Advanced Features of Duration Calculus and Their Applications in Sequential Hybrid Programs

He Jifeng and Xu Qiwen

International Institute for Software Technology, United Nations University, Macao SAR, P.R. China

Faculty of Science and Technology, University of Macau, Macao SAR, P.R. China

Keywords: Duration Calculus; Semantics; Verification; Hybrid systems

Abstract. Duration Calculus was introduced as a logic to specify real-time requirements of computing systems. It has been used successfully in a number of case studies. Moreover, many variants were proposed to deal with various features of real-time systems, including sequential communicating processes, sequential hybrid systems and imperative programming languages. This paper aims to integrate several variants of Duration Calculus, and to provide a semantic framework for real-time programming languages and sequential hybrid programs.

1. Introduction

Hybrid systems are interactive systems of continuous devices and real-time control programs. The design of a real-time control system is ideally decomposed into a progression of related phases. It starts with an analysis of requirements and properties of the process evolving within its environment. From these are derived formal specifications of the components of the system. A high level program generated in the later phase of the project is usually translated into machine code of the chosen computer. Additional application-specific hardware components may be needed to embed the computer into the system which it controls. Reliability of

---


2 On leave from East China Normal University, Shanghai, P.R. China.
the delivered system requires that all the conceptual gaps between specification and implementation be closed.

For hybrid systems, a variety of formal methods have been developed, among them Phase Transition System [HCH93, MMP92], Declarative Control [Koh88], the Extended State-Transition Graph [Nic92] and the Hybrid CSP [He93]. However, it remains a difficult task to mix the description of quantitative timing properties with that of discrete changes of sequential systems. An interval temporal logic with discrete time was investigated for presenting the kinds of temporal properties and signal transitions that occur in real-time control programs. The behaviour of hardware devices can often be decomposed into successively smaller intervals of activity [HMM83, Mos85]. Moreover, state transitions of programs can also be characterised by properties relating the initial and final values of variables over interval of times [Hal83, Mos86]. But in the treatment of hybrid systems where the physical world evolve continuously, this approach seems inappropriate. Furthermore, we lose some important algebraic laws of the Guarded Command Language [Dij76] in that framework, such as induction rules for recursions and the combination of assignments

\[(x := e; x := f) = (x := f[e/x])\]

This paper proposes a conservative extension of the untimed refinement calculus [HoH98, Jon90, Mor94] for hybrid systems, based on the model of the Duration Calculus (DC) [ZHR91, ZRH93]. Our research is inspired by the pioneering work presented in [PaR95, PaD88, PWX98, ZH96]. The main contribution of this paper includes

1. some novel features which play the vital role in formalising stable state of program variables and sequential composition operator,
2. a link between the untimed refinement calculus and our timed one, which preserves the laws of untimed programming,
3. a theory of Sequential Hybrid Programs (SHP).

The advantage of a model-oriented calculus is that it describes as directly as possible, using the full expressive power of mathematics, the observable and testable properties of the desired systems. These properties can be specified in independent modules, and can be assembled by simple combinators. Because both design notation and implementation languages are also given model-oriented semantics, correctness can be proved by deduction.

The remainder of the paper is organised as follows. Section 2 gives a brief account of DC. We introduce some advanced features and explore their algebraic properties in Section 3. Section 4 presents a link between the untimed refinement calculus [Jon90, HoH98, Mor94] and the timed one, and shows that the embedding from the former to the latter is a homomorphism. Section 5 presents a theory of Sequential Hybrid Programs. Those programs consist of phase statements and the conventional sequential programming constructs, where the former is used to specify the dynamic behaviours of controlled devices, and the latter describes the activity of control program. We provide a compositional semantics for SHP and study verification methods based on the advanced features of DC introduced in the earlier sections.
2. Preliminaries

In DC, time is continuous and represented by reals

\[ Time =_{df} \text{Real} \]

An interval is represented by \([t_1, t_2]\) where \(t_1, t_2 \in Time\) and \(t_1 \leq t_2\). We use \(I\) to stand for the set of intervals, and \(\sigma\) to range over intervals, and \(\sigma.b\) and \(\sigma.e\) to represent its left and right end points. Adjacent intervals can be merged using the concatenation operator \(\sim\).

\[ \sigma_1 \sim \sigma_2 =_{df} \sigma_1 \cup \sigma_2 \quad \text{if} \quad \sigma_1.e = \sigma_2.b \]

A model \(M\) assigns every symbol a constant of its associated type:

- Global variable \(x\) is assigned a real \(M(x)\).
- State variable \(V\) is interpreted as a time function \(M(V) : Time \to \text{Real}\).
- Temporal variable \(v\) is associated with an interval function \(M(v) : I \to \text{Real}\).

A specific temporal variable \(\ell\) denotes the length of interval

\[ M(\ell)(\sigma) =_{df} \sigma.e - \sigma.b \]

- \(M\) assigns an \(n\)-ary function name \(f\) a function \(M(f) : \text{Real}^n \to \text{Real}\).
- An \(n\)-ary predicate name \(p\) is associated with a predicate

\[ M(p) : \text{Real}^n \to \text{Bool} \]

Let \(h\) be a variable. Two models \(M\) and \(M0\) are said to be \(h\)-equivalent, denoted by \(M \equiv_h M0\), if for all variables \(v\) different from \(h\)

\[ M(v) = M0(v) \]

The terms of the language are defined by induction, and interpreted as real-valued functions over intervals.

- Global and temporal variables are terms.
- If \(r_1, \ldots, r_n\) are terms and \(f\) is an \(n\)-ary function name, then \(f(r_1, \ldots, r_n)\) is also a term

\[ M(f(r_1, \ldots, r_n))(\sigma) =_{df} M(f)(M(r_1)(\sigma), \ldots, M(r_n)(\sigma)) \]

Formulae are interpreted as functions from intervals to the Boolean values \(true\) and \(false\). The set of well-formed formulae is generated by the following rules:

- \(true\) and \(false\) are well-formed formulae.

\[ M(\text{true})(\sigma) =_{df} true \]

\[ M(\text{false})(\sigma) =_{df} false \]

- If \(r_1, \ldots, r_n\) are terms, and \(p\) is an \(n\)-ary predicate name, then \(p(r_1, \ldots, r_n)\) is a well-formed formula

\[ M(p(r_1, \ldots, r_n))(\sigma) =_{df} M(p)(M(r_1)(\sigma), \ldots, M(r_n)(\sigma)) \]

- If \(F\) and \(G\) are well-formed formulae, so are \(\neg F\) and \(F \wedge G\), \(F \sim G\) and \(\exists h \cdot F\), where \(h\) is a global variable.

\[ M(F \sim G)(\sigma) =_{df} \exists \sigma_1, \sigma_2 \cdot \sigma = (\sigma_1 \sim \sigma_2) \wedge M(F)(\sigma_1) \wedge M(G)(\sigma_2) \]
\[ \mathcal{M}(\exists h \cdot F)(\sigma) =_{df} \mathcal{M}(\exists h_0 \cdot (F \land (M \equiv_h M))) \]

All the usual logical connectives (disjunction, implication, etc.) and quantifiers can be defined

\[ F \lor G =_{df} \neg (\neg F \land \neg G) \]
\[ F \Rightarrow G =_{df} \neg F \lor G \]
\[ \forall h \cdot F =_{df} \neg (\exists h \cdot \neg F) \]

The formula \( F \triangleleft H \triangleright G \) presents a choice between \( F \) and \( G \) depending on \( H \).

\[ F \triangleleft H \triangleright G =_{df} (F \land H) \lor (G \land \neg H) \]

The modal operators \( \Diamond \) and \( \Box \) can be defined in terms of the chop operator. The formula \( \Diamond F \) holds on an interval if \( F \) does so on one of its subintervals.

\[ \Diamond F =_{df} \text{true} \triangleright (F \triangleright \text{true}) \]

The formula \( \Box F \) holds if \( F \) holds on all its subintervals.

\[ \Box F =_{df} \neg \Diamond (\neg F) \]

3. Advanced Features

This section introduces a set of advanced features, which play the vital role later in modelling sequential hybrid programs.

3.1. Initial and Final Values

**Definition 3.1** (Initial and final values)

Let \( S \) be a state expression. We introduce two temporal terms \( b.S \) and \( e.S \) defined by

\[ \mathcal{M}(b.S)(\sigma) =_{df} \mathcal{M}(S)(\sigma.b) \]
\[ \mathcal{M}(e.S)(\sigma) =_{df} \mathcal{M}(S)(\sigma.e) \]

**Example 3.2**

Let \( V \) be a state variable. Formula \( \text{stable}(V) \) holds on an interval if the value of \( V \) is constant throughout that interval

\[ \text{stable}(V) =_{df} \Box (b.V = e.V) \]

\( \text{stable}^{-}(V) \) holds on an interval if \( V \) remains unchanged except at the end of the interval

\[ \text{stable}^{-}(V) =_{df} \neg ((b.V \neq e.V) \triangleright (\ell > 0)) \]

**Theorem 3.3** \( b.V \) and \( e.V \) are subject to the following law:

\( (V-1) \)

\[ (F \land p(e.V)) \triangleright G = F \triangleright (p(b.V) \land G) \]

\( (V-2) \)

\[ (p(b.V) \land F) \triangleright G = p(b.V) \land (F \triangleright G) \]
(V-3) \( F \sim (G \land p(e,V)) = (F \sim G) \land p(e,V) \)
(V-4) \((\ell = 0) \land p(e,V) = p(b.V) \land (\ell = 0) \)

3.2. Stability

**Definition 3.4 (Stability)**
Let \( S \) be a Boolean state expression. Define
\[
[S] =_{df} \neg((\ell > 0) \sim \neg b.S \sim (\ell > 0))
\]
It expresses that \( S \) holds everywhere inside the interval. \( [\cdot] \) enjoys the following algebraic properties.

**Theorem 3.5**
1. \([true] = true\)
2. \([false] = (\ell = 0)\)
3. \([S \land T] = [S] \land [T]\)
4. \([S] = \Box([S])\)

**Proof.** we only show (4).
\[
\begin{align*}
\Box([S]) & \quad \{\text{Def of } \Box\}\nonumber \\
= \neg((true \sim \neg([S]) \sim true) \quad \{\text{Def of } [\cdot]\}\nonumber \\
= \neg((true \sim ((\ell > 0) \sim \neg b.S \sim (\ell > 0)) \sim true) \quad \{\text{associativity of } \sim\}\nonumber \\
= \neg(((\ell > 0) \sim \neg b.S \sim (\ell > 0)) \quad \{\text{Def of } [\cdot]\}\nonumber \\
= [S] 
\end{align*}
\]

**Notation**
We use the notation \([|S|]\) to abbreviate
\[
b.S \land [S] \land e.S
\]

3.3. Left and Right Neighbourhood Values

A state variable \( V \) is *finitary* if it can only change a finite number of times in any finite interval. We define its left neighbourhood value \( \overrightarrow{V} \) and right neighbourhood value \( \overleftarrow{V} \) by
\[
\begin{align*}
\overrightarrow{M}(V)(\sigma) = c & \quad \text{if } \exists \delta > 0 \bullet \overrightarrow{M}([V = c]([\sigma.b - \delta, \sigma,b])) = \text{true} \\
\overleftarrow{M}(V)(\sigma) = c & \quad \text{if } \exists \delta > 0 \bullet \overleftarrow{M}([V = c]([\sigma.e, \sigma.e + \delta])) = \text{true}
\end{align*}
\]
For a state expression \( E \), \( \overrightarrow{E} \) and \( \overleftarrow{E} \) can be defined in the same way.

The concepts of neighbourhood value are captured by the following laws:
(nv-1) \[ (F \land p(V)) \sim (G \land (\ell > 0)) \]
\[ = F \sim (G \land \exists x \cdot (p(x) \land ((V = x) \land \ell > 0) \sim \text{true})) \]
\[ \sim (F \land (\ell > 0)) \sim (p(V) \land G) \]
\[ = F \land \exists x \cdot (p(x) \land \text{true} \sim (V = x) \land \ell > 0) \sim G \]

(nv-2) \[ F \sim (G \land p(V)) = (F \sim G) \land p(V) \]
\[ \sim (F \land p(V)) \sim (G \land (\ell = 0)) = (F \sim (G \land (\ell = 0))) \land p(V) \]

(nv-3) \[ (F \land (\ell = 0)) \sim (p(V) \land G) = p(V) \sim ((F \land (\ell = 0)) \sim G) \]
\[ \sim (p(V) \land F) \sim G = p(V) \land (F \sim G) \]

3.4. Hiding State Variable

**Definition 3.6** (Hiding a state variable)

Let $V$ be a state variable. Define

\[ \mathcal{M}(\exists V \cdot F)(\sigma) =_{df} \exists M0 \cdot M0(F)(\sigma) \land (M \equiv_{\mathcal{M}} M0) \]

The following laws apply:

(\exists-1) If $(\sim V \notin F)$ and $(\sim V \notin G)$, and $(e.V \notin F)$ or $(b.V \notin G)$, then

\[ (\exists V \cdot F) \sim (\exists V \cdot G) = \exists V \cdot (F \sim G) \]

(\exists-2) If none of \{V, \sim V, \sim V\} occurs in both $F$ and $G$, then

\[ \exists V \cdot (F \land G) = (\exists V \cdot F) \land (\exists V \cdot G) \]

(\exists-3) If the state variable $V$ is not mentioned in $G$, then

\[ (\exists V \cdot F) \sim G = \exists V \cdot (F \sim G) \]
\[ G \sim (\exists V \cdot F) = \exists V \cdot (G \sim F) \]

(\exists-4) \[ F = \exists W \cdot (F[W/V] \land \|W = V\| \land (\sim W = \sim V) \land (\sim W = \sim \sim V)) \]

**Corollary**

If $\sim V \notin F$ and $\sim V \notin G$, then

\[ F = \exists W \cdot (F[W/V] \land \|W = V\| \land (\sim W = \sim V)) \]
\[ G = \exists W \cdot (G[W/V] \land \|W = V\| \land (\sim W = \sim V)) \]

**Proof.** From (\exists-1) and (\exists-2). \qed
3.5. Chopping points

Definition 3.7 (Super-dense chop)
Let $V$ be the state variables used in $F$ and $G$. Define

$$F \circ G = \exists x, V_F, V_G \cdot (F[V_F/V] \land \|V = V_F\| \land (\vec{V} = V_F) \land (V_F = x)) \sim$$

$$G[V_G/V] \land \|V = V_G\| \land (\vec{V} = V_G) \land (V_G = x)$$

The chop operator $\sim$ is used to compose the continuously evolving hybrid systems [ZHR91], whereas the relational composition operator is used to model the sequential composition of imperative programming languages [HoH98, Mor94]. The following theorem states that $\circ$ can be seen as the product of the chop operator and the relational composition operator.

Theorem 3.8

If $V$ and $V$ do not occur in $F$ nor $G$, then

$$(F \land p(V, V)) \circ (q(V, V) \land G) = (F \sim G) \land \exists x \cdot (p(V, x) \land q(x, V))$$

Theorem 3.9 ($\circ$ and $\sim$)

If $V \notin F$ and $V \notin G$, then $F \circ G = F \sim G$.

Proof.

$$F \circ G = \exists x, V_F, V_G \cdot \{\text{definition}\}$$

$$= (F[V_F/V] \land \|V = V_F\| \land (\vec{V} = V_F) \land (V_F = x)) \sim$$

$$G[V_G/V] \land \|V = V_G\| \land (\vec{V} = V_G) \land (V_G = x) \{\text{3-2}\}$$

$$= \exists x \cdot (\exists V_1 \cdot (F[V_1/V] \land \|V = V_1\| \land (\vec{V} = V_1)) \land (\exists V_1 \cdot (\vec{V}_1 = x))) \sim$$

$$\exists V_2 \cdot (G[V_2/V] \land \|V = V_2\| \land (\vec{V} = V_2)) \land (\exists V_2 \cdot (\vec{V}_2 = x)) \{\text{Coro of 3-4}\}$$

$$= \exists x \cdot (F \sim G) \land \{x \text{ not used by } F \text{ and } G\}$$

$$= F \sim G \quad \square$$

$\circ$ also enjoys the following familiar algebraic laws:

Theorem 3.10

1. $(F \circ G) \circ H = F \circ (G \circ H)$
2. $F \circ I = F = I \circ F$, where $I = df (\ell(0) \land (\vec{V} = \vec{V}))$
3. $F \circ (G \lor H) = (F \circ G) \lor (F \circ H)$
\[(G \lor H) \circ F = (G \circ F) \lor (H \circ F)\]

(4) \[F \circ \text{false} = \text{false} = \text{false} \circ G\]

(5) \[(p(\vec{V} \land F) \circ G = p(\vec{V}) \land (F \circ G)\]

(6) \[F \circ (G \land q(\vec{V})) = (F \circ G) \land q(\vec{V})\]

(7) \[(F \land r(\vec{V})) \circ G = F \circ (r(\vec{V} \land G)\]

(8) If \(\vec{V} \notin F\) and \(\vec{V} \notin G\), then

\[(F \land (\vec{V} = y)) \circ G = F \sim G\]

\[F \circ ((\vec{V} = y) \land G) = F \sim G\]

\[(F \land (\vec{V} = y)) \circ ((\vec{V} = z) \land G) = (F \sim G) \land (y = z)\]

3.6. Fixed Points

**Definition 3.11** (Greatest lower bound)

Let \(\mathcal{F}\) be a set of formulae. We define its greatest lower bound \(\sqcap \mathcal{F}\) by

\[
\mathcal{M}(\sqcap \mathcal{F})(\sigma) = \text{df} \quad \exists F \in \mathcal{F} \cdot \mathcal{M}(F)(\sigma)
\]

\(\sqcap\) can be defined algebraically by the following law:

\((\text{glb-1})\) \(G \sqsubseteq \sqcap \mathcal{F}\) if \(G \sqsubseteq F\) for all \(F \in \mathcal{F}\).

**Corollary**

(1) \(\sqcap \emptyset = \text{false}\)

(2) \(\sqcap (\mathcal{F}_1 \cup \mathcal{F}_2) = (\sqcap \mathcal{F}_1) \lor (\sqcap \mathcal{F}_2)\)

(3) \(\sqcap \{F\} = F\)

The greatest lower bound operator distributes over both \(\sim\) and \(\circ\).

\((\text{glb-2})\) Let \(\ast \in \{\sim, \circ\}\)

\[
(\sqcap \mathcal{F}) \ast G = \sqcap \{F \ast G \mid F \in \mathcal{F}\}
\]

\[
G \ast (\sqcap \mathcal{F}) = \sqcap \{G \ast F \mid F \in \mathcal{F}\}
\]

**Definition 3.12** (Weakest fixed point)

Let \(\Phi\) be a monotonic mapping of formulae. We define its weakest fixed point by

\[
\mu X \bullet \Phi(X) = \text{df} \quad \sqcap \{F \mid F \Rightarrow \Phi(F)\}
\]

From Tarski’s Fixed Point Theorem \([\text{Tarski55}]\) it follows that \(\mu X \bullet \Phi(X)\) is subject to the following laws.

\((\mu-1)\) \(\Phi(\mu X \bullet \Phi(X)) = \mu X \bullet \Phi(X)\)
\((\mu X)\) If \(F \Rightarrow \Phi(F)\), then \(F \Rightarrow \mu X \cdot \Phi(X)\).

**Definition 3.13** (Least upper bound)

Let \(\mathcal{F}\) be a set of formulae. We define its least upper bound \(\sqcup \mathcal{F}\) by

\[M(\sqcup \mathcal{F})(\sigma) = \forall F \in \mathcal{F} \cdot M(F)(\sigma)\]

\(\sqcup\) can be defined algebraically by the following law:

\((\text{lub-1}) G \Rightarrow \sqcup \mathcal{F} \text{ iff } G \Rightarrow F \text{ for all } F \in \mathcal{F}\).

**Corollary**

(1) \(\sqcup \emptyset = \text{true}\)

(2) \(\sqcup (\mathcal{F} \sqcup \mathcal{F}^2) = (\sqcup \mathcal{F}) \land (\sqcup \mathcal{F}^2)\)

(3) \(\sqcup \{F\} = F\)

4. **Link the Untimed Refinement Calculus with the Timed Calculus**

The meaning of a program of the Guarded Command Language [Dij76] is defined in [HoH98] by a predicate

\[\text{pre}(v) \vdash \text{post}(v, v') = \text{af} \ (ok \land \text{pre}(v)) \Rightarrow (ok' \land \text{post}(v, v'))\]

where

- Both \(ok\) and \(ok'\) are Boolean variables, where \(ok\) records the observation that the program has been started, and \(ok'\) records the observation that the program has terminated.
- \(v\) and \(v'\) respectively denote the initial and final values of the program variables \(v\).
- The predicate \(\text{pre}(v)\) is an assumption which the program can rely on when the program is initiated.
- The predicate \(\text{post}(v, v')\) is a commitment which must be true when the program terminates.

A design is a relation whose predicate is expressed in this form. We will use \(D\) to range over the set of designs in the remainder of this section. This is the approach used in the refinement calculus [HoH98, Mor94]. Designs can be combined by disjunction \(\lor\), conditional \(\triangleright\) and relational composition.

**Definition 4.1** (Relational composition)

\[Q(s') \cdot R(s) = \text{df} \ \exists m \cdot (Q(m) \land R(m))\]

**Theorem 4.2** (Laws of designs) [HoH98]

(1) \((b_1 \triangleright R_1) \lor (b_2 \triangleright R_2) = (b_1 \land b_2) \triangleright (R_1 \lor R_2)\)

(2) \((b_1 \triangleright R_1) \triangleright c \triangleright (b_2 \triangleright R_2) = (b_1 \land c) \triangleright (R_1 \land c \triangleright R_2)\)

(3) \((b_1 \triangleright R_1); (b_2 \triangleright R_2) = (b_1 \land \neg(R_1; \neg b_2)) \triangleright (R_1; R_2)\)
We next establish links between the untimed refinement calculus [Jon90, Mor94] and the timed one.

**Definition 4.3** (Mapping designs to DC formulae)
Let $D$ be a design. We define its timed interpretation by

$$
\mathcal{T}(D) = \exists v, v' \cdot ((\vec{V} = v) \land D \land (v' = \vec{v}))
$$

**Example 4.4**

$$
\mathcal{T}(\text{true} \vdash (v' = v + 1)) = ok \Rightarrow (ok' \land (\vec{V} = \vec{V} + 1))
$$

**Theorem 4.5**

$\mathcal{T}$ is injective, and

$$
\mathcal{T}(\text{pre}(v) \vdash \text{post}(v, v')) = (ok \land \text{pre}(\vec{V})) \Rightarrow (ok' \land \text{post}(\vec{V}, \vec{V}))
$$

**Definition 4.6** (Sequential composition of DC formulae)
Let $R1$ and $R2$ be formulae with global variables $ok$ and $ok'$. We define their sequential composition $R1; R2$ by

$$
R1; R2 = \exists o \cdot (R1[o/ok'] \circ R2[o/ok])
$$

**Definition 4.7** (Relational formulae)
A DC formula is relational if it can be expressed in the form

$$
P(\vec{V}) \vdash Q(\vec{V}, \vec{V}) = df (ok \land P(\vec{V})) \Rightarrow (ok' \land Q(\vec{V}, \vec{V}))
$$

**Theorem 4.8** (Closure of relational formulae)
If $R1$ and $R2$ are relational formulae, so are $R1 \lor R2$, $R1 \land b(\vec{V}) \land R2$ and $R1; R2$. □

The conclusion for disjunction generalises to an arbitrary set of relational formulae, and a similar law holds for arbitrary intersection.

**Theorem 4.9** (Closure of glb and lub)
(1) $\sqcap (P_i \vdash Q_i) = (\sqcup P_i) \vdash (\sqcap Q_i)$
(2) $\sqcup (P_i \vdash Q_i) = (\sqcap P_i) \vdash (\sqcup (P_i \Rightarrow Q_i))$

This means that relational formulae form a complete lattice under implication ordering. It contains a bottom element true and a top element $\neg ok$.

**Lemma 4.10** (Link between weakest fixed points) [Mat95]
If $f$ is universally conjunctive, i.e., for any set $S$,

$$
f(\sqcup S) = \sqcup \{f(X) \mid X \in S\}
$$

and $g$ and $h$ are monotonic functions on complete lattices satisfying $f \circ g = h \circ f$, then $f(\mu g) = \mu h$.

Finally we can prove $\mathcal{T}$ is a homomorphism.
**Theorem 4.11**

(1) \( \mathcal{T}(D_1 \lor D_2) = \mathcal{T}(D_1) \lor \mathcal{T}(D_2) \)

(2) \( \mathcal{T}(D_1 \land D_2) = \mathcal{T}(D_1) \land \mathcal{T}(D_2) \)

(3) \( \mathcal{T}(D_1 < b(v) \triangleright D_2) = \mathcal{T}(D_1) < \hat{b}(V) \triangleright \mathcal{T}(D_2) \)

(4) \( \mathcal{T}(D_1; D_2) = \mathcal{T}(D_1); \mathcal{T}(D_2) \)

(5) \( \mathcal{T}(\mu X \cdot ((D; X) < b(v) \triangleright II)) = \mu X \cdot (\mathcal{T}(D); X) < \hat{b}(V) \triangleright \mathcal{T}(II) \)

**Proof.** (4) Let \( D_i =_{df} \text{pre}_i(v) \vdash \text{post}_i(v, v') \) for \( i = 1, 2 \).

\[
\mathcal{T}(D_1); \mathcal{T}(D_2) \\
= (\text{pre}_1(V) \vdash \text{post}_1(V, V)); (\text{pre}_2(V) \vdash \text{post}_2(V, V)) \\
= (\text{pre}_1(V) \land \neg(\text{post}_2(V, V) \circ \neg\text{pre}_2(V))) \\
\vdash (\text{post}_1(V, V) \circ \text{post}_2(V, V)) \\
= (\text{pre}_1(V) \land \neg\exists x \cdot (\text{post}_2(V, x) \land \neg\text{pre}_2(x))) \\
\vdash \exists y \cdot (\text{post}_1(V, y) \land \text{post}_2(y, V)) \\
= \mathcal{T}(\text{pre}_1(V) \land \neg\exists x \cdot (\text{post}_2(v, x) \land \neg\text{pre}_2(x))) \\
\vdash \exists y \cdot (\text{post}_1(v, y) \land \text{post}_2(y, v')) \\
= \mathcal{T}(D_1; D_2) \]

(5) Let \( g(X) = (D; X) < b(v) \triangleright II \), and \( h(X) = (\mathcal{T}(D); X) < \hat{b}(V) \triangleright \mathcal{T}(II) \).

The conclusion follows from Lemma 4.10 and the fact that \( \mathcal{T} \) is universally conjunctive, and \( \mathcal{T} \circ g = h \circ \mathcal{T} \).

**Remark**

An alternative timed interpretation of a design \( \text{pre} \vdash \text{post} \) can be

\[ \mathcal{A}(\text{pre} \vdash \text{post}) =_{df} \text{pre} \vdash (t = 0) \land \text{post} \]

where execution of a termination program does not take time. Like \( \mathcal{T} \), \( \mathcal{A} \) is also a homomorphism.

Clearly, the theory of relational formulae is less able to describe things accurately than DC that contains more formulae. But for any DC formula \( F \), we can single out those relational formulae that approximate to \( D \). The best approximation to \( P \) is the least upper bound of this set.

**Definition 4.12** (Approximation)

\[ \text{approx}(F) =_{df} \{ P \vdash Q \mid F \Rightarrow (P \vdash Q) \} \]

From Theorem 4.9(2) it follows that \( \text{approx}(F) \) is relational.

**Theorem 4.13**

\( \text{approx} \) is monotonic and idempotent, and satisfies
\[ F \Rightarrow \text{approx}(F) \]

**Theorem 4.14**

1. \( \text{approx}(F_1 \lor F_2) \Rightarrow \text{approx}(F_1) \lor \text{approx}(F_2) \)
2. \( \text{approx}(F_1 < b(V) \triangleright F_2) \Rightarrow (\text{approx}(F_1) < b(V) \triangleright \text{approx}(F_2)) \)
3. \( \text{approx}(F_1 ; F_2) \Rightarrow \text{approx}(F_1) ; \text{approx}(F_2) \)

**Theorem 4.15** (Closure of weakest fixed point)

If \( \Phi \) is formed by relational formulae and operators \( \lor, <b> \) and \( ; \), then \( \mu X \Phi(X) \) is also a relational formula.

**Proof.**

\[
\begin{align*}
\text{approx}(\mu X \Phi(X)) & \quad \{ \text{fixed point} \} \\
= \quad & \text{approx}(\Phi(\mu X \Phi(X))) \quad \{ \text{Theorem 4.14} \} \\
\Rightarrow \quad & \Phi(\text{approx}(\mu X \Phi(X)))
\end{align*}
\]

from which and the law of weakest fixed point it follows that

\[
\text{approx}(\mu X \Phi(X)) \Rightarrow \mu X \Phi(X)
\]

The opposite inequation follows from Theorem 4.13.

---

5. Sequential Hybrid Programs

5.1. Syntax and Semantics

In this section, we study Sequential Hybrid Programs (SHP) initially proposed in [Xu97, PWX98]. As a reactive system, an SHP program may well be nonterminating. We now not only observe the initial and the final states, but also the intermediate behaviours of the system. As before, we use Boolean variable \( ok \) to denote that the program has been started. But instead of termination, \( ok' \) now denotes that the program is stable, i.e., the program is not in an endless loop of instantaneous transitions. To distinguish intermediate behaviours from those on termination, we introduce another two global variables \( wait, wait' : \text{Bool} \). When \( wait \) is true the program is started in an intermediate state, and when \( wait' \) is true the program has not terminated.

Taking new global variables \( wait \) and \( wait' \) into account, Definition 4.6 of sequential composition is generalised as follows

**Definition 5.1.** (Revised definition of sequential composition)

\[
P; Q =_{df} \exists w, o \cdot (P[w, o/\text{wait'}, ok'] \circ Q[w, o/wait, ok])
\]

Sequential composition has the unit

\[
11 =_{df} ok \Rightarrow (ok' \land (\ell = 0) \land (\triangledown = \triangledown') \land (wait = \text{wait}))
\]
The introduction of intermediate waiting state has implication for sequential composition; all the intermediate observation of $P$ are also intermediate observations of $(P; Q)$. If $Q$ is asked to start in a waiting state of $P$, it leaves the state unchanged, i.e., it satisfies the *healthiness condition*.

$\text{(H)} \quad Q = \mathcal{H} \triangleleft \text{wait} \triangleright Q$

**Theorem 5.2.** (Closure of healthy formulae)

If both $P_1$ and $P_2$ are healthy formulae, so are $(P_1 \lor P_2)$, $P_1 \triangleleft \hat{b}(V) \triangleright P_2$, and $(P_1; P_2)$.

*Proof.* We only show the case for sequential composition.

\[
\begin{align*}
(P_1; P_2) &= (\mathcal{H} \triangleleft \text{wait} \triangleright P_1); P_2 \quad \{\text{Theorem 3.10(3)}\} \\
&= (\mathcal{H}; P_2) \triangleleft \text{wait} \triangleright (P_1; P_2) \quad \{\mathcal{H}; X = X\} \\
&= P_2 \triangleleft \text{wait} \triangleright (P_1; P_2) \quad \{\text{Def of } \triangleleft \hat{b}\triangleright\} \\
&= \mathcal{H} \triangleleft \text{wait} \triangleright (P_1; P_2) \quad \square
\end{align*}
\]

We denote the set of all healthy formulae by $\mathcal{HD}$.

**Theorem 5.3.** $\mathcal{HD}$ forms a complete lattice.

*Proof.* $Q$ is healthy iff it satisfies

\[
Q = \mathcal{H}(Q)
\]

where

\[
\mathcal{H}(X) = \triangleleft \mathcal{H} \triangleleft \text{wait} \triangleright X
\]

The conclusion follows from Tarski’s Theory [Tar55].

For a mapping $F$ closed in $\mathcal{HD}$, i.e., $\phi \in \mathcal{HD}$ implies $F(\phi) \in \mathcal{HD}$, we use $\mu_{\mathcal{HD}} \cdot F$ to stand for the weakest fixed point in the complete lattice $\mathcal{HD}$.

We next study SHP using DC formulae satisfying the above healthiness condition. The language of SHP can be considered as an extension of the refinement calculus of Section 4 by a specification statement using DC formulae for describing possibly hybrid components with time-dependent dynamics. To express the programs more conveniently, we introduce a syntax which is similar to conventional programming notations. For simplicity, we assume $v$ is the only system variable. It is straightforward to extend to the case where there is a vector of variables.

\[
P ::= S \mid P_1; P_2 \mid P_1 \sqcap P_2 \mid (P_1 \triangleleft b \triangleright P_2) \mid \text{while } b \text{ do } P \text{ od}
\]

Denoting a design of an SHP program, $S$ is of the form $(\alpha \vdash_{\mathcal{H}} (\beta \triangleleft \text{wait} \triangleright \gamma))$ where $\alpha$, $\beta$ and $\gamma$ are DC formulae which do not mention $ok$, $ok'$, $wait$ and $wait'$; Sequential composition is represented by $P_1; P_2$ and its meaning is that if $P_1$ terminates then $P_2$ is executed immediately afterwards; $\sqcap$ is the non-deterministic choice; Iteration statement is executed repeatedly until the Boolean guard becomes false.

An SHP design is interpreted as a DC formula
**Definition 5.4.**

\[
(\alpha \vdash_H (\beta \lessdot \text{wait}' \triangleright \gamma)) =_{df} \mathcal{H}(\text{ok} \land \alpha \Rightarrow (\text{ok}' \land (\beta \lessdot \text{wait}' \triangleright \gamma)))
\]

The meaning is that if the program is started, then under the condition \(\alpha\), it guarantees that it will not diverge and the intermediate behaviours satisfy \(\beta\) while terminated behaviours satisfy \(\gamma\).

A particular form of SHP designs describes the dynamics that a phase of a hybrid system satisfies. We call this a phase statement, denoted by \(\ll C \gg\) where \(C\) is a DC formula. Continuous evolution can be specified by an invariant expressing the dynamical laws. Formally,

\[
[\ll C \gg] =_{df} (\text{true} \vdash_H C)
\]

The semantics of SHP program combinators is defined as usual

\[
\begin{align*}
\ll P_1; P_2 \gg & =_{df} \ll P_1 \gg; \ll P_2 \gg, \\
\ll P_1 \sqcap P_2 \gg & =_{df} \ll P_1 \gg \lor \ll P_2 \gg, \\
\ll (P_1 \lessdot b \triangleright P_2) \gg & =_{df} \ll (\ll P_1 \gg \lessdot b(\bar{V}) \triangleright \ll P_2 \gg), \\
[\text{while } b \text{ do } P \text{ od}] & =_{df} \mu_{HD}X \bullet ((\ll P \gg \lessdot b(V) \triangleright II).
\end{align*}
\]

As in Section 4, we map a design in refinement calculus, henceforth called a sequential design to distinguish from a design in SHP, into a DC formula.

**Definition 5.5.** (Revised interpretation of sequential design)

\[
\mathcal{R}(D) =_{df} \mathcal{H}(\mathcal{A}(D); (\text{ok} \Rightarrow (\text{ok}' \land \neg \text{wait}' \land II)))
\]

where \(\mathcal{A}(D)\) is the alternative interpretation of designs in Section 4. We take this interpretation because in SHP we express timing behaviours by DC formulae and assume other statements do not take time. It is easy to prove that

\[
\mathcal{R}(b \vdash R) = II \lessdot \text{wait} \triangleright ((\text{ok} \land b(V)) \Rightarrow (\text{ok}' \land \neg \text{wait}' \land R(V, \bar{V}) \land (\ell = 0))) \square
\]

It follows that \(\mathcal{R}(D)\) is an SHP design. The following theorem states \(\mathcal{R}\) is also a homomorphism over sequential designs.

**Theorem 5.6.**

1. \(\mathcal{R}(D_1 \lor D_2) = \mathcal{R}(D_1) \lor \mathcal{R}(D_2)\)
2. \(\mathcal{R}(D_1 \land D_2) = \mathcal{R}(D_1) \land \mathcal{R}(D_2)\)
3. \(\mathcal{R}(D_1 \lessdot b(v) \triangleright D_2) = \mathcal{R}(D_1) \lessdot b(\bar{V}) \triangleright \mathcal{R}(D_2)\)
4. \(\mathcal{R}(D_1; D_2) = \mathcal{R}(D_1); \mathcal{R}(D_2)\)
5. \(\mathcal{R}(\mu X \bullet ((D; X) \lessdot b(v) \triangleright II)) = \mu_{HD}X \bullet ((\mathcal{R}(D); X) \lessdot b(\bar{V}) \triangleright II)\) \square

**Theorem 5.7.** If \(D_1 \Rightarrow D_2\) holds in sequential refinement calculus, then \(\mathcal{R}(D_1) \Rightarrow \mathcal{R}(D_2)\). \square

This together with Theorem 5.6 indicates that laws of sequential refinement calculus are still valid when sequential designs are considered as SHP programs.

**Example 5.8.**
\[ x := x + 1; x := x + 2 = x := x + 3 \]
\[ x := e; (P \triangleright b(x) \triangleright Q) = ((x := e; P) \triangleright b(e) \triangleright (x := e; Q)) \]

We can define the Hoare triple as

\[ \{ \text{pre} (v) \} \; D \; \{ \text{post} (v) \} =_d D \Rightarrow (\text{pre} (v) \vdash \text{post} (v')). \]

Theorem 5.7 also says that one can still use the traditional Hoare logic to verify the discrete component of a SHP program.

5.2. Verification of SHP Programs

Refinement laws expressed in Theorem 4.2 can be generalised to SHP designs

**Theorem 5.9.**

1. \( (\alpha_1 \vdash_{\text{H}} (\beta_1 \triangleleft \text{wait'} \triangleright \gamma_1)) \lor (\alpha_2 \vdash_{\text{H}} (\beta_2 \triangleleft \text{wait'} \triangleright \gamma_2)) = (\alpha_1 \land \alpha_2 \vdash_{\text{H}} ((\beta_1 \lor \beta_2) \triangleleft \text{wait'} \triangleright (\gamma_1 \lor \gamma_2)),) \)
2. \( ((\alpha_1 \vdash_{\text{H}} (\beta_1 \triangleleft \text{wait'} \triangleright \gamma_1)) \land b(V) \triangleright (\alpha_2 \vdash_{\text{H}} (\beta_2 \triangleleft \text{wait'} \triangleright \gamma_2))) = ((\alpha_1 \land b(V) \triangleright \alpha_2) \vdash_{\text{H}} ((\beta_1 \triangleleft (\gamma_1 \lor \gamma_2)) \land b(V) \triangleright (\gamma_1 \lor \gamma_2)) \),
3. \( (\alpha_1 \vdash_{\text{H}} (\beta_1 \triangleleft \text{wait'} \triangleright \gamma_1)) ; (\alpha_2 \vdash_{\text{H}} (\beta_2 \triangleleft \text{wait'} \triangleright \gamma_2)) = (\neg (\alpha_1 ; \text{true}) \land (\alpha_2 ; \text{true}) \vdash_{\text{H}} ((\beta_1 \lor (\gamma_1 \lor \gamma_2)) \triangleleft \text{wait'} \triangleright (\gamma_1 \lor \gamma_2)) \). \]

We denote by \( \{ \alpha \} \; P \; \{ (\beta \triangleleft \text{wait'} \triangleright \gamma) \} \) that an SHP program \( P \) implements a design \( (\alpha \vdash_{\text{H}} (\beta \triangleleft \text{wait'} \triangleright \gamma)) \). This is interpreted as

\[ [P] \Rightarrow (\alpha \vdash_{\text{H}} (\beta \triangleleft \text{wait'} \triangleright \gamma)) \]

The following proof rules are derived directly from Theorem 5.9.

\[
\frac{\{ \alpha \} \; P_1 \; \{ (\beta \triangleleft \text{wait'} \triangleright \gamma) \}, \{ \alpha \} \; P_2 \; \{ (\beta \triangleleft \text{wait'} \triangleright \gamma) \}}{\{ \alpha \} \; P_1 \cap P_2 \; \{ (\beta \triangleleft \text{wait'} \triangleright \gamma) \}}
\]

\[
\frac{\{ \alpha \land b(V) \} \; P_1 \; \{ (\beta \triangleleft \text{wait'} \triangleright \gamma) \}, \{ \alpha \land \neg b(V) \} \; P_2 \; \{ (\beta \triangleleft \text{wait'} \triangleright \gamma) \}}{\{ \alpha \} \; (P_1 \triangleleft b \triangleright P_2) \; \{ (\beta \triangleleft \text{wait'} \triangleright \gamma) \}}
\]

\[
\frac{\alpha_2 \Rightarrow \alpha_1, \; \beta_1 \Rightarrow \beta_2, \; \gamma_1 \Rightarrow \gamma_2, \; \alpha_1 \; P \; \{ (\beta_1 \triangleleft \text{wait'} \triangleright \gamma_1) \}}{\alpha_2 \; P \; \{ (\beta_2 \triangleleft \text{wait'} \triangleright \gamma_2) \}}
\]

In the following, we study several special cases of SHP design formulae and investigate verification rules for them.

**Special case 1:** formula \( \alpha \) in the SHP design is a state predicate \( \text{pre} (V) \), denoting a precondition.

\[
\frac{\{ \text{pre} (V) \} \; P_1 \; \{ (\beta_1 \triangleleft \text{wait'} \triangleright (\gamma_1 \land \text{mid}(V))) \}, \{ \text{mid}(V) \} \; P_2 \; \{ (\beta_2 \triangleleft \text{wait'} \triangleright (\gamma_2 \land \text{post}(V))) \}}{\{ \text{pre} (V) \} \; P_1 \cap P_2 \; \{ ((\beta_1 \lor (\gamma_1 \land \text{mid}(V))) \triangleleft \text{wait'} \triangleright ((\gamma_1 \lor \gamma_2) \land \text{post}(V))) \}}
\]
**Special case 2:** the SHP design is of the form \((\text{pre}(V) \vdash_{H} (\alpha < \text{wait'} \triangleright (\alpha \land \text{post}(V))))\). Denote \(\{\text{pre}(V)\} \quad P \quad \{(\alpha < \text{wait'} \triangleright (\alpha \land \text{post}(V)))\}\) by \(\alpha : \{\text{pre}(V)\} \quad P \quad \{\text{post}(V)\}\).

\[
\begin{align*}
\alpha : \{\text{pre}(V)\} & \quad P_1 \quad \{\text{mid}(V)\}, \quad \beta : \{\text{mid}(V)\} \quad P_2 \quad \{\text{post}(V)\} \\
\{\text{pre}(V)\} \quad P_1 ; P_2 & \quad \{(\alpha \lor (\alpha ; \beta)) < \text{wait'} \triangleright ((\alpha ; \beta) \land \text{post}(V))\}\end{align*}
\]

\[
\alpha ; \beta \Rightarrow \alpha
\]

\[
\begin{align*}
\alpha : \{\text{pre}(V)\} & \quad P_1 \quad \{\text{mid}(V)\}, \quad \beta : \{\text{mid}(V)\} \quad P_2 \quad \{\text{post}(V)\} \\
\alpha : \{\text{pre}(V)\} & \quad P_1 \quad P_2 \quad \{\text{post}(V)\}\end{align*}
\]

\[
\alpha ; \alpha \Rightarrow \alpha, \quad \ell = 0 \Rightarrow \alpha \\
\{\text{(b \land r)}(V)\} \quad P \quad \{(\alpha < \text{wait'} \triangleright (\alpha \land \ell > c \land \text{r}(V)))\}\]
\[
\alpha : \{\text{r}(V)\} \quad \text{while b do P od} \quad \{\text{r}(V)\}\]

where \(c\) is a positive constant.

**Proof.** We only prove the last rule. First, we show

\[
(\neg \text{wait} \land \text{ok} \land r(\bar{V}) \land \left[\text{while b do P od}\right]) \Rightarrow (\text{ok'} \land (\alpha < \text{wait'} \triangleright (\alpha \land \text{r}(\bar{V}))))
\]

Let \(\left[\text{while b do P od}\right]\) in the proof below.

\[
\neg \text{wait} \land \text{ok} \land r(\bar{V}) \land \left[\text{W}\right] \quad \{\text{fixed point}\}
\]

\[
= \neg \text{wait} \land \text{ok} \land r(\bar{V}) \land (\left[\text{P}; \left[\text{W}\right]\right] < b(\bar{V}) \triangleright I) \quad \{\text{Def of } (<b> )\}
\]

\[
\Rightarrow ((\neg \text{wait} \land \text{ok} \land (b \land r)(\bar{V}) \land \left[\text{P}; \left[\text{W}\right]\right]) \lor (\neg \text{wait} \land \text{ok} \land r(\bar{V}) \land \left[\text{II}\right])) \quad \{\text{premises of the rule and Def of II}\}
\]

\[
\Rightarrow ((\neg \text{wait} \land \text{ok} \land (b \land r)(\bar{V}) \land \left[\text{W}\right]) \lor (\text{ok'} \land \neg \text{wait'} \land r(\bar{V}) \land \left[\text{W}\right]) \lor (\text{ok'} \land \neg \text{wait'} \land r(\bar{V}) \land \left[\text{II}\right])) \quad \{\text{predicate calculus}\}
\]

\[
\Rightarrow ((\text{ok'} \land (\alpha < \text{wait'} \triangleright \alpha \land \ell > c \land \text{r}(\bar{V})))\left[\text{W}\right]) \lor (\text{ok'} \land \neg \text{wait'} \land r(\bar{V}) \land \left[\text{W}\right]) \quad \{\text{Def of } (<\text{wait'} >)\}
\]

\[
\Rightarrow ((\text{ok'} \land \text{wait'} \land \left[\text{W}\right]) \lor (\text{ok'} \land \neg \text{wait'} \land r(\bar{V}) \land \left[\text{W}\right]) \lor (\text{ok'} \land \neg \text{wait'} \land r(\bar{V}) \land \left[\text{W}\right]) \quad \{\text{Def of } (<\text{wait'} >)\}
\]

\[
\Rightarrow ((\text{ok'} \land \alpha < \text{wait'} \triangleright \alpha \land \ell > c \land \text{r}(\bar{V}))) \lor ((\text{ok'} \land \neg \text{wait'} \land r(\bar{V}) \land \left[\text{W}\right]) \lor (\text{ok'} \land \neg \text{wait'} \land r(\bar{V}) \land \left[\text{W}\right]) \quad \{\text{Def of } (<\text{wait'} >)\}
\]
By induction, it follows that

\[
\neg \text{wait} \land \text{ok} \land r(\bar{V}) \land [W]
\]

\[
\Rightarrow \forall n \bullet ((ok^i \land (\alpha < \text{wait}' \triangleright (\alpha \land r(\bar{V}))))
\]

\[
\lor ((\alpha \land \ell > nc); (\neg \text{wait} \land \text{ok} \land r(\bar{V}) \land [W]))
\]

\[
= (ok^i \land (\alpha < \text{wait}' \triangleright (\alpha \land r(\bar{V}))))
\]

\[
\lor ((\alpha \land (\forall n \bullet \ell > nc)); (\neg \text{wait} \land \text{ok} \land r(\bar{V}) \land [W]))
\]

Since \((\forall n \bullet \ell > nc) = \text{false}\), we have

\[
\neg \text{wait} \land \text{ok} \land r(\bar{V}) \land [W] \Rightarrow (ok^i \land (\alpha < \text{wait}' \triangleright (\alpha \land r(\bar{V}))))
\]

Therefore,

\[
[W] \quad \lor \quad (\neg \text{wait} \land [W]) \quad \quad \text{\{predicate calculus\}}
\]

\[
\Rightarrow \text{wait} \land [II] \lor (\neg \text{wait} \land (\text{ok} \land r(\bar{V}) \Rightarrow (ok^i \land (\alpha < \text{wait}' \triangleright (\alpha \land r(\bar{V})))))) \quad \{\text{previous result}\}
\]

\[
\Rightarrow (II < \text{wait} \triangleright ((ok \land r(\bar{V}) \Rightarrow (ok^i \land (\alpha < \text{wait}' \triangleright (\alpha \land r(\bar{V}))))))) \quad \{\text{Def of } \triangleright \}
\]

\[
\Rightarrow (r(\bar{V}) \vdash_{\forall} (\alpha < \text{wait}' \triangleright \alpha \land r(\bar{V})))
\]

\[
\square
\]

5.3. A Simple Example

We consider the following toy example, named CRUISE for convenience in exposition. In the system, the speed of the car \( V \) is continuous and is controlled by a computer by periodically calculating the acceleration \( a \) which must be applied for the next time unit. The sampling of the current speed is represented by the statement \( v := V \). If \( v < 80 \), the car will accelerate and the rate of acceleration is computed by a control algorithm \( A \). If \( v \geq 80 \), the car will decelerate at a rate determined by a control algorithm \( D \). When accelerating, the behaviour of the car in the next time unit is defined by the phase statement \( \ll \exists x. x = \bar{a} \land ||V - x|| \leq 3 \land \bar{V} \geq 1 \land (\ell < 1 < \text{wait}' \triangleright \ell = 1) \gg \). Due to hardware limitations, the actual acceleration \( \bar{V} \) may not be exactly the same as specified. Condition \( ||V - x|| \leq 3 \land \bar{V} \geq 1 \) states that \( \bar{V} \) is within an error of 3 from the value of \( x \) and that it is at least 1. The design task is to construct components \( A \) and \( D \) such that the speed never exceeds 100.

The requirement can be expressed by the following correctness formula

\[
(||V < 100|| \lor \ell = 0) : \{\text{true}\} \quad \text{CRUISE} \quad \{\text{true}\}
\]

Due to compositionality of our method, we can verify the correctness of the system based on the specifications of \( A \) and \( D \). We record the verification by what is commonly called proof outline in program verification. In the outline, the formulae, marked by shaded boxes, represent conditions and conclusions of various verification rules used. For example, the formula in line (2) is obtained by applying the assignment axiom to the statement in line (1). The formula in turn serves as the pre-condition for the statement in line (3). Formula in line (6) is carried over from line (4), and is used as the pre-condition of the iteration.
body; line (17) denotes the property that the body has, and by the iteration rule, the iteration statement satisfies the property in line (19).

\[
V := 0; \quad \{V = 0\} \\
\text{alarm} := \text{off}; \quad \{V < 100\} \\
\text{while true do} \quad \{V < 100\} \\
\quad v := V; \quad \{V < 100 \land v = V\} \\
\quad \text{if } v < 80 \text{ then} \quad \{V < 80\} A; \quad \{V < 80 \land 0 < a \leq 15\} \\
\qquad (||V < 100\| \land \ell = 0\| < \text{wait} \land \ell = 1\|) \quad \{V < 100\} D; \quad \{V < 100\} \\
\qquad (||V < 100\| \land \ell = 0\| < \text{wait} \land \ell = 1\|) \quad (||V < 100\| \land l > 0.5\land \vec{V} < 100) \quad \text{else} \quad \{V < 100\} \\
\quad \text{fi} \quad (||V < 100\| \land \ell = 0\| < \text{wait} \land \ell = 1\|) \quad (||V < 100\| \land l > 0.5\land \vec{V} < 100) \quad \{V < 80\} A \quad \{V < 80 \land 0 < a \leq 15\} \\
\{true\} A \quad \{a > 0 \land a \geq 0.1 \ast e^{100/(v+20)} - 1 \land a \leq 0.1 \ast e^{100/(v+20)}\} \\
\text{Correctness of the system is maintained, because} \quad 0.1 \ast e^{100/(v+20)} \leq 15 \quad \text{for any } v \geq 0
\]
Finally, the control algorithm $A$ is developed. It can be verified using Hoare Logic.

\[
\begin{align*}
    c &:= 100/(v + 20); \\
    k &:= 1; \\
    a &:= 0.1; \\
    r &:= 0.6 * 5; \\
\{a = 0.1 * \sum_{n=0}^{k-1} \frac{1}{n!}(\frac{100}{v+20})^n \land r = 0.6 * 5^k/k!\} \\
\end{align*}
\]

\[
\text{while } k < 5 \lor r \geq 1 \text{ do} \\
\begin{align*}
    a &:= a + 0.1 * c; \\
    k &:= k + 1; \\
    c &:= (100 * c)/(v + 20) * k; \\
    r &:= (5 * r)/k \\
\end{align*}
\]

\[
\text{od} \\
\{a = 0.1 * \sum_{n=0}^{k-1} \frac{1}{n!}(\frac{100}{v+20})^n \land r = 0.6 * 5^k/k! \land k > 6 \land r < 1\} \\
\{a > 0 \land a \geq 0.1 * e^{100/(v+20)} - r \land a \leq 0.1 * e^{100/(v+20)}\}
\]

6. Conclusion

This paper adds a number of advanced features into DC, and aims to provide a semantical framework for sequential hybrid programs. In particular,

1. A high order hiding operator is introduced to model local variables in real-time programs.

2. A relational composition is formed to model the sequential composition operator of programming languages.

3. Neighbourhood values of state variable are used to represent the stable states of program variables.

We have shown that the super-dense chop is the product of the ITL chop and the relational composition operator, where the former models the sequential composition of continuously evolving physical environment, and the latter composes discrete computing agents.

This paper also presents a link between the theory of designs and DC. It enables us to discuss time-independent properties of real-time programs in untimed refinement calculus, and as a result greatly simplifies the reasoning task for hybrid systems.

In this paper, the advanced features of DC are employed in formalising SHP, and provide verification methods for reasoning about timing properties. The same notations have been used to tackle concurrent real-time systems and provide a denotational semantics for an industrial specification language [LiH99].

Acknowledgements: we thank our colleagues for discussions and in particular Richard Moore for comments.
References


