Derivation of Sequential, Real-Time, Process-Control Programs

[illegible] Keith Marzullo*
11-61-CR Fred B. Schneider**
Navin Budhiraja*

36457

p.15

91-1217
July 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 Contract N00140-87-C-8904 and a grant from Xerox Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.
**Supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under Grant No. CCR-9003440, and Digital Equipment Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

Derivation of Sequential,

Real-Time, Process-Control Programs

Keith Marzullo[*]
Fred B. Schneider[**]
Navin Budhiraja[*]

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

The use of weakest-precondition predicate transformers in the derivation of sequential, process-control software is discussed. Only one extension to Dijkstra's calculus for deriving ordinary sequential programs was found to be necessary: function-valued auxiliary variables. These auxiliary variables are needed for reasoning about states of a physical process that exist during program transitions.

1. Introduction

For the past few years, we have been exploring the use of assertional reasoning in the construction of process-control software. Our intent was to employ an existing method, perhaps with a few extensions, and systematically derive process-control programs from specifications. Use of an existing method had both a scientific and a pragmatic motivation. The scientific motivation was based on our expectation that the difficulties we encountered by using an extant method would provide insights into what distinguishes

[*] Supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 Contract N00140-87-C-8904 and a grant from Xerox Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

[**] Supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under Grant No. CCR-9003440, and Digital Equipment Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

process-control programs from ordinary sequential and concurrent programs. The pragmatic motivation was that extending a well understood method was likely to be easier than developing a new one.

Our investigations have been structured as a series of experiments. Each experiment is based on a simple process-control problem that (we feel) epitomizes some aspect of process-control programming. We started with the simplest process-control problem imaginable—a sequential control-program running on a single, fault-free processor. By reading sensors and writing to actuators, this program controls an on-going physical process. Solving such a problem requires reasoning about control-program execution times, something that has long been considered an integral part of process-control programming. We are well aware, however, that any conclusions from this experiment would have to be regarded as tentative. By considering a sequential control-program, problems arising due to resource contention are avoided; and by assuming a fault-free processor, complications associated with implementing fault-tolerance are being ignored.

Simplifying assumptions not withstanding, our first experiment did lead to some insights about the use of assertional reasoning in writing process-control programs. These insights are the subject of this paper. In section 2, we describe extensions to Dijkstra's weakest-precondition calculus [2] [3] that we found necessary for deriving sequential process-control programs. Section 3 illustrates the use of these extensions and the calculus by giving an example derivation of a control program. Conclusions appear in section 4.

2. Using Weakest Preconditions with Physical Processes

Process-control problems are often specified in terms of restrictions on permissible states of some physical system. By setting actuators to manipulate the process being controlled, a *control program* ensures that none of these proscribed states is ever entered. The actions of the control program are, therefore, closely linked to the state of the physical process being controlled. Consequently, when deriving a control program, it is necessary to reason about both the program state and the state of the physical process being controlled.

Assertional methods for deriving programs are based on manipulating logical formulae, called *assertions*, that characterize sets of program states. One way to employ assertional methods in the design of a process-control program is to augment the program state space so that it includes information about the state of the physical process being controlled. Doing so, however, requires extending the rules used to reason about program execution, as follows.

(1) While a program statement is executed, changes occur to the state of the physical process being controlled. Rules characterizing the effects of program execution must be modified to reflect these other state changes.

(2) Statements whose execution involves interaction with sensors and/or actuators must be axiomatized as rules relating states before, during, and after execution.

The remainder of this section discusses these extensions.

2.1. Reality Variables

The state space of physical system is usually defined by a collection of *state components*, each of which is indexed by some independent (physical) parameters. For example, the state of a railroad train at a time T can be characterized by its position $X(T)$, its speed $V(T)$, and its acceleration $A(T)$. Note that the choice of time as the independent parameter is arbitrary. If its velocity is always greater than 0, then a train at position X could equally well be described by time $T(X)$, speed $V(X)$, and acceleration $A(X)$. As physicists learned long ago, quantities that are convenient for the task at hand should be selected as the independent parameters.

The state space of a program can be augmented to include the state of a physical process. For each state component Q_i, we add to the program state space a function-valued program variable q_i, called a *reality variable*.[1] Each reality variable replicates (in the program's state space) information about a physical system during program execution. Initially, the domain of a reality variable q_i will be empty; as the independent parameter P_i for Q_i changes, the domain of q_i is extended to include the values over which P_i has ranged. Reality variables are entirely fictional. They allow us to describe and reason about the state of a physical system by using assertions, but they are not actually maintained in memory. Thus, they are a form of auxiliary variable [1].

In order to define and manipulate expressions involving function-valued program variables, like reality variables, it will be convenient to have some notation. Following [2], given a function f with domain $dom(f)$, the function expression

$$(f;\ x \in D : g(x))$$

is defined to be a function whose domain is $dom(f) \cup D$ and whose value at any point a is $g(a)$ if $a \in D$ and $f(a)$ otherwise. As a notational convenience, we define:

$$(f;\ x \in D : g(x);\ x \in D' : h(x)) \ = \ ((f;\ x \in D : g(x));\ x \in D' : h(x))$$

And, in specifying domains, we use the notation *low*.. ***high*** to denote the set $\{a \mid low \le a \le high\}$.

[1]In the sequel, we use upper-case identifiers to denote (physical) state components and the corresponding lower-case identifier to denote the reality variables that model these.

2.2. Preserving the Fiction: Updating Reality Variables

The state of a physical system is changed by a physical process. Typically, the changes can be characterized by a set of equations relating the current values of various state components to their recent values. We cannot expect a physical process to update the reality variables being used in modeling the state of a physical system. And, since the weakest-precondition calculus is based on the presumption that all changes to the truth of an assertion are the result of program execution, we have no choice but to regard the program itself as performing updates to reality variables. Program statements can compute these updates by using the equations that characterize the way the physical state components change.

Consider some physical state component $Q(P)$ being modeled by a reality variable $q(p)$, and suppose that as long as no actuator changes during some interval from P to $P+\delta$, changes to Q are characterized by the following continuous equation.

(2.1) $\quad Q(P+\Delta) = \mathcal{F}(Q(P), \Delta) \quad \text{for } 0 \le \Delta \le \delta$

Let $\langle S\rangle_\delta$ denote a statement whose execution coincides with a change of δ by parameter P. Then, execution of $\langle S\rangle_\delta$ is equivalent to executing S and, as part of the same atomic action, changing p and q in accordance with (2.1). This state change is modeled by a program fragment:

$$S;\ \ p, q := p+\delta, (q;\ i \in p \,..\, p+\delta\colon \mathcal{F}(q(p), i-p))$$

Using the weakest-precondition predicate transformers for multiple-assignment and statement composition, we obtain the following predicate transformer characterization for $\langle S\rangle_\delta$.

$$\begin{array}{ll} & wp(\langle S\rangle_\delta, R) \\ = & \qquad \text{«}wp \text{ definition of ";"»} \\ & wp(S, wp(p, q := p+\delta, (q;\ i \in p \,..\, p+\delta\colon \mathcal{F}(q(p), i-p)), R)) \\ = & \qquad \text{«}wp \text{ definition of ":="»} \\ & wp(S, R^{p,\,q}_{p+\delta,\,(q;\ i \in p \,..\, p+\delta\colon \mathcal{F}(q(p), i-p))}) \end{array}$$

Notice that when the independent parameter δ in $\langle S\rangle_\delta$ models the passage of time, $\langle S\rangle_\delta$ is a statement that executes for δ seconds. The definition of $wp(\langle S\rangle_\delta, R)$ then asserts that after executing $\langle S\rangle_\delta$ the current time has been incremented by δ and all other reality variables have been updated as if δ seconds had elapsed. However, our characterization of $\langle S\rangle_\delta$ also allows the independent parameter δ to be a quantity other than time, making it possible to reason in the coordinate system best suited for the problem at hand. Also notice that, according to our weakest precondition characterization of $\langle S\rangle_\delta$, an ordinary statement S must be regarded as being equivalent to $\langle S\rangle_0$. This is because

$$wp(\langle S\rangle_0, R) \ = \ wp(S, R)$$

holds, since $\mathcal{F}(q(p), 0) = q(p)$ according to (2.1).

To illustrate the use of $wp(\langle S\rangle_\delta, R)$ in an actual process-control programming problem, suppose we are interested in controlling the speed of a railroad train. Define reality variable $v(x)$ to be the speed of the train when it is at a given position x. From Newton's Laws of Motion, we know that if the train does not accelerate during an interval of δ seconds, then reality variable v can be characterized by the following equation:

(2.2) $\quad v(x+\Delta) = v(x) \quad \text{for } 0 \le \Delta \le v(x) * \delta$

Thus, according to our definition for $wp(\langle S\rangle_\delta, R)$, we have the following weakest precondition characterization for a statement $\langle S\rangle_\delta$ that takes duration δ seconds and is executed while a train is not accelerating.

$$\begin{array}{ll} & wp(\langle S\rangle_\delta, R) \\ = & \qquad «(2.2) \text{ and } wp \text{ definition for } \langle S\rangle_\delta» \\ & wp(S, R^{x,\ v}_{x+v(x)*\delta,\ (v;\ l \in x..x+v(x)*\delta:\ v(x))}) \end{array}$$

2.3. Interacting with a Physical Process

To have broad applicability, a method for reasoning about process-control programs must not restrict the types of sensors and actuators that it can handle. Rules for reasoning about sensors and actuators can be derived by

(1) modeling interactions with sensors and actuators by statements that read and update reality variables, and then

(2) using the rules provided for reasoning about ordinary statements to derive rules for reasoning about these models.

As long as reality variables correctly model the physical process, the resulting rules will be sound and can be used to reason about how a control program interacts with the process it controls.

To illustrate how sensors and actuators are modeled, we return to railroad control. Consider an actuator **go**(t) and a sensor **await**(c). Executing **go**(t) causes the train to accelerate/decelerate with some maximum constant acceleration ACC (say) until target speed t is reached; execution terminates only when the train reaches its target speed. **await**(c), if invoked while the train is not accelerating, delays execution of a program until the train is at location c.[2]

Define $Vlen(u, t)$ to be the distance that a train travels while it is accelerating from a speed u to target speed t:

$$Vlen(u, t) = |(u^2 - t^2)/(2 * ACC)|$$

[2]If **go**(t) is the only actuator that can cause acceleration, then the condition that **await**(c) is never executed while the train is accelerating is equivalent to stipulating that a train is controlled by a single sequential program.

Define $Vat(u, t, x)$ to be the speed of a train after having traveled x meters, $0 \le x \le Vlen(u, t)$, from the point at which it started accelerating from speed u to t:

$$Vat(u, t, x) = \begin{cases} \sqrt{u^2 + 2 * x * ACC} & \text{if } u < t \\ \sqrt{u^2 - 2 * x * ACC} & \text{if } u > t \\ u \quad \text{if } t = u \end{cases}$$

The effect of executing go(t) can be modeled as an update to reality variables x and v. The value of x is increased by $Vlen(v(x), t)$ and the domain of v is extended to include $x \,..\, x + Vlen(v(x), t)$:

$$\begin{aligned} \textbf{go}(t)\text{:}\quad & x, v := x + Vlen(v(x), t), \\ & \qquad (v;\ l \in x \,..\, x + Vlen(v(x), t) : Vat(v(x), t, l - x)) \end{aligned}$$

This multiple-assignment statement model provides a basis for calculating $wp(\textbf{go}(t), R)$:

$$\begin{aligned} & wp(\textbf{go}(t), R) \\ = & \quad \text{«model of } \textbf{go}(t)\text{»} \\ & wp(x, v := x + Vlen(v(x), t), \\ & \qquad (v;\ l \in x \,..\, x + Vlen(v(x), t) \text{: } Vat(v(x), t, l - x)), R) \\ = & \quad \text{«}wp \text{ definition of ":="»} \\ & R^{x,\, v}_{x + Vlen(v(x), t),\, (v;\ l \in x \,..\, x + Vlen(v(x), t) \text{: } Vat(v(x), t, l - x))} \end{aligned}$$

Similarly, **await**(c) can be modeled by an alternative command:

$$\textbf{await}(c)\text{:}\quad \textbf{if}\ x \le c \land 0 < v(x) \rightarrow x, v := c, (v;\ l \in x \,..\, c \text{: } v(x))\ \textbf{fi}$$

Our model for **await**(c) updates reality variables x and v if $x \le c$ and $0 < v(x)$ hold; otherwise, it delays forever. Using the weakest precondition for **if**, we can calculate a weakest precondition predicate transformer for **await**(c):

$$\begin{aligned} & wp(\textbf{await}(c), R) \\ = & \quad \text{«model of } \textbf{await}(c)\text{»} \\ & wp(\textbf{if}\ x \le c \land 0 < v(x) \rightarrow x, v := c, (v;\ l \in x \,..\, c \text{: } v(x))\ \textbf{fi}, R) \\ = & \quad \text{«}wp \text{ definition of } \textbf{if}\text{»} \\ & x \le c \land 0 < v(x) \\ & \quad \land (x \le c \land 0 < v(x) \Rightarrow wp(x, v := c, (v;\ l \in x \,..\, c \text{: } v(x)), R)) \\ = & \quad \text{«}wp \text{ definition of ":=" and predicate logic»} \\ & x \le c \land 0 < v(x) \land R^{x,\, v}_{c,\, (v;\ l \in x \,..\, c \text{: } v(x))} \end{aligned}$$

3. An Example

Other than the extensions mentioned above, the methodology of [2] and [3] for deriving ordinary sequential programs can be used, unchanged, for deriving sequential process-control programs. In this section, we illustrate that methodology with a simple railroad-control problem.

Railroad tracks are typically partitioned into segments, called *blocks*. Each block i, has an associated starting location b_i and ending location b_{i+1}, where $b_i \le b_{i+1}$, and a range of permissible speeds $mn_i .. mx_i$, where $0 \le mn_i < mx_i$. Desired is a program to control the speed of a point train[3] so that it travels from b_0 to b_n, maintaining safe speeds along the way. Use **go**(t) and **await**(c), as defined above, for interactions between a single sequential control program and the train.

First, we formalize the problem. The train has made a safe passage from location a to b provided the following holds.

$$\begin{aligned} \mathit{Safe}(a,b)\colon\ & a..b \sqsubseteq dom(v) \\ & \wedge\ (\forall l\colon a \le l \le b\colon b_i \le l \le b_{i+1} \Rightarrow mn_i \le v(l) \le mx_i) \end{aligned}$$

The first conjunct of $\mathit{Safe}(a,b)$ asserts that the train has actually traveled from a to b, and the second conjunct asserts that the train's speed satisfied the restrictions associated with each block it occupied. Using $\mathit{Safe}(a,b)$, we can specify the above railroad control problem in terms of weakest preconditions:

(3.1) $\quad x=b_0 \wedge v=(;\ b_0{:}v_0) \Rightarrow wp(S, \mathit{Safe}(b_0,b_n) \wedge x=b_n)$

This formula constrains S to be a program that terminates with the train at location b_n after having traveled at safe speeds to get there, provided S is started with the train at location b_0 traveling with speed v_0.[4]

3.1. A First Try

Having formalized the specification for a correct control program S, we now proceed with the derivation. The universal quantifier in conjunct $\mathit{Safe}(b_0,b_n)$ of the result assertion is a tip-off that S should be structured as a loop. Thus, we employ a standard hueristic from [2]—replacing a constant by a variable—and derive a loop invariant from the result assertion. Replacing n in the result assertion by a new program variable h (for "here") we get:

$$I\colon\ \mathit{Safe}(b_0,b_h)\ \wedge\ x=b_h\ \wedge\ 0 \le h \le n$$

Since $I \wedge h=n$ implies result assertion $\mathit{Safe}(b_0,b_n) \wedge x=b_n$, we conclude that the loop guard must be $h \ne n$ (or something that implies $h \ne n$) and conjecture that S has the following structure:

[3]Assuming a point train is not fundamental. It merely simplifies some of the derivation that follows. By using a configuration space transformation [4], the control problem for a length L train can be transformed to a control problem for a point train on a track with additional blocks.

[4]If the conjunct $x=b_n$ is omitted from the result assertion, then it would be permissible for control program S to terminate long after the train had passed point b_n. We have deemed such behavior unacceptable and so our specification prohibits it.

S: S_1 $\{I\}$
  do $h \neq n \rightarrow \{I \wedge h \neq n\}$ S_2 $\{I\}$ od $\{h=n \wedge I\}$
  $\{Safe(b_0, b_n)\}$

Program S will satisfy its specification provided we find statements S_1 and S_2 that satisfy the following specifications.

(3.2) $x=b_0 \wedge v=(;\ b_0{:}v_0) \Rightarrow wp(S_1, I)$

(3.3) $I \wedge h \neq n \Rightarrow wp(S_2, I)$

Formula (3.2) is the specification for the loop initialization; (3.3) is the specification for the loop body.

According to specification (3.2), S_1 must establish I. Observe that an easy way to establish I is by setting h to 0. So, we use wp to calculate an assertion that must hold before executing $h := 0$ in order for I to hold afterwards.

$wp(h := 0, I)$
= «wp definition of ":="»
$(Safe(b_0, b_h) \wedge x=b_h \wedge 0 \leq h \leq n)^h_0$
= «textual substitution»
$Safe(b_0, b_0) \wedge x=b_0$
= «definition of $Safe(a, b)$»
$b_0 \in dom(v) \wedge mn_0 \leq v(b_0) \leq mx_0 \wedge x=b_0$

Notice that $x=b_0 \wedge v=(;\ b_0{:}v_0)$, the antecedent of specification (3.1) for S, implies $wp(h := 0, I)$ only if $mn_0 \leq v_0 \leq mx_0$. Thus, executing $h := 0$ establishes the loop invariant only under certain conditions—the initial speed of the train must be safe for travel in block b_0. We identify this requirement explicitly.

Assumption AS1. $mn_0 \leq v_0 < mx_0$

In retrospect, this requirement should not be surprising. It is worth noting, however, that this implicit assumption was exposed simply by adhering to a rigorous calculus in deriving the program. Including this assumption in the program we have developed so far, we get:

S: $\{x=b_0 \wedge v=(;\ b_0{:}v_0) \wedge AS1\}$
  $h := 0$ $\{I\text{: } Safe(b_0, b_h) \wedge x=b_h \wedge 0 \leq h \leq n\}$
  do $h \neq n \rightarrow \{I \wedge h \neq n\}$ S_2 $\{I\}$ od $\{h=n \wedge I\}$
  $\{Safe(b_0, b_n)\}$

We now refine S_2, the body of the loop. Based on our choice of guard, we know that the loop will terminate when h equals n. Initially, h is 0. Thus, for S_2 to make progress towards termination, h must be increased; and for S_2 to satisfy specification (3.3), S_2 must reestablish I. To investigate the feasibility of increasing h by 1, we calculate $wp(h := h+1, I)$.

$$\begin{array}{ll}
 & wp(h := h+1, I) \\
= & \quad \text{«}wp \text{ definition of ":="»} \\
 & (Safe(b_0, b_h) \wedge x=b_h \wedge 0 \le h \le n)^h_{h+1} \\
= & \quad \text{«textual substitution»} \\
 & Safe(b_0, b_{h+1}) \wedge x=b_{h+1} \wedge 0 \le h+1 \le n \\
= & \quad \text{«}a \le b \le c \Rightarrow (Safe(a,c) = (Safe(a,b) \wedge Safe(b,c)))\text{»} \\
 & Safe(b_0, b_h) \wedge Safe(b_h, b_{h+1}) \wedge x=b_{h+1} \wedge 0 \le h+1 \le n
\end{array}$$

Since $I \wedge h \ne n$ holds at the start of S_2, we know that the first and last conjuncts of $wp(h := h+1, I)$ hold before S_2 executes. We must, therefore, arrange for the remaining conjuncts to hold.

$$\begin{array}{lll}
 & & Safe(b_h, b_{h+1}) \wedge x=b_{h+1} \\
 & = & \quad \text{«definition of } Safe(a,b)\text{»} \\
 & & b_h \,..\, b_{h+1} \subseteq dom(v) \\
 & & \quad \wedge\ (\forall l\colon\ b_h \le l \le b_{h+1}\colon\ b_i \le l \le b_{i+1} \\
 & & \qquad\qquad \Rightarrow mn_i \le v(l) \le mx_i) \wedge x=b_{h+1} \\
 & = & \quad \text{«}x=b_{h+1} \Rightarrow b_0 \,..\, b_{h+1} \subseteq dom(v)\text{»} \\
 & & (\forall l\colon\ b_h \le l \le b_{h+1}\colon\ b_i \le l \le b_{i+1} \Rightarrow mn_i \le v(l) \le mx_i) \wedge x=b_{h+1} \\
 & = & \quad \text{«predicate logic»} \\
(3.4) & & (\forall l\colon\ b_h \le l < b_{h+1}\colon\ b_i \le l \le b_{i+1} \Rightarrow mn_i \le v(l) \le mx_i) \\
 & & \quad \wedge\ \max(mn_h, mn_{h+1}) \le v(b_{h+1}) \le \min(mx_h, mx_{h+1}) \wedge x=b_{h+1}
\end{array}$$

We consider the final conjunct first. It is easy to establish this conjunct by executing **await**(b_{h+1}), so we compute:

$$\begin{array}{lll}
 & & wp(\mathbf{await}(b_{h+1}),\ (3.4)\,) \\
 & = & \quad \text{«}wp \text{ calculus»} \\
 & & x \le b_{h+1} \wedge 0 < v(x) \\
 & & \quad \wedge\ ((\forall l\colon\ b_h \le l < b_{h+1}\colon\ b_i \le l \le b_{i+1} \Rightarrow mn_i \le v(l) \le mx_i) \\
 & & \quad \wedge\ \max(mn_h, mn_{h+1}) \le v(b_{h+1}) \le \min(mx_h, mx_{h+1}) \\
 & & \quad \wedge\ x=b_{h+1})^{x,\,v}_{b_{h+1},\,(v;\ l \in x\,..\,b_{h+1}\colon v(x))} \\
 & = & \quad \text{«textual substitution and simplification»} \\
(3.5) & & x \le b_{h+1} \wedge 0 < v(x) \\
 & & \quad \wedge\ (\forall l\colon\ b_h \le l < b_{h+1}\colon\ b_i \le l \le b_{i+1} \Rightarrow \\
 & & \qquad\qquad mn_i \le (v;\ l \in x\,..\,b_{h+1}\colon v(x))(l) \le mx_i) \\
 & & \quad \wedge\ \max(mn_h, mn_{h+1}) \\
 & & \qquad \le (v;\ l \in x\,..\,b_{h+1}\colon v(x))(b_{h+1}) \\
 & & \qquad \le \min(mx_h, mx_{h+1})
\end{array}$$

Unfortunately, (3.5) is not implied by what is known to hold at the start of S_2, $I \wedge h \ne n$. We must therefore employ additional statements to transform the state from one satisfying $I \wedge h \ne n$ to one satisfying (3.5). The final conjunct of (3.5) can be established by executing **go**(τ), where τ is any speed that is safe and is attainable by accelerating from $v(b_h)$. That is, τ must satisfy:

$$\begin{array}{ll}
(3.6) & \max(mn_h, mn_{h+1}) \le \tau \le \min(mx_h, mx_{h+1}) \\
 & \quad \wedge\ Vlen(v(b_h), \tau) \le b_{h+1} - b_h
\end{array}$$

Nothing stated thus far implies that it should be possible to accelerate from any safe $v(b_h)$ to a safe $v(b_{h+1})$ in at most a distance of $b_{h+1}-b_h$, and so without making further assumptions about speed constraints, our control problem is unsolvable. We have uncovered another hidden assumption required to control a train:

Assumption AS2. $(\forall i, s\colon\ 0<i<n \wedge \max(mn_{i-1}, mn_i) \le s \le \min(mx_{i-1}, mx_i)\colon$
$(\exists s'\colon\ \max(mn_i, mn_{i+1}) \le s' \le \min(mx_i, mx_{i+1})\colon$
$Vlen(s, s') \le b_{i+1}-b_i)))$

Henceforth, we assume that speed constraints for blocks do satisfy AS2. (It is not difficult to prove that any control problem for which there is a safe path from b_0 to b_n can always be reformulated as one with more restrictive minimum and maximum speeds satisfying AS2.)

A target speed τ satisfying (3.6) can now be computed as follows. First, due to the definition of $Vlen(u, t)$, the set of attainable speeds s—both safe and unsafe—starting from position b_h is characterized by:

$$\sqrt{v(b_h)^2-2*ACC*(b_{h+1}-b_h)} \le s \le \sqrt{v(b_h)^2+2*ACC*(b_{h+1}-b_h)}$$

Second, the set of safe speeds s for location b_{h+1} is given by:

$$\max(mn_h, mn_{h+1}) \le s \le \min(mx_h, mx_{h+1})$$

The intersection of these sets, therefore, is the set of safe and attainable speeds; the maximum of this intersection is the greatest safe speed—time is money for a railroad.

$$\tau = \min(\sqrt{v(b_h)^2+2*ACC*(b_{h+1}-b_h)},\ mx_h,\ mx_{h+1})$$

Using this value of τ for the target speed ensures that the final conjunct of (3.5) will hold.

The penultimate conjunct of (3.5) now is implied by our choice of τ and $Safe(b_0, b_h)$. Thus, our only remaining obligation is the truth of the second conjunct of (3.5), $0<v(x)$. Recall that $0 \le mn_i < mx_i$ holds, by assumption. Thus, for all i, $mx_i \ne 0$ and so successive values of τ are each non-zero. Provided $v_0 \ne 0$, we can strengthen the loop invariant to include $0<v(x)$ as a conjunct. This results in the following program.

```
S: {$x=b_0 \wedge v=(;\ b_0{:}v_0) \wedge AS1 \wedge 0<v_0 \wedge AS2$}
   $h := 0$
   {$I$: $Safe(b_0, b_h) \wedge x=b_h \wedge 0 \le h \le n \wedge 0<v(x)$}
   do $h \ne n \rightarrow$ {$I \wedge h \ne n$}
           $S_2$: $t := \min(sqrt(v(b_h)^2+2*ACC*(b_{h+1}-b_h)),$
                          $mx_h,\ mx_{h+1})$;
               go($t$);
               await($b_{h+1}$);
               $h := h+1$
               {$I$}
   od {$h=n \wedge I$}
   {$Safe(b_0, b_n)$}
```

As the final step of the derivation, we delete references to reality variables from program statements. Recall, reality variables are auxiliary and, therefore, may not affect program execution. The only reference to a reality variable from within statements in the program above is the expression $v(b_h)$. We can maintain this value in a program variable *vel* by strengthening the loop invariant and adding assignments after each go statement. Making these changes results in the following control program; it solves the railroad control problem.

```
S: {$x=b_0 \wedge v=(;\ b_0{:}v_0) \wedge AS1 \wedge 0<v_0 \wedge AS2$}
   $h := 0$;  $vel := v_0$
   {$I$: $Safe(b_0, b_h) \wedge x=b_h \wedge 0 \le h \le n \wedge 0<v(x)$
        $\wedge\ vel=v(b_h)$}
   do $h \ne n \rightarrow$ {$I \wedge h \ne n$}
           $S_2$: $t := \min(sqrt(vel^2+2*ACC*(b_{h+1}-b_h)),$
                          $mx_h,\ mx_{h+1})$;
               go($t$);
               $vel := t$;
               await($b_{h+1}$);
               $h := h+1$
               {$I$}
   od {$h=n \wedge I$}
   {$Safe(b_0, b_n)$}
```

3.2. An Improved Control Program

Although correct, the control program just derived does not always permit a train to travel as quickly as possible. Modifying the derivation to maximize train speed is not difficult, however. First, we rewrite (3.4) as follows:

$$(\forall l{:}\ b_h<l<b_{h+1}{:}\ b_i \le l \le b_{i+1} \Rightarrow mn_i \le v(l) \le mx_i)$$
$$\wedge\ \max(mn_{h-1}, mn_h) \le v(b_h) \le \min(mx_{h-1}, mx_h)$$
$$\wedge\ \max(mn_h, mn_{h+1}) \le v(b_{h+1}) \le \min(mx_h, mx_{h+1})\ \wedge\ x=b_{h+1}$$

Then, rather than allowing the final conjunct to drive the derivation (as it did above), we concentrate on the penultimate conjunct. The loop body that results from this strategy is:

```
$S_2$: go($t_1$);
     await($b_{h+1} - Vlen(t_1, t_2)$);
     go($t_2$);
     $h := h+1$
```

where t_1 and t_2 are the largest speeds satisfying:

$$Vlen(v(b_h), t_1) + Vlen(t_1, t_2) \le b_{h+1} - b_h \ \wedge\ mn_h \le t_1 \le mx_h$$
$$\wedge\ \max(mn_h, mn_{h+1}) \le t_2 \le \min(mx_h, mx_{h+1})$$

Computing values for t_1 and t_2 and using this new S_2 as a loop body, we get the following revised control program.

```
$S$: $\{x=b_0 \wedge v=(;\ b_0{:}v_0) \wedge AS1 \wedge 0<v_0 \wedge AS2\}$
   $h := 0$;  $vel := v_0$
   $\{I$: $Safe(b_0, b_h) \ \wedge\ x=b_h \ \wedge\ 0 \le h \le n \ \wedge\ 0<v(x)$
        $\wedge\ vel = v(b_h)\}$
   do $h \ne n \rightarrow \{I \wedge h \ne n\}$
          $S_2$: $t_1 := \min(mx_h,$
                            $sqrt(vel^2 + 2*ACC*(b_{h+1} - b_h)),$
                            $sqrt(\frac{mx_{h+1}^2}{2} + \frac{vel^2}{2} + ACC*(b_{h+1} - b_h)))$;
              $t_2 := \min(t_1, ms_{h+1})$;
              go($t_1$);
              await($b_{h+1} - Vlen(t_1, t_2)$);
              go($t_2$);
              $vel := t_2$;
              $h := h+1$
              $\{I\}$
   od $\{h=n \wedge I\}$
   $\{Safe(b_0, b_n)\}$
```

4. Discussion

We were pleased to discover that only minor modifications were needed in order to employ Dijkstra's weakest-precondition calculus in deriving sequential, real-time, process-control programs. Dijkstra's calculus, unfortunately, is based on regarding a program as a relation between sets of states and, therefore, does not scale-up to concurrent and distributed programs, which are best thought of as "invariant maintainers". The extensions derived in section 2 for handling the state of a physical process—the contribution of this paper—do scale up. For example, we have been able to use them along with a logic for proving arbitrary safety properties of concurrent programs, Proof Outline Logic [5].

Second, both of the control programs we developed assumed that assignment statements are instantaneous. In reality, executing assignment statements does take time, and the state of the controlled process can change during that interval. It is not difficult to derive control programs for this more realistic setting. The predicate logic details become a bit messier as do the constants, but nothing about the structure of the derivation or resulting programs changes.

Reality variables are history variables—they encode in the current program state information about past system states. Using history variables for reasoning about programs is usually a bad idea, because it introduces distinctions that should be irrelevant. The current state—not how it was computed—should be of concern when reasoning about what a program will do next. In reasoning about process-control systems, however, one has no choice but to employ history variables of some sort. This is because the past instants for which the state of a physical process is defined is a strict superset of the past instants for which the state of a control program is defined. A program implements a discrete transition system, while a physical process is likely to implement a continuous transition system. History variables allow us to reason about all of the behavior of the physical process, including those states that exist while the program state is in transition, hence undefined.

Acknowledgment. Richard Brown read an early version of this paper and provided helpful comments.

References

[1] Clint, M. Program proving: Coroutines. *Acta Informatica 2*, 1 (1973), 50-63.

[2] Gries, D. *The Science of Programming*. Springer-Verlag, New York, 1981.

[3] Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, N.J., 1976.

[4] Lozano-Perez, T. Spatial planning: A configuration space approach. *IEEE Trans. on Computers C-32*, 2, 1983, 108-120.

[5] Schneider, F.B. and G. R. Andrews. Concepts for concurrent programming. In *Current Trends in Concurrency*. (J.W. de Bakker, W.P. de Roever, and G. Rozenberg, eds.) Lecture Notes in Computer Science, Volume 224, Springer-Verlag, New York, 1986, 669-716.