Programming with Infinitesimals: A WHILE-Language for Hybrid System Modeling[⋆]

Kohei Suenaga[1] and Ichiro Hasuo[2]

[1] JSPS Research Fellow, Kyoto University, Japan
[2] University of Tokyo, Japan

Abstract. We add, to the common combination of a WHILE-language and a Hoare-style program logic, a constant `dt` that represents an infinitesimal (i.e. infinitely small) value. The outcome is a framework for modeling and verification of *hybrid systems*: hybrid systems exhibit both continuous and discrete dynamics and getting them right is a pressing challenge. We rigorously define the semantics of programs in the language of *nonstandard analysis*, on the basis of which the program logic is shown to be sound and relatively complete.

1 Introduction

Hybrid systems are systems that deal with both discrete and continuous data. They have rapidly gained importance since more and more physical systems—cars, airplanes, etc.—are controlled with computers. Their sensors will provide physical, continuous data, while the behavior of controller software is governed by discrete data. Those information systems which interact with a physical ambience are more generally called *cyber-physical systems (CPS)*; hybrid systems are an important building block of CPSs.

Towards the goal of getting hybrid systems right, the research efforts have been mainly from two directions. *Control theory*—originally focusing on continuous data and their control e.g. via integration and differentiation—is currently extending its realm towards hybrid systems. The de facto standard SIMULINK tool for hybrid system modeling arises from this direction; it employs the block diagram formalism and offers simulation functionality—aiming at *testing* rather than *verification*. The current work is one of the attempts from the other direction—from the *formal verification* community—that advance from discrete to continuous.

Hybrid systems exhibit two kinds of dynamics: continuous *flow* and discrete *jump*. Hence for a formal verification approach to hybrid systems, the challenge is: 1) to incorporate flow-dynamics; and 2) to do so at the lowest possible cost, so that the discrete framework smoothly transfers to hybrid situations. A large body of existing work includes differential equations explicitly in the syntax; see the discussion of related work below. What we propose, instead, is to introduce a constant `dt` for an *infinitesimal* (i.e. infinitely small) value and *turn flow into jump*. With `dt`, the continuous operation of integration can be represented by a `while` loop, to which existing discrete techniques like Hoare-style program logics readily apply. For a rigorous mathematical development we employ *nonstandard analysis (NSA)* beautifully formalized by Robinson.

[⋆] We are greteful to Naoki Kobayashi and Toshimitsu Ushio for helpful discussions.

Concretely, in this paper we take the common combination of a WHILE-language, a first-order assertion language and a Hoare logic (e.g. in the textbook [10]); and add a constant `dt` to obtain a modeling and verification framework for hybrid systems. Its three ingredients are called $\textsc{While}^{\mathtt{dt}}$, $\textsc{Assn}^{\mathtt{dt}}$ and $\textsc{Hoare}^{\mathtt{dt}}$. These are connected by denotational semantics defined in the language of NSA. We prove soundness and relative completeness of the logic $\textsc{Hoare}^{\mathtt{dt}}$. Underlying the technical development is the idea of what we call *sectionwise execution*, illustrated by the following example.

Example 1.1 Let c_{elapse} be the following program; $\equiv$ denotes the syntactic equality.

$$c_{\mathsf{elapse}} \quad :\equiv \quad \Big[\ t := 0\ ;\quad \mathtt{while}\ t \le 1\ \mathtt{do}\ t := t + \mathtt{dt}\ \Big]$$

The value designated by `dt` is infinitesimal; therefore the `while` loop will not terminate within finitely many steps. Nevertheless it is intuitive to expect that after an "execution" of this program (which takes an infinitely long time), the value of t should be infinitely close to 1. How can we turn this intuition into a mathematical argument?

Our idea is to think about *sectionwise execution*. For each natural number i we consider the *i-th section* of the program c_{elapse}, denoted by $c_{\mathsf{elapse}}|_i$. Concretely, $c_{\mathsf{elapse}}|_i$ is defined by replacing the infinitesimal `dt` in c_{elapse} by $\frac{1}{i+1}$:

$$c_{\mathsf{elapse}}|_i \quad :\equiv \quad \Big[\ t := 0\ ;\quad \mathtt{while}\ t \le 1\ \mathtt{do}\ t := t + \tfrac{1}{i+1}\ \Big]\ .$$

Informally $c_{\mathsf{elapse}}|_i$ is the "i-th approximation" of the original c_{elapse}.

A section $c_{\mathsf{elapse}}|_i$ does terminate within finite steps and yields $1 + \frac{1}{i+1}$ as the value of t. Now we collect the outcomes of sectionwise executions and obtain a sequence

$$(\,1+1,\ 1+\tfrac{1}{2},\ 1+\tfrac{1}{3},\ \dots,\ 1+\tfrac{1}{i},\ \dots)$$

which is thought of as an incremental approximation of the actual outcome of the original program c_{elapse}. Indeed, in the language of NSA, the sequence represents a *hyperreal number r* that is infinitely close to 1.

We note that, as is clear from this example, a program of $\textsc{While}^{\mathtt{dt}}$ is not executable in general. We would rather regard $\textsc{While}^{\mathtt{dt}}$ as a modeling language for hybrid systems, with a merit of being close to a common programming style.

The idea of *turning flow into jump* with `dt` and NSA seems applicable to other discrete modeling/verification techniques than `while`-language and Hoare logic. We wish to further explore this potentiality. Adaptation of more advanced techniques for deductive program verification—such as invariant generation and type systems—to the presence of `dt` is another important direction of future work.

Due to the lack of space all the proofs are deferred to the extended version [9].

Related work There have been extensive research efforts towards hybrid systems from the formal verification community. Unlike the current work where we turn flow into jump via `dt`, most of them feature acute distinction between flow- and jump-dynamics.

Hybrid automaton [1] is one of the most successful approaches to verification of hybrid systems. A number of model-checking algorithms have been invented for automatic verification. The deductive approach in the current work—via theorem-proving in

HOARE$^{\mathtt{dt}}$—has an advantage of handling parameters (i.e. universal quantifiers or free variables that range over an infinite domain) well; model checking in such a situation necessarily calls for some abstraction technique. Our logic HOARE$^{\mathtt{dt}}$ is compositional too: a property of a whole system is inferred from the ones of its constituent parts.

Deductive verification of hybrid systems has seen great advancement through a recent series of work by Platzer and his colleagues, including [5, 6]. Their formalism is variations of *dynamic logic*, augmented with differential equations to incorporate flow-dynamics. They have also developed advanced techniques aimed at automated theorem proving, resulting in a sophisticated tool called KeYmaera [7]. We expect our approach (namely incorporating flow-dynamics via `dt`) offer a smoother transfer of existing discrete verification techniques to hybrid applications. Additionally, our preliminary observations suggest that some of the techniques developed by Platzer and others can be translated into the techniques that work in our framework.

The use of NSA as a foundation of hybrid system modeling is not proposed for the first time; see e.g. [2, 8]. Compared to these existing work, we claim our novelty is a clean integration of NSA and the widely-accepted programming style of `while`-languages, with an accompanying verification framework by HOARE$^{\mathtt{dt}}$.

2 A Nonstandard Analysis Primer

This section is mainly for fixing notations. For more details see e.g. [4].

The type of statements "there exists $i_0 \in \mathbb{N}$ such that, for each $i \geq i_0$, $\varphi(i)$ holds" is typical in analysis. It is often put as "for sufficiently large $i \in \mathbb{N}$." This means: the set $\{i \in \mathbb{N} \mid \varphi(i) \text{ holds}\} \subseteq \mathbb{N}$ belongs to the family $\mathcal{F}_0 := \{S \subseteq \mathbb{N} \mid \mathbb{N} \setminus S \text{ is finite}\}$.

In NSA, the family $\mathcal{F}_0$ is extended to so-called an *ultrafilter* $\mathcal{F}$. The latter is a convenient domain of "i-indexed truth values": notably for each set $S \subseteq \mathbb{N}$, exactly one out of S and $\mathbb{N} \setminus S$ belongs to $\mathcal{F}$.

Definition 2.1 (Ultrafilter) A *filter* is a family $\mathcal{F} \subseteq \mathcal{P}(\mathbb{N})$ such that: 1) $X \in \mathcal{F}$ and $X \subseteq U$ implies $U \in \mathcal{F}$; 2) $X \cap Y \in \mathcal{F}$ if $X, Y \in \mathcal{F}$. A nonempty filter $\mathcal{F} \neq \emptyset$ is said to be *proper* if it does not contain $\emptyset \subseteq \mathbb{N}$; equivalently, if $\mathcal{F} \neq \mathcal{P}(\mathbb{N})$. An *ultrafilter* is a maximal proper filter; equivalently, it is a filter $\mathcal{F}$ such that for each $S \subseteq \mathbb{N}$, exactly one out of S and $\mathbb{N} \setminus S$ belongs to $\mathcal{F}$.

A filter $\mathcal{F}'$ can be always extended to an ultrafilter $\mathcal{F} \supseteq \mathcal{F}'$; this is proved using Zorn's lemma. Since the family $\mathcal{F}_0$ is easily seen to be a filter, we have:

Lemma 2.2 *There is an ultrafilter $\mathcal{F}$ that contains $\mathcal{F}_0 = \{S \subseteq \mathbb{N} \mid \mathbb{N} \setminus S$ is finite$\}$.*

Throughout the rest of the paper we fix such $\mathcal{F}$. Its properties to be noted: 1) $\mathcal{F}$ is closed under finite intersections and infinite unions; 2) exactly one of S or $\mathbb{N} \setminus S$ belongs to $\mathcal{F}$, for each $S \subseteq \mathbb{N}$; and 3) if S is such that $\mathbb{N} \setminus S$ is finite, then $S \in \mathcal{F}$.

We say "$\varphi(i)$ holds for almost every $i \in \mathbb{N}$" for the fact that the set $\{i \mid \varphi(i) \text{ holds}\}$ belongs to $\mathcal{F}$. For its negation we say "for negligibly many i."

Definition 2.3 (Hypernumber $\boldsymbol{d} \in {}^*\mathbb{D}$) For a set $\mathbb{D}$ (typically it is $\mathbb{N}$ or $\mathbb{R}$), we define the set ${}^*\mathbb{D}$ by ${}^*\mathbb{D} := \mathbb{D}^{\mathbb{N}} / \sim_{\mathcal{F}}$. It is the set of infinite sequences on $\mathbb{D}$ modulo the following equivalence $\sim_{\mathcal{F}}$: we define $(d_0, d_1, \dots) \sim_{\mathcal{F}} (d'_0, d'_1, \dots)$ by

$$d_i = d'_i \text{ “for almost every } i\text{,” that is, } \{i \in \mathbb{N} \mid d_i = d'_i\} \in \mathcal{F}.$$

An equivalence class $\big[(d_i)_{i\in\mathbb{N}}\big]_{\sim_{\mathcal{F}}} \in {}^*\mathbb{D}$ shall be also denoted by $\big[(d_i)_{i\in\mathbb{N}}\big]$ or $(d_i)_{i\in\mathbb{N}}$ when no confusion occurs. An element $\boldsymbol{d} \in {}^*\mathbb{D}$ is called a *hypernumber*; in contrast $d \in \mathbb{D}$ is a *standard number*. Hypernumbers will be denoted in boldface like $\boldsymbol{d}$.

We say that $(d_i)_{i\in\mathbb{N}}$ is a *sequence representation* of $\boldsymbol{d} \in {}^*\mathbb{D}$ if $\boldsymbol{d} = [(d_i)_i]$. Note that, given $\boldsymbol{d} \in {}^*\mathbb{D}$, its sequence representation is not unique. There is a canonical embedding $\mathbb{D} \hookrightarrow {}^*\mathbb{D}$ mapping d to $[(d, d, \dots)]$; the latter shall also be denoted by d.

Definition 2.4 (Operations and relations on ${}^*\mathbb{D}$) An operation $f : \mathbb{D}^k \to \mathbb{D}$ of any finite arity k (such as $+ : \mathbb{R}^2 \to \mathbb{R}$) has a canonical “pointwise” extension $f : ({}^*\mathbb{D})^k \to {}^*\mathbb{D}$. A binary relation $R \subseteq \mathbb{D}^2$ (such as $<$ on real numbers) also extends to $R \subseteq ({}^*\mathbb{D})^2$.

$$f\big(\,\big[\,(d_i^{(0)})_{i\in\mathbb{N}}\big], \dots, \big[\,(d_i^{(k-1)})_{i\in\mathbb{N}}\big]\,\big) := \Big[\,\big(f(d_i^{(0)}, \dots, d_i^{(k-1)})\big)_{i\in\mathbb{N}}\Big]\ ,$$
$$\big[\,(d_i)_{i\in\mathbb{N}}\big]\ R\ \big[\,(d'_i)_{i\in\mathbb{N}}\big] \quad \overset{\text{def.}}{\Longleftrightarrow} \quad d_i\ R\ d'_i \quad \text{for almost every } i.$$

These extensions are well-defined since $\mathcal{F}$ is closed under finite intersections.

Example 2.5 (ω and ω^{-1}) By ω we denote the hypernumber $\omega = [\,(1, 2, 3, \dots)\,] \in {}^*\mathbb{N}$. It is bigger than (the embedding of) any (standard) natural number $n = [\,(n, n, \dots)\,]$, since we have $n < i$ for all i except for finitely many. The presence of ω shows that $\mathbb{N} \subsetneq {}^*\mathbb{N}$ and $\mathbb{R} \subsetneq {}^*\mathbb{R}$. Its inverse $\omega^{-1} = [\,(1, \frac{1}{2}, \frac{1}{3}, \dots)\,]$ is positive ($0 < \omega^{-1}$) but is smaller than any (standard) positive real number $r > 0$.

These hypernumbers—*infinite* ω and *infinitesimal* ω^{-1}—will be heavily used.

For the set $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ of Boolean truth values we have the following. Therefore a “hyper Boolean value” does not make sense.

Lemma 2.6 *Assume that $\mathbb{D}$ is a finite set $\mathbb{D} = \{a_1, \dots, a_n\}$. Then the canonical inclusion map $\mathbb{D} \hookrightarrow {}^*\mathbb{D}$ is bijective. In particular we have ${}^*\mathbb{B} \cong \mathbb{B}$ for $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$.* $\square$

3 Programming Language WHILE$^{\mathrm{dt}}$

3.1 Syntax

We fix a countable set **Var** of *variables*.

Definition 3.1 (WHILE$^{\mathrm{dt}}$, WHILE) The syntax of our target language WHILE$^{\mathrm{dt}}$ is:

$$\begin{array}{lll}
\mathbf{AExp} \ni & a ::= & x \mid \mathtt{c}_r \mid a_1\ \mathtt{aop}\ a_2 \mid \mathtt{dt} \mid \infty \\
& & \text{where } x \in \mathbf{Var},\ \mathtt{c}_r \text{ is a constant for } r \in \mathbb{R}, \text{ and } \mathtt{aop} \in \{+, -, \cdot, \hat{}\,\} \\
\mathbf{BExp} \ni & b ::= & \mathtt{true} \mid \mathtt{false} \mid b_1 \wedge b_2 \mid \neg b \mid a_1 < a_2 \\
\mathbf{Cmd} \ni & c ::= & \mathtt{skip} \mid x := a \mid c_1; c_2 \mid \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ b\ \mathtt{do}\ c
\end{array}$$

An expression in $\mathbf{AExp}$ is said to be *arithmetic*; one in $\mathbf{BExp}$ is *Boolean* and one in $\mathbf{Cmd}$ is a *command*. The operator $a^\wedge b$ designates "a to the power of b" and will be denoted by a^b. The operator $^\wedge$ is included as a primitive for the purpose of relative completeness (Thm. 5.4). We will often denote the constant $\mathtt{c}_r$ by r.

By WHILE, we denote the fragment of WHILE$^{\mathtt{dt}}$ without the constants $\mathtt{dt}$ and ∞.

The language WHILE is much like usual programming languages with a `while` construct, such as $\mathbf{IMP}$ in the textbook [10]. Its only anomaly is a constant $\mathtt{c}_r$ for any real number r: although unrealistic from the implementation viewpoint, it is fine because WHILE is meant to be a modeling language. Then our target language WHILE$^{\mathtt{dt}}$ is obtained by adding $\mathtt{dt}$ and ∞: they designate an infinitesimal ω^{-1} and an infinite ω.

The relations $>, \le, \ge$ and $=$ are definable in WHILE$^{\mathtt{dt}}$: $x > y$ as $y < x$; $\le$ as the negation of $>$; and $=$ as the conjunction of $\le$ and $\ge$. So are all the Boolean connectives such as $\vee$ and $\Rightarrow$, using $\neg$ and $\wedge$. We emphasize that $\mathtt{dt}$ is by itself a constant and has nothing to do with a variable t. We could have used a more neutral notation like ∂; however the notation $\mathtt{dt}$ turns out to be conveniently intuitive in many examples.

Definition 3.2 (Section of WHILE$^{\mathtt{dt}}$ expression) Let e be an expression of WHILE$^{\mathtt{dt}}$, and $i \in \mathbb{N}$. The *i-th section* of e, denoted by $e|_i$, is obtained by replacing each occurrence of $\mathtt{dt}$ and ∞ in e by the constants $\mathtt{c}_{1/(i+1)}$ and $\mathtt{c}_{i+1}$, respectively. Obviously $e|_i$ is an expression of WHILE.

Example 3.3 (Train control) Our first examples model small fragments of the European Train Control System (ETCS); this is also a leading example in [5]. The following command c_{accel} models a train accelerating at a constant acceleration a, until the time ε is reached. The variable v is for the train's velocity; and z is for its position.

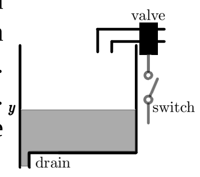
$$\begin{array}{lll} c_{\mathsf{accel}} & :\equiv & \left[\mathtt{while}\ t < \varepsilon\ \mathtt{do}\ c_{\mathsf{drive}}\right] \qquad \text{where} \\ c_{\mathsf{drive}} & :\equiv & \left[t := t + \mathtt{dt}\,;\ v := v + a \cdot \mathtt{dt}\,;\ z := z + v \cdot \mathtt{dt}\right] \end{array} \tag{1}$$

The following command $c_{\mathsf{chkAndBrake}}$ models a train that, once its distance from the boundary m gets within the safety distance s, starts braking with the force $b > 0$. However the check if the train is within the safety distance is done only every ε seconds.

$$\begin{array}{lll} c_{\mathsf{chkAndBrake}} & :\equiv & \left[\mathtt{while}\ v > 0\ \mathtt{do}\ (\,c_{\mathsf{corr}};\ c_{\mathsf{accel}}\,)\right] \qquad \text{where} \\ c_{\mathsf{corr}} & :\equiv & \left[t := 0\,;\ \mathtt{if}\ m - z < s\ \mathtt{then}\ a := -b\ \mathtt{else}\ a := 0\right] \end{array} \tag{2}$$

Example 3.4 (Water-level monitor) Our second example is an adaptation from [1]. Compared to the above train example, it exhibits simpler flow-dynamics (the first derivative is already a constant) but more complex jump-dynamics.

There is a water tank with a constant drain (2 cm per second). When the water level y gets lower than 5 cm the switch is turned on, which eventually opens up the valve but only after a time lag of two seconds. While the valve is open, the water level y rises by 1 cm per second. Once y reaches 10 cm the switch is turned off, which will shut the valve but again after a time lag of two seconds.



In the following command c_{water}, the variable x is a timer for a time lag. The `case` construct is an obvious abbreviation of nested `if ... then ... else ...`.

$$c_{\mathsf{water}} \;:\equiv\; \left[\begin{array}{l} x := 0;\ y := 1;\ s := 1;\ v := 1; \\ \mathtt{while}\ t < t_{\mathsf{max}}\ \mathtt{do}\quad \{ \\ \quad x := x + \mathtt{dt};\quad t := t + \mathtt{dt}; \\ \quad \mathtt{if}\ v = 0\ \mathtt{then}\ y := y - 2\cdot\mathtt{dt}\ \mathtt{else}\ y := y + \mathtt{dt}; \\ \quad \mathtt{case}\quad \{ \begin{array}{ll} s = 0 \wedge v = 0 \wedge y \le 5: & s := 1;\ x := 0; \\ s = 1 \wedge v = 0 \wedge x \ge 2: & v := 1; \\ s = 1 \wedge v = 1 \wedge 10 \le y: & s := 0;\ x := 0; \\ s = 0 \wedge v = 1 \wedge x \ge 2: & v := 0; \\ \mathtt{else} & \mathtt{skip}\quad \}\} \end{array} \end{array}\right] \tag{3}$$

3.2 Denotational Semantics

We follow [10] and interpret a command of $\textsc{While}^{\mathsf{dt}}$ as a transformer on memory states. Our state stores hyperreal numbers such as the infinitesimal $\omega^{-1} = [\,(1, \frac{1}{2}, \frac{1}{3}, \dots)\,]$, hence is called a *hyperstate*.

Definition 3.5 (Hyperstate, state) A *hyperstate* $\boldsymbol{\sigma}$ is either $\boldsymbol{\sigma} = \bot$ ("undefined") or a function $\boldsymbol{\sigma} : \mathbf{Var} \to {}^*\mathbb{R}$. A *state* is a standard version of a hyperstate: namely, a *state* σ is either $\sigma = \bot$ or a function $\sigma : \mathbf{Var} \to \mathbb{R}$.

We denote the collection of hyperstates by $\mathbf{HSt}$; that of (standard) states by $\mathbf{St}$.

The definition of (hyper)state as a total function—rather than a partial function with a finite domain—follows [10]. This makes the denotational semantics much simpler. Practically, one can imagine there is a fixed default value (say 0) for any variable.

The following definition is as usual.

Definition 3.6 (State update) Let $\boldsymbol{\sigma} \in \mathbf{HSt}$ be a hyperstate, $x \in \mathbf{Var}$ and $\boldsymbol{r} \in {}^*\mathbb{R}$. We define an *updated hyperstate* $\boldsymbol{\sigma}[x \mapsto \boldsymbol{r}]$ as follows. When $\boldsymbol{\sigma} = \bot$, we set $\bot[x \mapsto \boldsymbol{r}] := \bot$. Otherwise: $\big(\boldsymbol{\sigma}[x \mapsto \boldsymbol{r}]\big)(x) := \boldsymbol{r}$; and for $y \neq x$, $\big(\boldsymbol{\sigma}[x \mapsto \boldsymbol{r}]\big)(y) := \boldsymbol{\sigma}(y)$.

An *updated (standard) state* $\sigma[x \mapsto r]$ is defined analogously.

Definition 3.7 (Sequence representation) Let $(\sigma_i)_{i\in\mathbb{N}}$ be a sequence of (standard) states. It gives rise to a hyperstate—denoted by $[(\sigma_i)_{i\in\mathbb{N}}]$ or simply by $(\sigma_i)_{i\in\mathbb{N}}$—in the following way. We set $(\sigma_i)_{i\in\mathbb{N}} := \bot$ if $\sigma_i = \bot$ for almost all i. Otherwise $[\,(\sigma_i)_{i\in\mathbb{N}}\,] \neq \bot$ and we set $\big[\,(\sigma_i)_{i\in\mathbb{N}}\,\big](x) := \big[\,(\sigma_i(x))_{i\in\mathbb{N}}\,\big]$, where the latter is the hyperreal represented by the sequence $(\sigma_i(x))_i$ of reals. For $i \in \mathbb{N}$ such that $\sigma_i = \bot$, the value $\sigma_i(x)$ is not defined; in this case we use an arbitrary real number (say 0) for $\sigma_i(x)$. This does not affect the resulting hyperstate since $\sigma_i(x)$ is defined for almost all i.

Let $\boldsymbol{\sigma} \in \mathbf{HSt}$ be a hyperstate, and $(\sigma_i)_{i\in\mathbb{N}}$ be a sequence of states. We say $(\sigma_i)_{i\in\mathbb{N}}$ is a *sequence representation* of $\boldsymbol{\sigma}$ if it gives rise to $\boldsymbol{\sigma}$, that is, $\big[(\sigma_i)_{i\in\mathbb{N}}\big] = \boldsymbol{\sigma}$. In what follows we shall often denote a sequence representation of $\boldsymbol{\sigma}$ by $(\boldsymbol{\sigma}|_i)_{i\in\mathbb{N}}$. We emphasize that given $\boldsymbol{\sigma} \in \mathbf{HSt}$, its sequence representation $(\boldsymbol{\sigma}|_i)_i$ is not unique.

The denotational semantics of WHILE$^{\mathrm{dt}}$ is a straightforward adaptation of the usual semantics of WHILE, except for the `while` clauses where we use sectionwise execution (see Ex. 1.1). As we see later in Lem. 3.10, however, the idea of sectionwise execution extends to the whole language WHILE$^{\mathrm{dt}}$.

Definition 3.8 (Denotational semantics for WHILE$^{\mathrm{dt}}$) For expressions of WHILE$^{\mathrm{dt}}$, their *denotation*

$$\begin{array}{lll} [\![a]\!] : & \mathbf{HSt} \longrightarrow {}^*\mathbb{R} \cup \{\bot\} & \text{for } a \in \mathbf{AExp}, \\ [\![b]\!] : & \mathbf{HSt} \longrightarrow \mathbb{B} \cup \{\bot\} & \text{for } b \in \mathbf{BExp}, \text{ and} \\ [\![c]\!] : & \mathbf{HSt} \longrightarrow \mathbf{HSt} & \text{for } c \in \mathbf{Cmd} \end{array}$$

is defined as follows. Recall that $\bot$ means "undefined" (cf. Def. 3.5); that $\mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$ is the set of Boolean truth values; and that ${}^*\mathbb{B} \cong \mathbb{B}$ (Lem. 2.6).

If $\boldsymbol{\sigma} = \bot$, we define $[\![e]\!]\bot := \bot$ for any expression e. If $\boldsymbol{\sigma} \neq \bot$ we define

$$\begin{array}{lll} [\![x]\!]\boldsymbol{\sigma} & := \boldsymbol{\sigma}(x) & [\![\mathtt{c}_r]\!]\boldsymbol{\sigma} := r \quad \text{for each } r \in \mathbb{R} \\ [\![a_1 \;\mathtt{aop}\; a_2]\!]\boldsymbol{\sigma} & := [\![a_1]\!]\boldsymbol{\sigma} \;\mathtt{aop}\; [\![a_2]\!]\boldsymbol{\sigma} & \\ [\![\mathtt{dt}]\!]\boldsymbol{\sigma} & := \omega^{-1} = \big[\,(1, \tfrac{1}{2}, \tfrac{1}{3}, \dots)\,\big] & [\![\infty]\!]\boldsymbol{\sigma} := \omega = \big[\,(1, 2, 3, \dots)\,\big] \end{array}$$

$$\begin{array}{ll} [\![\mathtt{true}]\!]\boldsymbol{\sigma} := \mathrm{tt} & [\![\mathtt{false}]\!]\boldsymbol{\sigma} := \mathrm{ff} \\ [\![b_1 \wedge b_2]\!]\boldsymbol{\sigma} := [\![b_1]\!]\boldsymbol{\sigma} \wedge [\![b_2]\!]\boldsymbol{\sigma} & [\![\neg b]\!]\boldsymbol{\sigma} := \neg([\![b]\!]\boldsymbol{\sigma}) \\ [\![a_1 < a_2]\!]\boldsymbol{\sigma} := [\![a_1]\!]\boldsymbol{\sigma} < [\![a_2]\!]\boldsymbol{\sigma} & \end{array}$$

$$[\![\mathtt{skip}]\!]\boldsymbol{\sigma} := \boldsymbol{\sigma} \qquad [\![x := a]\!]\boldsymbol{\sigma} := \boldsymbol{\sigma}\big[\,x \mapsto [\![a]\!]\boldsymbol{\sigma}\,\big] \qquad [\![c_1; c_2]\!]\boldsymbol{\sigma} := [\![c_2]\!]\big(\,[\![c_1]\!]\boldsymbol{\sigma}\,\big)$$

$$[\![\mathtt{if}\; b \;\mathtt{then}\; c_1 \;\mathtt{else}\; c_2]\!]\boldsymbol{\sigma} := \begin{cases} [\![c_1]\!]\boldsymbol{\sigma} & \text{if } [\![b]\!]\boldsymbol{\sigma} = \mathrm{tt} \\ [\![c_2]\!]\boldsymbol{\sigma} & \text{if } [\![b]\!]\boldsymbol{\sigma} = \mathrm{ff} \end{cases}$$

$$\begin{array}{l} [\![\mathtt{while}\; b \;\mathtt{do}\; c]\!]\boldsymbol{\sigma} := \Big(\, \big[\!\big[\, (\mathtt{while}\; b \;\mathtt{do}\; c)|_i \,\big]\!\big] (\boldsymbol{\sigma}|_i) \Big)_{i \in \mathbb{N}} \;, \\ \text{where } (\boldsymbol{\sigma}|_i)_{i \in \mathbb{N}} \text{ is an arbitrary sequence representation of } \boldsymbol{\sigma} \text{ (Def. 3.7)} \end{array} \tag{4}$$

Here $\mathtt{aop} \in \{+, -, \times, \hat{}\,\}$ and $<$ are interpreted on ${}^*\mathbb{R}$ as in Def. 2.4. For each $e \in \mathbf{AExp} \cup \mathbf{BExp}$, we obviously have $[\![e]\!]\boldsymbol{\sigma} = \bot$ if and only if $\boldsymbol{\sigma} = \bot$. It may happen that $[\![c]\!]\boldsymbol{\sigma} = \bot$ with $\boldsymbol{\sigma} \neq \bot$, due to nontermination of `while` loops.

In the semantics of `while` clauses (4), the section $(\mathtt{while}\; b \;\mathtt{do}\; c)|_i$ is a command of WHILE (Def. 3.2); and $\boldsymbol{\sigma}|_i$ is a (standard) state. Thus the state $\big[\!\big[\, (\mathtt{while}\; b \;\mathtt{do}\; c)|_i \,\big]\!\big] (\boldsymbol{\sigma}|_i)$ can be defined by the usual semantics of `while` constructs (see e.g. [10]). That is,

$$\begin{array}{l} [\![\mathtt{while}\; b' \;\mathtt{do}\; c']\!]\sigma = \sigma' \qquad \overset{\text{def.}}{\Longleftrightarrow} \\ \left\{ \begin{array}{l} \text{– } \sigma = \sigma' = \bot; \\ \text{– there exists a finite sequence } \sigma = \sigma_0, \sigma_1, \dots, \sigma_n = \sigma' \text{ such that: } [\![b']\!]\sigma_n = \\ \quad \mathrm{ff}; \text{ and for each } j \in [0, n).\ \big(\,[\![b']\!]\sigma_j = \mathrm{tt} \;\&\; [\![c']\!]\sigma_j = \sigma_{j+1}\,\big); \text{ or} \\ \text{– such a finite sequence does not exist and } \sigma' = \bot. \end{array} \right. \end{array} \tag{5}$$

By bundling these up for all i, and regarding it as a hyperstate (Def. 3.7), we obtain the right-hand side of (4). The well-definedness of (4) is proved in Lem. 3.9.

Lemma 3.9 *The semantics of* `while` *clauses (4) is well-defined, being independent of the choice of a sequence representation* $(\boldsymbol{\sigma}|_i)_i$ *of the hyperstate* $\boldsymbol{\sigma}$. □

In proving the lemma it is crucial that: the set $\{x_1, \ldots, x_n\}$ of variables that are relevant to the execution of the command is finite and statically known. This would not be the case with a programming language that allows dynamical creation of fresh variables.

We have chosen not to include the division operator / in $\text{WHILE}^{\mathsf{dt}}$; this is to avoid handling of *division by zero* in the semantics, which is cumbersome but seems feasible.

Here is one of our two key lemmas. Its proof is by induction.

Lemma 3.10 (Sectionwise Execution Lemma) *Let e be any expression of $\text{WHILE}^{\mathsf{dt}}$; $\boldsymbol{\sigma}$ be a hyperstate; and $(\boldsymbol{\sigma}|_i)_{i\in\mathbb{N}}$ be an arbitrary sequence representation of $\boldsymbol{\sigma}$. We have*

$$[\![e]\!]\boldsymbol{\sigma} = \left[\,\left(\,[\![e|_i]\!](\boldsymbol{\sigma}|_i)\,\right)_{i\in\mathbb{N}}\right] \ .$$

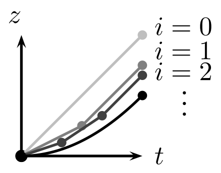
Here the denotational semantics $[\![e|_i]\!]$ of a WHILE *expression $e|_i$ is defined in a usual way (i.e. like in Def. 3.8; for* `while` *clauses see (5)).* □

Example 3.11 Consider c_{accel} in Ex. 3.3. For simplicity we fix the parameters: $c_{\mathsf{accel1}} :\equiv [\,t := 0;\ \varepsilon := 1;\ a := 1;\ v := 0;\ z := 0;\ c_{\mathsf{accel}}\,]$. Its i-th section $c_{\mathsf{accel1}}|_i$ has the obvious semantics. For any (standard) state $\sigma \neq \bot$, the real number $([\![c_{\mathsf{accel1}}|_i]\!]\sigma)(z)$—the traveled distance—is easily calculated as

$$\tfrac{1}{i+1}\cdot\tfrac{1}{i+1} + \tfrac{1}{i+1}\cdot\tfrac{2}{i+1} + \cdots + \tfrac{1}{i+1}\cdot\tfrac{i+1}{i+1} = \tfrac{(i+1)(i+2)}{2(i+1)^2} = \tfrac{1}{2}\cdot\tfrac{i+2}{i+1} \ .$$

Therefore by (4), for any hyperstate $\boldsymbol{\sigma} \neq \bot$, the hyperreal $([\![c_{\mathsf{accel1}}]\!]\boldsymbol{\sigma})(z)$ is equal to

$$\left[\,\left(1,\ \tfrac{3}{4},\ \tfrac{2}{3},\ \tfrac{5}{8},\ \ldots,\ \tfrac{1}{2}\cdot\tfrac{i+2}{i+1},\ \ldots\right)\right] \ ;$$



this is a hyperreal that is infinitely close to $1/2$.

Much like Ex. 1.1, one way to look at this sectionwise semantics is as an incremental approximation. Here it approximates the solution $z = \frac{1}{2}t^2$ of the differential equation $z'' = 1$, obtained via the Riemann integral. See the above figure.

Remark 3.12 (Denotation of `dt`) We fixed ω^{-1} as the denotation of `dt`. However there are more infinitesimals, such as $(\pi\omega)^{-1} = (\frac{1}{\pi}, \frac{1}{2\pi}, \frac{1}{3\pi}, \ldots)$ with $(\pi\omega)^{-1} < \omega^{-1}$. The choice of `dt`'s denotation does affect the behavior of the following program $c_{\mathsf{nonintegrable}}$:

$$c_{\mathsf{nonintegrable}} \quad :\equiv \quad \left[\ x := 1;\quad \texttt{while}\ x \neq 0\ \texttt{do}\ x := x - \texttt{dt}\ \right] \ .$$

When we replace `dt` by $\frac{1}{i+1}$ the program terminates with $x = 0$; hence by our semantics the program yields a non-$\bot$ hyperstate with $x \mapsto 0$. However, replacing `dt` by $\frac{1}{(i+1)\pi}$ with π irrational, the program terminates for no i and it leads to the hyperstate $\bot$.

In fact, indifference to the choice of an infinitesimal value (violated by $c_{\mathsf{nonintegrable}}$) is a typical condition in nonstandard analysis, found e.g. in the characterization of differentiability or Riemann integrability (see [4]). In this sense the program $c_{\mathsf{nonintegrable}}$ is “nonintegrable”; we are yet to see if we can check integrability by syntactic means.

The program $c_{\mathsf{nonintegrable}}$ can be modified into the one with more reasonable behavior, by replacing the guard $x \neq 0$ by $x > 0$. One easily sees that, while different choices of `dt`'s denotation (e.g. ω^{-1} vs. $(\pi\omega)^{-1}$) still lead to different post-hyperstates, the differences lie within infinitesimal gaps. The same is true of all the “realistic” programs that we have looked at.

4 Assertion Language $\textsc{Assn}^{\mathtt{dt}}$

Definition 4.1 ($\textsc{Assn}^{\mathtt{dt}}$, $\textsc{Assn}$) The syntax of our assertion language $\textsc{Assn}^{\mathtt{dt}}$ is:

$$\begin{array}{llll}
\mathbf{AExp} \ni & a & ::= & x \mid \mathtt{c}_r \mid a_1 \,\mathtt{aop}\, a_2 \mid \mathtt{dt} \mid \infty \qquad \text{(the same as in } \textsc{While}^{\mathtt{dt}}\text{)} \\
\mathbf{Fml} \ni & A & ::= & \mathtt{true} \mid \mathtt{false} \mid A_1 \wedge A_2 \mid \neg A \mid a_1 < a_2 \mid \\
 & & & \forall x \in {}^*\mathbb{N}.\, A \mid \forall x \in {}^*\mathbb{R}.\, A \qquad \text{where } x \in \mathbf{Var}
\end{array}$$

An expression in the family $\mathbf{Fml}$ is called an *(assertion) formula*.

We introduce existential quantifiers as notational conventions: $\exists x \in {}^*\mathbb{D}.\, A :\equiv \neg\forall x \in {}^*\mathbb{D}.\, \neg A$, where $\mathbb{D} \in \{\mathbb{N}, \mathbb{R}\}$.

By $\textsc{Assn}$ we designate the language obtained from $\textsc{Assn}^{\mathtt{dt}}$ by: 1) dropping the constants $\mathtt{dt}, \infty$; and 2) replacing the quantifiers $\forall x \in {}^*\mathbb{N}$ and $\forall x \in {}^*\mathbb{R}$ by $\forall x \in \mathbb{N}$ and $\forall x \in \mathbb{R}$, respectively, i.e. by those which range over standard numbers.

Formulas of $\textsc{Assn}^{\mathtt{dt}}$ are the Boolean expressions of $\textsc{While}^{\mathtt{dt}}$, augmented with quantifiers. The quantifier $\forall x \in {}^*\mathbb{N}$ ranging over hyper-*natural* numbers plays an important role in relative completeness of $\textsc{Hoare}^{\mathtt{dt}}$ (Thm. 5.4).

It is essential that in $\textsc{Assn}^{\mathtt{dt}}$ we have only *hyper*quantifiers like $\forall x \in {}^*\mathbb{R}$ and not *standard* quantifiers like $\forall x \in \mathbb{R}$. The situation is much like with the celebrated *transfer principle* in nonstandard analysis [4, Thm. II.4.5]. There the validity of a standard formula φ is transferred to that of its $*$-transform ${}^*\varphi$; and in ${}^*\varphi$ only hyperquantifiers, and no standard quantifiers, are allowed to occur.

Remark 4.2 (Absence of standard quantifiers) The lack of standard quantifiers does restrict the expressive power of $\textsc{Assn}^{\mathtt{dt}}$. Notably we cannot assert that two hypernumbers x, y are infinitely close, that is, $\forall \varepsilon \in \mathbb{R}.\, (\varepsilon > 0 \Rightarrow -\varepsilon < x - y < \varepsilon)$.[3] However this assertion is arguably unrealistic since, to check it against a physical system, one needs measurements of arbitrarily progressive accuracy. The examples in §6 indicate that $\textsc{Assn}^{\mathtt{dt}}$ is sufficiently expressive for practical verification scenarios, too.

Definition 4.3 (Section of $\textsc{Assn}^{\mathtt{dt}}$ expression) Let e be an expression of $\textsc{Assn}^{\mathtt{dt}}$ (arithmetic or a formula), and $i \in \mathbb{N}$. The *i-th section* of e, denoted by $e|_i$, is obtained by: 1) replacing every occurrence of $\mathtt{dt}$ and ∞ by the constant $\mathtt{c}_{1/(i+1)}$ and $\mathtt{c}_{i+1}$, respectively; and 2) replacing every hyperquantifier $\forall x \in {}^*\mathbb{D}$ by $\forall x \in \mathbb{D}$. Here $\mathbb{D} \in \{\mathbb{N}, \mathbb{R}\}$.

Obviously a section $e|_i$ is an expression of $\textsc{Assn}$.

Definition 4.4 (Semantics of $\textsc{Assn}^{\mathtt{dt}}$) We define the relation $\boldsymbol{\sigma} \models A$ ("$\boldsymbol{\sigma}$ satisfies A") between a hyperstate $\boldsymbol{\sigma} \in \mathbf{HSt}$ and an $\textsc{Assn}^{\mathtt{dt}}$ formula $A \in \mathbf{Fml}$ as usual.

[3] By replacing $\forall \varepsilon \in \mathbb{R}$ by $\forall \varepsilon \in {}^*\mathbb{R}$ we obtain a legitimate $\textsc{Assn}^{\mathtt{dt}}$ formula, but it is satisfied only when the two hypernumbers x, y are equal.

Namely, if $\boldsymbol{\sigma} = \bot$ we define $\bot \models A$ for each $A \in \mathbf{Fml}$. If $\boldsymbol{\sigma} \neq \bot$, the definition is by the following induction on the construction of A.

$$\begin{array}{lll}
\boldsymbol{\sigma} \models \mathtt{true} & \boldsymbol{\sigma} \not\models \mathtt{false} & \\
\boldsymbol{\sigma} \models A_1 \wedge A_2 & \overset{\text{def.}}{\Longleftrightarrow} & \boldsymbol{\sigma} \models A_1 \ \&\ \boldsymbol{\sigma} \models A_2 \\
\boldsymbol{\sigma} \models \neg A & \overset{\text{def.}}{\Longleftrightarrow} & \boldsymbol{\sigma} \not\models A \\
\boldsymbol{\sigma} \models a_1 < a_2 & \overset{\text{def.}}{\Longleftrightarrow} & [\![a_1]\!]\boldsymbol{\sigma} < [\![a_2]\!]\boldsymbol{\sigma} \qquad \text{where } [\![a_i]\!]\boldsymbol{\sigma} \text{ is as defined in Def. 3.8} \\
\boldsymbol{\sigma} \models \forall x \in {}^*\mathbb{D}.\, A & \overset{\text{def.}}{\Longleftrightarrow} & \boldsymbol{\sigma}[x \mapsto \boldsymbol{d}] \models A \quad \text{for each } \boldsymbol{d} \in {}^*\mathbb{D} \qquad (\mathbb{D} \in \{\mathbb{N}, \mathbb{R}\})
\end{array}$$

Recall that $\boldsymbol{\sigma}[x \mapsto \boldsymbol{d}]$ denotes an updated hyperstate (Def. 3.6).

An ASSN$^{\mathsf{dt}}$ formula $A \in \mathbf{Fml}$ is said to be *valid* if $\boldsymbol{\sigma} \models A$ for any $\boldsymbol{\sigma} \in \mathbf{HSt}$. We denote this by $\models A$. Validity of an ASSN formula is defined similarly.

Lemma 4.5 (Sectionwise Satisfaction Lemma) *Let $A \in \mathbf{Fml}$ be an* ASSN$^{\mathsf{dt}}$ *formula; $\boldsymbol{\sigma}$ be a hyperstate; and $(\boldsymbol{\sigma}|_i)_{i \in \mathbb{N}}$ be an arbitrary sequence representation of $\boldsymbol{\sigma}$. We have*

$$\boldsymbol{\sigma} \models A \quad \textit{if and only if} \quad \big(\boldsymbol{\sigma}|_i \models A|_i \quad \textit{for almost every } i\,\big) \ ,$$

where the latter relation $\models$ between standard states and ASSN *formulas is defined in the usual way (i.e. like in Def. 4.4).* □

This is our second key lemma. We note that it fails once we allow standard quantifiers in ASSN$^{\mathsf{dt}}$. For example, let $A :\equiv (\exists y \in \mathbb{R}.\, 0 < y < x)$ and $\boldsymbol{\sigma}$ be a hyperstate such that $\boldsymbol{\sigma}(x) = \omega^{-1}$. Then we have $\boldsymbol{\sigma}|_i \models A|_i$ for every i but $\boldsymbol{\sigma} \not\models A$.

The validity of an ASSN$^{\mathsf{dt}}$ formula A, if A is ($\mathtt{dt}$, ∞)-free, can be reduced to that of an ASSN formula. This is the *transfer principle* for ASSN$^{\mathsf{dt}}$ which we now describe.

Definition 4.6 ($*$-transform) Let A be an ASSN formula. We define its *$*$-transform*, denoted by ${}^*A$, to be the ASSN$^{\mathsf{dt}}$ formula obtained from A by replacing every occurrence of a standard quantifier $\forall x \in \mathbb{D}$ by the corresponding hyperquantifier $\forall x \in {}^*\mathbb{D}$.

It is easy to see that: 1) $({}^*A)|_i \equiv A$ for each ASSN formula A; 2) $A \equiv {}^*(A|_i)$ for each ASSN$^{\mathsf{dt}}$ formula A that is *($\mathtt{dt}$, ∞)-free*—that is, $\mathtt{dt}$ or ∞ does not occur in it. Then the following is an immediate consequence of Lem. 4.5.

Proposition 4.7 (Transfer principle) *1. For each* ASSN *formula A, $\models A$ iff $\models {}^*A$.*
2. *For any ($\mathtt{dt}$,∞)-free* ASSN$^{\mathsf{dt}}$ *formula A, the following are equivalent: a) $\models A|_i$ for each $i \in \mathbb{N}$; b) $\models A|_i$ for some $i \in \mathbb{N}$; c) $\models A$.* □

5 Program Logic HOARE$^{\mathsf{dt}}$

We now introduce a Hoare-style program logic HOARE$^{\mathsf{dt}}$ that is devised for the verification of WHILE$^{\mathsf{dt}}$ programs. It derives *Hoare triples* $\{A\}c\{B\}$.

Definition 5.1 (Hoare triple) A *Hoare triple* $\{A\}c\{B\}$ of HOARE$^{\mathsf{dt}}$ is a triple of ASSN$^{\mathsf{dt}}$ formulas A, B and a WHILE$^{\mathsf{dt}}$ command c.

A Hoare triple $\{A\}c\{B\}$ is said to be *valid*—we denote this by $\models \{A\}c\{B\}$—if, for any hyperstate $\boldsymbol{\sigma} \in \mathbf{HSt}$, $\boldsymbol{\sigma} \models A$ implies $[\![c]\!]\boldsymbol{\sigma} \models B$.

As usual a Hoare triple $\{A\}c\{B\}$ asserts *partial correctness*: if the execution of c starting from $\boldsymbol{\sigma}$ does not terminate, we have $[\![c]\!]\boldsymbol{\sigma} = \bot$ hence trivially $[\![c]\!]\boldsymbol{\sigma} \models B$. The formula A in $\{A\}c\{B\}$ is called a *precondition*; B is a *postcondition*.

The rules of HOARE$^{\mathsf{dt}}$ are the same as usual; see e.g. [10].

Definition 5.2 (HOARE$^{\mathsf{dt}}$) The deduction rules of HOARE$^{\mathsf{dt}}$ are as follows.

$$\frac{}{\{A\}\,\texttt{skip}\,\{A\}}\ (\textsc{Skip}) \qquad \frac{}{\{\,A[a/x]\,\}\ x := a\,\{A\}}\ (\textsc{Assign})$$

$$\frac{\{A\}\,c_1\,\{C\} \quad \{C\}\,c_2\,\{B\}}{\{A\}\,c_1; c_2\,\{B\}}\ (\textsc{Seq}) \qquad \frac{\{A \wedge b\}\,c_1\,\{B\} \quad \{A \wedge \neg b\}\,c_2\,\{B\}}{\{A\}\,\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\,\{B\}}\ (\textsc{If})$$

$$\frac{\{A \wedge b\}\,c\,\{A\}}{\{A\}\,\texttt{while}\ b\ \texttt{do}\ c\,\{A \wedge \neg b\}}\ (\textsc{While}) \qquad \frac{\models A \Rightarrow A' \quad \{A'\}\,c\,\{B'\} \quad \models B' \Rightarrow B}{\{A\}\,c\,\{B\}}\ (\textsc{Conseq})$$

In the rule (ASSIGN), $A[a/x]$ denotes the capture-avoiding substitution of a for x in A. Recall that $\mathbf{BExp}$ of WHILE$^{\mathsf{dt}}$ is a fragment of $\mathbf{Fml}$ of ASSN$^{\mathsf{dt}}$. Therefore in the rules (IF) and (WHILE), an expression b is an ASSN$^{\mathsf{dt}}$ formula.

We write $\vdash \{A\}c\{B\}$ if the triple $\{A\}c\{B\}$ can be derived using the above rules.

Soundness is a minimal requirement of a logic for verification. The proof makes an essential use of the key "sectionwise" lemmas (Lem. 3.10 and Lem. 4.5).

Theorem 5.3 (Soundness) $\vdash \{A\}c\{B\}$ *implies* $\models \{A\}c\{B\}$. □

We also have a "completeness" result. It is called *relative completeness* [3] since completeness is only modulo the validity of ASSN$^{\mathsf{dt}}$ formulas (namely those in the (CONSEQ) rule); and checking such validity is easily seen to be undecidable. The proof follows the usual method (see e.g. [10, Chap. 7]); namely via explicit description of weakest preconditions.

Theorem 5.4 (Relative completeness) $\models \{A\}c\{B\}$ *implies* $\vdash \{A\}c\{B\}$. □

6 Verification with HOARE$^{\mathsf{dt}}$

We present a couple of examples. Its details as well as some lemmas that aid finding loop invariants will be presented in another venue, due to the lack of space.

Example 6.1 (Water-level monitor) For the program c_{water} in Ex. 3.4, we would like to prove that the water level y stays between 1 cm and 12 cm. It is not hard to see, after some trials, that what we can actually prove is: $\vdash \{\texttt{true}\}c_{\mathsf{water}}\{1 - 4 \cdot \texttt{dt} < y < 12 + 2 \cdot \texttt{dt}\}$. Note that the additional infinitesimal gaps like $4 \cdot \texttt{dt}$ have no physical meaning. In the proof, we use the following formula A as a loop invariant.

$$\begin{array}{lcl}
A & :\equiv & A_s \ \wedge\ A_0 \ \wedge\ A_1 \ \wedge\ A_2 \ \wedge\ A_3 \\
A_s & :\equiv & (s = 0 \vee s = 1) \ \wedge\ (v = 0 \vee v = 1) \\
A_0 & :\equiv & s = 1 \ \wedge\ v = 1 \quad \Rightarrow \quad 1 - 4 \cdot \texttt{dt} < y < 10 \\
A_1 & :\equiv & s = 0 \ \wedge\ v = 1 \quad \Rightarrow \quad 0 \le x < 2 \quad \wedge \quad 10 \le y < 10 + x + \texttt{dt} \\
A_2 & :\equiv & s = 0 \ \wedge\ v = 0 \quad \Rightarrow \quad 5 < y < 12 + 2 \cdot \texttt{dt} \\
A_3 & :\equiv & s = 1 \ \wedge\ v = 0 \quad \Rightarrow \quad 0 \le x < 2 \quad \wedge \quad 5 - 2x - 2 \cdot \texttt{dt} < y \le 5
\end{array}$$

Example 6.2 (Train control) Take the program $c_{\mathsf{chkAndBrake}}$ in Ex. 6.2; we aim at the postcondition that the train does not travel beyond the boundary m, that is, $z \leq m$. For simplicity let us first consider $c_{\mathsf{constChkAndBrake}} :\equiv (\varepsilon := \mathtt{dt}\,;\ c_{\mathsf{chkAndBrake}})$. This is the setting where the check is conducted constantly. Indeed we can prove that $\vdash \{v^2 \leq 2b(z-m)\}c_{\mathsf{constChkAndBrake}}\{z \leq m\}$, with a loop invariant $v^2 \leq 2b(z-m)$.

The invariant (and the precondition) $v^2 \leq 2b(z-m)$ is what is derived in [5] by solving a differential equation and then eliminating quantifiers. Using HOARE$^{\mathtt{dt}}$ we can also derive it: roughly speaking, a differential equation in [5] becomes a recurrence relation in our NSA framework. The details and some general lemmas that aid invariant generation are deferred to another venue.

In the general case where $\varepsilon > 0$ is arbitrary, we can prove $\vdash \{v^2 \leq 2b(z-m-v\cdot\varepsilon)\}c_{\mathsf{chkAndBrake}}\{z \leq m\}$ in HOARE$^{\mathtt{dt}}$.

An obvious challenge in verification with HOARE$^{\mathtt{dt}}$ is finding loop invariants. It is tempting—especially with "flow-heavy" systems, i.e. those with predominant flow-dynamics—to assert a differential equation's solution as a loop invariant. This does not work: it is a loop invariant only modulo infinitesimal gaps, a fact not expressible in ASSN$^{\mathtt{dt}}$ (Rem. 4.2). We do not consider this as a serious drawback, for two reasons. Firstly, such "flow-heavy" systems could be studied, after all, from the control theory perspective that is continuous in its origin. The formal verification approach is supposed to show its strength against "jump-heavy" systems, for which differential equations are hardly solvable. Secondly, verification goals are rarely as precise as the solution of a differential equation: we would aim at $z \leq m$ in Ex. 6.2 but not at $z = \frac{1}{2}at^2$.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comp. Sci. 138(1), 3–34 (1995)
2. Bliudze, S., Krob, D.: Modelling of complex systems: Systems as dataflow machines. Fundam. Inform. 91(2), 251–274 (2009)
3. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Journ. Comput. 7(1), 70–90 (1978)
4. Hurd, A.E., Loeb, P.A.: An Introduction to Nonstandard Real Analysis. Academic Press (1985)
5. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reasoning 41(2), 143–189 (2008)
6. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. 20(1), 309–352 (2010)
7. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. Lect. Notes Comp. Sci., vol. 5195, pp. 171–178. Springer (2008)
8. Rust, H.: Operational Semantics for Timed Systems: A Non-standard Approach to Uniform Modeling of Timed and Hybrid Systems, Lect. Notes Comp. Sci., vol. 3456. Springer (2005)
9. Suenaga, K., Hasuo, I.: Programming with infinitesimals: A WHILE-language for hybrid system modeling. Extended version with proofs, available online (April 2011)
10. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1993)