A Semantic Foundation for TCOZ in Unifying Theories of Programming

Shengchao Qin[1], Jin Song Dong[2], and Wei-Ngan Chin[1,2]

[1] Singapore-MIT Alliance, National University of Singapore
[2] School of Computing, National University o Singapore
{qinsc,dongjs,chinwn}@comp.nus.edu.sg

Abstract. Unifying Theories of Programming (UTP) can provide a formal semantic foundation not only for programming languages but also for more expressive specification languages. We believe UTP is particularly well suited for presenting the formal semantics for integrated specification languages which often have rich language constructs for state encapsulation, event communication and real-time modeling. This paper uses UTP to formalise the semantics of Timed Communicating Object Z (TCOZ) and captures some TCOZ new features for the first time. In particular, a novel unified semantic model of the channel based synchronisation and sensor/actuator based asynchronisation in TCOZ is presented. This semantic model will be used as a reference document for developing tools support for TCOZ and as a semantic foundation for proving soundness of those tools.

Keywords: UTP, semantics, integrated formal specifications

1 Introduction

Formal semantics of specification languages provide foundations for language understanding, reasoning and tools construction. Various formal specification languages are often integrated for modeling large and complex systems. The development of the formal semantics for those integrated formal specifications provides some challenges due to the richness of the language constructs that facilitate complex states encapsulation, communication and real-time modeling. Hoare and He's Unifying Theories of Programming (UTP) [6] can present formal semantics not only for programming languages but also for specification languages. We believe UTP is particularly well suited for giving formal semantics for the integrated specification languages. One integrated formal notation namely Timed Communicating Object Z (TCOZ) [8] builds on the strengths of Object-Z [4, 16] and Timed CSP [13, 2] notations in order to provide a single notation for modeling both the state and process aspects of complex systems. In addition to CSP's channel-based communication mechanism (where messages represent discrete synchronisations between processes), TCOZ has recently been extended with asynchronous interface inspired by process control theory, sensors and actuators [7]. Based on the infinite failure model of Timed CSP, an enhanced semantics for TCOZ has been proposed [9] where the process behavioural aspects are focused. However, other important aspects of TCOZ were left out. In particular, it does not cover the semantics of the asynchronous communication mechanism of sensors and actuators. It is difficult to extend

K. Araki, S. Gnesi, and D. Mandrioli (Eds.): FME 2003, LNCS 2805, pp. 321–340, 2003.

that semantics to cover sensors and actuators because the meta framework used is based on events (channel), which is incompatible with the shared-variable nature of sensors and actuators.

This paper demonstrates how UTP can be used for constructing a formal observation-oriented model for TCOZ. In particular, a novel unified semantic model for both channel and sensors/actuators based communications is presented. This UTP model not only covers the TCOZ communication and process aspects, but also other features, such as class encapsulation, inheritance, dynamic binding and extended TCOZ timing constructs (deadline and waituntil commands), which have not been covered by the previous semantics. This semantic model will be used as a reference document and a semantic foundation for developing sound tools support for TCOZ. Our philosophy on tools support for integrated formal methods is to reuse/link existing tools especially graphical tools as much as possible. For example, one approach is to develop transformation rules from TCOZ to Timed Automata (TA) so that existing TA tools can be used to model check TCOZ timing properties, or to Message Sequence Chart (MSC) so that MSC tools can be used to analyse TCOZ's message passing and interaction behaviour. The proof of the soundness of those transformation rules can be based on this UTP semantic framework.

The remainder of the paper is organised as follows. Section 2 outlines the TCOZ syntax with a simple example. Section 3 starts with a brief introduction to UTP then presents the UTP observation model with meta variables. Section 4 develops the UTP semantics for TCOZ operations and processes. Section 5 presents the UTP semantics for TCOZ classes. Section 6 addresses related works with a conclusion and points out some future directions.

2 The TCOZ's Syntax and Example

The abstract syntax of TCOZ is given as follows.

Specification ::= CDecl; $\cdots$; CDecl

CDecl ::= $\upharpoonright$ VisibList; InheritC; StateSch; INIT; StaOp*; ProOp*; [MAIN]

VisibList ::= VisibAttr; VisibOp

VisibAttr ::= AttrName*

VisibOp ::= OpName*

InheritC ::= **Inherits** CName*

StateSch ::= VarDecl*; ChanDecl*; SenDecl*; ActDecl*

VarDecl ::= v : **T**

ChanDecl ::= ch : **chan**

SenDecl ::= sv : **T sensor**

ActDecl ::= sv : **T actuator**

StaOp ::= Δ(AttrName* | ActName*), VarDecl* $\bullet$ Pred($\underline{u}$,$\underline{v}$')

ProOp ::= VarDecl* $\bullet$ Process

MAIN ::= Process

$$\begin{array}{rl}
\mathsf{Process} ::= & \mathsf{Skip} \mid \mathsf{Stop} \mid \mathsf{Chaos}\ (\textit{primitives}) \\
& \mid \mathsf{StaOp}\ (\textit{state update})\ \mid \mathsf{Comm} \rightarrow \mathsf{Process}\ (\textit{communication}) \\
& \mid \mathsf{b} \bullet \mathsf{Process}\ (\textit{state guard})\ \mid \mathsf{Process} \rhd\{\mathsf{t}\}\ \mathsf{Process}\ (\textit{timeout}) \\
& \mid \textsc{Wait}\ \mathsf{t}\ (\textit{wait})\ \mid \mathsf{Process} \bullet \textsc{Deadline}\ \mathsf{t}\ (\textit{deadline}) \\
& \mid \mathsf{Process} \bullet \textsc{WaitUntil}\ \mathsf{t}\ (\textit{waituntil}) \\
& \mid \mathsf{Process};\ \mathsf{Process}\ (\textit{sequential composition}) \\
& \mid \mathsf{Process} \Box \mathsf{Process}\ (\textit{external choice}) \\
& \mid \mathsf{Process} \sqcap \mathsf{Process}\ (\textit{internal choice}) \\
& \mid \mathsf{Process}\ |[\mathsf{E}]|\ \mathsf{Process}\ (\textit{parallel composition}) \\
& \mid \mathsf{Process} \backslash \mathsf{E}\ (\textit{hiding})\ \mid \mu\,\mathsf{X} \bullet \mathsf{Process}\ (\textit{recursion})
\end{array}$$

$$\mathsf{Comm} ::= \mathsf{ch!e}\ (\textit{chan. ouput}) \mid \mathsf{b} \bullet \mathsf{ch?x}\ (\textit{chan. input}) \mid \mathsf{b} \bullet \mathsf{sv?x}\ (\textit{sensor read})$$

where b is a boolean condition, t is a time expression, E is a finite set of communication events, e is a message, and x is a variable.

Let us use a simple timed message queue system to illustrate the TCOZ notation. The behaviour of the following timed message queue system is that it can receive a new message (of type $[MSG]$) through an input channel '*in*' within a time duration 'T_j' or remove a message and send it through an output channel '*out*' within a time duration 'T_l'. If there is no interaction with environment within a certain time 'T_o', then a message will be removed from the current list but stored in a (window like) actuator list (*lost*) so that other objects (un-specified) with a sensor '*lost*' can read it at any time. The message queue has a FIFO property.

TimedQueue

$items : \mathrm{seq}\, MSG$
$in, out : \mathbf{chan}$
$lost : \mathrm{seq}\, MSG\ \mathbf{actuator}$
$T_l, T_j, T_o : \mathbb{N}$

INIT
$items = lost = \langle\ \rangle$

RecLost
$\Delta(lost)$
$lost' = \langle head(items) \rangle ^\frown lost$

Add
$\Delta(items)$
$i? : MSG$
$items' = items ^\frown \langle i? \rangle$

Del
$\Delta(items)$
$i! : MSG$
$items \neq \langle\ \rangle \Rightarrow items = \langle i! \rangle ^\frown items'$
$items = \langle\ \rangle \Rightarrow items' = \langle\ \rangle$

$Join \mathrel{\widehat=} [i : MSG] \bullet in?i \rightarrow Add \bullet \textsc{Deadline}\, T_j$
$Leave \mathrel{\widehat=} [items \neq \langle\ \rangle] \bullet out!head(items) \rightarrow Del \bullet \textsc{Deadline}\, T_l$
$\textsc{Main} \mathrel{\widehat=} \mu\, Q \bullet (Join \Box Leave) \rhd\{T_o\}\ (RecLost;\ Del) \bullet \textsc{Deadline}\, T_l;\ Q$

3 The UTP Observation Model

In the Unifying Theories of Programming (UTP), the relational/predicate calculus is adopted as a fundamental basis for unifying various programming theories across three dimensions: different computational paradigms, different levels of abstraction, and distinct mathematical representations. For each programming paradigm, specifications, designs, and programs are all interpreted as relations between an initial observation and a subsequent (intermediate stable or final) observation of the behaviour of their executions. Program correctness and refinement calculus can be represented by inclusion of relations. All the laws in a relational calculus are also valid in reasoning about correctness in all theories and languages.

Formal theories differ from one another by their alphabet, signature, and healthiness conditions. The *alphabet* of a theory is just a set of names used to record external observations of the behaviour. The names for initial observations are undecorated, whereas the names for subsequent observations are primed. The *signature* gives the way to represent the elements of the theory by taking primitives directly as elements and using operations to construct elements in an inductive manner. The *healthiness conditions* help filter out required elements for a sub-theory from those of a larger theory in which it is embedded. For example, in a top-down design process, programs are just a subset of intermediate designs, while designs are a subset of specifications.

To give a semantic model for the timed communicating language TCOZ, we need to choose an appropriate model of time. There are two typical models: a discrete model and a continuous one. The continuous model is very expressive and closer to the nature of real time. However, it is difficult to implement exactly for digital computer systems. On the other hand, the discrete model is implementable and closer to an untimed model. Timed CSP has a denotational semantics based on continuous time [2], and the existing semantics for TCOZ also adopts the continuous model [9]. However, to follow the objective of making our model simple and apt for exploration of algebraic refinement laws, we choose the discrete model. The discrete time model has also been adopted by the Sherif and He's work [14] on the semantics for timed Circus [17], which naturally extends Woodcock and Cavalcanti's semantics for Circus [18]. Although the general approach of the timed Circus semantics is adopted in our UTP semantic model for TCOZ processes, our semantic model contains many new aspects especially the formal treatment of both channel and sensor/actuator communication interfaces.

3.1 The Meta Process Model and Variables

TCOZ is mainly used to specify complex reactive systems. The behaviour of such a system can be modeled by observations of two kinds. The initial observation reflects the state of the system when the system starts to run. The follow-up observation records the state of the system when the system reaches a stable state. A stable state is either a termination state, in which the system terminates and the corresponding observation is called the final observation, or an intermediate waiting state, in which the system has no interaction with its environment and does not have infinite internal active events (not divergent) [6].

The process model starts with the above observations: at the initial and final (or intermediate stable) states of the system. Due to the timing feature of TCOZ, the observations on the interactions with the environment are enriched by adding time information. The existing model for Timed CSP and TCOZ attaches an explicit time stamp on each observation. The discrete model of time allows us to add time information implicitly. The interactions of a system with its environment are recorded as a sequence of tuples, each element of the sequence representing the observations over a single time unit. The first component of the tuple is a sequence of communication events or shared-variable updates which occur during a time unit. The second component represents a set of refused events (refusal) at the end of the time unit.

The following meta variables are introduced in the alphabet of the observations of the TCOZ process behaviour, some of them are similar to those in the previous UTP semantic frameworks [6, 14, 18]. The key difference is that timed trace has now been encoded with a set of shared-variable updates (due to sensors/actuators).

- ok, ok' : $\mathsf{Boolean}$. In order to analyse explicitly the phenomena of process initiation and termination, these variables are introduced to denote these observations.
  ok records the observation that the process has started. When ok is *false*, the process has not started, so no observation can be made.
  ok' records the observation that the process has terminated or has reached an intermediate stable state. The process is divergent when ok' is *false*.
- $wait, wait'$: $\mathsf{Boolean}$. Because of the requirement for synchronisation, an active process will usually engage in alternate periods of internal activity (computation) and periods of quiescence or stability, while it is waiting for a reaction or an acknowledgement from its environment. We therefore introduce a variable $wait'$, which is *true* just when a process is waiting in such quiescent periods. Its main purpose is to distinguish intermediate observations from the observations made on termination. $wait$ is used in the initial observation, which is *true* when the process starts in an intermediate state.
- $state, state'$: $\mathsf{Var} \to \mathsf{Value}$. In order to record the state of data variables (class attributes/local variables) that occur in a process, these two variables are introduced to associate resp. every variable with its value in the corresponding observations.
- tr, tr' : $\mathsf{seq}(\mathsf{seq}\,(Event \cup Update) \times \mathbb{P}\,Event)$. Each of these two variables records a sequence of observations on the process's interactions with its environment. tr records the observations that occur before the process starts, and tr' records the observations that take place so far. Each element of the sequence denotes the observations over one time unit, which is specified by a tuple. The first component of the tuple is the sequence of communication events or updates on sensor-actuator variables that occur during the time unit, the second is an associated set of refusals at the end of the time unit.
  The set *Event* denotes all possible communicating events. The set *Update*, defined as $Update =_{df} ((\mathsf{SV} \to \mathsf{Value}) \times \mathsf{Tag})$, represents the set of all possible updates (states) of all sensor-actuator variables (SV). The binary set $\mathsf{Tag} =_{df} \{0, 1\}$ shows which process is making the current update: 1 indicates that current update is made by the current process, whereas 0 indicates that current update is due to an environmental process.
- $trace$: $\mathsf{seq}\,(Event \cup Update)$. This variable is used to record a sequence of events/updates that take place so far since the last observation. It can be derived from tr, tr' by taking their difference as follows:

  $flat(tr) ^\frown trace = flat(tr')$, where $^\frown$ is the concatenation operator, and
  $flat : \mathsf{seq}(\mathsf{seq}\,(Event \cup Update) \times \mathbb{P}\,Event) \to \mathsf{seq}\,(Event \cup Update)$

$$\mathit{flat}(\langle\rangle) =_{df} \langle\rangle \qquad \mathit{flat}(\langle(es, ref)\rangle \frown tr) =_{df} es \frown \mathit{flat}(tr)$$

Two auxiliary functions $cs(trace), ds(trace)$ are adopted to extract resp. the subsequences of communication events and shared-variable states from the sequence *trace*. The function *cs* is defined as

$$cs(\langle\rangle) =_{df} \langle\rangle \qquad cs(\langle e\rangle \frown tail) =_{df} \begin{cases} \langle e\rangle \frown cs(tail), & \text{if } e \in Event, \\ cs(tail), & \text{otherwise.} \end{cases}$$

The function *ds* can be defined similarly.

- $gs : \mathsf{SV} \to \mathsf{Value}$. This variable is used to hold the latest updated state of all shared sensor-actuator variables.

In our semantics model, the observation-based semantics for a TCOZ process will be described by a predicate whose alphabet contains the above variables [6].

A binary relation $\overset{t}{\preceq}$ is defined over two sequences of observations as follows.

$$tr_1 \overset{t}{\preceq} tr_2 =_{df} (\mathit{front}(tr_1) \preceq tr_2) \wedge (\pi_1(\mathit{last}(tr_1)) \preceq \pi_1(tr_2(\#tr_1)))$$

where $\preceq$ is the ordinary subsequence relation between sequences of the same type. $\mathit{front}(tr)$ is the initial part of *tr* obtained by dropping those observations recorded in last time unit. $\mathit{last}(tr)$ gets the last element of the sequence *tr*. $\pi_1(tup)$ returns the first component of the tuple *tup*. $\#tr$ is the number of elements in *tr*, while $tr(n)$ returns the *n*th element.

This definition states that, given two timed traces, tr_1 and tr_2, tr_2 is an expansion of tr_1, if the initial part of tr_1 is a subsequence of tr_2, and the untimed traces recorded at the last time unit of tr_1 is a subsequence of the untimed traces at the same time in tr_2.

Since the execution of a process can never *undo* any action performed previously, each trace can only get longer. The current value of *tr* must therefore always be an expansion of its initial value. Hereby, the semantics predicate *P* for any process P should satisfy the healthiness condition **R** defined as follows:

$$\mathbf{R}(P) =_{df} P = (P \wedge tr \overset{t}{\preceq} tr')$$

3.2 The Class Model

TCOZ has two kinds of classes, active and passive ones. The behaviour of (an object of) an active class can be specified by a record of its continuous interactions with its environment via its MAIN process, whereby any update on its data state is hidden. Passive class does not have its own thread of control and its state and operations (processes) are available for use by its controlling object. We model an active class as a predicate with an assumption and a commitment (also known as design in [6]), and a passive class as a service provider, which provides a set of services to its environment.

In order to address issues like class encapsulation and dynamic typing that are essential for object-orientation, the following TCOZ features are considered in the UTP model.

1. An object-oriented specification contains not only variables of simple types but also objects. To ensure a legal access to a variable, the model is equipped with a set of visible attributes/operations.
2. Due to the subclass mechanism, an object can lie in a subclass of its originally declared one. Therefore, the behaviour of its operations will depend on its current type. To support such a dynamic binding mechanism for operation calls, our model keeps track of the dynamic type for each object. This enables us to validate operations in a framework where the type of each variable is properly recorded.
3. A value of an object variable is a finite tuple, which may record the current type of the object, and the values of its attributes. Since an object may contain attributes of object types, its value is often defined with nested recursions.

In order to address the above issues clearly, the following meta variables are introduced to keep track of the class information.

- *CN* and *super* are used to record the contextual information on classes and their relationships. *CN* is the set of classes already declared, *super* is a partial function which maps a class to the set of its direct superclasses. For example, $C_1 \in super(C_2)$ states that C_1 is a direct superclass of C_2. C is a *superclass* of C' if there exists a finite sequence of classes $C_0, \cdots, C_n$, such that $C = C_n$ and $C' = C_0$ and $C_{i+1} \in super(C_i)$ for all $0 \leqslant i < n$. We use the set $super^+(C)$ to denote all superclasses of C, and $super^*(C)$ to present all superclasses of C and itself. Note that $super^*(C) =_{df} super^+(C) \cup \{C\}$.
- For each class $C \in CN$, we use the following notations to denote its structure and record different variables involved in its specification.
  - The set of state attributes of class C, $attr(C) = \{\langle a_1 : T_1\rangle, \cdots, \langle a_m : T_m\rangle\}$, comprises both the attributes declared in C and those that C inherits from its superclasses, where T_i stands for the type of attribute a_i of class C, and will be referred by $type(C.a_i)$. The set of channels declared in class C is denoted by $chan(C) = \{ch_1, \cdots, ch_n : \mathbf{chan}\}$.
  - The set of operations declared or inherited by C, $op(C) = op_s(C) \cup op_p(C)$. It is composed of a set of state operations ($op_s(C)$) and a set of process operations ($op_p(C)$).
  - *senvar*, *actvar*: the set of sensor and actuator variables declared in current class or inherited from its superclasses. They provide an interface between the control system and its controlled system.
  - *locvar*: the set of local definitions, $\{v_1 : T_1, \cdots, v_m : T_m\}$;
  - *visibattr*, *visibop*: the set of visible state attributes and visible operations.

For notational convenience, we assume the following four sets of names are pairwise disjoint: classes, attributes, operations and (local or shared) variables.

A state binds variables to their current values. A variable of a primitive data type can take any value of that type. The value of an object variable is composed of the values of its attributes together with its current type (as in [5]):

$$\{a \mapsto value \mid a \in attr(C)\} \cup \{myclass \mapsto C\}$$

In what follows, we investigate the observation-based semantics of TCOZ processes, and as well explore some associated algebraic laws. After that, we formalise the TCOZ class semantics. Following the notation style in UTP [6], we adopt the italic format to represent semantic notations (e.g., predicates), whereas we use the sans serif format to denote syntactic notations (e.g., specifications) in this paper. For instance, the semantics of a process P is simply represented by a predicate P, rather than $[\![\mathsf{P}]\!]$.

4 Process Semantics

In this section, the observation model for TCOZ processes is developed. Some process models that are similar to [14] are moved to the Appendix.

4.1 Communication

This subsection is devoted to communications. Other primitives Chaos, Skip and Stop are presented in the Appendix.

A synchronisation $\mathsf{ch.e}$ can take place only if an output event $\mathsf{ch!e}$ is ready, an input event $\mathsf{b} \bullet \mathsf{ch?x}$ is also ready, and the message to be passed satisfies the condition b.

In order to describe the behaviour of these two primitives, we introduce two auxiliary predicates, $com\_blk(ch)$ and $com\_syn(ch)$, to represent the waiting behaviour for communication and the synchronised communication respectively.

$$\begin{array}{ll} com\_blk(ch) =_{df} & ok' \wedge wait' \wedge no\_interact(trace) \wedge not\_ref(tr, tr', ch) \\ com\_syn(ch.e) =_{df} & ok' \wedge \neg wait' \wedge trace = \langle ch.e \rangle \wedge \#tr' = \#tr \end{array}$$

Note that predicate $not\_ref(tr, tr', ch)$ is true if any events with respect to channel ch do not occur in the refusals of the observations recorded from tr to tr'.

$$not\_ref(tr, tr', ch) =_{df} \forall n : \#tr \leq n \leq \#tr' \bullet ch \notin \pi_2(tr'(n))$$

The predicate $no\_interact(trace)$ denotes that there are no communication events recorded in $trace$, while the shared-variable updates recorded in $trace$ (if any) are due to the environmental process. That is, for any $s \in \mathsf{seq}(Event \times Update)$,

$$no\_interact(s) =_{df} cs(s) = \langle\rangle \wedge \forall u \in ds(s) \bullet \pi_2(u) = 0$$

An output primitive $\mathsf{ch!e}$ stays in a waiting state before some other process becomes ready to receive a message via the channel ch, or finishes the communication instantaneously once the receiver is ready.

$$ch!e =_{df} com\_blk(ch) \vee (com\_blk(ch) \circ (com\_syn(ch.e) \wedge state' = state))$$

where the operator $\circ$ is the composition of two sequentially made observations. For two observation predicates $P(\underline{v}, \underline{v}')$, $Q(\underline{v}, \underline{v}')$, where $\underline{v}, \underline{v}'$ represent respectively the initial and final versions of all observation variables, the composition of them is

$$P(\underline{v}, \underline{v}') \circ Q(\underline{v}, \underline{v}') =_{df} \exists \underline{v}_0 \bullet P(\underline{v}, \underline{v}_0) \wedge Q(\underline{v}_0, \underline{v}')$$

Note that the final observation from P coincides with the initial observation from Q.

For the input primitive b • ch?x, if the message to be passed does not satisfy the condition b, it results in deadlock. Once this communication occurs, the value passed along the channel will be assigned to the variable x and recorded in the state.

$$\begin{aligned} b \bullet ch?x =_{df}\ & com\_blk(ch) \ \vee \ (com\_blk(ch) \circ \\ & (b[e/x] \wedge com\_syn(ch.e) \wedge state' = state \oplus \{x \mapsto e\} \ \vee \\ & \neg b[e/x] \wedge Stop)) \end{aligned}$$

The guarded sensor read command b(x) • sv?x is defined in terms of the following recursive process. Intuitively, it consecutively reads values from the sensor (once per time unit) until the sensed value meets the guard.

$$\mathsf{b(x)} \bullet \mathsf{sv?x} =_{df} \ \mu\, \mathsf{X} \bullet \mathsf{sv?x} \rightarrow ((\mathsf{b(x)} \bullet \mathsf{Skip}) \,\Box\, (\neg \mathsf{b(x)} \bullet (\mathrm{WAIT}\ 1;\ \mathsf{X})))$$

where the simple read sv?x obtains the latest value of the sensor-actuator variable sv.

$$sv?x =_{df} \ ok' \wedge \neg wait' \wedge tr' = tr \wedge state' = state \oplus \{x \mapsto gs(sv)\}$$

The simple prefix process Comm→ P is explained as a sequential composition of the communication behaviour and the behaviour of the process that follows.

$$Comm \rightarrow P =_{df} \ Comm;\ P$$

Semantics for sequential composition is presented in the Appendix.

4.2 State Operation

There are two kind of state operations, one only updates the local state of the current class, whereas the other updates the global state, i.e., the sensor-actuator variables that it is in charge of.

Local State Update. A local state operation $\Delta(\underline{\mathsf{y}}), \underline{\mathsf{x}} : \underline{\mathsf{T}} \bullet \mathsf{Pred}(\underline{\mathsf{u}}, \underline{\mathsf{v}}')$ enlarges the state with its local definitions and updates the state afterwards.

$$\begin{aligned} & \Delta(\underline{y}), \underline{x} : \underline{T} \bullet Pred(\underline{u}, \underline{v}') =_{df} \ ok' \wedge \neg wait' \wedge no\_interact(trace) \wedge \\ & ((\exists\, \underline{val}_1 \bullet state' = state \oplus \{\underline{x} \mapsto \underline{val}_1\}) \circ \\ & \ (\exists\, \underline{val} \bullet state' = state \oplus \{\underline{v} \mapsto \underline{val}\} \wedge Pred(state(\underline{u}), state'(\underline{v})))) \end{aligned}$$

Actuator Update. An actuator update operation $\Delta(\underline{\mathsf{sv}}), \underline{\mathsf{x}} : \underline{\mathsf{T}} \bullet \mathsf{Pred}(\underline{\mathsf{u}}, \underline{\mathsf{sv}}, \underline{\mathsf{sv}}')$ specifies that expected values can be assigned to the sensor-actuator variables $\underline{\mathsf{sv}}$.

$$\begin{aligned} & \Delta(\underline{sv}), \underline{x} : \underline{T} \bullet Pred(\underline{u}, \underline{sv}, \underline{sv}') =_{df} \ ok' \wedge \neg wait' \wedge \#tr' = \#tr \wedge \\ & \exists\, \underline{val} \bullet gs' = gs \oplus \{\underline{sv} \mapsto \underline{val}\} \wedge ((\exists\, \underline{val}_1 \bullet state' = state \oplus \{\underline{x} \mapsto \underline{val}_1\}) \circ \\ & Pred(state(\underline{u}), gs(\underline{sv}), gs'(\underline{sv}))) \wedge trace = \langle (gs', 1) \rangle \end{aligned}$$

where gs and gs' indicate the value of the variable gs resp. before and after the update.

In our model, consecutive actuator update operations are combined into one atomic update operation. Therefore, the above update list can be a list of actuator variables.

4.3 Timeout Process

The timeout process $\mathsf{P} \rhd\{t\}\ \mathsf{Q}$ behaves as P if P has no interaction with the environment at all but terminates within time t, or it reacts to the environment within time t, otherwise it behaves as Q.

$$\begin{array}{l} P \rhd\{t\}\ Q =_{df} (P \wedge no\_interact(trace) \wedge \#tr' - \#tr \leqslant t) \vee \\ (\exists\, k : \#tr < k \leqslant \#tr{+}t, \exists\, \tilde{tr} \bullet \pi_1(tr'(k)) \neq \langle\rangle \wedge tr \preceq \tilde{tr} \wedge \#\tilde{tr} - \#tr = k \wedge \\ \ (\forall\, i : \#tr < i < \#tr{+}k \bullet no\_interact(\pi_1(tr'(i))) \wedge \tilde{tr}(i) = tr'(i)) \wedge P[\tilde{tr}/tr]) \ \vee \\ (\exists\, \tilde{tr} \bullet tr \preceq \tilde{tr} \wedge \#\tilde{tr} - \#tr = t \wedge \\ \ (\forall\, i : \#tr < i < \#tr{+}t \bullet no\_interact(\pi_1(tr'(i))) \wedge \tilde{tr}(i) = tr'(i)) \wedge Q[\tilde{tr}/tr]) \end{array}$$

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.

The following are some algebraic laws that can be derived from our semantic definition. For simplicity, the proofs are omitted.

T1. $\mathsf{P} \rhd\{t\}\ \mathsf{P} = \mathsf{P}$
T2. $\mathsf{Skip} \rhd\{t\}\ \mathsf{P} = \mathsf{Skip}$
T3. $(\mathsf{a} \to \mathsf{P}) \rhd\{t\}\ (\mathsf{b} \to \mathsf{P}) = ((\mathsf{a} \to \mathsf{Skip}) \rhd\{t\}\ (\mathsf{b} \to \mathsf{Skip}));\ \mathsf{P}$
T4. $\mathsf{P} \rhd\{t\}\ (\mathsf{Q} \sqcap \mathsf{R}) = (\mathsf{P} \rhd\{t\}\ \mathsf{Q}) \sqcap (\mathsf{P} \rhd\{t\}\ \mathsf{R})$
T5. $(\mathsf{P} \sqcap \mathsf{Q}) \rhd\{t\}\ \mathsf{R} = (\mathsf{P} \rhd\{t\}\ \mathsf{R}) \sqcap (\mathsf{Q} \rhd\{t\}\ \mathsf{R})$

4.4 Wait

The process WAIT t just waits for t time units to pass before terminating immediately. It can be defined as follows in terms of timeout construct defined in section 4.3.

$$Wait\ \ t\ =_{df}\ Stop \rhd\{t\}\ Skip$$

It is subject to the following laws.

W1. $\textsc{Wait}\ \ \mathsf{t}_1;\ \textsc{Wait}\ \ \mathsf{t}_2 = \textsc{Wait}\ \ (\mathsf{t}_1 + \mathsf{t}_2)$
W2. $(\textsc{Wait}\ \ \mathsf{t}_1)\ |[\mathsf{E}]|\ (\textsc{Wait}\ \ \mathsf{t}_2) = \textsc{Wait}\ \ (\mathbf{max}(\mathsf{t}_1, \mathsf{t}_2))$
W3. $\mathsf{Stop} \rhd\{t\}\ \mathsf{P} = \textsc{Wait}\ \ \mathsf{t};\ \mathsf{P}$

4.5 Deadline

The *Deadline* construct P • DEADLINE t imposes a timing constraint on a specification P, which requires the computation of P to be finished within time t.

$$P \bullet Deadline\ t\ =_{df}\ P \wedge (\#tr' - \#tr \leqslant t)$$

It enjoys the following properties.

D1. $\mathsf{P} \bullet \textsc{Deadline}\ \ \mathsf{t}_1 \bullet \textsc{Deadline}\ \ \mathsf{t}_2\ =\ \mathsf{P} \bullet \textsc{Deadline}\ \ \mathbf{min}(\mathsf{t}_1, \mathsf{t}_2)$
D2. $(\mathsf{P} \sqcap \mathsf{Q}) \bullet \textsc{Deadline}\ \ \mathsf{t}\ =\ (\mathsf{P} \bullet \textsc{Deadline}\ \ \mathsf{t}) \sqcap (\mathsf{Q} \bullet \textsc{Deadline}\ \ \mathsf{t})$

4.6 WaitUntil

In case that P terminates within time t, the *WaitUntil* construct $\mathsf{P} \bullet \textsc{WaitUntil}\ \mathsf{t}$ has to keep waiting after the termination of P until t time units have passed.

$$\begin{aligned} P \bullet \mathit{WaitUntil}\ t \ =_{df} \ & (\exists \tilde{tr}' \bullet tr \preceq \tilde{tr}' \preceq tr' \wedge (\#\tilde{tr}' - \#tr < t) \wedge \\ & (P[\tilde{tr}'/tr', \mathit{true}/ok', \mathit{false}/\mathit{wait}'] \circ \\ & (\mathit{Wait}\ (t - (\#\tilde{tr}' - \#tr))[\tilde{tr}/tr]))) \ \vee \ P \wedge (\#tr' - \#tr \geqslant t) \end{aligned}$$

It enjoys the following properties.

U1. $\mathsf{P} \bullet \textsc{WaitUntil}\ \mathsf{t}_1 \bullet \textsc{WaitUntil}\ \mathsf{t}_2 \ = \ \mathsf{P} \bullet \textsc{WaitUntil}\ \mathbf{max}(\mathsf{t}_1, \mathsf{t}_2)$
U2. $(\mathsf{P} \sqcap \mathsf{Q}) \bullet \textsc{WaitUntil}\ \mathsf{t} \ = \ (\mathsf{P} \bullet \textsc{WaitUntil}\ \mathsf{t}) \sqcap (\mathsf{Q} \bullet \textsc{WaitUntil}\ \mathsf{t})$

4.7 State-Guarded Process

The state-guarded process $\mathsf{b} \bullet \mathsf{P}$ behaves as P if the condition b is initially satisfied, otherwise it waits for ever (like the process Stop).

$$b \bullet P \ =_{df} \ b \wedge P \vee \neg b \wedge \mathit{Stop}$$

It satisfies the following properties.

G1. $\mathit{false} \bullet \mathsf{P} = \mathsf{Stop}$
G2. $\mathit{true} \bullet \mathsf{P} = \mathsf{P}$
G3. $\mathsf{b} \bullet \mathsf{Stop} = \mathsf{Stop}$
G4. $\mathsf{b} \bullet (\mathsf{c} \bullet \mathsf{P}) = (\mathsf{b} \wedge \mathsf{c}) \bullet \mathsf{P}$
G5. $\mathsf{b} \bullet (\mathsf{P};\ \mathsf{Q}) = (\mathsf{b} \bullet \mathsf{P});\ \mathsf{Q}$

4.8 Parallel Composition

The parallel composition of two processes represents all the possible behaviours of both processes which are not only synchronised on a specific set of events and on the time when these events occur, but also coincide with each other on the state of sensor-actuator variables at each update. The overall process will terminate when both component processes do.

The parallel composition is defined in terms of the general parallel merge operator $\|_M$ in UTP [6], where the predicate M denotes the way to merge two observations.

In the following definition, our new merge predicate $M(E)$ is in charge of both channel based communications and shared-variable updates, due to the existence of two distinct communication mechanisms (channel and sensor/actuator) in TCOZ.

$$\begin{aligned} P \ |[E]|\ Q \ =_{df} \ & (((P;\ \mathit{idle}) \|_{M(E)} Q) \vee (P \|_{M(E)} (Q;\ \mathit{idle}))); \\ & ((ok \Rightarrow \mathit{Skip}) \wedge (\neg ok \Rightarrow tr \overset{t}{\preceq} tr')) \end{aligned}$$

An *idle* process, which may either wait or terminate, follows after each of the two processes. This is to allow each of the processes to wait for its partner to terminate.

$$\mathit{idle} \ =_{df} \ ok' \wedge \mathit{no\_interact}(\mathit{trace}) \wedge \mathit{state}' = \mathit{state}$$

The merge predicate $M(E)$ is defined as

$$\begin{aligned} M(E) =_{df}\ & ok' = (0.ok \wedge 1.ok) \wedge wait' = (0.wait \vee 1.wait) \wedge \\ & state' = (0.state \oplus 1.state) \wedge \\ & tr' \in syn(0.tr, 1.tr, E) \wedge \#tr' = \#0.tr = \#1.tr \ \wedge \\ & \forall i : \#tr..\#tr' \bullet consistent(ds(\pi_1(0.tr(i))), ds(\pi_1(1.tr(i)))) \end{aligned}$$

Given two timed traces tr_1, tr_2, and a set of events E, the set $syn(tr_1, tr_2, E)$ is defined inductively as follows.

$$\begin{aligned} & syn(tr_1, tr_2, E) =_{df} syn(tr_2, tr_1, E) \\ & syn(\langle\rangle, \langle\rangle, E) =_{df} \{\langle\rangle\} \\ & syn(\langle(t, r)\rangle, \langle\rangle, E) =_{df} \{\langle(t', r)\rangle \mid t' \in (t \underset{E\,U}{\parallel} \langle\rangle)\} \\ & syn(\langle(t_1, r_1)\rangle ^\frown tr_1, \langle(t_2, r_2)\rangle ^\frown tr_2, E) =_{df} \\ & \qquad \{\langle(t', r')\rangle ^\frown u \mid t' \in (t_1 \underset{E\,U}{\parallel} t_2) \wedge r' = r_1 \cup r_2 \wedge u \in syn(tr_1, tr_2, E)\} \end{aligned}$$

The predicate $consistent(s_1, s_2)$ specifies that two sequences of updates on shared variables are consistent. It is used in the above definition to ensure that two individual records of shared-variable updates coincide with each other in every time unit.

$$\begin{aligned} consistent(s_1, s_2) =_{df}\ & \#s_1 = \#s_2 \wedge \forall i : 1..\#s_1 \bullet (\pi_1(s_1(i)) = \pi_1(s_2(i)) \wedge \\ & \pi_2(s_1(i)) + \pi_2(s_2(i)) \neq 2) \end{aligned}$$

$s \underset{E\,U}{\parallel} t$ is used to merge untimed traces s and t into one untimed trace, where E is the set of events to be synchronised, U is the set of possible shared-variable updates. In comparison to Roscoe's model for the parallel merge of untimed traces [12], the following definition is more sophisticated as it also captures the shared variable communications. In the following clauses, e, e_1, e_2 are representative elements of E (events), u, u_1, u_2 are representative elements of U (updates), whereas x, x_1, x_2 represent communication events not residing in E.

$$\begin{aligned} & s \underset{E\,U}{\parallel} t =_{df} t \underset{E\,U}{\parallel} s \qquad \langle\rangle \underset{E\,U}{\parallel} \langle\rangle =_{df} \{\langle\rangle\} \qquad \langle e\rangle \underset{E\,U}{\parallel} \langle\rangle =_{df} \{\} \\ & \langle u\rangle \underset{E\,U}{\parallel} \langle\rangle =_{df} \{\} \qquad \langle x\rangle \underset{E\,U}{\parallel} \langle\rangle =_{df} \{\langle x\rangle\} \\ & \langle x\rangle ^\frown s \underset{E\,U}{\parallel} \langle e\rangle ^\frown t =_{df} \{\langle x\rangle ^\frown l \mid l \in (s \underset{E\,U}{\parallel} \langle e\rangle ^\frown t)\} \\ & \langle e\rangle ^\frown s \underset{E\,U}{\parallel} \langle e\rangle ^\frown t =_{df} \{\langle e\rangle ^\frown l \mid l \in (s \underset{E\,U}{\parallel} t)\} \\ & \langle e_1\rangle ^\frown s \underset{E\,U}{\parallel} \langle e_2\rangle ^\frown t =_{df} \{\}, \text{ where } e_1 \neq e_2 \\ & \langle u_1\rangle ^\frown s \underset{E\,U}{\parallel} \langle u_2\rangle ^\frown t =_{df} \begin{cases} \{\}, \text{ if } \neg consistent(\langle u_1\rangle, \langle u_2\rangle) \\ \{u ^\frown l \mid join(\langle u\rangle, \langle u_1\rangle, \langle u_2\rangle) \wedge l \in (s \underset{E\,U}{\parallel} t)\}, \text{ otherwise} \end{cases} \\ & \langle x\rangle ^\frown s \underset{E\,U}{\parallel} \langle u\rangle ^\frown t =_{df} \{\langle x\rangle ^\frown l \mid l \in (s \underset{E\,U}{\parallel} \langle u\rangle ^\frown t)\} \\ & \langle e\rangle ^\frown s \underset{E\,U}{\parallel} \langle u\rangle ^\frown t =_{df} \{\} \\ & \langle x_1\rangle ^\frown s \underset{E\,U}{\parallel} \langle x_2\rangle ^\frown t =_{df} \{\langle x_1\rangle ^\frown l \mid l \in (s \underset{E\,U}{\parallel} \langle x_2\rangle ^\frown t)\} \cup \{\langle x_2\rangle ^\frown l \mid l \in (\langle x_1\rangle ^\frown s \underset{E\,U}{\parallel} t)\} \end{aligned}$$

The predicate $join(s, s_1, s_2)$ merges two consistent sequences of updates (s_1 and s_2) into one overall sequence (s).

$$\begin{aligned} join(s, s_1, s_2) =_{df}\ & consistent(s_1, s_2) \wedge \#s = \#s_1 \wedge \\ & \forall\, i : 1..\#s_1 \bullet (\pi_1(s(i)) = \pi_1(s_1(i)) \wedge \\ & \pi_2(s(i)) = \pi_2(s_1(i)) + \pi_2(s_2(i))) \end{aligned}$$

The following are some properties that parallel composition owns.

P1. $\mathsf{Chaos} \mathbin{|[\mathsf{E}]|} \mathsf{P} = \mathsf{Chaos}$
P2. $\mathsf{Stop} \mathbin{|[\mathsf{E}]|} \mathsf{P} = \mathsf{Stop}$
P3. $\mathsf{P} \mathbin{|[\mathsf{E}]|} \mathsf{Q} = \mathsf{Q} \mathbin{|[\mathsf{E}]|} \mathsf{P}$
P4. $\mathsf{P} \mathbin{|[\mathsf{E}_1]|} (\mathsf{Q} \mathbin{|[\mathsf{E}_2]|} \mathsf{R}) = (\mathsf{P} \mathbin{|[\mathsf{E}_1]|} \mathsf{Q}) \mathbin{|[\mathsf{E}_2]|} \mathsf{R}$
P5. $\mathsf{P} \mathbin{|[\mathsf{E}]|} (\mathsf{Q} \sqcap \mathsf{R}) = (\mathsf{P} \mathbin{|[\mathsf{E}]|} \mathsf{Q}) \sqcap (\mathsf{P} \mathbin{|[\mathsf{E}]|} \mathsf{R})$

Definitions for *sequential composition*, *internal/external choices*, *recursion*, and *hiding* are presented in the Appendix, which are similar to the definitions in [14].

5 Class Semantics

This section aims to deal with class declarations, their well-definedness and their composition.

Given a class declaration *cdecl* as follows.

C

$\upharpoonright (\mathsf{VisibAttr}, \mathsf{VisibOp})$

Inherits $\underline{\mathsf{C}'}$

$\underline{\mathsf{lv}} : \underline{\mathbf{T}}$

$\underline{\mathsf{a}} : \underline{\mathsf{T}}^a$
$\underline{\mathsf{ch}} : \mathbf{chan}$
$\underline{\mathsf{sv}}_1 : \underline{\mathsf{T}}_1^s$ **sensor**
$\underline{\mathsf{sv}}_2 : \underline{\mathsf{T}}_2^s$ **actuator**

INIT
b

m_1
$\Delta(\underline{\mathsf{y}}_1),\ \underline{\mathsf{x}}_1 : \underline{\mathsf{T}}_1^p \bullet \mathsf{Pred}(\mathsf{u}_1, \mathsf{v}_1')$

...

m_k
$\Delta(\underline{\mathsf{y}}_k),\ \underline{\mathsf{x}}_k : \underline{\mathsf{T}}_k^p \bullet \mathsf{Pred}(\mathsf{u}_k, \mathsf{v}_k')$

$\mathsf{m}_{k+1} \mathrel{\widehat{=}} [\underline{\mathsf{x}}_{k+1} : \underline{\mathsf{T}}_{k+1}^p] \bullet \mathsf{P}_{k+1}$
...
$\mathsf{m}_n \mathrel{\widehat{=}} [\underline{\mathsf{x}}_n : \underline{\mathsf{T}}_n^p] \bullet \mathsf{P}_n$

$[\mathrm{MAIN} \mathrel{\widehat{=}} \mathsf{P}]$

where

- C is the name of the class which is declared as a direct subclass of classes $\underline{\mathsf{C}'}$.
- The names of visible attributes and operations are listed in VisibList(resp. in VisibAttr and VisibOp).
- $\mathsf{m}_1, \cdots, \mathsf{m}_n$ are operations declared in C. $\Delta(\underline{\mathsf{y}}_i)$ states that only attributes (or actuators) $\underline{\mathsf{y}}_i$ can be modified by m_i. $\underline{\mathsf{x}}_i : \underline{\mathsf{T}}_i^p$ are the parameters of the operation m_i. The set of operations is divided into two parts, the first part, $\mathsf{m}_1, \cdots, \mathsf{m}_k$, called *state*

operations, represent operations in Object-Z style, where the body is specified by a predicate. The second part, $\mathsf{m}_{k+1}, \cdots, \mathsf{m}_n$, called *process operations*, are operations in process style, where the body is specified by a process.
- the MAIN operation is optional. If it is present in the definition, the class is called an *active* class. Otherwise, it is called a *passive* class.

We first discuss the passive class where the MAIN operation is absent. A passive class declaration *cdecl* is well-defined, denoted by $\mathcal{WD}(cdecl)$, if it satisfies the following conditions: (1) C is distinct from $\underline{\mathsf{C}}'$, (2) the following names are distinct: local variables, state attributes, channels, sensors, actuators, operations, operation parameters, (3) each state operation can only modify the attributes or actuators in its Δ-list, (4) the $\mathsf{VisibAttr}$ and $\mathsf{VisibOp}$ are resp. subsets of the attributes and operations declared in the current class or inherited from its superclasses, (5) each Δ-list in state operations should be names of attributes or actuators (declared in current class or inherited from superclasses), (6) the set of sensors and the set of actuators should also include those inherited from superclasses. The last three conditions cannot be tested based on an individual class declaration, but can be checked at the end of all class declarations. Formally, the well-definedness of the above class declaration given for C is defined by the following predicate.

$$\mathcal{WD} =_{df} \left\{ \begin{array}{l} \mathsf{C} \notin \{\underline{\mathsf{C}}'\} \wedge type(\underline{\mathsf{a}}) = \underline{\mathsf{T}}^a \wedge \#\underline{\mathsf{a}} = \#\underline{\mathsf{T}}^a \wedge type(\underline{\mathsf{lv}}) = \underline{\mathbf{T}} \wedge \#\underline{\mathsf{lv}} = \#\underline{\mathbf{T}} \\ type(\underline{\mathsf{sv}}_1) = \underline{\mathsf{T}}_1^s \wedge \#\underline{\mathsf{sv}}_1 = \#\underline{\mathsf{T}}_1^s \wedge type(\underline{\mathsf{sv}}_2) = \underline{\mathsf{T}}_2^s \wedge \#\underline{\mathsf{sv}}_2 = \#\underline{\mathsf{T}}_2^s \wedge \\ \forall\, i \bullet (dif(\langle \underline{\mathsf{lv}} \rangle ^\frown \langle \underline{\mathsf{a}} \rangle ^\frown \langle \underline{\mathsf{ch}} \rangle ^\frown \langle \underline{\mathsf{sv}}_1 \rangle ^\frown \langle \underline{\mathsf{sv}}_2 \rangle ^\frown \langle \mathsf{m}_1, \cdots, \mathsf{m}_n \rangle ^\frown \langle \underline{\mathsf{x}}_i \rangle) \\ \wedge \#\underline{\mathsf{x}}_i = \#\underline{\mathsf{T}}_i^p) \wedge \forall\, i : 1..k \bullet \{\mathsf{v}_i\} \subseteq \{\underline{\mathsf{y}}_i\} \cup \{\underline{\mathsf{x}}_i\} \end{array} \right\}$$

where $dif(\langle e_1, \cdots, e_n \rangle) =_{df} \forall\, i, j : 1..n \bullet i \neq j \Rightarrow e_i \neq e_j$.

The class declaration *cdecl* provides the structural information of class C to the state of the system, and its role is specified by the following design.

$$cdecl =_{df} \mathcal{WD} \vdash \left\{ \begin{array}{l} locvar' = \{\mathsf{C} \mapsto \{\underline{\mathsf{lv}} : \underline{\mathbf{T}}\}\} \wedge CN' = \{\mathsf{C}\} \wedge \\ super' = \{\mathsf{C} \mapsto \mathsf{C}_i \mid \mathsf{C}_i \in \underline{\mathsf{C}}\text{'}\} \wedge \\ visibattr' = \{\mathsf{C} \mapsto \mathsf{VisibAttr}\} \wedge \\ visibop' = \{\mathsf{C} \mapsto \mathsf{VisibOp}\} \wedge attr' = \{\mathsf{C} \mapsto \{\underline{\mathsf{a}} : \underline{\mathsf{T}}^a\}\} \wedge \\ senvar' = \{\mathsf{C} \mapsto \{\underline{\mathsf{sv}}_1 : \underline{\mathsf{T}}_1^s\}\} \wedge \\ actvar' = \{\mathsf{C} \mapsto \{\underline{\mathsf{sv}}_2 : \underline{\mathsf{T}}_2^s\}\} \wedge \\ chan' = \{\mathsf{C} \mapsto \{\underline{\mathsf{ch}}\}\} \wedge op' = op_s' \cup op_p' \wedge \\ op_s' = \{\, \mathsf{C} \mapsto \{\mathsf{m}_1 \mapsto (\langle \underline{\mathsf{x}}_1 : \underline{\mathsf{T}}_1^p \rangle, \{\underline{\mathsf{y}}_1\}, \mathsf{Pred}(\mathsf{u}_1, \mathsf{v}_1')), \\ \qquad \cdots, \mathsf{m}_k \mapsto (\langle \underline{\mathsf{x}}_k : \underline{\mathsf{T}}_k^p \rangle, \{\underline{\mathsf{y}}_k\}, \mathsf{Pred}(\mathsf{u}_k, \mathsf{v}_k'))\}\} \wedge \\ op_p' = \{\, \mathsf{C} \mapsto \{\mathsf{m}_{k+1} \mapsto (\langle \underline{\mathsf{x}}_{k+1} : \underline{\mathsf{T}}_{k+1}^p \rangle, \mathsf{P}_{k+1}), \\ \qquad \cdots, \mathsf{m}_n \mapsto (\langle \underline{\mathsf{x}}_n : \underline{\mathsf{T}}_n^p \rangle, \mathsf{P}_n)\}\} \end{array} \right\}$$

The design $P \vdash Q =_{df} ok \wedge P \Rightarrow ok' \wedge Q$ as in UTP [6].

The above environment generated by an individual class declaration *cdecl*, only records the names of those variables, attributes and operations. The complete information will be generated at the end of the class declaration section when class dependencies are also available.

The well-definedness of the operation bodies can not be determined by the *individual* class declaration itself, and it will be defined at the end of all class declarations. As a result, the logic variable $op(\mathsf{C})$ binds each operation m_i to its body rather than its meaning. The meaning of m_i will be calculated at the end of the declarations.

We now turn our attention to active classes. The MAIN operation is used to determine the behaviour of objects of an active class after initialisation. Objects of an active class have their own thread of control and their mutable state attributes and operation definitions are fully encapsulated. This condition should be reflected in the well-definedness of the definition of an active class.

Suppose the MAIN process is present in the above definition *cdecl* for class C. The well-definedness is specified by

$$\mathcal{WD}(cdecl) \;=_{df}\; \mathcal{WD} \wedge \mathsf{VisibAttr} = \varnothing \wedge \mathsf{VisibOp} = \{\textsc{Main}\}$$

where the predicate $\mathcal{WD}$ is defined as above.

The MAIN operation part: $\textsc{Main} \mapsto (b \bullet \mathsf{P})$ should be added into the value of the logic variable $op_p(\mathsf{C})$ in the above definition of the design *cdecl*, where b is the condition declared in INIT schema. However, when we calculate the set of process operations for a class later, MAIN is implicitly removed from the set of process operations of any of its active superclass, since TCOZ does not allow MAIN process to be inherited.

5.1 Composing Class Declarations

All class definitions *cdecls* for a specification is a composition of a number of class declarations

$$cdecls \;=_{df}\; cdecl_1;\; \cdots\; ;\; cdecl_k$$

Based on these complete definitions, we derive the whole context information for the specification by composing all the class declarations. This is done by simply adding up the contents of the current environment generated by the component class declarations provided that there is no redefinition of a class in its scope. It is also defined by the parallel merge operator:

$$cdecl_1;\; cdecl_2 \;=_{df}\; cdecl_1 \;\|_M\; cdecl_2$$

where the merge predicate M is defined as the following design

$$M \;=_{df}\; (CN_1 \cap CN_2 = \varnothing) \vdash \left\{ \begin{array}{l} CN' = CN_1 \cup CN_2 \wedge \\ super' = super_1 \cup super_2 \wedge \\ visibattr' = visibattr_1 \cup visibattr_2 \wedge \\ visibop' = visibop_1 \cup visibop_2 \wedge \\ locvar' = locvar_1 \cup locvar_2 \wedge \\ senvar' = senvar_1 \cup senvar_2 \wedge \\ actvar' = actvar_1 \cup actvar_2 \wedge \\ attr' = attr_1 \cup attr_2 \wedge op' = op_s' \cup op_p' \wedge \\ op_s' = op_{s1} \cup op_{s2} \wedge op_p' = op_{p1} \cup op_{p2} \end{array} \right\}$$

5.2 Well-Definedness of the Class Declarations

A sequence of class declarations for a specification is well-defined if the contents of the environment it has generated meet the following well-definedness conditions:

- The visible attributes (resp. operations) declared in a class should be members of the state attributes (resp. operations) in the current class or in any of its superclasses.

  $$\begin{aligned}\mathcal{WD}_1 =_{df}\ & \forall\, \mathsf{C} \in CN \bullet \mathsf{VisibAttr}(\mathsf{C}) \subseteq attr(super^*(\mathsf{C})) \\ & \wedge\ \mathsf{VisibOp}(\mathsf{C}) \subseteq op(super^*(\mathsf{C}))\end{aligned}$$

  where $super^*(\mathsf{C})$ is composed of all superclasses of C and C itself as before, and

  $$attr(\{\mathsf{C}_1, \cdots, \mathsf{C}_n\}) =_{df} \bigcup_{i:1..n} attr(\mathsf{C}_i),\ op(\{\mathsf{C}_1, \cdots, \mathsf{C}_n\}) =_{df} \bigcup_{i:1..n} op(\mathsf{C}_i)$$

- Multiple inheritances are allowed in TCOZ. However, distinct direct superclasses of any class are not permitted to have any common process operations (i.e. process operations with the same name and signature).

  $$\begin{aligned}&\mathcal{WD}_2 =_{df}\ \forall\, \mathsf{C} \in CN \bullet \#super(\mathsf{C}) > 1 \Rightarrow (\forall\, \mathsf{C}_1, \mathsf{C}_2 \in super(\mathsf{C}) \bullet \\ &(\mathsf{C}_1 \neq \mathsf{C}_2 \Rightarrow dom(op_p(super^*(\mathsf{C}_1))) \cap dom(op_p(super^*(\mathsf{C}_2))) = \varnothing \\ &\wedge\ \pi_1(ran(op_p(super^*(\mathsf{C}_1)))) \cap \pi_1(ran(op_p(super^*(\mathsf{C}_2)))) = \varnothing))\end{aligned}$$

- The Δ-list in each state operation can only comprise attributes or actuator variables declared in the current class or inherited from any superclass.

  $$\begin{aligned}\mathcal{WD}_3 =_{df}\ & \forall\, \mathsf{C} \in CN, \mathsf{m} \in op_s(C) \bullet \pi_2(ran(m)) \subseteq attr(super^*(\mathsf{C})) \vee \\ & \pi_2(ran(m)) \subseteq actvar(super^*(\mathsf{C}))\end{aligned}$$

- No parallel process operation is allowed to update any actuator variable in more than one component.

  $$\begin{aligned}\mathcal{WD}_4 =_{df}\ & \forall\, \mathsf{C} \in CN, (\mathsf{P}_1\ |[\mathsf{E}_1]|\ \cdots\ |[\mathsf{E}_{n\text{-}1}]|\ \mathsf{P}_n) \in op_p(C) \bullet \\ & \forall\, i,j : 1..n \bullet i \neq j \Rightarrow avar(\mathsf{P}_i) \cap avar(\mathsf{P}_j) = \varnothing\end{aligned}$$

  where $avar(\mathsf{P})$ is the set of actuators employed by P.
- In addition, other well-definedness conditions, such as the inheritance relation does not contain circularity, are omitted here, since similar conditions have been discussed in He, Liu and Li's work [5] for Java-like object-oriented languages.

5.3 Formalising the Behaviour of Class Operations

The dynamic behaviour of class operations is defined as the least fixed point of a set of recursive equations due to the inheritance (dependency) relation among the declared classes. We deal with the state operations and the process operations separately, since the former follow the inheritance rules of Object-Z, whereas the latter do not.

State Operations. For each class $\mathsf{C} \in CN$ and every state operation $m \in \{op_s(\mathsf{C}') \mid \mathsf{C}' \in super^*(\mathsf{C})\}$, it contains an equation $D(\mathsf{C.m}) = f(D)$, which is defined with respect to the following cases.

Case (1): m is newly introduced, i.e., it is declared in C, but not in any superclasses. Suppose the declaration of m is $\Delta(\underline{\mathsf{y}}),\ \underline{\mathsf{x}} : \underline{\mathsf{T}} \bullet \mathsf{Pred}(\underline{\mathsf{u}}, \underline{\mathsf{v}}')$.

$$D(\mathsf{C.m}) \ =_{df}\ \Delta(\underline{y}),\ \underline{x} : \underline{T} \bullet Pred(\underline{u}, \underline{v}')$$

The right-hand side is the semantic predicate defined in section 4.2.

Case (2): m is not declared in C but in its "nearest" superclasses, $\mathsf{C}_1, \cdots, \mathsf{C}_r$, i.e.,

$$\mathsf{m} \notin op_s(\mathsf{C}) \wedge \forall\, i : 1..r \bullet (\mathsf{m} \in op_s(\mathsf{C}_i) \wedge \mathsf{C}_i \in super^+(\mathsf{C}))$$

We can always assume none of these classes is a superclass of others, i.e., $\mathsf{C}_i \notin super^*(\mathsf{C}_j)$, for any $i, j : 1..r$. Otherwise, we remove C_i from the list if $\mathsf{C}_i \in super^*(\mathsf{C}_j)$. We also assume that each C_i is the nearest one to C that defines m in the corresponding dependence path, i.e.,

$$\forall\, i : 1..r, \neg \exists\, \mathsf{C}' \bullet \mathsf{C}' \in super^+(\mathsf{C}) \wedge \mathsf{C}_i \in super^+(\mathsf{C}') \wedge \mathsf{m} \in op_s(\mathsf{C}')$$

The equation for $D(\mathsf{C.m})$ is

$$D(\mathsf{C.m}) = \bigwedge\nolimits_{i:1..r} D(\mathsf{C}_i.\mathsf{m})$$

Case (3): m is defined in class C as $\Delta(\underline{\mathsf{y}}),\ \underline{\mathsf{x}} : \underline{\mathsf{T}} \bullet \mathsf{Pred}(\underline{\mathsf{u}}, \underline{\mathsf{v}}')$, but also defined in some "nearest" superclasses, $\mathsf{C}_1, \cdots, \mathsf{C}_r$, i.e.,

$$m \in op_s(\mathsf{C}) \wedge \forall\, i : 1..r \bullet (m \in op_s(\mathsf{C}_i) \wedge \mathsf{C}_i \in super^+(\mathsf{C}))$$

Using the same assumption as in case (2), the equation for $D(\mathsf{C.m})$ is

$$D(\mathsf{C.m}) = (\Delta(\underline{y}),\ \underline{x} : \underline{T} \bullet Pred(\underline{u}, \underline{v}')) \wedge \bigwedge\nolimits_{i:1..r} D(\mathsf{C}_i.\mathsf{m})$$

Process Operations. Given a class name C, and a process operation m, there are two cases to deal with.

Case (1). The process is not defined in C, but in a superclass C' of C. Then simply

$$D(\mathsf{C.m}) = D(\mathsf{C}'.\mathsf{m})$$

Case (2). The process operation is defined in C. Its dynamic behaviour is captured by its body and the environment in which it is executed. The design $D(\mathsf{C.m})$ is thus subject to the equation $D(\mathsf{C.m}) = \varphi(body(\mathsf{C.m}))$. φ is used to pass the actual parameters to their corresponding formal parameters, and generate the semantics predicate afterwards, as discussed in section 4.

The function φ distributes over operators and is inductively defined as:

$$\varphi(\mathsf{P}_1\, \mathbf{op}\, \mathsf{P}_2) \ =_{df}\ \varphi(\mathsf{P}_1)\, op\, \varphi(\mathsf{P}_2),\ \text{where } \mathbf{op} \in \{;\ , \Box, \sqcap, \ |[\mathsf{E}]|\ , \rhd\{\mathsf{t}\}, \rightarrow, \bullet\}$$
$$\varphi(\mathsf{P} \bullet \textsc{Deadline}\ \mathsf{t}) \ =_{df}\ \varphi(\mathsf{P}) \bullet Deadline\ t$$
$$\varphi(\mathsf{P} \bullet \textsc{WaitUntil}\ \mathsf{t}) \ =_{df}\ \varphi(\mathsf{P}) \bullet WaitUntil\ t$$
$$\varphi(\mu \mathsf{X} \bullet \mathsf{P}) \ =_{df}\ \mu X \bullet \varphi(\mathsf{P}), \quad \varphi(\mathsf{P} \backslash \mathsf{E}) \ =_{df}\ \varphi(\mathsf{P}) \backslash E$$
$$\varphi(\mathsf{x}) = x,\ \varphi(\mathsf{f}(\underline{\mathsf{e}})) = f(\underline{\varphi(\mathsf{e})}),$$

where f can be any legal arithmetic operator ($+, -, *, /, \leqslant, \neq, \cdots$), logical connector ($\wedge, \vee, \neg, \Rightarrow, \cdots$), or set operator ($\in, \notin, \subseteq, \cdots$).

An operation invocation $\mathsf{o.m}$ is mapped by φ to

$$\varphi(\mathsf{o.m}(\underline{\mathsf{val}})) \;=_{df}\; \Box\{\mathsf{o}(myclass) = \mathsf{C'} \wedge \mathsf{m} \in visibop(\mathsf{C'}) \bullet D(\mathsf{C'.m})[\underline{val}/\underline{x}]\}$$

where $\underline{\mathsf{x}}$ is the parameters of the operation $\mathsf{C'.m}$.

5.4 The Behaviour of Active Classes

This subsection is devoted to formalising the behaviour of active classes. The behaviour of a system specified in TCOZ is determined by the MAIN processes of active classes.

Given a sequence of class declarations $\mathsf{cdecls} =_{df} \mathsf{cdecl}_1, \cdots, \mathsf{cdecl}_n$, where cdecl_n is an active class of interest which may depend on (inherit from) the other classes. The behaviour of (any objects of) this active class is defined as the following predicate:

$$cdecls;\; initial;\; D(cdecl_n.\textsc{Main})$$

The design *initial* performs the following tasks: (1) to check the well-definedness of the complete declaration section; (2) to derive the final values of the logical variables; (3) to define the dynamic behaviour of every operation.

$$initial \;=_{df}\; \bigwedge\nolimits_i \mathcal{WD}_i \vdash \left\{\begin{array}{l} super' = super \wedge CN' = CN \wedge \forall\,\mathsf{C} \in CN \bullet \\ locvar'(\mathsf{C}) = locvar(super^*(\mathsf{C})) \wedge attr'(\mathsf{C}) = attr(super^*(\mathsf{C})) \wedge \\ senvar'(\mathsf{C}) = senvar(super^*(\mathsf{C})) \wedge actvar'(\mathsf{C}) = actvar(super^*(\mathsf{C})) \wedge \\ op_s'(\mathsf{C}) = \{(\mathsf{m} \mapsto (\langle \underline{\mathsf{x}} : \underline{\mathsf{T}} \rangle, \Delta(\underline{\mathsf{y}}), D(\mathsf{C.m}))) \mid \exists\,\mathsf{Pred} \bullet \\ \quad (\mathsf{m} \mapsto (\langle \underline{\mathsf{x}} : \underline{\mathsf{T}} \rangle, \Delta(\underline{\mathsf{y}}), \mathsf{Pred})) \in op_s(\mathsf{C'}) \wedge \mathsf{C'} \in super^*(\mathsf{C})\} \wedge \\ op_p'(\mathsf{C}) = \{(\mathsf{m} \mapsto (\langle \underline{\mathsf{x}} : \underline{\mathsf{T}} \rangle, D(\mathsf{C.m}))) \mid \exists\,\mathsf{P} \bullet \\ \quad (\mathsf{m} \mapsto (\langle \underline{\mathsf{x}} : \underline{\mathsf{T}} \rangle, \mathsf{P})) \in op_p(\mathsf{C'}) \wedge \mathsf{C'} \in super^*(\mathsf{C})\} \wedge \\ visibattr' = \{\mathsf{C} \mapsto (attr(super^*(\mathsf{C})) \upharpoonright visibattr(\mathsf{C})) \mid \mathsf{C} \in CN\} \wedge \\ visibop' = \{\mathsf{C} \mapsto (op(super^*(\mathsf{C})) \upharpoonright visibop(\mathsf{C})) \mid \mathsf{C} \in CN\} \end{array}\right\}$$

where $\mathcal{WD}_i$ is the well-definedness condition discussed in section 5.2. $D(\mathsf{C.m})$ discussed in last section defines the dynamic behaviour of the operation m of class C.

6 Related Work, Conclusion and Future Work

The semantics of Object-Z has been investigated earlier. For example, Object-Z has a fully abstract semantics [3, 15]. Timed CSP's semantics has also been well studied [2, 10, 11]. The process model used by TCOZ [9] presented a conservative extension to the basic timed failures model [10]. The semantic model of TCOZ in this paper is based on the UTP framework. The most closely related works are the UTP timed [14] and untimed [18] semantic models of Circus and the UTP semantic model [5] of object-oriented programming languages. A significant contribution of this paper is the unified semantic model for both channel and sensor/actuators based communications in TCOZ. This new model is far more complete. It not only covers the communication and process aspects of TCOZ, but also other features, such as class encapsulation, inheritance, dynamic binding and extended TCOZ timing constructs (deadline and waituntil commands), which have not been covered by the previous result [9].

This paper also demonstrates that UTP can provide a formal semantic foundation not only for programming languages but also for much more expressive specification languages. In particular, UTP is well suited for capturing formal semantics for integrated specification languages (i.e., TCOZ) which often have rich language constructs for state encapsulation, event communication and real-time modeling. Our semantic model will be used as a reference document for developing tools support for TCOZ. For example, in the semantic model, the well formed rules can be used as precise requirements for developing a type checking system. Various laws for the language constructs can be encoded as theorems to support a reasoning system.

The semantic model presented in this paper is a discrete time model which can readily be connected to an untimed model, so that model checker like FDR [12] can also be used to check untimed properties of TCOZ. For checking timing properties, we have recently developed transformation rules from TCOZ to Timed Automata (TA) so that various TA tools, i.e. UPPAAL [1], can be applied to check timing properties. We plan to give a UTP semantic model for TA, and to prove the soundness of our transformation rules based on UTP semantics for both TCOZ and TA.

Another further research work would be to develop operational and data refinement techniques for TCOZ and to look into transforming TCOZ to object-oriented programming languages, e.g., Java. This work should be achievable given that UTP semantics for Java-like language has already been formulated in [5].

Acknowledgement

We would like to thank Jifeng He for helpful comments and inspiring related work. We are also grateful to anonymous referees for many helpful comments.

References

1. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and Y. Wang. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. *Hybrid Systems*, LNCS 1066, pages 232–243. Springer-Verlag, 1996.
2. J. Davies and S. Schneider. A brief history of Timed CSP. *Theoret. Comput. Sci.*, 138:243–271, 1995.
3. D. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, eds., *VDM'90: VDM and Z!*, LNCS 428, pages 242–262. Springer-Verlag, 1990.
4. R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z.* Cornerstones of Computing Series. Macmillan, March 2000.
5. J. He, Z. Liu, and X. Li. A relational model for specification of object-oriented systems. Technical Report 262, UNU/IIST, October 2002.
6. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

7. B. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. In J. Wing, J. Woodcock, and J. Davies, eds., *FM99: World Congress on Formal Methods*, LNCS 1709, pages 1166–1185, 1999.
8. B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.
9. B. Mahony and J. S. Dong. Overview of the semantics of TCOZ. In K. Araki, A. Galloway, and K. Taguchi, eds, *IFM'99: Integrated Formal Methods*, pages 66–85. Springer-Verlag, 1999.
10. M. Mislove, A. Roscoe, and S. Schneider. Fixed Points Without Completeness. *Theoret. Comput. Sci.*, 138:273–314, 1995.
11. G. Reed and A. Roscoe. A timed model for communicating sequential processes. *Theoret. Comput. Sci.*, 58:249–261, 1988.
12. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
13. S. Schneider, J. Davies, D. Jackson, G. Reed, J. Reed, and A. Roscoe. Timed CSP: Theory and practice. *Real-Time: Theory in Practice*, LNCS 600, pages 640–675. Springer-Verlag, 1992.
14. A. Sherif and J. He. Towards a timed model for circus. In C. George and H. Miao, eds., *ICFEM'02 Formal Methods and Software Engineering*, LNCS 2495, pages 613–624. Springer-Verlag, 2002.
15. G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
16. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
17. J. Woodcock and A. Cavalcanti. Circus: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Oxford OX1 3QD, UK, July 2001.
18. J. Woodcock and A. Cavalcanti. The Semantics of Circus. In D. Bert, J. Bowen, M. Henson and K. Robinson, eds., *2nd International Conference on Z and B*, LNCS 2272, pages 184–203. Springer-Verlag, 2002.

Appendix

The semantics for the process constructs (e.g., primitives, internal/external choices, etc.) that are similar to Sherif and He's work[14] are listed here.

$$\begin{array}{l}
Skip \;=_{df}\; ok' \wedge \neg wait' \wedge tr' = tr \wedge state' = state \\
Stop \;=_{df}\; ok' \wedge wait' \wedge state' = state \wedge no\_interact(trace) \\
Chaos \;=_{df}\; \mathbf{R}(true) \\
P;\ Q \;=_{df}\; P[false/ok'] \;\vee\; P \wedge wait' \;\vee\; P[true, false/ok', wait'] \circ Q \\
P \sqcap Q \;=_{df}\; P \vee Q \\
P \Box Q \;=_{df}\; (P \wedge Q \wedge wait' \wedge trace = \langle\rangle) \vee \\
\qquad\qquad (((P \wedge Q \wedge ok' \wedge wait' \wedge trace = \langle\rangle \wedge state' = state) \vee Skip) \\
\qquad\qquad\; \circ (\neg wait' \vee (\neg(tr \preceq tr') \wedge trace \neq \langle\rangle))) \wedge (P \vee Q);\ Skip \\
\mu \mathsf{X} \bullet \mathsf{F(X)} \;=_{df}\; \sqcap \{\mathsf{X} \mid \mathsf{X} \sqsupseteq \mathsf{F(X)}\} \\
P \backslash \mathsf{E} \;=_{df}\; (\exists \tilde{tr} \bullet P[\tilde{tr}/tr'] \;\wedge\; \forall k : \#tr \leqslant k \leqslant \#tr' \bullet \\
\qquad\qquad\quad \pi_1(tr'(k)) = \pi_1(\tilde{tr}) \upharpoonright (Event - \mathsf{E}) \;\wedge \\
\qquad\qquad\quad \pi_2(\tilde{tr}(k)) = \pi_2(tr'(k)) \cup \mathsf{E});\ Skip
\end{array}$$