Exercises in *Nonstandard Static Analysis* of Hybrid Systems

Ichiro Hasuo[1] and Kohei Suenaga[2]

[1] University of Tokyo, Japan
[2] Kyoto University, Japan

Abstract. In formal verification of hybrid systems, a big challenge is to incorporate continuous *flow* dynamics in a discrete framework. Our previous work proposed to use *nonstandard analysis (NSA)* as a vehicle from discrete to hybrid; and to verify hybrid systems using a Hoare logic. In this paper we aim to exemplify the potential of our approach, through transferring *static analysis* techniques to hybrid applications. The transfer is routine via the *transfer principle* in NSA. The techniques are implemented in our prototype automatic precondition generator.

1 Introduction

Hybrid systems exhibit both discrete (digital) *jump* and continuous (physical) *flow* dynamics. They are of paramount importance now that so many physical systems—cars, airplanes, etc.—are controlled with computers. For an approach to hybrid systems from the formal verification community, 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 uses *differential equations* explicitly in the syntax; see the discussion of related work below.

In [21], instead, we proposed to introduce a constant $\mathtt{dt}$ for an *infinitesimal* (i.e. infinitely small) value, and *turn flow into jump*. With $\mathtt{dt}$, the continuous operation of integration can be represented by a while-loop, to which existing discrete techniques such as Hoare-style program logics readily apply. For a rigorous mathematical development we employed *nonstandard analysis (NSA)* beautifully formalized by Robinson [15].

Concretely, in [21] we took the common triple of a WHILE-language, a first-order assertion language and a Hoare logic (e.g. in the textbook [22]); and added a constant $\mathtt{dt}$ to obtain a modeling and verification framework for hybrid systems. Its three components are called $\text{WHILE}^{\mathtt{dt}}$, $\text{ASSN}^{\mathtt{dt}}$ and $\text{HOARE}^{\mathtt{dt}}$. These are connected by denotational semantics defined in the language of NSA. We proved soundness and relative completeness of the logic $\text{HOARE}^{\mathtt{dt}}$. 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 program on the right. The value of $\mathtt{dt}$ is infinitesimal; therefore the `while` loop will not terminate within finitely many steps. Nevertheless it is somehow intuitive to expect that after an "execution" of this program, the value of t should be infinitesimally close to 1.

$$
\begin{array}{l}
t := 0\ ; \\
\texttt{while}\ t \leq 1\ \texttt{do} \\
\quad t := t + \mathtt{dt}
\end{array}
$$

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 $c_{\mathsf{elapse}}|_i$ and shown on the right. Concretely, $c_{\mathsf{elapse}}|_i$ is obtained by replacing the infinitesimal $\mathtt{dt}$ in c_{elapse} with $\frac{1}{i+1}$. Informally $c_{\mathsf{elapse}}|_i$ is the "i-th approximation" of the original c_{elapse}.

$$t := 0\ ;\quad \mathtt{while}\ t \leq 1\ \mathtt{do}\quad t := t + \tfrac{1}{i+1}$$

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

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

which is thought of as a progressive approximation of the actual outcome of the original program c_{elapse}. Indeed, in the language of NSA, the sequence (1) represents a *hyperreal number* $\boldsymbol{r}$ that is infinitesimally close to 1.

We note that a program in WHILE$^{\mathtt{dt}}$ is *not* executable in general: the program c_{elapse} executes infinitely many iterations. It is however a merit of *static* approaches to verification, that programs need not be executed to prove their correctness. Instead, well-defined mathematical semantics suffices and supports deductive reasoning. This is what we do, with the denotational semantics of WHILE$^{\mathtt{dt}}$ exemplified in Example 1.1.

This paper is a first step towards serious use of our framework of [21]. We employ various discrete strategies for reasoning about programs—in Hoare-style logics, the concern is mostly *invariant discovery*—and verify small, but nontrivial, examples. Such discrete techniques are actively pursued in program verification, or, in the *static analysis* community. This paper aims to exemplify our framework's potential for transferring static analysis techniques to hybrid applications. In it we rely on nonstandard analysis; hence our venture is called *nonstandard static analysis*. We have implemented a prototype that generates a precondition, given a program and a postcondition.

In what follows we present the strategies for simplification and invariant/precondition discovery that we implemented, and prove their correctness. In fact we only present the strategies' *standard* version—i.e. for WHILE and HOARE, as opposed to the *nonstandard* version that is for WHILE$^{\mathtt{dt}}$ and HOARE$^{\mathtt{dt}}$. This is because the transfer from the former to the latter is routine work. Soundness of the nonstandard version follows from that of the standard one, almost trivially via the *sectionwise lemmas*. We will demonstrate this process using one of the strategies, in §4.2.

A related issue is the legitimacy of use of *Mathematica* for symbolic computation in our prototype. What we prove in *Mathematica* are formulas about real numbers, not hyperreals; and this needs justification. In §5.2 we address this issue and show that proofs in *Mathematica* indeed prove formulas about hyperreal numbers. The key is the *transfer principle*, a celebrated result in NSA (see §2 later).

We defer most of the proofs to the appendix.

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 $\mathtt{dt}$, most of them feature acute distinction between flow- and jump-dynamics.

Model-checking approaches to hybrid systems have been studied for quite a while, with the successful formalism of *hybrid automaton* [1], on the one hand. On the other hand, deductive approaches have seen great advancement through a recent series of work by Platzer and his colleagues (including [12, 14]), resulting in the automated

prover KeYmaera. Our *nonstandard static analysis* approach currently falls much short of theirs in scalability and sophistication. However some of their techniques can be translated into the techniques in our framework; one example is the *differential invariant* strategy (§5.3). Interestingly in [14] it is argued that: being hybrid imposes no additional burden to verification in principle. This concurs with our claim.

Research in static analysis resulted in a huge number of verification techniques—[4, 7, 18] to name just a few. Some of them have been already used for hybrid applications (modeled with explicit differential equations) [16, 17, 19]. Our thesis is that these discrete techniques can be transferred to hybrid applications *as they are*, via NSA. In §4.1 we transfer the *phase split* technique in [20].

The use of NSA as a foundation of hybrid system modeling is not proposed for the first time; see e.g. [3, 5, 8]. Compared to this existing work, we claim our novelty is the use of NSA machinery (notably the transfer principle) in actual, automatic verification.

One recent research program (resulting e.g. in [6]) aims to employ *continuous* techniques—from the theory of dynamical systems—in purely discrete programs and applications. This is opposite to our approach (discrete techniques applied to continuous applications). We believe, however, that the two directions are not contending ones. It is not at all our intention here to champion the superiority of discrete techniques; our point instead is that the collection of available discrete techniques has much wider applicability. It is indeed our future work to *combine* discrete and continuous techniques.

2 Preliminaries

We summarize our previous work [21], focusing on providing intuitions. We denote the syntactic equality by $\equiv$.

Nonstandard Analysis. For detailed expositions see e.g. [9, 11].

We fix an *ultrafilter* $\mathcal{F} \subseteq \mathcal{P}(\mathbb{N})$ that extends the cofinite filter $\mathcal{F}_{\mathrm{c}} := \{S \subseteq \mathbb{N} \mid \mathbb{N} \setminus S \text{ is finite}\}$. Its properties to be noted: 1) for any $S \subseteq \mathbb{N}$, exactly one of S and $\mathbb{N} \setminus S$ belongs to $\mathcal{F}$; 2) if S is *cofinite* (i.e. $\mathbb{N} \setminus S$ is finite), then S belongs to $\mathcal{F}$.

Definition 2.1 (Hypernumber $d \in {}^*\mathbb{D}$) 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

$$\{i \in \mathbb{N} \mid d_i = d'_i\} \in \mathcal{F} \ , \quad \text{for which we say "} d_i = d'_i \text{ for almost every } i\text{."}$$

Therefore, given that two sequences $(d_i)_i$ and $(d'_i)_i$ coincide except for finitely many indices i, they represent the same hypernumber. Other predicates (such as $<$) are defined in the same way. For example a hyperreal $\omega^{-1} := [\,(1, \frac{1}{2}, \frac{1}{3}, \dots)\,]$ is positive ($0 < \omega^{-1}$) but is smaller than any (standard) positive real $r = [(r, r, \dots)]$.

The framework of WHILE$^{\mathtt{dt}}$, ASSN$^{\mathtt{dt}}$ and HOARE$^{\mathtt{dt}}$. We take the standard combination (e.g. in [22]) of a while-style programming language WHILE, a first-order assertion language ASSN and a Hoare-style program logic HOARE (we equip them with constants for all real numbers). Then we augment the framework with a constant $\mathtt{dt}$ that denotes

a specific infinitesimal $\omega^{-1} = [\,(1, \frac{1}{2}, \frac{1}{3}, \dots)\,]$, and obtain the *nonstandard* framework consisting of WHILE$^{\mathtt{dt}}$, ASSN$^{\mathtt{dt}}$ and HOARE$^{\mathtt{dt}}$.

Definition 2.2 (WHILE$^{\mathtt{dt}}$, WHILE, ASSN$^{\mathtt{dt}}$, ASSN) The syntax of WHILE$^{\mathtt{dt}}$ is:

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

Here $\mathbf{AExp}$ is the set of *arithmetic expressions*; $\mathbf{BExp}$ and $\mathbf{Cmd}$ are those of *Boolean* and *command* expressions. The operator $\lceil r \rceil$ denotes the smallest natural number which is larger than or equal to the real number r. We will use this operator in §5.4. The command $\mathtt{assert}\ b$ amounts to $\mathtt{skip}$ if b is true; an infinite loop (divergence) otherwise.

Our first-order assertion language[3] ASSN$^{\mathtt{dt}}$ consists of the following *formulas*.

$$\mathbf{Fml} \ni A ::= \mathtt{true} \mid \mathtt{false} \mid A_1 \wedge A_2 \mid \neg A \mid a_1 < a_2 \mid \forall x \in {}^*\mathbb{N}.\, A \mid \forall x \in {}^*\mathbb{R}.\, A \qquad \text{where } x \in \mathbf{Var} \text{ and } a_i \in \mathbf{AExp}$$

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

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

It is essential that in ASSN$^{\mathtt{dt}}$ we allow only *hyper*quantifiers $\forall x \in {}^*\mathbb{R}$ and not *standard* ones $\forall x \in \mathbb{R}$. This is much like with the *transfer principle* in NSA [11, Thm. II.4.5].

Definition 2.3 (Section $e|_i$) Let e be an expression of WHILE$^{\mathtt{dt}}$ or ASSN$^{\mathtt{dt}}$, and $i \in \mathbb{N}$. The *i-th section* of e, denoted by $e|_i$, is obtained by: 1) replacing every occurrence of $\mathtt{dt}$ with the constant $1/(i+1)$; and 2) replacing every hyperquantifier $\forall x \in {}^*\mathbb{D}$ with $\forall x \in \mathbb{D}$. Here $\mathbb{D} \in \{\mathbb{N}, \mathbb{R}\}$. Obviously a section $e|_i$ is an expression of WHILE or ASSN.

Definition 2.4 (HOARE$^{\mathtt{dt}}$, HOARE) HOARE$^{\mathtt{dt}}$ is a system that derives *Hoare triples* $\{A\}c\{B\}$ (a triple of ASSN$^{\mathtt{dt}}$ formulas A, B and a WHILE$^{\mathtt{dt}}$ command c) using the following rules. The rules are the same for HOARE.

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

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

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

$$\frac{}{\{b \Rightarrow A\}\,\mathtt{assert}\ b\,\{A\}}\ (\textsc{Assert})$$

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

[3] The term *assertion language* has little to do with the command $\mathtt{assert}\ b$ in WHILE$^{\mathtt{dt}}$.

We turn to semantics. Denotational semantics of WHILE$^{\mathtt{dt}}$ is defined following the intuition in Example 1.1. There the semantics $[\![c]\!]$ of a command c is a *hyperstate transformer* that maps a hyperstate $\boldsymbol{\sigma}$ (a state that stores hypernumbers) to a hyperstate $[\![c]\!]\boldsymbol{\sigma}$.

Semantics of an assertion $A \in \mathbf{Fml}$ is defined in the usual way; we write $\models A$ and say A is *valid* if $\boldsymbol{\sigma} \models A$ for each hyperstate $\boldsymbol{\sigma}$. A Hoare triple is *valid* ($\models \{A\}c\{B\}$) if, for any hyperstate $\boldsymbol{\sigma}$ such that $\boldsymbol{\sigma} \models A$, we have $[\![c]\!]\boldsymbol{\sigma} \models B$.

Three *sectionwise lemmas* play central roles in our framework. They correspond to *Łoś's theorem* in NSA. We only present the following one, that is the most relevant to the current paper. Recall the meaning of "for almost every i" via an ultrafilter (§2).

Lemma 2.5 (Sectionwise validity of Hoare triples) *Let A, B be* ASSN$^{\mathtt{dt}}$ *formulas, and $c \in \mathbf{Cmd}$ be a* WHILE$^{\mathtt{dt}}$ *command. We have*

$$\models \{A\}c\{B\} \qquad \Longleftrightarrow \qquad \models \{A|_i\}\, c|_i\, \{B|_i\} \ \textit{for almost every } i. \qquad \square$$

Definition 2.6 ($*$-transform) Let A be an ASSN formula. We define its $*$*-transform*, denoted by ${}^*A$, to be the ASSN$^{\mathtt{dt}}$ formula obtained from A by replacing every occurrence of a standard quantifier $\forall x \in \mathbb{D}$ with the corresponding hyperquantifier $\forall x \in {}^*\mathbb{D}$.

Proposition 2.7 (Transfer principle) *1. For each* ASSN *formula A, $\models A$ iff $\models {}^*A$.*
2. For any `dt`*-free* ASSN$^{\mathtt{dt}}$ *formula A, the following are equivalent: 1) $\models A|_i$ for each $i \in \mathbb{N}$; 2) $\models A|_i$ for some $i \in \mathbb{N}$; 3) $\models A$.* $\square$

3 A Leading Example: ETCS

We use verification of (a simplified version of) the *European Train Control System (ETCS)* as a leading example—this is done also in many chapters of [13]. We go step by step, introducing the strategies as the need arises.

Our target program $\mathtt{ETCS}_0$ is shown on the right. It contains small fragments of the European Train Control System (ETCS). Here `a`, `t`, `v`, and `z` are variables that represent acceleration rate, time, velocity, and position, respectively. The symbols `m`, `s`, `b`, `a0`, `eps` are all constants, all of which are assumed to be (strictly) positive. Here, `m` is the position beyond which the train must not run ("a wall"); `s` is the *safety distance* such that, once the train's distance from the wall is less than `s`, it starts braking; `b` is the rate used in braking; `a0` is the (positive) acceleration rate used unless the train is braking. The check if $\mathtt{m} - \mathtt{z} < \mathtt{s}$ occurs once every `eps` seconds.

Fig. 1. The original program $\mathtt{ETCS}_0$

```
while (v > 0) {
  if m - z < s
    then a := -b
    else a := a0;
  t := 0;
  while (t < eps && v > 0) {
    z := z + v * dt;
    v := v + a * dt;
    t := t + dt }}
(: z < m :)
```

The (post)condition to be guaranteed after the execution of $\mathtt{ETCS}_0$ is the safety property $\mathtt{z} < \mathtt{m}$; it is inserted in Fig. 1 as an annotation, between `(:` and `:)`. We set out to calculate a precondition—a relationship among constants as well as (the initial values of) the variables—that ensures this postcondition.

The program $\mathtt{ETCS}_0$ is itself a while-loop; thus to calculate a precondition we need to discover a (loop) invariant. However the loop is a complicated one with an if branch and another while-loop inside. We first preprocess $\mathtt{ETCS}_0$ and simplify it, employing some program transformation strategies studied in the field of static analysis.

4 Program Simplification Strategies

4.1 Phase Split

Looking at $\texttt{ETCS}_0$ (Fig. 1), we can imagine that once the train starts braking, it never starts to accelerate. That is, we could transform the outer loop in $\texttt{ETCS}_0$ into two successive loops, with the former loop solely for accelerating and the latter loop for braking.

It is the program transformation technique proposed in [20] (closely related to those in [2, 10]) that makes the above intuition rigorous. It eliminates an if-branch inside a while loop using a *phase splitter predicate* b_s, as is shown on the right. This simplification strategy shall be referred to as *phase split*.

$$\texttt{while}\ b_\mathrm{g}\ \texttt{do}\ \boxed{\ldots(\texttt{if}\ldots)\ldots}$$

$$\text{into}\quad \left[\begin{array}{l} \texttt{while}\ b_\mathrm{g} \wedge \neg b_\mathrm{s}\ \texttt{do}\ \boxed{\ldots}\ ; \\ \texttt{while}\ b_\mathrm{g} \wedge b_\mathrm{s}\ \texttt{do}\ \boxed{\ldots} \end{array}\right]$$

First we follow [20] and describe the strategy formalized in WHILE and HOARE. In §4.2 we use the sectionwise lemmas and show that the nonstandard framework (WHILE$^{\mathtt{dt}}$ and HOARE$^{\mathtt{dt}}$), too, admits the same strategy.

Definition 4.1 (Holed command, pre-hole fragment) The set $\mathbf{Cmd}_{[\_]}$ of *holed commands* of WHILE—those which contain exactly one hole $[\_]$ for a guard of `if`—is defined by the BNF expression below. Here $c, c_1, c_2 \in \mathbf{Cmd}$ are commands (without holes). The result of replacing $[\_]$ with $b \in \mathbf{BExp}$ in $h \in \mathbf{Cmd}_{[\_]}$ is denoted by $h[b]$.

$$\begin{array}{rl} \mathbf{Cmd}_{[\_]} \ni h \ ::= & \texttt{if}\ [\_]\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid h; c \mid c; h \mid \\ & \texttt{if}\ b\ \texttt{then}\ h\ \texttt{else}\ c \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ h \end{array}$$

$$\overline{\texttt{if}\ [\_]\ \texttt{then}\ c_1\ \texttt{else}\ c_2} :\equiv \texttt{skip} \qquad \overline{h; c} :\equiv \overline{h} \qquad \overline{c; h} :\equiv c; \overline{h}$$

$$\overline{\texttt{if}\ b\ \texttt{then}\ h\ \texttt{else}\ c} :\equiv \texttt{assert}\ b\ ; \overline{h} \qquad \overline{\texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ h} :\equiv \texttt{assert}\ \neg b\ ; \overline{h}$$

For each holed command h, its *pre-hole fragment* $\overline{h}$ is defined inductively as above. Intuitively, it is the fragment of h that is executed before hitting the hole $[\_]$.

Lemma 4.2 (Phase split [20]) *If a Boolean expression* $b_\mathrm{s} \in \mathbf{BExp}$ *satisfies*

$$\models \{b_\mathrm{s}\}\, \overline{h}\, \{b_\mathrm{c}\}\ , \quad \models \{\neg b_\mathrm{s}\}\, \overline{h}\, \{\neg b_\mathrm{c}\}\ , \quad \textit{and} \quad \models \{b_\mathrm{g} \wedge b_\mathrm{s}\}\, h[b_\mathrm{c}]\, \{\neg b_\mathrm{g} \vee b_\mathrm{s}\}\ , \quad (2)$$

then we have $[\![c_0]\!] = [\![c_1]\!]$ *between the commands*

$$\begin{array}{lcl} c_0 & :\equiv & \texttt{while}\ b_\mathrm{g}\ \texttt{do}\ h[b_\mathrm{c}]\ , \quad \textit{and} \\ c_1 & :\equiv & \left[\ \texttt{while}\ (b_\mathrm{g} \wedge \neg b_\mathrm{s})\ \texttt{do}\ h[\texttt{false}]\ ;\ \texttt{while}\ (b_\mathrm{g} \wedge b_\mathrm{s})\ \texttt{do}\ h[\texttt{true}]\ \right]\ . \end{array} \quad (3)$$

□

Such b_s is called a *phase splitter predicate*. In applying this lemma to $\texttt{ETCS}_0$, an obvious candidate for a phase splitter predicate is $\mathtt{m} - \mathtt{z} < \mathtt{s}$, the guard of the if-branch itself. (Its negation $\mathtt{m} - \mathtt{z} \geq \mathtt{s}$ is a candidate too; but the side conditions (2) cannot be discharged.)

To discharge the side conditions (2) is not hard: in its course we use the *differential invariant* strategy described later in §5.3. We further apply obvious simplifications, such as $[\![\texttt{if true then}\ c_1\ \texttt{else}\ c_2]\!] = [\![c_1]\!]$ and constant propagation; consequently we are led to the program $\texttt{ETCS}_1$ (on the right) that is equivalent to $\texttt{ETCS}_0$.

Fig. 2. The program $\texttt{ETCS}_1$

```
while (v > 0 && m - z >= s) {
  a := a0;   t := 0;
  while (t < eps && v > 0) {
    z := z + v * dt;
    v := v + a0 * dt;
    t := t + dt }};
while (v > 0 && m - z < s) {
  a := -b;   t := 0;
  while (t < eps && v > 0) {
    z := z + v * dt;
    v := v - b * dt;
    t := t + dt }}
(: z < m :)
```

4.2 From Standard to Nonstandard I: Modular Transfer

As stated in the introduction, the current paper presents strategies (for simplification and invariant/precondition discovery) only for the standard framework (WHILE and HOARE), leaving out the corresponding nonstandard strategies (for $\text{WHILE}^{\mathtt{dt}}$ and $\text{HOARE}^{\mathtt{dt}}$). This is because the transfer from the former to the latter is straightforward. Here we describe the process of such transfer, with the phase split strategy (§4.1) as an example. Here the *sectionwise lemmas* (in particular Lem. 2.5) play the key role.

The syntactic notions of holed command and pre-hole fragment are defined in $\text{WHILE}^{\mathtt{dt}}$, in the exactly the same way as in WHILE (Def. 4.1).

Lemma 4.3 (Nonstandard phase split) *Assume* $b_{\mathrm{s}} \in \mathbf{BExp}$ *satisfies, in* $\text{HOARE}^{\mathtt{dt}}$,

$$\models \{b_{\mathrm{s}}\}\, \overline{h}\, \{b_{\mathrm{c}}\} \ , \quad \models \{\neg b_{\mathrm{s}}\}\, \overline{h}\, \{\neg b_{\mathrm{c}}\} \ , \quad \textit{and} \quad \models \big\{b_{\mathrm{g}} \wedge b_{\mathrm{s}}\big\}\, h[b_{\mathrm{c}}]\, \big\{\neg b_{\mathrm{g}} \vee b_{\mathrm{s}}\big\} \ .$$

Let c_0 *and* c_1 *be as in (3), now in* $\text{WHILE}^{\mathtt{dt}}$. *We have* $[\![c_0]\!] = [\![c_1]\!]$. □

Proof. Let $\boldsymbol{\sigma} \in \mathbf{HSt}$ be a hyperstate; and $(\boldsymbol{\sigma}|_i)_{i\in\mathbb{N}}$ be its arbitrary sequence representation. By the definition of hypernumbers, it suffices to show that

$$[\![c_0|_i]\!](\boldsymbol{\sigma}|_i) = [\![c_1|_i]\!](\boldsymbol{\sigma}|_i) \qquad \text{for almost every } i. \tag{4}$$

By the assumptions and Lem. 2.5, each one of the following holds for almost every i.

$$\models \{b_{\mathrm{s}}|_i\}\, \overline{h}|_i\, \{b_{\mathrm{c}}|_i\} \ \models \{(\neg b_{\mathrm{s}})|_i\}\, \overline{h}|_i\, \{(\neg b_{\mathrm{c}})|_i\} \ \models \big\{(b_{\mathrm{g}} \wedge b_{\mathrm{s}})|_i\big\}\, h[b_{\mathrm{c}}]|_i\, \big\{(\neg b_{\mathrm{g}} \vee b_{\mathrm{s}})|_i\big\} \ .$$

Therefore, by the definition of ultrafilter, all three of the above hold simultaneously, for almost every i. That is, for almost every i, we have all of the following three hold:

$$\models \{b_{\mathrm{s}}|_i\}\, \overline{h|_i}\, \{b_{\mathrm{c}}|_i\} \ \models \{\neg(b_{\mathrm{s}}|_i)\}\, \overline{h|_i}\, \{\neg(b_{\mathrm{c}}|_i)\} \ \models \big\{b_{\mathrm{g}}|_i \wedge b_{\mathrm{s}}|_i\big\}\, h|_i[b_{\mathrm{c}}|_i]\, \big\{\neg(b_{\mathrm{g}}|_i) \vee b_{\mathrm{s}}|_i\big\}$$

where we also used an obvious fact $\overline{h}\,|_i \ \equiv\ \overline{(h|_i)}$. To each such i we apply the standard version of the strategy (Lem. 4.2) and obtain, for almost every i,

$$\begin{aligned}&\big[\!\big[\mathtt{while}\ b_{\mathrm{g}}|_i\ \mathtt{do}\ h|_i[b_{\mathrm{c}}|_i]\big]\!\big](\boldsymbol{\sigma}|_i)\\ &= \big[\!\big[\mathtt{while}\ (b_{\mathrm{g}}|_i \wedge \neg b_{\mathrm{s}}|_i)\ \mathtt{do}\ h|_i[\mathtt{false}]\ ;\ \mathtt{while}\ (b_{\mathrm{g}}|_i \wedge b_{\mathrm{s}}|_i)\ \mathtt{do}\ h|_i[\mathtt{true}]\big]\!\big](\boldsymbol{\sigma}|_i)\end{aligned}$$

Using another obvious fact that $h[b_{\mathrm{s}}]\,|_i \ \equiv\ \big(h|_i\big)[b_{\mathrm{s}}|_i]$, this is equivalent to (4). □

Note the *modularity* in the proof: the standard version's proof (Lem. 4.2) is completely hidden. For any other (standard) strategy presented in this paper, its nonstandard version follows in the same way via sectionwise arguments.

4.3 Superfluous Guard Elimination

In $\mathtt{ETCS}_1$ each while-loop has its guard consisting of two atomic inequalities. This causes problems with our toolkit for invariant/precondition discovery, especially with the *counting iterations* strategy (§5.4). In fact, some of the conditions are seemingly superfluous: for example, in the second (internal) while-loop, $\mathtt{m} - \mathtt{z} < \mathtt{s}$ is an invariant.

Lemma 4.4 (Superfluous guard elimination) *For commands*

$$\begin{array}{lcl} c_0 & :\equiv & \texttt{while}\ (b \wedge b_{\mathrm{sf}})\ \texttt{do}\ c\ , \text{ and} \\ c_1 & :\equiv & \texttt{if}\ b_{\mathrm{sf}}\ \texttt{then}\ \big(\texttt{while}\ b\ \texttt{do}\ c\big)\ \texttt{else skip}\ , \end{array}$$

if we have $\models \{b \wedge b_{\mathrm{sf}}\}\, c\, \{b_{\mathrm{sf}}\}$, *then* $[\![c_0]\!] = [\![c_1]\!]$. □

This strategy, together with some straightforward simplifications, transforms $\texttt{ETCS}_1$ into $\texttt{ETCS}_2$ shown below. For each of the three instances of the strategy, the side condition is easily discharged using the *differential invariant* strategy (§5.3).

The program $\texttt{ETCS}_2$:

```
if (v > 0)
  then
    while (m - z >= s) {
      a := a0;   t := 0;
      while (t < eps) {
        z := z + v * dt;
        v := v + a0 * dt;
        t := t + dt }}
  else skip;
```

```
while (v > 0) {
  a := -b;   t := 0;
  while (t < eps && v > 0) {
    z := z + v * dt;
    v := v - b * dt;
    t := t + dt }}
(: z < m :)
```

4.4 Time Elapse

The challenge in verifying WHILE programs is in while-loops. The second half of $\texttt{ETCS}_2$ is a loop with another loop inside; we wish to simplify this.

A close look reveals that the inside loop is vacuous: since the inside guard and the outside guard share the same condition $\texttt{v} > 0$, the condition $\texttt{t} < \texttt{eps}$ has in fact no effect.[4] We often encounter this situation in the verification of $\text{WHILE}^{\texttt{dt}}$ programs—other simplification strategies easily lead to such vacuous nested loops.

Lemma 4.5 (Time elapse) *Let* t *be a variable that does not occur in* $b \in \mathbf{BExp}$ *or* $c \in \mathbf{Cmd}$*;* ε *be a positive constant; and*

$$\begin{array}{lcl} c_0 & :\equiv & \texttt{while}\ b\ \texttt{do}\ \big(\ t := 0;\quad \texttt{while}\ (t < \varepsilon\ \wedge\ b)\ \texttt{do}\ (t := t + \texttt{dt};\ c)\ \big)\ , \\ c_1 & :\equiv & \texttt{while}\ b\ \texttt{do}\ c\ . \end{array}$$

Then the denotations $[\![c_0]\!]$ *and* $[\![c_1]\!]$ *coincide on all variables but* t. □

Lem. 4.5 allows several straightforward generalizations. Currently we do not need them.

Since t does not occur in the postcondition $\texttt{z} < \texttt{m}$, Lem. 4.5 ensures that *time elapse* is a sound transformation. It simplifies the second half of $\texttt{ETCS}_2$ into $\texttt{ETCS}_3$ below.

The program $\texttt{ETCS}_3$:

```
if (v > 0)
  then
    while (m - z >= s) {
      a := a0;   t := 0;
      while (t < eps) {
        z := z + v * dt;
        v := v + a0 * dt;
        t := t + dt }}
  else skip;
```

```
while (v > 0) {
  a := -b;
  z := z + v * dt;
  v := v - b * dt }
(: z < m :)
```

[4] This is not the case with the first half of $\texttt{ETCS}_2$: it might be $\texttt{m} - \texttt{z} < \texttt{s}$ but still the train can accelerate for eps seconds. Therefore the nested loops cannot be suppressed so easily.

5 Precondition/Invariant Discovery Strategies

We now describe three strategies aimed at the discovery of suitable preconditions/invariants for while-loops. In fact they return directly a precondition (when they succeed).

We desire their output to be quantifier-free, since quantifiers often incur prohibiting performance penalties. This is what we do in our prototype: quantifiers appear only in the *Mathematica* backend. In what follows several quantifiers do occur; but they are in the correctness proofs of the strategies, i.e. on the meta level.

5.1 Invariant via Quantifier Elimination

We now compute a precondition `precond1` that, after the execution of the last while-loop in $\texttt{ETCS}_3$ (which we denote by c_{brake}), ensures the postcondition $\texttt{z} < \texttt{m}$ hold.

Recall our thesis: via nonstandard analysis, we can represent continuous flow by discrete jumps, so that (discrete) verification techniques readily apply. However, once we cast a *continuous* look at c_{brake}, it obviously represents a simple flow dynamics governed by the differential equations $\dot{v}(t) = \ddot{z}(t) = -b$. In this "continuous" perspective, one would solve the current question in the following way.

- First the differential equations are solved analytically, obtaining $v(t) = v_0 - bt$ and $z(t) = z_0 + v_0 t - \frac{1}{2}bt^2$. Here v_0 and z_0 are initial values.
- Using these analytic solutions, the desired precondition is nothing other than

$$\forall t \in \mathbb{R}.\ \big(\ v_0 - bt > 0\ \wedge\ t \geq 0 \quad \Longrightarrow \quad z_0 + v_0 t - \frac{1}{2}bt^2 < m\ \big)\ . \tag{5}$$

  One would then apply *quantifier elimination* and obtain a quantifier-free formula that is equivalent to (5). Some algorithms are known including *cylindrical algebraic decomposition (CAD)*. In *Mathematica* `Resolve` offers this functionality; it returns

$$v_0 \leq 0\ \vee\ (\ z_0 < m\ \wedge\ 2bm - v_0^2 - 2bz_0 \geq 0\ )\ .$$

This procedure of "first solve differential equations, then eliminate quantifiers" is common in the deductive verification of hybrid systems, such as in Platzer's [13].

The next strategy (Lem. 5.2) is a discrete variation of this procedure, where differential equations are replaced with *difference equations* (also called *recurrence relations*).

We need preparation. In Lem. 5.2 we need to have a formula $A\overbrace{[a/x][a/x]\cdots[a/x]}^{y \text{ times}}$ as a formula *with y as a free variable*. This is not automatic—note that substitution $[a/x]$ and the number y live on the meta level. However, let us say that A is a formula $A \equiv (x > 0)$ and $a \equiv x + 1$. Then we have, for any $y \in \mathbb{N}$,

$$\models \qquad \big(\ x > 0\ \big)\ [x + 1/x]\,[x + 1/x]\cdots[x + 1/x] \quad \Leftrightarrow \quad x + y > 0\ ,$$

where substitution is repeated y times. Thus the formula $x + y > 0$ *effectively* represents the formula $(\ x > 0\ )\ [x + 1/x]\,[x + 1/x]\cdots[x + 1/x]$. In other words, $x + y > 0$ is a *homogeneous representation* of substitutions $x > 0, x + 1 > 0, (x + 1) + 1 > 0, \ldots$.

The general definition is as follows. There A_{rs} corresponds to $x + y > 0$ in the above example; it is a formula that must be discovered somehow. In our prototype it is done by solving a recurrence relation.

Definition 5.1 (Repeated substitution) Let x be a variable, A be a ASSN formula, a be an arithmetic expression, and y be a variable not occurring in A, a or x. A formula A_{rs} is said to be a *homogeneous representation* of A and $[a/x]$ if the following holds.

$$\models \quad A_{\mathrm{rs}}[0/y] \Leftrightarrow A \quad \wedge \quad \forall u \in \mathbb{N}.\big(\, A_{\mathrm{rs}}[u+1/y] \Leftrightarrow \big(A_{\mathrm{rs}}[u/y]\big)[a/x] \,\big) \qquad (6)$$

We abuse notations by denoting such A_{rs} by $A[a/x]^y$.

Lemma 5.2 (QE invariant)

$$\models \left\{ \begin{array}{l} (\neg b \Rightarrow A) \quad \wedge \\ \forall y \in \mathbb{N}.\big(\, (\, b[a/x]^y \wedge \neg b[a/x]^{y+1} \,) \Rightarrow A[a/x]^{y+1} \,\big) \end{array} \right\} \ \texttt{while}\ b\ \texttt{do}\ x := a \ \ \{A\} \ .$$

The lemma is presented in the simplest form: it is straightforward to allow a sequence $x_1 := a_1; \ldots ; x_n := a_n$ of assignments inside the loop.

Proof. Let P be the precondition $(\neg b \Rightarrow A) \ \wedge \ \forall y \in \mathbb{N}.\big(\, \big(\, b[a/x]^y \wedge \neg b[a/x]^{y+1} \,\big) \Rightarrow A[a/x]^{y+1} \,\big)$. By the (WHILE) rule of HOARE, it suffices to show that: 1) P is a loop invariant, that is, $\models \{P \wedge b\} x := a \{P\}$; and 2) P is strong enough, i.e. $\models P \wedge \neg b \Rightarrow A$.

To prove 1), assume $\sigma \models P \wedge b$. The goal is $[\![x := a]\!]\sigma \models P$, which is equivalent to $\sigma \models P[a/x]$, since $P[a/x]$ is the weakest precondition for $x := a$ and the postcondition P. This is further reduced to: $\sigma \models b[a/x]^{n+1} \wedge \neg b[a/x]^{n+2} \Rightarrow A[a/x]^{n+2}$ for any $n \in \mathbb{N}$ (which follows from $\sigma \models P$); and $\sigma \models \neg b[a/x] \Rightarrow A[a/x]$ (which follows from $\sigma \models b$ and $\sigma \models (\, b \wedge \neg b[a/x] \,) \Rightarrow A[a/x]$).

The condition 2) is obvious since $\models P \Rightarrow (\neg b \Rightarrow A)$. This concludes the proof. □

In our prototype, we first compute homogeneous representations $b[a/x]^y$ and $A[a/x]^y$ by solving recurrence relations in y such as $A[a/x]^{y+1} = \big(A[a/x]^y\big)[a/x]$. Using this we form the precondition in Lem. 5.2. Then we further eliminate the quantifier $\forall y \in \mathbb{N}$ and obtain a quantifier-free precondition.

For example, in the case of c_{brake} in $\texttt{ETCS}_3$, we discover a homogeneous representation $A[\overrightarrow{a}/\overrightarrow{x}]^y :\equiv \frac{1}{2}(b\mathtt{dt}^2 y + 2\mathtt{dt}vy - b\mathtt{dt}^2 y^2 + 2z) < m$. As a quantifier-free precondition we obtain the following, which is henceforth denoted by P_1.

$$P_1 :\equiv (v > 0 \vee m > z) \wedge \big(b^2\mathtt{dt}^2 + 4b\mathtt{dt}v + 8bz + 4v^2 < 8bm \vee b\mathtt{dt}v + 2bz + v^2 \leq 2bm\big) \qquad (7)$$

This strategy (Lem. 5.2) has the merit of having no side conditions. But in our experience it is computationally expensive: a slightly complicated postcondition (say, a combination of a few inequalities) makes the quantifier elimination step infeasible. In fact it does not currently work for the loops in $\texttt{ETCS}_3$ other than the last one c_{brake}.

5.2 From Standard to Nonstandard II: The Transfer Principle and Symbolic Computation in *Mathematica*

Here again we demonstrate how a standard strategy can be transferred to a nonstandard one. The strategy *QE invariant* (Lem. 5.2) employs *Mathematica* in two stages: solving recurrence relations and eliminating quantifiers, both done over standard numbers (instead of hyperreals). Our focus is therefore on how this can be justified.

Definition 5.3 (Repeated substitution in ASSN$^{\mathtt{dt}}$) Let x, A, a, y be as in Def. 5.1, but now in ASSN$^{\mathtt{dt}}$. A formula A'_{rs} is said to be a *homogeneous representation* of A and $[a/x]$, and is denoted by $A[a/x]^y$, if the following holds.

$$\models \quad A'_{\mathrm{rs}}[0/y] \Leftrightarrow A \quad \wedge \quad \forall u \in {}^*\mathbb{N}.\big(\ A'_{\mathrm{rs}}[u+1/y] \Leftrightarrow \big(A'_{\mathrm{rs}}[u/y]\big)[a/x]\ \big) \qquad (8)$$

The sole difference from Def. 5.1 is the quantifier (ranging over hypernatural numbers).

How do we find A'_{rs} that satisfies (8)? We rely on the following lemma.

Lemma 5.4 *In the setting of Def. 5.3, let A' and a' be the expressions obtained from A and a by replacing the constant* `dt` *by a fresh variable d (and also by making hyperquantifiers standard). If A_{rs} is a homogeneous representation $A'[a'/x]^y$ in* ASSN *(Def. 5.1), then ${}^*A_{\mathrm{rs}}[\mathtt{dt}/d]$ is a homogeneous representation $A_{\mathrm{rs}}[a/x]^y$ in* ASSN$^{\mathtt{dt}}$. □

Therefore it essentially suffices to prove (6). This is in ASSN hence *Mathematica* is capable of proving it.

Lemma 5.5 (Nonstandard QE invariant) *We have, in* HOARE$^{\mathtt{dt}}$,

$$\models \left\{ \begin{array}{l} (\neg b \Rightarrow A) \wedge \\ \forall y \in {}^*\mathbb{N}.\big(\ \big(\ b[a/x]^y \wedge \neg b[a/x]^{y+1}\ \big) \Rightarrow A[a/x]^{y+1}\ \big) \end{array} \right\} \ \mathtt{while}\ b\ \mathtt{do}\ x := a\ \ \{A\}\ .$$

Proof. By Lem. 2.5, the following suffices: in HOARE,

$$\models \left\{ \left(\begin{array}{l} (\neg b \Rightarrow A) \wedge \\ \forall y \in {}^*\mathbb{N}.\big(\ \big(\ b[a/x]^y \wedge \neg b[a/x]^{y+1}\ \big) \Rightarrow A[a/x]^{y+1}\ \big) \end{array} \right)\Big|_i \right\} \begin{array}{l} \mathtt{while}\ b|_i\ \mathtt{do} \\ x := a|_i \end{array} \{A|_i\}$$

holds for almost every $i \in \mathbb{N}$. By Def. 5.3, Def. 5.1 and [21, Lem. 4.5], it is easy that $\models \big(A[a/x]^y\big)|_i \Leftrightarrow A|_i[a|_i/x]^y$ for almost every $i \in \mathbb{N}$. Therefore it suffices to show

$$\models \left\{ \begin{array}{l} (\neg b|_i \Rightarrow A|_i) \quad \wedge \\ \forall y \in \mathbb{N}.\big(\ \big(\ b|_i[a|_i/x]^y \wedge \neg b|_i[a|_i/x]^{y+1}\ \big) \Rightarrow A|_i[a|_i/x]^{y+1}\ \big) \end{array} \right\} \begin{array}{l} \mathtt{while}\ b|_i\ \mathtt{do} \\ x := a|_i \end{array} \{A|_i\}$$

for almost every i. The last statement (in HOARE) is Lem. 5.2 itself. □

The second use of *Mathematica*—in quantifier elimination—is justified much like in Lem. 5.4, via the transfer principle.

5.3 Differential Invariant

Most of the simplification strategies in §4 come with side conditions to be discharged. In many such cases, the postcondition to be established is itself an invariant. An example is the condition $\mathtt{v} > 0$ in the first half of $\mathtt{ETCS}_1$.

The following lemma provides an efficient method for proving such Hoare triples.[5]

[5] The name is taken from Platzer's extensive treatment of a similar notion [13]. The correspondence is as follows: when the while-loop below represents continuous dynamics, the condition $a_{\mathrm{c}}[a/x] < a_{\mathrm{c}}$ means that the first derivative of a_{c} is negative.

Lemma 5.6 (Differential Invariant) *Assume that an arithmetic expression a_{c} satisfies $\models b \Rightarrow a_{\mathrm{c}}[a/x] < a_{\mathrm{c}}$. Then $\models \{a_{\mathrm{c}} < 0\}$ `while` b `do` $x := a \{a_{\mathrm{c}} < 0\}$.* □

In our current prototype a candidate for the invariant $a_{\mathrm{c}} < 0$ is chosen simply out of the disjuncts of the postcondition. More sophisticated candidate generation methods—such as in [13]—are left as future work.

5.4 Iteration Count

Verification of the leading example ETCS (§3) has been reduced, in §5.1, to the situation below on the left. Here postcondition P_1 is as in (7). Our prototype further simplifies the situation into the one on the right.

```
if (v > 0)
 then
   while (m - z >= s) {
     a := a0;   t := 0;
     while (t < eps) {
       z := z + v * dt;
       v := v + a0 * dt;
       t := t + dt }}
 else skip
(: P_1 :)
```

```
while (m - z >= s) {
 a := a0;   t := 0;
 while (t < eps) {
   z := z + v * dt;
   v := v + a0 * dt;
   t := t + dt }}
(: P_2 :)
```

where $P_2 :\equiv b^2\mathtt{dt}^2 + 4b\mathtt{dt}v + 8bz + 4v^2 < 8bm \vee b\mathtt{dt}v + 2bz + v^2 \leq 2bm$ (9)

We now face a nested loop to which none of the previous strategies applies. Our last strategy estimates the number of iterations in each loop—such as `eps`/`dt` for the inner loop[6]—and approximates commands by repeated applications of assignments. The strategy involves nontrivial side conditions but we experienced its wide applicability.

The formalization of the full *iteration count* strategy is cumbersome and buries its idea in overwhelming details. Here we only present the following restricted version that deals with a single loop. The full version that applies to nested loops is deferred to the appendix.

Lemma 5.7 *Let a_{i} be an arithmetic expression. Assume the following conditions.*

1. $\models \neg b[a/x]^{a_{\mathrm{i}}}$
2. *For some fresh variable y,* $\models \{\neg b[a/x]^{y+1}\}\, c\, \{\neg b[a/x]^{y}\}$
3. $\models \forall u, v \in \mathbb{R}.\, (\, (\, a_{\mathrm{i}} \leq u < v < a_{\mathrm{i}} + 1 \wedge \neg b[a/x]^{u}\,) \Rightarrow \neg b[a/x]^{v}\,)$
4. *For some fresh variable y,* $\models \{A[a/x]^{y+1}\}\, c\, \{A[a/x]^{y}\}$
5. $\models \forall u, v \in \mathbb{R}.\, (\, (\, u < v \wedge A[a/x]^{v}\,) \Rightarrow A[a/x]^{u}\,)$

Then we have

$$\models \quad \{A[a/x]^{a_{\mathrm{i}}+1} \wedge a_{\mathrm{i}} \geq 0\}\ \mathtt{while}\ b\ \mathtt{do}\ c\ \{A\}\ . \tag{10}$$

□

Here are some intuitions. The expression a_{i} denotes an estimated number of iterations: for example, $a_{\mathrm{i}} \equiv \mathtt{eps}/\mathtt{dt}$ for the inner loop on the right in (9). The estimation is done by solving the equation $a_{\mathrm{i}} \cdot \mathtt{dt} = \mathtt{eps}$. We are also *simulating* execution of c by a

[6] Note that the number `eps`/`dt` makes sense in NSA: it means $(i+1) \cdot \mathtt{eps}$ in the i-th section.

substitution $[a/x]$; the conditions 2. and 4. assert that this simulation is sound. Note that these conditions are trivially satisfied if c is an assignment command $x := a$.

The conditions 3. and 5. impose suitable *monotonicity* requirements on the guard b and the postcondition A. Such monotonicity is needed for the following reason. The estimated number a_{i} of iterations need not be exactly some (hyper)natural number—it can lie between two (hyper)natural numbers, i.e. $\lceil a_{\mathrm{i}} \rceil - 1 < a_{\mathrm{i}} < \lceil a_{\mathrm{i}} \rceil$. This is why we are *conservative* in (10); we use $a_{\mathrm{i}} + 1$, instead of a_{i}, as a number of iterations. The monotonicity conditions (3. & 5.) ensure that this conservative approximation is sound.

The full version of this *iteration count* strategy (Lem. 5.7) succeeds in the remaining bit of the leading example (on the right in (9)). Its output is a Boolean combination of inequalities; we denote this formula by P_3. It is complicated—not least due to occurrences of $\mathtt{dt}$—and we would rather not write it down explicitly.

5.5 Cast to Shadow

We have established that $\models \ \{P_3\}\ \mathtt{ETCS_0}\ \{\mathtt{z} < \mathtt{m}\}$. As the last step, we wish to eliminate the occurrences of $\mathtt{dt}$ in the precondition P_3. This results in a much simpler precondition. The intuition is: the condition $\varepsilon + \mathtt{dt} < 0$ is implied by $\varepsilon < 0$ if ε is a constant denoting a standard real number. To put it in rigorous terms:

Lemma 5.8 (Cast to Shadow) *Let a be an arithmetic expression in* WHILE$^{\mathtt{dt}}$*, d be a fresh variable, and $a[d/\mathtt{dt}]$ be the expression in* WHILE *obtained by substituting d for $\mathtt{dt}$ in a. Assume the following.*

1. *The expression a is* closed*, that is, it has no occurrences of variables.*
2. *Let σ be a (standard) state that is not $\bot$; it determines a function*[7] *$f_\sigma : \mathbb{R} \to \mathbb{R}$ by $r \mapsto [\![a[d/\mathtt{dt}]]\!]\big(\sigma[d \mapsto r]\big)$. We assume that f_σ is continuous at $r = 0$ for any given $\sigma \neq \bot$. Here $[\![a[d/\mathtt{dt}]]\!]$ is defined by denotational semantics [21, §3.2]; and $\sigma[d \mapsto r]$ is an updated state.*

Then $\models \big(\, a[d/\mathtt{dt}]\,\big)[0/d] < 0 \Longrightarrow a < 0$. □

Note that, by the definition of $a[d/\mathtt{dt}]$, the expression $\big(\, a[d/\mathtt{dt}]\,\big)[0/d]$ is the result of replacing $\mathtt{dt}$ in a with 0—i.e. of "eliminating $\mathtt{dt}$." In using the lemma, the closedness assumption (Cond. 1.) can be enforced by setting the initial values of variables as constants (that necessarily represent standard reals by the definition of WHILE$^{\mathtt{dt}}$). The continuity assumption (Cond. 2.) in the lemma is most of the time satisfied since the arithmetic operations of WHILE are all continuous except for zero-division.

In nonstandard analysis, a hyperreal number r' that is not infinite has a unique (standard) real number r that is infinitesimally close to r' (we write $r \simeq r'$). Such r is called the *shadow* of r'; hence the name of the strategy.

This strategy can be used for simplifying the precondition P_3 (or, more generally, a Boolean combination of inequalities) in the following way.

– First, P_3 is converted to CNF (or DNF; it does not matter): $\models P_3 \Leftrightarrow \bigwedge_i \bigvee_j L_{i,j}$. Note that every literal $L_{i,j}$ occurs positively.

[7] In general a partial function since zero-division might occur.

- Among all the literals $L_{i,j}$, the negative ones (like $L_{i,j} \equiv \neg(a_{i,j} < b_{i,j})$) is turned into a positive one (like $a_{i,j} \geq b_{i,j}$), by reversing inequalities.
- Each inequality is turned into the form of either $c_{i,j} < 0$ or $c_{i,j} \leq 0$. For example: we turn $a_{i,j} \geq b_{i,j}$ into $b_{i,j} - a_{i,j} \leq 0$. Thus we have obtained $\models P_3 \Leftrightarrow \bigwedge_i \bigvee_j c_{i,j} \lhd_{i,j} 0$, where each $\lhd_{i,j}$ is either $<$ or $\leq$.
- Obviously $\models c_{i,j} < 0 \Rightarrow c_{i,j} \lhd_{i,j} 0$; thus $\models \bigwedge_i \bigvee_j c_{i,j} < 0 \Rightarrow P_3$.
- Finally, to each inequality $c_{i,j} < 0$ we apply Lem. 5.8 (given that the side conditions are discharged). This establishes $\models \bigwedge_i \bigvee_j \big(c_{i,j}[d/\mathtt{dt}] \big)[0/d] < 0 \Rightarrow P_3$; thus we obtain a "simplified" precondition that has no occurrences of `dt`.

We note that thus obtained precondition $\bigwedge_i \bigvee_j \big(c_{i,j}[d/\mathtt{dt}] \big)[0/d] < 0$ is not necessarily equivalent to P_3, but is stronger (i.e. less general). In our experience, however, the lost generality is most of the time marginal.

After all, the following is the (core part of the long and extensive) outcome of our prototype when it is fed with $\mathtt{ETCS}_0$.

$$a_0\big(2\varepsilon\sqrt{2a_0(m-s-z_0)+v_0^2} + b\epsilon^2 + 2m - 2s - 2z_0\big) \\ + 2b\varepsilon\sqrt{2a_0(m-s-z_0)+v_0^2} + a_0^2\epsilon^2 + v_0^2 < 2bs$$

Here v_0 and z_0 are constants that designate the initial values of v and z, respectively.

6 Implementation and Experiments

Our prototype implementation takes a WHILE$^{\mathtt{dt}}$ program and a postcondition as its input, and tries to calculate a precondition. The tool consists of the following components.

- A native verification condition generator implemented in OCaml. It combines the standard verification condition generation for Hoare-type logics (such as in [22]) and the previously described strategies. It relies on the *Mathematica* backend for most of the strategies as well as for (symbolic) arithmetic proofs.
- A *Mathematica* backend that provides functions for the strategies in §4–5. It also implements some proof strategies for arithmetic formulas.
- A Perl script that serves as an interface between the above two.

Our prototype employs the standard backward reasoning for Hoare-type logics. When it encounters a while loop, it tries the precondition discovery strategies in §5. If one or more of those strategies succeed and produce preconditions $P_1, \ldots, P_n$, then our prototype continues with the disjunction $P_1 \vee \cdots \vee P_n$ of those preconditions. If not, our prototype tries simplification strategies in §4; the order is *phase split*, *superfluous guard elimination* and then *time elapse*. If the simplification succeeds, it again tries the precondition discovery strategies. If not, our prototype reports failure.

Although we sometimes use numeric computations for finding counterexamples, the proofs are established in purely symbolic means. This is a crucial fact for the correctness guarantee via the transfer principle.

We have tested our prototype against the following WHILE$^{\mathtt{dt}}$ programs:

- `etcs.while`: the ETCS example used throughout the paper, and

– `zeno.while`: the behavior of a bouncing ball.

We conducted both experiments on Fujitsu HX600 with Quad Core AMD Opteron 2.3GHz CPU and 32GB memory. We used *Mathematica* 7.0 for Linux x86 (64-bit).[8]

The current implementation is premature. The goals of the experiments have been: 1) the feasibility test of our methodology of nonstandard static analysis, and 2) to drive theoretical development of static analysis strategies suited for hybrid applications. Enhancing scalability and efficiency of the prototype is left as future work.

`etcs.while`. Our prototype completed precondition generation for `etcs.while` in 40.96 seconds (31.97 seconds in the user process and 8.99 in the kernel process), generating the precondition described in §5.5 as an outcome.

`zeno.while`. We have modeled the behavior of a bouncing ball in WHILE$^{\mathtt{dt}}$ and run our prototype on the program. This program includes the following three parameters:

– t_0: ending time of the system;
– b: elastic coefficient between the ground and the ball;
– h_0: *safety height*—the ball should not reach higher than this height.

The postcondition is that the peak height of the last bounce is not higher than h_0.

Although the current prototype does not fully succeed to compute a precondition, it manages to simplify the program substantially. For example, the ascending and descending phases in the movement of the ball are discovered automatically, using Lem. 4.2. With a manual insertion of one condition (an inequality), our prototype runs through and outputs a symbolic precondition (which is too long to present here).

Though this bouncing ball example was not fully-automatic, we still find its result interesting and encouraging. The example exhibits the *Zeno behavior*, a big challenge for many verification techniques (see e.g. [13, Section 3.3.3]). We expect our framework to be useful in Zeno-type examples, since: 1) it does not distinguish flow dynamics from jump dynamics (dynamics is always "discrete," or "all flow dynamics is Zeno"); and 2) we do not have a governing notion of time (t is a variable just like others).

Discussion. Efficiency or speed of verification is currently not our primary concern; still we noticed several methods for speeding up. For example, calls to the four precondition discovery strategies could be parallelized. Another performance drawback of the current prototype is that the OCaml program and *Mathematica* communicate via a file written to storage, which takes time. A tighter connection via the script mode usage of *Mathematica*, or *Mathlink*, is currently looked at.

Acknowledgments Thanks are due to all the reviewers for their careful reading and useful comments; particularly to the reviewer who pointed out an error in Lem. 5.8. I.H. is supported by Grants-in-Aid for Young Scientists (A) No. 24680001, JSPS, and by Aihara Innovative Mathematical Modelling Project, FIRST Program, JSPS/CSTP; K.S. is supported by Grants-in-Aid for JSPS Fellows 23·571.

[8] The source code of our precondition generator and the input WHILE$^{\mathtt{dt}}$ programs are available at `http://www-mmm.is.s.u-tokyo.ac.jp/~ichiro/papers/vcgen.tgz`.

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. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: EMSOFT. pp. 49–58 (2009)
3. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: Non-standard semantics of hybrid systems modelers. J. Comput. Syst. Sci. 78(3), 877–910 (2012)
4. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) PLDI. pp. 300–309. ACM (2007)
5. Bliudze, S., Krob, D.: Modelling of complex systems: Systems as dataflow machines. Fundam. Inform. 91(2), 251–274 (2009)
6. Chaudhuri, S., Gulwani, S., Lublinerman, R., NavidPour, S.: Proving programs robust. In: Gyimóthy, T., Zeller, A. (eds.) SIGSOFT FSE. pp. 102–112. ACM (2011)
7. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Jr., W.A.H., Somenzi, F. (eds.) CAV. Lect. Notes Comp. Sci., vol. 2725, pp. 420–432. Springer (2003)
8. Gamboa, R.A., Kaufmann, M.: Nonstandard analysis in ACL2. J. Autom. Reason. 27(4), 323–351 (Nov 2001)
9. Goldblatt, R.: Lectures on the Hyperreals: An Introduction to Nonstandard Analysis. Springer-Verlag (1998)
10. Gopan, D., Reps, T.W.: Guided static analysis. In: Nielson, H.R., Filé, G. (eds.) SAS. Lect. Notes Comp. Sci., vol. 4634, pp. 349–365. Springer (2007)
11. Hurd, A.E., Loeb, P.A.: An Introduction to Nonstandard Real Analysis. Academic Press (1985)
12. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. 20(1), 309–352 (2010)
13. Platzer, A.: Logical Analysis of Hybrid Systems—Proving Theorems for Complex Dynamics. Springer (2010)
14. Platzer, A.: The complete proof theory of hybrid systems. Tech. Rep. CMU–CS–11–144, Carnegie-Mellon Univ., Pittsburgh PA 15213 (2011)
15. Robinson, A.: Non-standard analysis. Princeton Univ. Press (1996)
16. Rodríguez-Carbonell, E., Tiwari, A.: Generating polynomial invariants for hybrid systems. In: Morari, M., Thiele, L. (eds.) HSCC. Lect. Notes Comp. Sci., vol. 3414, pp. 590–605. Springer (2005)
17. Sankaranarayanan, S.: Automatic invariant generation for hybrid systems using ideal fixed points. In: Johansson, K.H., Yi, W. (eds.) HSCC. pp. 221–230. ACM (2010)
18. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: Jones, N.D., Leroy, X. (eds.) POPL. pp. 318–329. ACM (2004)
19. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. Formal Meth. in Sys. Design 32(1), 25–55 (2008)
20. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. Lect. Notes Comp. Sci., vol. 6806, pp. 703–719. Springer (2011)
21. Suenaga, K., Hasuo, I.: Programming with infinitesimals: A while-language for hybrid system modeling. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP (2). Lect. Notes Comp. Sci., vol. 6756, pp. 392–403. Springer (2011)
22. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1993)

A Appendix

A.1 Proofs

Proof of Lem. 2.5 The proof is much like parts of the proofs in [21]. In particular, the 'only if' part can be proved much like in [21, Lem. 4.5]. The proof is presented anyway for completeness.

For the 'if' part: assume a hyperstate $\boldsymbol{\sigma}$ satisfies A; and let $(\boldsymbol{\sigma}|_i)_{i\in\mathbb{N}}$ be an arbitrary sequence representation. By [21, Lem. 4.5], we have $\boldsymbol{\sigma}|_i \models A|_i$ for almost every i. It follows from the assumption that $[\![c|_i]\!](\boldsymbol{\sigma}|_i) \models B|_i$. By [21, Lem. 3.10 & 4.5] this yields $[\![c]\!]\boldsymbol{\sigma} \models B$.

For the 'only if' part: assume the negation of the right-hand side. By the definition of ultrafilter, this means:

$$\not\models \{A|_i\}\, c|_i\, \{B|_i\} \qquad \text{for almost every } i,$$

which in turn is equivalent to:

$$\text{for almost every } i\text{, there exists } \sigma_i \in \mathbf{St} \text{ such that } \sigma_i \models A|_i \text{ and } [\![c|_i]\!]\sigma_i \not\models B|_i.$$

Pick such $\sigma_i \in \mathbf{St}$ for each such i; for other i's let σ_i be, say, $\bot$. Then let $\boldsymbol{\sigma}$ be the hyperstate $\boldsymbol{\sigma} := [\,(\sigma_i)_i\,]$. This hyperstate $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\sigma} \models A$ and $[\![c]\!]\boldsymbol{\sigma} \not\models B$, by [21, Lem. 3.10 & 4.5]. This contradicts the assumption that $\models \{A\}c\{B\}$. □

Proof of Lem. 4.2 We follow [20] and prove the lemma using the following sublemma.[9]

Sublemma A.1 *Assume a state $\sigma \in \mathbf{St}$ is such that $[\![\overline{h}]\!]\sigma \models b_{\mathrm{c}}$. Then $[\![h[b_{\mathrm{c}}]]\!]\sigma = [\![h[\mathtt{true}]]\!]\sigma$.*

Proof. By induction on the construction of $h \in \mathbf{Cmd}_{[\_]}$.

When $h \equiv \mathtt{if}\ [\_]\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$: we have $\overline{h} \equiv \mathtt{skip}$; hence by the assumption, $\sigma \models b_{\mathrm{c}}$. Therefore

$$\begin{aligned}[\![h[b_{\mathrm{c}}]]\!]\sigma &= [\![\mathtt{if}\ b_{\mathrm{c}}\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!]\sigma \\ &= [\![c_1]\!]\sigma && \text{by } \sigma \models b_{\mathrm{c}} \\ &= [\![\mathtt{if}\ [\mathtt{true}]\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!]\sigma = [\![h[\mathtt{true}]]\!]\sigma\ .\end{aligned}$$

When $h \equiv h'; c$, since $\overline{h} \equiv \overline{h'}$ we have $[\![\overline{h'}]\!]\sigma \models b_{\mathrm{c}}$. Therefore by the induction hypothesis, we have $[\![h'[b_{\mathrm{c}}]]\!]\sigma = [\![h'[\mathtt{true}]]\!]\sigma$. This is used in $(*)$ below.

$$[\![h[b_{\mathrm{c}}]]\!]\sigma = [\![h'[b_{\mathrm{c}}]; c]\!]\sigma = [\![c]\!]\big([\![h'[b_{\mathrm{c}}]]\!]\sigma\big) \overset{(*)}{=} [\![c]\!]\big([\![h'[\mathtt{true}]]\!]\sigma\big) = [\![h[\mathtt{true}]]\!]\sigma\ .$$

[9] The statement here looks different from that of the original lemma [20, Lem. 1]. The latter holds only if the assertion language is sufficiently expressive; the statement here holds regardless.

When $h \equiv c : h'$, since $\overline{h} \equiv c; \overline{h'}$ we have $[\![c; \overline{h'}]\!]\sigma \models b_{\mathrm{c}}$, that is, $[\![\overline{h'}]\!]([\![c]\!]\sigma) \models b_{\mathrm{c}}$. Therefore by the induction hypothesis, we have $[\![h[b_{\mathrm{c}}]]\!]([\![c]\!]\sigma) = [\![h[\mathtt{true}]]\!]([\![c]\!]\sigma)$. This is used in $(*)$ below.

$$[\![h[b_{\mathrm{c}}]]\!]\sigma = [\![h'[b_{\mathrm{c}}]]\!]([\![c]\!]\sigma) \overset{(*)}{=} [\![h'[\mathtt{true}]]\!]([\![c]\!]\sigma) = [\![h[\mathtt{true}]]\!]\sigma \ .$$

When $h \equiv \mathtt{if}\ b\ \mathtt{then}\ h'\ \mathtt{else}\ c$, the pre-hole fragment $\overline{h}$ is $\overline{h} \equiv \mathtt{assert}\ b; \overline{h'}$. Therefore the assumption $[\![\overline{h}]\!]\sigma \models b_{\mathrm{c}}$ is equivalent to:

$$\sigma \not\models b \qquad \text{or} \qquad [\![\overline{h'}]\!]\sigma \models b_{\mathrm{c}} \ . \tag{11}$$

Now assume $\sigma \not\models b$. Then

$$\begin{aligned}
[\![h[b_{\mathrm{c}}]]\!]\sigma &= [\![\mathtt{if}\ b\ \mathtt{then}\ h'[b_{\mathrm{c}}]\ \mathtt{else}\ c]\!]\sigma \\
&= [\![c]\!]\sigma && \text{by } \sigma \not\models b \\
&= [\![\mathtt{if}\ b\ \mathtt{then}\ h'[\mathtt{true}]\ \mathtt{else}\ c]\!]\sigma \\
&= [\![h[\mathtt{true}]]\!]\sigma \ .
\end{aligned}$$

Otherwise, by (11) we have $[\![\overline{h'}]\!]\sigma \models b_{\mathrm{c}}$; by the induction hypothesis this yields $[\![h'[b_{\mathrm{c}}]]\!]\sigma = [\![h'[\mathtt{true}]]\!]\sigma$. This is used in $(*)$ below.

$$\begin{aligned}
[\![h[b_{\mathrm{c}}]]\!]\sigma &= [\![\mathtt{if}\ b\ \mathtt{then}\ h'[b_{\mathrm{c}}]\ \mathtt{else}\ c]\!]\sigma \\
&= [\![h'[b_{\mathrm{c}}]]\!]\sigma && \text{by } \sigma \models b \\
&\overset{(*)}{=} [\![h'[\mathtt{true}]]\!]\sigma \\
&= [\![\mathtt{if}\ b\ \mathtt{then}\ h'[\mathtt{true}]\ \mathtt{else}\ c]\!]\sigma \\
&= [\![h[\mathtt{true}]]\!]\sigma \ .
\end{aligned}$$

The other case when $h \equiv \mathtt{if}\ b\ \mathtt{then}\ c\ \mathtt{else}\ h'$ is similar. This concludes the proof. □

Proof. (Of Lem. 4.2) Let us say that a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_k$ is an *execution trace* of c_0 if:

- $[\![b_{\mathrm{g}}]\!]\sigma_i = \mathrm{tt}$ for each $i \in [0, k)$ and $[\![b_{\mathrm{g}}]\!]\sigma_k = \mathrm{ff}$;
- $\sigma_{i+1} = [\![h[b_{\mathrm{c}}]]\!]\sigma_i$ for each $i \in [0, k)$; and
- $\sigma_i \neq \bot$ for each $i \in [0, k]$.

By the definition of denotational semantics, it is obvious that $[\![c_0]\!]\sigma \neq \bot$ if and only if an execution trace of c_0 with $\sigma = \sigma_0$ exists; moreover, if that is the case, $[\![c_0]\!]\sigma = \sigma_k$.

Similarly, we define an *execution trace* of c_1 to be a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_k$ such that: there exists $l \in [0, k]$ and

- $[\![b_{\mathrm{g}} \wedge \neg b]\!]\sigma_i = \mathrm{tt}$ for each $i \in [0, l)$ and $[\![b_{\mathrm{g}} \wedge \neg b]\!]\sigma_l = \mathrm{ff}$;
- $[\![b_{\mathrm{g}} \wedge b]\!]\sigma_i = \mathrm{tt}$ for each $i \in [l, k)$ and $[\![b_{\mathrm{g}} \wedge b]\!]\sigma_k = \mathrm{ff}$;
- $\sigma_{i+1} = [\![h[\mathtt{false}]]\!]\sigma_i$ for each $i \in [0, l)$;

– $\sigma_{i+1} = [\![h[\mathtt{true}]]\!]\sigma_i$ for each $i \in [l, k)$; and
– $\sigma_i \neq \bot$ for each $i \in [0, k]$.

The number l designates how many times the body of the first loop in c_1 is executed. It is again obvious that: $[\![c_1]\!]\sigma \neq \bot$ if and only if an execution trace with $\sigma = \sigma_0$ exists; and that if that is the case $[\![c_1]\!]\sigma = \sigma_k$.

It suffices to prove that the above two notions of execution trace (of c_0 and of c_1) coincide. The latter is an easy consequence of the following three properties, which are to be proved.

1. In an execution trace of c_0 we have:

$$\sigma_i \models b \quad \text{implies} \quad \sigma_{i+1} \models b \ , \qquad \text{for each } i \in [0, k-1).$$

2. In an execution trace of c_0 or c_1 we have:

$$\sigma_i \models b \quad \text{implies} \quad [\![h[b_{\mathrm{c}}]]\!]\sigma_i = [\![h[\mathtt{true}]]\!]\sigma_i \ , \qquad \text{for each } i \in [0, k).$$

3. In an execution trace of c_0 or c_1 we have:

$$\sigma_i \not\models b \quad \text{implies} \quad [\![h[b_{\mathrm{c}}]]\!]\sigma_i = [\![h[\mathtt{false}]]\!]\sigma_i \ , \qquad \text{for each } i \in [0, k).$$

To prove the property 1., assume $\sigma_i \models b_{\mathrm{g}}$ with $i \in [0, k-1)$. By the definition of execution trace of c_0 we have $\sigma_i \models b$; therefore $\sigma_i \models b_{\mathrm{g}} \wedge b$. By the assumption that $\models \{b_{\mathrm{g}} \wedge b\}\, h[b_{\mathrm{c}}]\, \{\neg b_{\mathrm{g}} \vee b\}$ this implies

$$[\![h[b_{\mathrm{c}}]]\!]\sigma_i \models \neg b_{\mathrm{g}} \vee b \ , \quad \text{that is,} \quad \sigma_{i+1} \models \neg b_{\mathrm{g}} \vee b \ .$$

Now, since $i + 1 \in [0, k)$, by the definition of execution trace we have $\sigma_{i+1} \models b_{\mathrm{g}}$; therefore $\sigma_{i+1} \models b$.

To prove the property 2., by the assumption that $\models \{b\}\, \overline{h}\, \{b_{\mathrm{c}}\}$ we have $[\![\overline{h}]\!]\sigma_i \models b_{\mathrm{c}}$; thus by Lem. A.1 we have $[\![h[b_{\mathrm{c}}]]\!]\sigma_i = [\![h[\mathtt{true}]]\!]\sigma_i$. The property 3. is similar. □

Proof of Lem. 4.5 The lemma follows immediately from the following facts, which are easily proved, and the definition of denotational semantics for while-loops. First, for any command c, if a variable t does not occur in c then

$$[\![c]\!](\sigma[t \mapsto r]) = ([\![c]\!]\sigma)[t \mapsto r] \ ,$$

for any state σ and any real number $r \in \mathbb{R}$. Second, if a Boolean expression $b \in \mathbf{BExp}$ has no occurrence of t,

$$[\![b]\!](\sigma[t \mapsto r]) = [\![b]\!]\sigma \ ,$$

for any state σ and any real number $r \in \mathbb{R}$.

Proof of Lem. 5.4 $A_{\mathrm{rs}} = A'[a'/x]^y$ means that the following holds. Note the formula is $\mathtt{dt}$-free.

$$\models\ A_{\mathrm{rs}}[0/y] \Leftrightarrow A'\ \wedge\ \forall u \in \mathbb{N}.\big(\ A_{\mathrm{rs}}[u+1/y] \Leftrightarrow \big(A_{\mathrm{rs}}[u/y]\big)[a'/x]\ \big) \tag{12}$$

From this the following is trivial.

$$\models \forall d \in \mathbb{R}.\ \big(\ A_{\mathrm{rs}}[0/y] \Leftrightarrow A'\ \wedge\ \forall u \in \mathbb{N}.\big(\ A_{\mathrm{rs}}[u+1/y] \Leftrightarrow \big(A_{\mathrm{rs}}[u/y]\big)[a'/x]\ \big)\ \big) \tag{13}$$

Here we appeal to the *transfer principle* (Prop. 2.7) and obtain, in $\textsc{Assn}^{\mathtt{dt}}$,

$$\models \forall d \in {}^*\mathbb{R}.\ \big(\ {}^*A_{\mathrm{rs}}[0/y] \Leftrightarrow {}^*A'\ \wedge\ \forall u \in {}^*\mathbb{N}.\big(\ {}^*A_{\mathrm{rs}}[u+1/y] \Leftrightarrow \big({}^*A_{\mathrm{rs}}[u/y]\big)[a'/x]\ \big)\ \big)\ . \tag{14}$$

Now that we are in $\textsc{Assn}^{\mathtt{dt}}$, we take the instance of (14) with $d = \mathtt{dt}$ and obtain

$$\begin{aligned}\models \Big(\ & ({}^*A_{\mathrm{rs}}[\mathtt{dt}/d])[0/y] \Leftrightarrow A\ \wedge \\ & \forall u \in {}^*\mathbb{N}.\big(\ ({}^*A_{\mathrm{rs}}[\mathtt{dt}/d])[u+1/y] \Leftrightarrow \big(\ ({}^*A_{\mathrm{rs}}[\mathtt{dt}/d])[u/y]\big)[a/x]\ \big)\ \Big)\ ,\end{aligned} \tag{15}$$

where we used obvious syntactic equalities ${}^*A'[\mathtt{dt}/d] \equiv A$ and $a'[\mathtt{dt}/d] \equiv a$. The condition (15) means that ${}^*A_{\mathrm{rs}}[\mathtt{dt}/d]$ is indeed a homogeneous representation (Def. 5.3). □

Proof of Lem. 5.6 It suffices to show $\models \{a_{\mathrm{c}} < 0 \wedge b\}\ x := a\ \{a_{\mathrm{c}} < 0\}$; this follows immediately from the side condition. □

Proof of Lem. 5.7 We claim that the following formula P

$$\begin{aligned} P\ &:\equiv\ \exists z \in \mathbb{N}.\, Q \qquad \text{where} \\ Q\ &:\equiv\ \forall u \in \mathbb{N}.\,\big(\ u < z \Rightarrow b[a/x]^u\ \big)\ \wedge\ \neg b[a/x]^z\ \wedge\ A[a/x]^z \end{aligned}$$

is a loop invariant.

Firstly we prove that P is indeed a loop invariant. Assume $\sigma \models b \wedge P$. Then there exists a natural number $n_{\mathrm{lc}} \in \mathbb{N}$ such that $\sigma \models Q[n_{\mathrm{lc}}/z]$. If $n_{\mathrm{lc}} = 0$ then $\sigma \models b \wedge \neg b$; this forces $\sigma = \top$ and hence trivially $[\![c]\!]\sigma \models P$.

Now assume $n_{\mathrm{lc}} > 0$; we aim at showing $[\![c]\!]\sigma \models P$. By $\sigma \models \neg b[a/x]^{n_{\mathrm{lc}}}$, $n_{\mathrm{lc}} > 0$ and Cond. 2, we have $[\![c]\!]\sigma \models \neg b[a/x]^{n_{\mathrm{lc}}-1}$. This means there exists at least one natural number $n \in \mathbb{N}$ such that $[\![c]\!]\sigma \models \neg b[a/x]^n$; take the smallest such and denote it by n'_{lc}. Obviously $n'_{\mathrm{lc}} \leq n_{\mathrm{lc}} - 1$; by Cond. 5 we have $\sigma \models A[a/x]^{n'_{\mathrm{lc}}+1}$; this further yields $[\![c]\!]\sigma \models A[a/x]^{n'_{\mathrm{lc}}}$ by Cond. 4. By the minimalness requirement on n'_{lc} we have $[\![c]\!]\sigma \models \forall u \in \mathbb{N}.\big(\ u < n'_{\mathrm{lc}} \Rightarrow b[a/x]^u\ \big)$. Now we have shown $[\![c]\!]\sigma \models Q[n'_{\mathrm{lc}}/z]$, therefore $[\![c]\!]\sigma \models P$.

Secondly we prove that the precondition $P' :\equiv A[a/x]^{a_{\mathrm{i}}+1} \wedge a_{\mathrm{i}} \geq 0$ in (10) subsumes P. Assume $\sigma \models P'$. By Cond. 1 and 3 we have $\sigma \models \neg b[a/x]^{n''_{\mathrm{lc}}}$, where $n''_{\mathrm{lc}} := \lceil [\![a_{\mathrm{i}}]\!]\sigma \rceil$ is the smallest natural number that is larger than or equal to the non-negative real number $[\![a_{\mathrm{i}}]\!]\sigma$. Therefore at least for one $n \in \mathbb{N}$ we have $\sigma \models \neg b[a/x]^n$; take the smallest such and denote it by n'''_{lc}. Obviously $n'''_{\mathrm{lc}} \leq n''_{\mathrm{lc}} = \lceil [\![a_{\mathrm{i}}]\!]\sigma \rceil$. Then by $\sigma \models P'$ and Cond. 5 we have $\sigma \models A[a/x]^{n'''_{\mathrm{lc}}}$ and this implies $\sigma \models Q[n'''_{\mathrm{lc}}/z]$, hence $\sigma \models P$.

Lastly we have to show $\models P \wedge \neg b \Rightarrow A$, which is obvious. □

Proof of Lem. 5.8 It is well-known that the following are equivalent (see e.g. [9, Cor. 7.1.2]): 1) a function f is continuous at $c \in \mathbb{R}$; 2) $f(x) \simeq f(c)$ whenever $x \simeq c$. Here $\simeq$ means that the two hyperreal numbers are *infinitely* (or *infinitesimally*) *close* to each other; and the function f in 2) is to be precise the *-transfer ${}^*f$ of the original function [9, §3.11]. We have, for each hyperstate $\boldsymbol{\sigma} \neq \bot$,

$$[\![a]\!]\boldsymbol{\sigma} = [\![a[d/\mathtt{dt}]]\!](\boldsymbol{\sigma}[d \mapsto \mathtt{dt}]) \overset{(*)}{\simeq} [\![a[d/\mathtt{dt}]]\!](\boldsymbol{\sigma}[d \mapsto 0]) = [\![\big(\, a[d/\mathtt{dt}]\,\big)[0/d]]\!]\boldsymbol{\sigma} \ , \tag{16}$$

where $(*)$ holds from the continuity assumption (Cond. 2.) and that $\mathtt{dt} \simeq 0$.

The expression $\big(a[d/\mathtt{dt}]\big)[0/d]$ is $\mathtt{dt}$-free; thus by Cond. 1., the value $[\![\big(a[d/\mathtt{dt}]\big)[0/d]]\!]\boldsymbol{\sigma}$ is in fact a standard real number. It is straightforward that, given a standard real number r and a hyperreal number r' such that $r \simeq r'$, we have $r < 0$ imply $r' < 0$. Thus (16) yields the claim. □

A.2 The Full *Iteration Count* Strategy

Lemma A.2 ((Full) iteration count) *Let a_{ii} and a_{io} be arithmetic expressions. Assume the following conditions.*

1. $\models b_{\mathrm{o}}\big([f/y]^{a_{\mathrm{ii}}}[e/x]\big)^{a_{\mathrm{io}}}$
2. $\models \forall u, u', v, v', w, w' \in \mathbb{R}.\, \big(0 \leq u' < u \wedge 0 \leq v' < v \wedge 0 \leq w' < w \wedge A\big([f/y]^{u}[e/x]\big)^{v}[f/y]^{w} \Rightarrow A\big([f/y]^{u'}[e/x]\big)^{v'}[f/y]^{w'}\big)$
3. $\models \forall u, u', v, v' \in \mathbb{R}.\, \big(0 \leq u < u' \wedge 0 \leq v < v' \wedge \neg b_{\mathrm{o}}\big([f/y]^{v}[e/x]\big)^{u} \Rightarrow \neg b_{\mathrm{o}}([f/y]^{v'}[e/x])^{u'}\big)$
4. $\models \neg b_{\mathrm{i}}[f/y]^{a_{\mathrm{ii}}}$
5. $\models \forall u, v \in \mathbb{R}.\, (u < v \wedge \neg b_{\mathrm{i}}[f/y]^{u} \Rightarrow \neg b_{\mathrm{i}}[f/y]^{v})$
6. $\models \forall u \in \mathbb{R}.\, (0 \leq u < a_{\mathrm{ii}} \Rightarrow b_{\mathrm{i}}[f/y]^{u})$

Then we have

$$\Big\{\ A\big(\,[f/y]^{a_{\mathrm{ii}}+1}[e/x]\,\big)^{a_{\mathrm{io}}+1} \wedge a_{\mathrm{ii}} \geq 0 \wedge a_{\mathrm{io}} \geq 0\ \Big\}\ c_0\ \{A\}\ , \quad \textit{where}\quad c_0 \ :\equiv\ \mathtt{while}\ b_{\mathrm{o}}\ \mathtt{do}\ \big(x := e;\quad \mathtt{while}\ b_{\mathrm{i}}\ \mathtt{do}\ y := f\big)\ . \tag{17}$$

Here expressions like $A\big([f/y]^{v}[e/x]\big)^{u}$ are the obvious generalization of Def. 5.1.

In the proof we will use two lemmas: one is Lem. 5.7 and the other is Sublem. A.3 below. Each of them is about estimating the number of iterations for a single loop.

The operator $\lceil \_ \rceil$ occurs in the proofs. We note that its use is confined to the backend and it does not occur in Lem. A.2. Therefore in the actual verification using Lem. A.2, we do not have to deal with the discontinuous (hence inconvenient) function $\lceil \_ \rceil$.

Sublemma A.3 *Let a_{i} be an arithmetic expression. Assume the following conditions.*

1. $\models \neg b[a/x]^{a_{\mathrm{i}}}$
2. $\models \forall u \in \mathbb{N}.\, (0 \leq u < a_{\mathrm{i}} \Rightarrow b[a/x]^{a_{\mathrm{i}}})$
3. *For some fresh variable y,* $\models \big\{b[a/x]^{y+1}\big\}\, c\, \big\{b[a/x]^{y}\big\}$
4. *For some fresh variable y,* $\models \big\{\neg b[a/x]^{y+1}\big\}\, c\, \big\{\neg b[a/x]^{y}\big\}$
5. $\models \forall u, v \in \mathbb{R}.\, (a_{\mathrm{i}} \leq u < v < a_{\mathrm{i}} + 1 \wedge \neg b[a/x]^{u} \Rightarrow \neg b[a/x]^{v})$

6. For some fresh variable y, $\models \{A[a/x]^{y+1}\}\, c\, \{A[a/x]^{y}\}$

Then we have

$$\models \quad \{A[a/x]^{\lceil a_{\mathrm{i}} \rceil} \wedge a_{\mathrm{i}} \geq 0\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{A\}\ . \tag{18}$$

Proof. The proof goes much like the one for Lem. 5.7, with the same invariant P. Cond. 1–4 forces a_{i} to be a good estimate of the number of iterations; in fact we can prove the latter to be precisely $\lceil a_{\mathrm{i}} \rceil$. □

Proof. (Of Lem. A.2) It suffices to prove

$$\models \quad A\big([f/y]^{a_{\mathrm{ii}}+1}[e/x]\big)^{a_{\mathrm{io}}+1} \Rightarrow A\big([f/y]^{\lceil a_{\mathrm{ii}} \rceil}[e/x]\big)^{a_{\mathrm{io}}+1} \qquad \text{and} \tag{19}$$

$$\models \quad \Big\{\, A\big([f/y]^{\lceil a_{\mathrm{ii}} \rceil}[e/x]\big)^{a_{\mathrm{io}}+1} \wedge a_{\mathrm{ii}} \geq 0 \wedge a_{\mathrm{io}} \geq 0 \,\Big\}\ c_0\ \{A\}\ . \tag{20}$$

Cond. 2 immediately yields (19). The remaining subgoal (20) is proved using Lem. 5.7; its side conditions are new proof obligations. Among them, the following two do not follow immediately from the assumptions of Lem. A.2:

$$\models \quad \Big\{\, A\big([f/y]^{\lceil a_{\mathrm{ii}} \rceil}[e/x]\big)^{y+1} \wedge a_{\mathrm{ii}} \geq 0 \,\Big\}\ x := e;\ \texttt{while}\ b_{\mathrm{i}}\ \texttt{do}\ y := f \quad \big\{A\big([f/y]^{\lceil a_{\mathrm{ii}} \rceil}[e/x]\big)^{y}\big\}\ , \tag{21}$$

$$\models \quad \Big\{\, \neg b_{\mathrm{o}}\big([f/y]^{\lceil a_{\mathrm{ii}} \rceil}[e/x]\big)^{y+1} \wedge a_{\mathrm{ii}} \geq 0 \,\Big\}\ x := e;\ \texttt{while}\ b_{\mathrm{i}}\ \texttt{do}\ y := f \quad \big\{\neg b_{\mathrm{o}}\big([f/y]^{\lceil a_{\mathrm{ii}} \rceil}[e/x]\big)^{y}\big\}\ . \tag{22}$$

The obligation (21) is discharged via Lem. 5.7; the obligation (22) is via Lem. A.3. □