Deductive Verification with Ghost Monitors

Martin Clochard, Claude Marché, Andrei Paskevich

Deductive Verification with Ghost Monitors

MARTIN CLOCHARD, ETH Zürich, Switzerland
CLAUDE MARCHÉ, Université Paris-Saclay, Inria, France
ANDREI PASKEVICH, LRI, Université Paris-Sud & CNRS, France

2

We present a new approach to deductive program verification based on auxiliary programs called *ghost monitors*. This technique is useful when the syntactic structure of the target program is not well suited for verification, for example, when an essentially recursive algorithm is implemented in an iterative fashion. Our approach consists in implementing, specifying, and verifying an auxiliary program that monitors the execution of the target program, in such a way that the correctness of the monitor entails the correctness of the target. The ghost monitor maintains the necessary data and invariants to facilitate the proof. It can be implemented and verified in any suitable framework, which does not have to be related to the language of the target programs. This technique is also applicable when we want to establish relational properties between two target programs written in different languages and having different syntactic structure.

We then show how ghost monitors can be used to specify and prove fine-grained properties about the *infinite behaviors* of target programs. Since this cannot be easily done using existing verification frameworks, we introduce a dedicated language for ghost monitors, with an original construction to *catch* and handle divergent executions. The soundness of the underlying program logic is established using a particular flavor of transfinite games. This language and its soundness are formalized and mechanically checked.

CCS Concepts: • **Theory of computation → Logic and verification**; **Hoare logic**.

Additional Key Words and Phrases: Deductive program verification, Ghost code, Unstructured programs, Infinite behaviors, Games, Floyd-Hoare logic, Predicate transformers.

ACM Reference Format:
Martin Clochard, Claude Marché, and Andrei Paskevich. 2020. Deductive Verification with Ghost Monitors. *Proc. ACM Program. Lang.* 4, POPL, Article 2 (January 2020), 26 pages. https://doi.org/10.1145/3371070

1 INTRODUCTION

The traditional approach to deductive program verification, as embodied by the Floyd-Hoare logic and the weakest precondition calculus, ties the verification process to the syntactic structure of the program under consideration: contracts are attached to subprogram boundaries, loop invariants are placed at a fixed place in the loop body, the program state at previous loop iterations is inaccessible, etc. Sometimes, however, the target program is not written in a way that makes its proof straightforward. An algorithm implemented as a simple loop may be best explained through a complex recursive scheme that establishes links between individual loop iterations. Various optimizations may alter the control flow making the original loop invariants inadequate.

We propose a new method to handle such hard cases by introducing a level of indirection (nod to David Wheeler's aphorism intended). Instead of a frontal assault on the target program, we write, specify, and verify an auxiliary program, called *ghost monitor*, whose control flow restores an

Authors' addresses: Martin Clochard, Department of Computer Science, ETH Zürich, Zürich, 8092, Switzerland, martin.clochard@inf.ethz.ch; Claude Marché, Université Paris-Saclay, Inria, Palaiseau, 91120, France, claude.marche@inria.fr; Andrei Paskevich, LRI, Université Paris-Sud & CNRS, Orsay, 91405, France, andrei.paskevich@lri.fr.

2475-1421/2020/1-ART2
https://doi.org/10.1145/3371070

algorithmic structure more appropriate to formal proof, allowing us to place the right invariants in the right places. In this regard, a ghost monitor is a reimplementation of the target code in a way best suited for verification. However, the monitor does not perform computations on its own (apart from auxiliary computations needed purely for verification purposes). Instead, it follows—monitors—the execution of the target program, from one breakpoint to another, so that the execution of the monitor and the target advance in lockstep. The only observable side effect of the monitor is when it commands the target program: "continue until the next breakpoint". In other words, the monitor program is akin to a debugger for the target program: the former maintains the auxiliary data and the invariants needed for the proof, while the latter performs the actual computations and synchronizes with the monitor at the breakpoints.

It is important to note that we do not imply any kind of run-time checking here. Ghost monitors are subject to the standard static analysis and deductive verification procedures. Since the monitor may observe the state of the target, we can express—as the monitor's specification—all the properties of the target code we may desire to prove. And since the monitor cannot alter the execution of the target program in any way (it may only command the target to continue), proving the correctness of the monitor ensures that the target program indeed satisfies the required properties.

It may seem that we do not make the user's job any easier: after all, they have to write and verify a whole new program. However, we are going to show on several examples that the monitor's code and specification, put together, may well be clearer and easier to come up with than the invariants and assertions tied to the control flow of the target code.

Let us demonstrate the essence of our method on a simple example. Consider the following two code fragments, one written in OCaml, and the other in C:

```
let rec mcc91 (n: int) : int =
  if n > 100 then
    n - 10
  else
    mcc91 (
      mcc91 (
        n + 11 ))
```

```
e = 1;
while (1) {
  if (n > 100) {
    n = n - 10; e = e - 1;
    if (e == 0) break; }
  else {
    n = n + 11; e = e + 1; }}
```

The recursive function on the left is the famous "91 function" by John McCarthy [Manna and McCarthy 1970]: `mcc91` terminates on every input n and computes $f_{91}(n) = 91$ if $n \leq 100$ and $n - 10$ otherwise. To prove that function `mcc91` terminates and computes f_{91} is a matter of milliseconds using any modern automated program verifier, backed up by an SMT solver. The trickiest though not difficult part here is the proof of termination, which requires finding an appropriate *variant* (i.e., decreasing measure), for example $101 - n$.

The iterative C code on the right computes the same function (leaving the result in `n`), and can in fact be obtained from the recursive code by manual de-recursification, using an extra variable e to represent the number of pending recursive calls. Verifying the iterative version, however, is a little harder [Clochard et al. 2018a]: it requires discovering a significantly more complex variant (lexicographic order on $(101 - n + 10e, e)$) and a non-trivial loop invariant ($f_{91}^{e}(n) = f_{91}(\textbf{old}\ n)$).

Our method allows us to verify the code on the right by "observing" its execution from a ghost monitor program whose control flow would be taken from the recursive code on the left. We start by placing four breakpoints in the C code: breakpoint 0 at the beginning, breakpoint 1 before `if (n > 100)...`, breakpoint 2 before `if (e = 0)...`, and breakpoint 3 at the end. We choose the breakpoints in such a way that the evolution of the symbolic state of the target program between two breakpoints can be computed statically, e.g., via symbolic execution. In particular, breakpoints

```
let rec mcc91_monitor()
  requires { pc = 1 ∧ e > 0 }          (* start at breakpoint 1 *)
  variant  { 101 - n }                 (* same variant as in mcc91 *)
  ensures  { pc' = 2 ∧ e' = e - 1 }    (* stop at breakpoint 2 *)
  ensures  { n' = f91(n) }
= if n > 100 then
    NEXT();             (* no recursive calls, move to breakpoint 2 *)
  else begin
    NEXT();             (* update n and e, move to breakpoint 1 *)
    mcc91_monitor();    (* inner mcc91 call, stop at breakpoint 2 *)
    NEXT();             (* e is non-zero, so move to breakpoint 1 *)
    mcc91_monitor()     (* outer mcc91 call, stop at breakpoint 2 *)
  end

let main_monitor()
  requires { pc = 0 }      (* start at the beginning *)
  ensures  { pc' = 3 }     (* stop at the end *)
  ensures  { n' = f91(n) } (* main functional property *)
= NEXT();                  (* initialize e, move to breakpoint 1 *)
  mcc91_monitor();         (* compute mcc91 once, stop at breakpoint 2 *)
  NEXT()                   (* exit the while loop *)
```

Fig. 1. Ghost monitor for the iterative McCarthy 91 function.

must break loops. Also, we need breakpoints at the beginning and at the end, so that we can express and verify that the ghost monitor observes the entire execution of the target program.

The evolution of the symbolic state is translated into the following specification that establishes the relation between the pre- and post-state of the slices of the C code (variables with primes represent the post-state, the additional variable *pc* denotes the breakpoint number):

$$pc = 0 \longrightarrow pc' = 1 \ \wedge \ e' = 1 \ \wedge \ n' = n$$
$$pc = 1 \ \wedge \ n > 100 \longrightarrow pc' = 2 \ \wedge \ e' = e - 1 \ \wedge \ n' = n - 10$$
$$pc = 1 \ \wedge \ n \leq 100 \longrightarrow pc' = 1 \ \wedge \ e' = e + 1 \ \wedge \ n' = n + 11$$
$$pc = 2 \ \wedge \ e = 0 \longrightarrow pc' = 3 \ \wedge \ e' = e \ \wedge \ n' = n$$
$$pc = 2 \ \wedge \ e \neq 0 \longrightarrow pc' = 1 \ \wedge \ e' = e \ \wedge \ n' = n$$

We attach this specification (with an added precondition $0 \leq pc \leq 2$) to a procedure named `NEXT`, and define the ghost monitor as shown in Figure 1. In the monitor code, the variables that refer to the state of the target C program are written in italic. The proof of this program is performed automatically using any modern automated program verifier based on SMT solvers.

The control flow of `mcc91_monitor` follows that of `mcc91`. However, the actual computations are performed as side effects of `NEXT`, which simulates the execution of the C program up to the next breakpoint. Each call to `mcc91_monitor` starts at the beginning of the loop body (breakpoint 1), and ends when variable `e` is decremented (breakpoint 2). Remember that `e` represents the number of pending recursive calls to `mcc91`, and, indeed, at the end of a call to `mcc91_monitor`, the variable `n` contains the result of the McCarthy 91 function applied to the pre-state value of `n`. The function `main_monitor` initiates and finalizes the computation. Its contract ensures that the execution of

the monitor begins and ends at the same time as the execution of the target code. Since the monitor does not affect the target program state in any other way than by calling NEXT, we can conclude that the iterative C program does indeed compute the 91 function.

Contributions and outline. The first contribution of our paper is a new method of deductive program verification that relies on an external auxiliary program, a ghost monitor, to make explicit the underlying algorithm of the target program. This liberty to choose a different syntactic structure can significantly simplify the discovery of appropriate contracts and invariants, as shown in Section 2 on a new proof of the Schorr-Waite in-place graph traversal algorithm [Schorr and Waite 1967]. Notably, this technique can be applied to verification of programs written in unstructured, assembly-like languages. Our approach is very different from the traditional use of ghost data and ghost code (as explored, e.g., in [Filliâtre et al. 2016]), where the auxiliary variables and computations are implanted inside the target code, and the verification process still has to follow the original control flow.

One advantage of our method is that the monitor does not need to be written in the language of the target program. Whatever verification framework we use to prove the monitor's correctness, it never interacts with the target program's code. All needed knowledge about the target's behavior and the semantics of the target's language is stated in the specification of the NEXT operation. This specification can be generated mechanically, for example, by means of symbolic execution of the target code between the chosen breakpoints. More precisely, we build the control flow graph of the target program and, assuming all cycles contain at least one breakpoint, each path from one breakpoint to another gives us one case for the postcondition of NEXT. Note that specifying NEXT is a separate effort, not related to the implementation or verification of the monitor.

This approach also works when we want to establish *relational properties*, like forward or backward simulation, between two target programs. Indeed, we only need to provide the monitor with two NEXT operations, one for each target, and make it follow the execution of both, in an appropriate cadence. In the presence of non-determinism, the choices made by one target may be transferred to the other (as arguments given to NEXT). Again, we can prove that verifying the monitor program ensures the corresponding properties of the targets. We demonstrate our method applied to the proof of relational properties in Section 3.

Going even further, we show in Section 3.2 how one can handle the properties of potentially non-terminating programs. To do this, however, we need to provide the monitor program with the means to catch and handle divergent executions. This motivates the second contribution of this paper: a dedicated programming language for ghost monitors that allows us to specify and prove properties of infinite executions of target programs, after a transfinite number of calls to NEXT.

This language, presented in Section 4, supports arbitrary recursion, continuations, and fine-grained specification of non-terminating behaviors, both in the targets and in the monitor. In Section 4.2, we define a program logic for this language in the form of an appropriate weakest precondition calculus. The expected soundness properties of our approach are stated in Section 4.3.

In order to establish these properties, we develop an original theoretical framework based on transfinite games; this constitutes the third contribution of our paper, presented in Section 5. Besides being well suited for reasoning about infinite executions, this framework also allows us to adopt a generalized approach with respect to the interpretation of non-determinism in target programs. A universal interpretation of non-determinism, commonly termed *demonic*, is used to prove program correctness for every possible behavior. On the other hand, an existential, *angelic*, interpretation is used to prove the existence of specific behaviors, or the existence of valid implementation choices. The game-theoretic framework allows us to mix both kinds of non-determinism when reasoning about given target programs.

```
typedef struct struct_node {
  unsigned int m :1, c :1;
  struct struct_node *l, *r;
} * node;

void schorr_waite(node root) {
  /* breakpoint 0 */ node t = root, p = NULL;
  while (/* breakpoint 1 */ p != NULL || (t != NULL && !t->m)) {
    if (t == NULL || t->m) {
      if (p->c) { /* pop */
        node q = t; t = p; p = p->r; t->r = q }
      else {      /* swing */
        node q = t; t = p->r; p->r = p->l; p->l = q; p->c = 1 } }
    else {        /* push */
      node q = p; p = t; t = t->l; p->l = q; p->m = 1; p->c = 0 }
  } /* breakpoint 2 */
}
```

Fig. 2. Schorr-Waite graph traversal in C, with breakpoints.

In order to increase confidence in our results, we mechanized the program logic underlying our language for ghost monitors using the Why3 tool. This development is available online [Clochard 2018a].

2 EXTENDED EXAMPLE: SCHORR-WAITE IN-PLACE GRAPH TRAVERSAL

Since our approach does not exploit the syntactical structure of the target code, it works well for programs written in a low-level or unstructured language, including assembly. A representative example is the Schorr-Waite in-place graph traversal algorithm [Schorr and Waite 1967], a landmark example for evaluating proof methods dealing with pointer aliasing [Bornat 2000]. To our knowledge, Leino was the first to propose a proof using only automated theorem provers [Leino 2010].

Let us consider the implementation of this algorithm written in a low-level pointer-manipulating C code, given in Figure 2. Our objective is to traverse the graph of nodes reachable from the given `root` via the pointers `l` and `r`. The specificity is to avoid using extra memory by modifying `l` and `r` to perform backtracking. All pointers are restored to their initial values at the end. The informal specification contains three parts: (i) The graph structure induced by pointers `l` and `r` is restored at the end of the procedure; (ii) Assuming all nodes in the graph are initially unmarked (`m` is 0), the ones reachable from `root` get marked at the end; (iii) The nodes unreachable from `root` have their mark unchanged. Figure 3 shows a formal specification, using Bornat-style *component-as-array model* [Bornat 2000] to represent heap memory.

Proving directly that the low-level code meets this specification requires quite complex loop invariants [Leino 2010]. With the ghost monitor approach, we write a new ghost code, shown in Figure 4, following the standard structure of a recursive depth-first traversal. As for the McCarthy function, we assume first a `NEXT` procedure is defined, with the breakpoints set as shown in the code of Figure 2. Our monitor is annotated with the usual pre- and postconditions expected for a classical depth-first traversal. This annotated code is fully proved with automated solvers [Clochard and Marché 2018]. Unlike Leino [Leino 2010], we do not need complex loop invariants to make explicit the hidden notion of backtracking stack behind Schorr-Waite algorithm. Among the existing proofs

```
type loc (* abstract type for memory locations *)
constant null : loc
type memory = { (* component-as-array modeling of the heap *)
  mutable l, r: loc → loc;
  mutable m, c: loc → bool; }

(* paths *)
predicate edge (h : memory) (x y : loc) = x ≠ null ∧ (h.l x = y ∨ h.r x = y)
inductive path memory loc loc =
  | path_nil : ∀h, x. path h x x
  | path_cons : ∀h, x, y, z. edge h x y ∧ path h y z → path h x z

(* unchanged_structure only concerns the graph shape, not the marks *)
predicate unchanged_structure (h1 h2 : memory) =
  ∀x. x ≠ null → h2.l x = h1.l x ∧ h2.r x = h1.r x

(* global instance for the memory *)
val heap : memory

let schorr_waite (root : loc) (ghost graph : set loc) : unit
  requires { (* root belongs to the graph *)
    root ∈ graph }
  requires { (* the graph is closed with respect to its edges *)
    ∀x. x ∈ graph ∧ x ≠ null → (heap.l x) ∈ graph ∧ (heap.r x) ∈ graph }
  requires { (* the graph starts fully unmarked *)
    ∀x. x ∈ graph → ¬(heap.m x) }
  ensures  { (* the graph structure is left unchanged *)
    unchanged_structure heap heap' }
  ensures  { (* every location reachable from the root is marked *)
    ∀x. path heap root x ∧ x ≠ null → heap'.m x }
  ensures  { (* every other location keeps its previous marking *)
    ∀x. ¬(path heap root x) ∧ x ≠ null → heap'.m x = heap.m x }
```

Fig. 3. Schorr-Waite graph traversal, main specification.

of Schorr-Waite algorithm, the closest to ours might be the one proposed by Yang [Yang 2007], based on a refinement approach. We argue however that our approach based on ghost monitors goes significantly further than classical refinement. Quoting the last paragraph of page 20 of Yang's paper: *"One evident shortcoming of our logic is that the proof rules assume that two commands have similar control structures. When this assumption breaks, our new rules for quadruples do not help, and we mostly have to reason about C and C' individually in separation logic"*. Indeed, our method does not have this limitation. Notably, Yang's proof relates two iterative versions of the Schorr-Waite algorithm, while with our method we relate an iterative version to a more abstract recursive version, in which the stack is implicit. Our proof is fully automated, and we argue that the extra annotations we need to add, as postconditions to the monitor, are considerably simpler than Leino's ones [Leino 2010].

```
(* DFS invariant refers to different parts of heap at different time:
   the graph structure is given via the initial heap memory h,
   while the coloring is given via the current heap memory m *)
predicate well_colored (graph gray : set loc) (h : memory) (m : loc → bool) =
  gray ⊆ graph ∧ (∀x. x ∈ gray → m x) ∧
  (∀x y. x ∈ graph ∧ edge h x y ∧ y ≠ null ∧ m x → x ∈ gray ∨ m y)

let schorr_waite (root : loc) (ghost graph : set loc) : unit =
  let ghost initial_heap = heap in (* ghost copy of the initial memory *)
  let rec recursive_monitor (ghost gray_nodes : set loc) : unit
    requires { pc = 1 ∧ t ∈ graph }
    requires { (* assume DFS invariant *)
      well_colored graph gray_nodes initial_heap heap.m }
    requires { (* non-marked nodes have unchanged structure *)
      ∀x. x ≠ null ∧ ¬(heap.m x) → heap.l x = initial_heap.l x ∧ heap.r x = initial_heap.r x }
    ensures { pc' = 1 ∧ t' = t ∧ p' = p }
    ensures { (* pointer buffer is overall left unchanged *)
      unchanged_structure heap heap' }
    ensures { (* maintain DFS invariant *)
      well_colored graph gray_nodes initial_heap heap.m }
    ensures { (* the top node gets marked *)
      t' ≠ null → heap.m t' }
    ensures { (* cannot mark unreachable nodes or change marked nodes *)
      ∀x. x ≠ null → ¬(path initial_heap t' x) ∨ heap.m x →
        heap'.m x = heap.m x ∧ heap'.c x = heap.c x }
    variant { |graph| − |gray_nodes| }
  = if t = null || heap.m t then () else
    let ghost new_gray = {t} ∪ gray_nodes in
    NEXT (); (* push *)
    recursive_monitor new_gray; (* traverse the left child *)
    NEXT (); (* swing *)
    recursive_monitor new_gray; (* traverse the right child *)
    NEXT ()  (* pop *)
  in
  NEXT (); (* initialize *)
  recursive_monitor ∅;
  NEXT ()  (* exit *)
```

Fig. 4. Ghost monitor for Schorr-Waite graph traversal.

3 RELATIONAL PROPERTIES AND INFINITE BEHAVIORS

The ghost monitor method applies to relational properties like program equivalence. We illustrate this aspect by proving the equivalence of two programs parsing well-parenthesized words (Dyck language): they take an infinite stream of parentheses as input and stop upon finding an unmatched closing parenthesis. The first program (C), written in C, uses a counter to keep track of the number of currently opened parentheses:

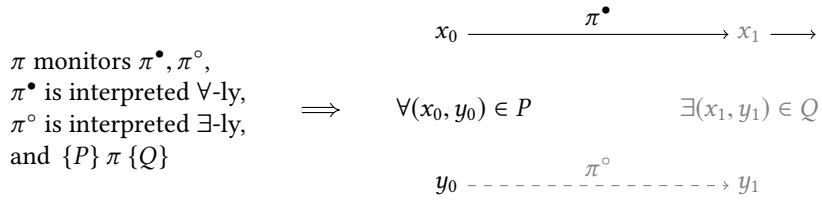

Fig. 5. Transfer property yielding a simulation.

```
n = 0; while (n >= 0) if (getchar() == '(') n++; else n--;
```

The second program (`M`), written in ML, uses recursion:

```
let rec scan () = while getchar () = '(' do scan () done in scan ()
```

We prove that the two programs are equivalent under two simplifying hypotheses: unbounded stack and mathematical integers.

To that end, we write a monitor calling one `NEXT` procedure for each program. Instead of representing the input streams explicitly, as two global variables in the monitor, from which the `NEXT` procedures would take the next character returned by `getchar`, we opt for a non-deterministic semantics of `getchar`, in order to illustrate the handling of non-determinism in ghost monitors. The challenge now consists in synchronizing the execution of the two programs, ensuring that they actually process the same input.

3.1 Equivalence of Non-Deterministic Programs

The straightforward generalization of ghost monitors to non-determinism is to make `NEXT` perform non-deterministic choices whenever the target program does. In order to ensure that (`C`) and (`M`) read the same input values, the non-deterministic choices of one program must be transferred to the other. We achieve this by adding arguments to one of the `NEXT` operations, endowing the ghost monitor with the ability to drive the execution of the corresponding target program in the direction that matches the behavior of the other target. In essence, we replace the *demonic* interpretation of non-determinism for one of the two programs by an *angelic* one.

In this setting, contracts proved for the ghost monitor are transferred to simulation properties.

Definition 3.1 (Transfer Property). The transfer property (see Figure 5) states that for every

- ghost monitor π for target programs $\pi^\bullet$ and π°, where non-determinism is interpreted as demonic (universal) for $\pi^\bullet$ and as angelic (existential) for π°,
- valid Hoare triple $\{P\}\,\pi\,\{Q\}$ (total correctness contract),
- pair of states x_0, y_0 for $\pi^\bullet$ and π°, respectively, that satisfy P,
- complete (potentially infinite) execution trace of $\pi^\bullet$ starting from x_0,

there exists a pair of states (x_1, y_1) satisfying Q such that x_1 belongs to the chosen execution trace of $\pi^\bullet$ and there exists an execution trace of π° leading from y_0 to y_1.

In other words, once we prove the correctness of our monitor for a given specification, whenever the execution of the monitored programs starts in a (combined) state that satisfies the precondition, for every behaviour of the demonic program, there exists a behaviour of the angelic program such that together they eventually satisfy the postcondition.

When P (resp. Q) expresses that states of $\pi^\bullet$ and π° are initial (resp. final) and equivalent to each other, this simulation result means that all behaviors of the 'demonic' program can be translated

```
/* breakpoint 0 */
n = 0;
while ( /* breakpoint 1 */
        n >= 0 ) {
  if (getchar() == '(') n++;
  else n--;
} /* breakpoint 2 */
```

```
(* breakpoint 0 *)
let rec scan () =
  while (* breakpoint 1 *)
        getchar() = '('
  do scan () done
  (* breakpoint 2 *) in
scan() (* breakpoint 3 *)
```

Fig. 6. Programs (C) and (M) augmented with breakpoints.

$$
\begin{aligned}
&angel_C \rightarrow read_C = toRead_C \\
&pc_C = 0 \rightarrow pc'_C = 1 \ \wedge\ n'_C = 0 \ \wedge\ in'_C = in_C \\
&pc_C = 1 \ \wedge\ n_C < 0 \rightarrow pc'_C = 2 \ \wedge\ n'_C = n_C \ \wedge\ in'_C = in_C \\
&pc_C = 1 \ \wedge\ n_C \geq 0 \rightarrow\ pc'_C = 1 \ \wedge\ in'_C = in_C + [read_C] \ \wedge \\
&\qquad\qquad (read_C = \text{'('} \ \rightarrow\ n'_C = n_C + 1) \wedge \\
&\qquad\qquad (read_C \neq \text{'('} \ \rightarrow\ n'_C = n_C - 1)
\end{aligned}
$$

$$
\begin{aligned}
&angel_M \rightarrow read_M = toRead_M \\
&pc_M = 0 \rightarrow pc'_M = 1 \ \wedge\ stack'_M = [3] + stack_M \ \wedge\ in'_M = in_M \\
&pc_M = 1 \rightarrow\ in'_M = in_M + [read_M] \ \wedge \\
&\qquad\qquad (read_M = \text{'('} \ \rightarrow\ stack'_M = [1] + stack_M \ \wedge\ pc'_M = 1) \ \wedge \\
&\qquad\qquad (read_M \neq \text{'('} \ \rightarrow\ stack'_M = stack_M \ \wedge\ pc'_M = 2) \\
&pc_M = 2 \rightarrow in'_M = in_M \ \wedge\ stack'_M = \text{tail}\ stack_M \ \wedge\ pc'_M = \text{head}\ stack_M
\end{aligned}
$$

Fig. 7. Specifications of NEXT_C and NEXT_M.

as behaviors of the 'angelic' program. By proving such translations in both directions, we obtain equivalence. To avoid writing a separate ghost monitor for each direction, we write a single monitor, parametric in the roles of the two programs. To represent these roles, we augment the state with two immutable Boolean variables $angel_C$ and $angel_M$ that indicate whether non-determinism is interpreted as angelic for the corresponding program.

Let us now build the monitor. First, we identify the relevant breakpoints in both programs, shown in Figure 6. As these breakpoints break all execution cycles, we can obtain proper specifications for NEXT_C and NEXT_M, shown in Figure 7. To describe the state of each target program, we use variables *pc* and *in* that denote, respectively, the current location and the input log (the list of all characters read up to this point). Since program (M) is recursive, we also have to model its stack. We represent it as a sequence of stack frames, each containing the return address in the form of a breakpoint number: 3 when scan is called from the main expression in (M), and 1 when scan is called recursively.

To model the non-deterministic nature of getchar, each NEXT procedure receives an argument *toRead* and returns a value *read* which represents the character that has just been read, if any. When the program runs in the demonic mode (the corresponding *angel* variable is false), this return value is not specified in any way, reflecting the fact that *read* was picked in a non-deterministic manner. When the program runs in the angelic mode (the corresponding *angel* variable is true), *read* assumes the value of the *toRead* argument, allowing the monitor to control the non-deterministic choice.

```
let monitor ()
  requires { pc_C = 0 ∧ pc_M = 0 }   (* both start at the beginning *)
  requires { in_M = in_C }           (* start with same initial reads *)
  requires { angel_C ≠ angel_M }     (* one of the programs is angelic *)
  ensures  { pc'_C = 2 ∧ pc'_M = 3 } (* both stop at the end *)
  ensures  { in'_M = in'_C }         (* after reading the same input *)
= let ign = '(' in  (* used when the choice is ignored by NEXT *)
  NEXT_C (ign); NEXT_M (ign);
  let rec sync () : unit
    requires { pc_C = 1 ∧ pc_M = 1 }  (* at the entry of loop iteration *)
    requires { in_M = in_C }          (* same inputs so far *)
    requires { n_C ≥ 0 }              (* program (C) enters the loop *)
    ensures  { pc'_C = 1 ∧ pc'_M = 2 } (* at the end of loop body / call *)
    ensures  { stack'_M = stack_M }   (* same recursive call *)
    ensures  { in'_M = in'_C }        (* same inputs so far *)
    ensures  { n'_C = n_C - 1 }       (* counter decremented *)
  = if angel_M then NEXT_M (NEXT_C (ign)) else NEXT_C (NEXT_M (ign));
    if pc_M = 1 then begin sync (); NEXT_M (ign); sync () end
  in
  sync (); NEXT_C (ign); NEXT_M (ign)
```

Fig. 8. Monitor for equivalence of (C) and (M).

We are now ready to write the monitor program. We could use the structure of (M), but to reduce annotations we choose a slightly different structure, which fuses the iterations of the loop in (M) in a single tail-recursive call. We give the monitor in Figure 8. The core of the monitor is the recursive function sync, which relates, on one hand, the loop iterations in (C) until the counter has decreased, and, on the other hand, one full recursive call of scan in (M).

While this monitor can be automatically proved without any trouble, there is a catch. Indeed, we did not prove that this monitor terminates, as we do not have any measure to control execution time. However, we can only use the transfer hypothesis to derive program equivalence when we have a total correctness property for the ghost monitor. This means that we need additional ingredients to deal with non-terminating behaviors.

3.2 Infinite Behaviors

The easiest way to recover the total correctness for the ghost monitor is to assume that the program with demonic non-determinism terminates, e.g., by using a variable representing execution fuel. While this would lead to a terminating monitor, we would not obtain the right simulation result. Indeed, to exploit such a termination hypothesis, we need to weaken the transfer property by restricting the universal execution range to terminating executions, i.e., finite maximal executions. In particular, we would only obtain as a result that the two programs have the same terminating behaviors, but may have different non-terminating behaviors.

To make the distinction clear, suppose that we change the semantics of (C) so that getchar() may have a third possible behavior, starting an infinite loop. Then (C) and (M) still have the same terminating behaviors (we may adjust a fuel-based monitor to prove this), but (C) now has extra non-terminating behaviors in which it reads a finite number of inputs, while (M) must read infinitely many ones.

Hence we need a way to relate infinite executions of the two programs. To do so, we extend our ghost monitors with means to handle divergence. We consider the program clock as a possibly transfinite number and we treat the transition to a limit ordinal as an exceptional situation that can be intercepted in the monitor code. Of course, this cannot be realized in a physically executable code, but ghost monitors only serve for verification and are never executed. In this way, a ghost monitor may call `NEXT` an infinite number of times *and then* proceed to an exit point, covered by the monitor's postcondition. This allows us to formulate and prove statements about infinite executions of the targets, by considering them in a virtual "limit state" after an infinite number of steps.

To be able to proceed with that plan, we have to augment the language of ghost monitors with several ingredients (below, we focus on infinite recursion, but loops can be treated in a similar way or simply encoded as recursion).

Representation of limit states. First, we first need a way to actually represent the aforementioned limit program state. We achieve this using *least upper bounds*: the state of the program after an infinite execution is considered to be the least upper bound of the states during said execution. This perfectly matches the intuition for event traces like our input logs, as the least upper bound becomes the (potentially infinite) trace of events during the entire infinite execution. For our example, we want the limit state to contain only the observable effects of the program, i.e., the limits of in_C/in_M. We augment the state with time counters t_C and t_M, which increase at each execution step. We then remove program variables (only n_C in the example) from states with infinite time counters, so that least upper bounds of infinite executions represent faithfully observable effects of the execution. By convention, we also set pc_C and pc_M to ∞ in any state with infinite time counters.

Abstraction of caught infinite executions. Second, we need a way to describe infinite executions as they will be caught by divergence handling. For this, we replace decreasing well-founded termination measures with increasing progression measures, and represent a diverging behavior by its complete trace. This trace is an infinite call stack for infinite recursion. Instead of representing a termination argument, progression measures serve as non-local constraints over diverging execution traces. For our running example, we need to force time counters to increase between recursive calls to make sure that whenever `sync` diverges, so do the target programs. Similarly, we need to force input logs to grow to rule out behaviors in which a program stops performing input. Note that we do not impose any particular condition on the progression ordering, in particular no condition closely related to well-foundedness.

Divergence handlers. Third, we equip recursive definitions with divergence handling (**`diverges`**) clauses. These clauses introduce a statement intended to handle any form of divergence issuing from recursion. Divergence handlers are entered with the target program state after an infinite recursive descent, that is, the least upper bound of the states along the call stack leading to divergence. This call stack is given to the divergence handling clause as a parameter, and is assumed to respect the progression clause/the precondition, as well as being infinite. Divergence handlers should cut the stack at some point of their choice, which corresponds to completing a leftover recursive call. For verification purposes, this means that the handler should establish the postcondition of this particular call from the assumptions on the diverging behavior. Note that this is more general than simply recovering from an exception thrown by the recursive function, which is equivalent to closing the bottom call. Indeed, the recursive function may continue executing even after divergence has been caught (potentially infinitely many times through local recursive functions).

We can now use this framework to turn our earlier ghost monitor into a ghost monitor that proves the equivalence of the two programs, taking into account non-terminating behavior. However, we

```
rec monitor ( _ : 𝕌 ) : ⟨P1 ↪ _ : 𝕌. Q1⟩
  progress { ∅ }   (* function monitor is not recursive *)
= let ign = '(' in
  let _ = NEXT_C (ign) in
  let _ = NEXT_M (ign) in
  rec sync ( _ : 𝕌 ) : ⟨P2 ↪ _ : 𝕌. Q2⟩
    (* progression order compares states at the entry of recursive calls *)
    progress { {(s1, s2) | s1.t_C < s2.t_C ∧ s1.t_M < s2.t_M ∧ |s1.in_C| < |s2.in_C|} }
  = let _ = switch
        case _ : 𝕌. angel_M in
          let ch = NEXT_C(ign) in NEXT_M(ch)
        case _ : 𝕌. ¬ angel_M in
          let ch = NEXT_M(ign) in NEXT_C(ch)
      end-switch
    in
    switch
      case _ : 𝕌. pc_M = 1 in
        let _ = sync ( ◊ ) in
        switch
          case _ : 𝕌. pc_M ≠ ∞ in
            let _ = NEXT_M (ign) in sync ( ◊ )
          case _ : 𝕌. pc_M = ∞ in ◊
        end-switch
      case _ : 𝕌. pc_M ≠ 1 in ◊
    end-switch
    diverges ( unbounded_stack ) ->
      switch
        case c:call. c ∈ unbounded_stack in (c, ◊)
      end-switch
  in
  let _ = sync ( ◊ ) in
  switch
    case _ : 𝕌. pc_C ≠ ∞ in
      let _ = NEXT_C (ign) in NEXT_M (ign)
    case _ : 𝕌. pc_C = ∞ in ◊
  end-switch
  diverges ( _ ) ->
    switch end-switch  (* unreachable code *)
```

$$
\begin{aligned}
\text{where}\quad P_1 &\equiv pc_C = 0 \wedge pc_M = 0 \wedge in_M = in_C \wedge angel_C \neq angel_M \\
Q_1 &\equiv ((pc'_C = 2 \wedge pc'_M = 3) \vee (pc'_C = \infty \wedge pc'_M = \infty)) \wedge in'_M = in'_C \\
P_2 &\equiv pc_C = 1 \wedge pc_M = 1 \wedge in_M = in_C \wedge n_C \geq 0 \\
Q_2 &\equiv ((pc'_C = 1 \wedge pc'_M = 2) \vee (pc'_C = \infty \wedge pc'_M = \infty)) \wedge \\
&\qquad (pc'_C \neq \infty \rightarrow stack'_M = stack_M \wedge n'_C = n_C - 1) \wedge in'_M = in'_C
\end{aligned}
$$

Fig. 9. Monitor for equivalence, with divergence handling.

cannot write the monitor using an existing verification language as we need divergence handlers. We use our own minimalist language of ghost monitors, which will be covered in detail in Section 4. We show the monitor adapted to divergence handling in Figure 9. Except for minor syntactic details due to the language change (most notably, expressing conditionals with variable-binding guarded commands), this is the monitor from Figure 8 equipped with divergence-handling code. We denote the unit type with $\mathbb{U}$ and the single inhabitant of this type with $\Diamond$. We equip recursive function sync with a progression clause that requires the clocks of both target programs to advance and the length of the input log to increase with each recursive call. We also add to sync a divergence handler that completes some arbitrary unfinished call in the infinite call stack. To account for the extra behaviors, we also add a few tests after calls to sync to treat infinite states appropriately.

Design choices for divergence handlers. So far, we have not explained how the call stack leading to divergence was represented in the divergence handler, in the sense of which mathematical values were used. There are several choices, and not all of them are equivalent from the perspective of mechanized proofs, nor are they equally powerful. In Section 4, we make the choice of the most general construction available, by representing this stack as a non-empty set of call parameters/state pairs. We constrain this stack to be totally ordered by the chosen progression order, as well as having no maximum. As this set-based representation does not limit the stack to countable sets, we are able to allow recursive calls inside the divergence handler itself, as long as the progression order increases with respect to the elements of the stack.

However, this general choice is typically not the most practical. In most examples, we would prefer to sacrifice the power of making recursive calls inside the divergence handler, as well as the power to return from the handler at any suspended recursive call, in return for consequent advantages for mechanized proof. First, this lets us represent the stack as a sequence of call parameters/state pairs instead of a set. Second, as the divergence handler cannot return at intermediate recursive calls anymore, we may exclude alternative cases arising due to divergence from the post-condition of recursive calls (as well as extra monitor code to handle those cases, like the extra test in Figure 9). Third, with handlers exiting the entire recursion scheme directly, it is easier to link aforesaid alternative cases to the initial parameters and states of the recursion stack. Fourth, the use of a sequence for the stack allows us to generalize the progression order to any relation between recursive calls occurring in immediate succession.

Of course, this more practical divergence handler construction can be derived as syntactic sugar from the more general one, by turning the sequence of successive recursive states into a parameter of the monitor and using prefix order as the progression order. Other intermediate schemes may be derived as well in a similar way. Thus we focus on the more general construction in Section 4.

4 A VERIFICATION LANGUAGE SUPPORTING INFINITE BEHAVIORS

While we believe that pretty much any method or tool for deductive verification can be used to develop and verify ghost monitors (with some extensions if divergence handlers are desired), a formal foundation is still needed to describe the conditions under which our approach is applicable, ensuring that the proof of the monitor implies the desired properties of the target program or programs. To that end, we define a language, denoted $\mathcal{M}$, that can serve as an underlying semantics for ghost monitors, and we state and prove the transfer properties for programs expressed in this language. We consider an almost purely functional language, where functions are first-order and equipped with contracts. The only non-functional feature of $\mathcal{M}$ is a global mutable variable now, which represents the combined current state of the target programs. Each target program P is assigned an associated predefined procedure $\mathtt{NEXT}_P$ which lets P advance until the next breakpoint; and only these procedures are allowed to update now. To handle angelic and demonic

pgm	::=	$fun(expr)$	function call
	|	let var = pgm in pgm	program sequence
	|	switch $case^*$ end-switch	guarded choice
	|	cont fun in pgm end-cont	continuation capture
	|	rec $fun\,(var : typ) : contract$ progress($expr$) = pgm diverges(var) : pgm in pgm	function definition
$case$	::=	case $var : typ.\ formula$ in pgm	guarded command
$contract$	::=	$\langle formula \hookrightarrow var : typ.\ formula \rangle$	function contract

Fig. 10. Syntax of $\mathcal{M}$.

non-determinism, we allow these procedures to take arguments and return values, respectively, as shown in the previous section.

4.1 Syntax and Typing

The syntax of $\mathcal{M}$ is given in Figure 10. Syntactic categories var, fun, and typ correspond to variable names, function names, and data types, respectively. Non-terminals $expr$ and $formula$ refer respectively to arbitrary mathematical expressions and formulas in some logical language, whose exact nature is irrelevant here. Expressions and formulas can access any variable (provided they occur in the variable's lexical scope), including the global variable `now`. Formulas in contract postconditions (to the right of the arrow $\hookrightarrow$) can also access the special variable `old`, which refers to the state of `now` before the execution of a function. All variables except `now` are immutable.

We deliberately keep the number of language constructs as small as possible. We exploit the fact that most usual programming constructions (like loops, conditional, assertions, escape mechanisms, and others) with their usual verification rules can be defined as syntactic sugar in terms of the core constructions. For example, we define an $\mathcal{M}$ program e that returns the value of expression e using continuation capture and a function call: cont k in $k(e)$ end-cont. The same continuation-binding mechanism can be used to implement constructions like `break`, `continue` or `return`. As for loops, they can be easily implemented as tail-recursive functions.

In the guarded choice construction, each case introduces a variable that can be used both in the guard formula and the guarded command. This represents an arbitrary and non-deterministic choice: the variable can be bound to any value that satisfies the guard formula (if there is no such value, the guarded command will not be selected). This generalization of guarded choice is useful when we want to use an existentially quantified guard and also give the guarded command the access to the witness value. An empty guarded choice (switch end-switch) corresponds to an "assert false" statement that must never be reachable in an execution.

We treat the data types assigned to variables and return values as sets and denote them with letters A, B, and C. The type of `now` and `old` is denoted G. Tuples of data values and relations over them are typed as elements of Cartesian products $A \times B$ and power sets $\mathcal{P}(A \times B)$, respectively. Logical formulas have type $\{\top, \bot\}$. Functions of $\mathcal{M}$, denoted f, g, and h, are necessarily first-order, and their types are of the form $A \to B$, where A is the type of the function parameter and B is the type of its return value. Functions with empty return type $\emptyset$ do not return a value. We denote such functions with letter k and call them continuations.

$$\frac{\Sigma \mid \Gamma \vdash \pi_0 : A \qquad \Sigma \mid \Gamma, v : A \vdash \pi_1 : B}{\Sigma \mid \Gamma \vdash \mathsf{let}\ v = \pi_0\ \mathsf{in}\ \pi_1 : B} \qquad \frac{\Sigma, k : B \to \emptyset \mid \Gamma \vdash \pi : B}{\Sigma \mid \Gamma \vdash \mathsf{cont}\ k\ \mathsf{in}\ \pi\ \mathsf{end\text{-}cont} : B}$$

$$\frac{\Sigma \mid \Gamma, v : A \vdash \pi : B \qquad \Gamma, v : A, \mathsf{now} : G \vdash_\circ \varphi : \{\top, \bot\}}{\Sigma \mid \Gamma \vdash \mathsf{case}\ v : A.\ \varphi\ \mathsf{in}\ \pi : B} \qquad \frac{\Gamma, \mathsf{now} : G \vdash_\circ e : A}{\Sigma, f : A \to B \mid \Gamma \vdash f(e) : B}$$

$$\frac{\Sigma \mid \Gamma \vdash c_1 : B \quad \cdots \quad \Sigma \mid \Gamma \vdash c_n : B}{\Sigma \mid \Gamma \vdash \mathsf{switch}\ c_1 \ldots c_n\ \mathsf{end\text{-}switch} : B} \qquad \frac{\Gamma, \mathsf{now} : G \vdash_\circ e : A}{\Sigma, k : A \to \emptyset \mid \Gamma \vdash k(e) : B}$$

$$\frac{\begin{array}{ll}\Sigma, f : A \to B \mid \Gamma, v : A \vdash \pi_0 : B & \Sigma, f : A \to B \mid \Gamma, S : \mathcal{P}(A \times G) \vdash \pi_1 : (A \times G) \times B \\ \Sigma, f : A \to B \mid \Gamma \vdash \pi_2 : C & \Gamma \vdash_\circ r : \mathcal{P}((A \times G) \times (A \times G)) \\ \Gamma, v : A, \mathsf{now} : G \vdash_\circ \varphi_0 : \{\top, \bot\} & \Gamma, v : A, u : B, \mathtt{old} : G, \mathsf{now} : G \vdash_\circ \varphi_1 : \{\top, \bot\}\end{array}}{\Sigma \mid \Gamma \vdash \mathsf{rec}\ f(v : A) : \langle \varphi_0 \hookrightarrow u : B.\ \varphi_1 \rangle\ \mathsf{progress}(r)\ = \pi_0\ \mathsf{diverges}(S) : \pi_1\ \mathsf{in}\ \pi_2\ :\ C}$$

Fig. 11. Typing rules for $\mathcal{M}$.

In order to characterize the behavior of target programs across infinite executions, we require the set G to be equipped with a partial order $\preccurlyeq$ such that every non-empty chain $X \subseteq G$ admits a least upper bound[1]. We shall require that the target program state now can only grow with respect to $\preccurlyeq$ during the execution. This can be achieved, e.g., by including the wall clock in now, as in the example in Section 3.2, or by treating G as the set of execution traces, as we do below in Section 4.3.

The typing system for language $\mathcal{M}$ is defined using a standard relation $\Sigma \mid \Gamma \vdash \pi : B$, where π is the program being typed, B is the type of its return value, and the contexts Γ and Σ assign types to variables and functions, respectively. We assume that mathematical expressions and logical formulas are typed according to some preexisting typing relation $\vdash_\circ$. The typing rules for $\mathcal{M}$ are shown in Figure 11. Notice that we do not provide any special rules for the NEXT functions. As far as typing is concerned, these functions are just ordinary inhabitants of Σ. Also notice the special rule for continuation calls, which allows the call $k(e)$ to assume arbitrary types.

Let us now discuss the most intricate construction in our verification language: the function definition, which introduces a (possibly recursive) function equipped with a handler to process the result of diverging behavior. The function f receives a parameter v of type A and produces a return value of type B. During its execution, f may modify the state of the target program(s) stored in the global variable now, by calling, directly or indirectly, one of the NEXT functions in Σ. The behavior of f is specified by its contract, which consists of a precondition φ_0 and a postcondition φ_1, in which the returned value is named u. Function f is implemented using a program π_0 which may recursively call f.

The progress clause specifies a strict progression order r, with respect to which the parameter of f and the target program state must increase with each recursive call. Recursion behaves in a special way in case of infinite execution, as presented in Section 3. If the recursion stack grows up to a limit state, the divergence handler π_1 is called. The recursion stack is passed to the handler in the form of a set S of parameter/state pairs at the entry of uncompleted calls to f. This set is totally ordered with respect to r. The target program state now is set at the entry of the divergence handler to the least upper bound (w.r.t. $\preccurlyeq$) of the earlier states registered in S.

[1] We do not require a least upper bound for the empty chain, as it is irrelevant for our formalism. Thus $\preccurlyeq$ does not have to be a chain-complete order for G.

The divergence handler is allowed to make recursive calls of f, on condition that the parameter/state pair at the entry of the new call is r-greater than any pair in S. It is thus possible for the recursion stack to reach the next limit state, so that the divergence handler will be called again.

Upon returning, the divergence handler chooses some unfinished recursive call on the stack, characterized by a parameter/state pair $(v, \mathtt{old}) \in S$, and completes this call, returning a value u of type B, such that v, old, and u satisfy the postcondition φ_1. In this way, the divergence handler cuts out an infinite suffix of the call stack and restores the normal execution flow.

4.2 Predicate Transformer Semantics

We give a predicate transformer semantics for $\mathcal{M}$, using weakest precondition transformers in the style of Dijkstra [Dijkstra 1975]. Indeed, since ghost monitors are not intended for execution but only for verification, such proof-oriented semantics is better suited for our purposes. First of all, we need to provide interpretations for the variables and functions in the execution context. As noted earlier, we transparently treat data types as sets.

Definition 4.1. A Γ-*valuation* is a mapping σ that maps each variable $v : A$ in Γ to a value in A. Given a variable valuation σ, we write $[\![e]\!]_\sigma$ and $[\![\varphi]\!]_\sigma$ to denote the standard interpretation of a mathematical expression e and a logical formula φ, respectively.

Definition 4.2. A Σ-*specification context* is a mapping Φ that maps each function $f : A \to B$ in Σ to a pair (P, Q), where $P \subseteq A \times G$ and $Q \subseteq (A \times G) \times (B \times G)$. The set P represents the precondition of f, that is, the set of parameter/state pairs allowed at the entry of f. The set Q represents the postcondition of f which relates the entry parameter/state pairs to the exit result/state pairs.

Informally speaking, functions are interpreted as *contracts*. Notice that the relation Q does not need to be functional, that is, f may have multiple outcomes for the same entry parameter/state pair. Also, f may have admissible entry parameter/state pairs in P that do not correspond to any outcome in Q, e.g., when f escapes by calling a continuation.

We define the semantics of $\mathcal{M}$ as an operator $\mathrm{WP}\langle\Phi, \sigma\rangle(\pi, Q)$, parameterized by a Γ-valuation σ, a Σ-specification context Φ, a well-typed program π with type derivation $\Sigma \mid \Gamma \vdash \pi : B$, and a set $Q \subseteq B \times G$ (the postcondition); the result of WP is a subset of G (the weakest precondition). To simplify notation, we treat the type derivation $\Sigma \mid \Gamma \vdash \pi : B$ as an implicit, rather than explicit, parameter of WP. Intuitively, WP returns a set P of target program states such that for every possible execution of π starting with now in P, the final result and target state will be in Q.

To handle function definitions, we also define a predicate $\mathrm{PC}\langle(\Phi_x)_{x\in G}, \sigma\rangle(\pi : \langle\varphi_0 \hookrightarrow u : B.\ \varphi_1\rangle)$, expressing the correctness of π with respect to the contract $\langle\varphi_0 \hookrightarrow u : B.\ \varphi_1\rangle$. Unlike WP, this correctness predicate is parameterized by a family of specification contexts $(\Phi_x)_{x\in G}$; this allows us to handle the progression preconditions for recursive function calls. We also use the following definition to describe non-terminating executions:

Definition 4.3. For a given type A and a strict ordering relation R on $A \times G$, a set $H \subseteq A \times G$ is called a *limit stack* with respect to R, if the following conditions hold:

(a) H is totally ordered by R,
(b) H is non-empty and does not have a maximum element,
(c) for any two pairs $(a_0, x_0), (a_1, x_1)$ in H, if $((a_0, x_0), (a_1, x_1)) \in R$ then $x_0 \preccurlyeq x_1$.

The *top state* of H, denoted $\mathrm{top}(H)$, is then defined as the least upper bound of $\{x \mid (a, x) \in H\}$.

The full definitions of WP and PC are given in Figure 12. Let us provide some informal explanations about these rules.

$$\mathrm{WP}\langle\Phi,\sigma\rangle(f(e),\,Q) \;\triangleq\; \{\, x \in G \mid (a_x, x) \in P_f \wedge \forall b, y.\, ((a_x, x), (b, y)) \in Q_f \Rightarrow (b, y) \in Q \,\}$$
$$\text{where } (P_f, Q_f) = \Phi(f) \text{ and } a_x = [\![e]\!]_{\sigma[\mathsf{now}\leftarrow x]}$$

$$\mathrm{WP}\langle\Phi,\sigma\rangle(\text{let } v = \pi_0 \text{ in } \pi_1,\,Q) \;\triangleq\; \mathrm{WP}\langle\Phi,\sigma\rangle(\pi_0,\,\{\,(a,x) \in A \times G \mid x \in \mathrm{WP}\langle\Phi,\sigma[v \leftarrow a]\rangle(\pi_1, Q)\,\})$$
$$\text{where } \Sigma \mid \Gamma \vdash \pi_0 : A$$

$$\mathrm{WP}\langle\Phi,\sigma\rangle(\text{cont } k \text{ in } \pi \text{ end-cont},\,Q) \;\triangleq\; \mathrm{WP}\langle\Phi[k \leftarrow (Q,\emptyset)],\sigma\rangle(\pi, Q)$$

$$\mathrm{WP}\langle\Phi,\sigma\rangle(\text{case } v : A.\ \varphi \text{ in } \pi,\,Q) \;\triangleq\; \{\, x \in G \mid \forall a \in A.\, [\![\varphi]\!]_{\sigma[v\leftarrow a,\mathsf{now}\leftarrow x]} \Rightarrow x \in \mathrm{WP}\langle\Phi,\sigma[v \leftarrow a]\rangle(\pi, Q)\,\}$$

$$\mathrm{Guard}\langle\sigma\rangle(\text{case } v : A.\ \varphi \text{ in } \pi) \;\triangleq\; \{\, x \in G \mid \exists a \in A.\, [\![\varphi]\!]_{\sigma[v\leftarrow a,\mathsf{now}\leftarrow x]}\,\}$$

$$\mathrm{WP}\langle\Phi,\sigma\rangle(\text{switch } c_1 \ldots c_n \text{ end-switch},\,Q) \;\triangleq\; \Big(\bigcup_{i=1}^{n} \mathrm{Guard}\langle\sigma\rangle(c_i)\Big) \cap \Big(\bigcap_{i=1}^{n} \mathrm{WP}\langle\Phi,\sigma\rangle(c_i, Q)\Big)$$

$$\mathrm{PC}\langle(\Phi_x)_{x\in G},\sigma\rangle(\pi : \langle\varphi_0 \hookrightarrow u : B.\ \varphi_1\rangle) \;\triangleq\; \forall x \in G.\, [\![\varphi_0]\!]_{\sigma[\mathsf{now}\leftarrow x]} \Rightarrow x \in \mathrm{WP}\langle\Phi_x,\sigma\rangle(\pi, Q_x)$$
$$\text{where } Q_x = \{\,(b,y) \in B \times G \mid [\![\varphi_1]\!]_{\sigma[u\leftarrow b,\mathsf{now}\leftarrow y,\mathtt{old}\leftarrow x]}\,\}$$

$$\mathrm{WP}\langle\Phi,\sigma\rangle(\text{rec } f(v : A) : \langle\varphi_0 \hookrightarrow u : B.\ \varphi_1\rangle \text{ progress}(r) = \pi_0 \text{ diverges}(S) : \pi_1 \text{ in } \pi_2,\,Q) \;\triangleq\; \begin{cases} \mathrm{WP}\langle\Phi^{\mathrm{rec}}_{\emptyset},\sigma\rangle(\pi_2, Q) & \text{if the conditions (a)-(c) below hold,} \\ \emptyset & \text{otherwise} \end{cases}$$

(a) $[\![r]\!]_\sigma$ is a strict order on $A \times G$
(b) $\forall a \in A.\ \mathrm{PC}\langle(\Phi^{\mathrm{rec}}_{\{(a,x)\}})_{x\in G},\sigma[v \leftarrow a]\rangle(\pi_0 : \langle\varphi_0 \hookrightarrow u : B.\ \varphi_1\rangle)$
(c) for all $H \subseteq A \times G$ such that H is a limit stack w.r.t. $[\![r]\!]_\sigma$,
$\big(\forall (a,x) \in H.\, [\![\varphi_0]\!]_{\sigma[v\leftarrow a,\mathsf{now}\leftarrow x]}\big) \;\Rightarrow\; \mathrm{top}(H) \in \mathrm{WP}\langle\Phi^{\mathrm{rec}}_H,\sigma[S \leftarrow H]\rangle\big(\pi_1, Q^{\mathrm{lim}}_H\big)$

where
$$\begin{aligned} \Phi^{\mathrm{rec}}_H &= \Phi[f \leftarrow (P^{\mathrm{rec}}_H, Q^{\mathrm{rec}})] \\ P^{\mathrm{rec}}_H &= \{\,(a,x) \in A \times G \mid [\![\varphi_0]\!]_{\sigma[v\leftarrow a,\mathsf{now}\leftarrow x]} \wedge \forall p \in H.\, (p,(a,x)) \in [\![r]\!]_\sigma\,\} \\ Q^{\mathrm{rec}} &= \{\,((a,x),(b,y)) \in (A \times G) \times (B \times G) \mid [\![\varphi_1]\!]_{\sigma[v\leftarrow a,u\leftarrow b,\mathsf{now}\leftarrow y,\mathtt{old}\leftarrow x]}\,\} \\ Q^{\mathrm{lim}}_H &= \{\,(((a,x),b),y) \in ((A \times G) \times B) \times G \mid (a,x) \in H \wedge [\![\varphi_1]\!]_{\sigma[v\leftarrow a,u\leftarrow b,\mathsf{now}\leftarrow y,\mathtt{old}\leftarrow x]}\,\} \end{aligned}$$

Fig. 12. Predicate transformer semantics for $\mathcal{M}$.

In the rule for function call $f(e)$, the WP operator returns the set of states x such that the value of expression e in state x (denoted a_x) satisfies the precondition of function f and all possible outcomes of f applied to a_x in state x (according to the postcondition of f) satisfy the desired postcondition Q. This matches well the treatment of function calls in standard definitions of a weakest precondition transformer.

Similarly, our rule for program sequence (let-in) is a straightforward set-based rendition of a standard weakest precondition rule.

The rule for continuation capture simply binds continuation k to a contract whose precondition is the desired postcondition Q and whose postcondition is empty, meaning that k cannot return to the caller. This rule adapts in a natural way the typing scheme of the `call/cc` operator.

The semantics of the guarded choice requires that, first, at least one case is eligible, and, second, for every possible realization of each guard (that is, for every possible witness that satisfies the guard's condition), the execution of the selected branch leads to an outcome in Q. The first part is expressed as the disjunction of the guard conditions, and the second part is expressed as the conjunction of the weakest preconditions for each case.

The program correctness operator PC is a straightforward adaptation of a Hoare triple: any execution of program π starting in a state that satisfies precondition φ_0 will lead to an outcome that satisfies postcondition φ_1. Since we also capture divergent executions, this is a statement of total correctness. The only peculiarity of our definition is that the specification context Φ is parametrized by the starting state.

Finally, in the rule for recursive function definitions, the definition itself is admitted under three conditions (a)-(c):

(a) The progression relation r is indeed a strict order.

(b) The main body of function f, denoted π_0, is correct with respect to the given specification. Notice that for every parameter/state pair (a, x), we verify π_0 under a dedicated specification context $\Phi^{\mathrm{rec}}_{\{(a,x)\}}$, which enforces an additional precondition for each recursive call of f, namely, that the parameter/state pair at the call entry is strictly greater than (a, x) with respect to the progression order r.

(c) The divergence handler of f, denoted π_1, is correct for every limit stack H in which every parameter/state pair satisfies φ_0, and the corresponding postcondition Q^{lim}_H. This postcondition requires π_1 to return a pair $(a, x) \in H$ together with a value b. The pair (a, x) indicates an unfinished call in the stack that the handler will complete (thus cutting out the infinite sequence of unfinished recursive calls after (a, x) in the stack), and the return value b must satisfy the postcondition of the completed call. The state in which π_1 starts its execution is the least upper bound of states in H, denoted $\mathrm{top}(H)$. If function f is called during the execution of π_1, the specification context Φ^{rec}_H will ensure progression with respect to every unfinished call in H.

4.3 Linking a Monitor to Target Programs

For a ghost monitor, each target program is presented as a transition system that corresponds to a small-step[2] operational semantics of this program. Under this view, each `NEXT` procedure amounts to one step in the transition system. In order to ensure the existence of least upper bounds for infinite behaviors, it is convenient to consider `now` as a combination of traces (rather than individual states) in these transition systems.

Definition 4.4. For a transition system $\mathcal{S} = (S, \rightarrow)$, we define the *ordered set of traces* $(G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}})$ as the set of potentially infinite sequences of transitions in S, ordered by the prefix order:

- $G_{\mathcal{S}} = \{ (s_n)_{n \in I} \mid I \leqslant \omega \wedge \forall n \in I.\, (n+1) \in I \Rightarrow s_n \rightarrow s_{n+1} \}$
- $(s_n)_{n \in I} \preccurlyeq_{\mathcal{S}} (s'_n)_{n \in J}$ if and only if $I \leqslant J \leqslant \omega$ and $\forall n \in I.\, s_n = s'_n$.

We denote traces with t and we write juxtaposition ty to denote the extension of trace t with a state $y \in S$ (which implies that both t and ty are finite).

Notice the difference with the example in Fig. 9, where the values of `now` are the target programs' states augmented with wall clocks t_C and t_M. The advantage of treating `now` as a combination of

[2]Or rather "medium-step", as we progress from one breakpoint to another, and not instruction by instruction.

traces, rather than states, is that we obtain, in a systematic way, an appropriate order relation and meaningful least upper bounds for G. At the same time, ghost monitors that manipulate traces are more difficult to write and to understand: traces provide a good theoretical framework but a less than convenient practical language. We may have the best of both worlds by projecting traces (finite or infinite) to clock-augmented states: then a ghost monitor working with individual states as in Fig. 9 can be implicitly converted to one formulated in terms of traces. In the finite case, the representative state is simply the last state in the execution trace, and the wall clock in the length of the trace. In the limit case, we use the approach described in Section 3.2, paragraph "Representation of limit states", which defines the representative state as the least upper bound of the infinite trace, restricted to the components of the state that are observable at the limit point of execution. However, we cannot infer which parts of the state must be accumulated and preserved in the limit state from the transition system alone; additional information is required. A generic framework for passing from traces to clock-augmented states via game simulation (in presence of said additional information) is described in the technical report [Clochard et al. 2018b, Definitions 4.1 and 4.2]. This framework is also implemented in our mechanization [Clochard 2018a, File `ordered_transition.mlw`].

We can now state the desired *transfer properties*, which express the soundness of the ghost monitor approach. These are presented in the following three theorems which will be proved in the next section.

The next theorem translates the axiomatic semantics of $\mathcal{M}$ to the total correctness statement for transition systems, and thus justifies the earlier proofs of the Schorr-Waite algorithm and of the iterative implementation of McCarthy's 91 function. As in Figure 9, $\mathbb{U}$ stands for the unit type, and $\Diamond$ for its unique inhabitant.

Theorem 4.5 (Transfer property, demonic case). *Let $\mathcal{S} = (S, \rightarrow)$ be a transition system. Let Σ be a single-element typing context that binds the procedure name* `NEXT` *to the type $\mathbb{U} \rightarrow S$. Let specification context Φ bind* `NEXT` *to the contract*

$$\{(\Diamond, t) \mid \exists y.ty \in G_{\mathcal{S}}\},\ \{((\Diamond, t), (y, ty)) \mid ty \in G_{\mathcal{S}}\}.$$

Let π be a well-typed program with type derivation $\Sigma \mid \Gamma \vdash \pi : \mathbb{U}$. Let σ be a Γ-valuation and $Q \subseteq G_{\mathcal{S}}$, a postcondition set. Suppose that the weakest precondition set $\mathrm{WP}\langle\Phi, \sigma\rangle(\pi, \mathbb{U} \times Q)$ is not empty and let t_0 be a trace in this set. Let t be an arbitrary maximal trace in $G_{\mathcal{S}}$ that extends t_0, that is, $t_0 \preccurlyeq_{\mathcal{S}} t$ and $\forall y.ty \notin G_{\mathcal{S}}$. Then there exists a trace $t_1 \in G_{\mathcal{S}}$ such that $t_0 \preccurlyeq_{\mathcal{S}} t_1 \preccurlyeq_{\mathcal{S}} t$ and $t_1 \in Q$.

Informally, if the target program starts from a trace t_0 generated by WP for postcondition Q, then it will eventually produce a trace in Q, either after a finite number of transitions or in the limit state. The contract of `NEXT` signifies that it can only be called upon a trace that can be extended, i.e., either an empty trace or a finite trace $s_0 s_1 \ldots s_n$, whose last element s_n is in the domain of $\rightarrow$, and for every such trace and every possible transition $s_n \rightarrow s_{n+1}$, `NEXT` may proceed by setting `now` to $s_0 s_1 \ldots s_n s_{n+1}$ and returning s_{n+1}.

In the setting of Theorem 4.5, we consider a demonic interpretation of non-determinism, where the monitor does not choose the next transition. If we consider an angelic interpretation instead, we obtain a transfer property that translates the semantics of $\mathcal{M}$ to the properties of trace existence, in accordance with the intuition that the monitor fully drives the execution of the target code.

Theorem 4.6 (Transfer property, angelic case). *Let $\mathcal{S} = (S, \rightarrow)$ be a transition system. Let Σ be a single-element typing context that binds the procedure name* `NEXT` *to the type $S \rightarrow \mathbb{U}$. Let specification context Φ bind* `NEXT` *to the contract*

$$\{(y, t) \mid ty \in G_{\mathcal{S}}\},\ \{((y, t), (\Diamond, ty)) \mid ty \in G_{\mathcal{S}}\}.$$

Let π be a well-typed program with type derivation $\Sigma \mid \Gamma \vdash \pi : \mathbb{U}$. Let σ be a Γ-valuation and $Q \subseteq G_{\mathcal{S}}$, a postcondition set. Suppose that $\mathrm{WP}\langle\Phi, \sigma\rangle(\pi, \mathbb{U} \times Q)$ is not empty and let t_0 be a trace in this set. Then there exists a trace $t_1 \in G_{\mathcal{S}}$ such that $t_0 \preccurlyeq_{\mathcal{S}} t_1$ and $t_1 \in Q$.

We can also state a transfer property that mixes the two kinds of non-determinism, justifying the simulation diagram in Section 3. Here, we need to handle the state of two target programs.

Theorem 4.7 (Transfer property, mixed case). *Let $\mathcal{S}^\bullet = (S^\bullet, \rightarrow^\bullet)$ and $\mathcal{S}^\circ = (S^\circ, \rightarrow^\circ)$ be a pair of transition systems. Let G be the set of trace pairs $G_{\mathcal{S}^\bullet} \times G_{\mathcal{S}^\circ}$ ordered by the product order $(t_0^\bullet, t_0^\circ) \preccurlyeq (t_1^\bullet, t_1^\circ) \equiv (t_0^\bullet \preccurlyeq_{\mathcal{S}^\bullet} t_1^\bullet \wedge t_0^\circ \preccurlyeq_{\mathcal{S}^\circ} t_1^\circ)$. Let Σ be a two-element typing context that binds $\mathsf{NEXT}^\bullet$ to the type $\mathbb{U} \rightarrow S^\bullet$ and NEXT° to $S^\circ \rightarrow \mathbb{U}$. Let specification context Φ bind $\mathsf{NEXT}^\bullet$ to the contract*

$$\{(\Diamond, (t^\bullet, t^\circ)) \mid \exists y. t^\bullet y \in G_{\mathcal{S}^\bullet} \wedge t^\circ \in G_{\mathcal{S}^\circ}\},\ \{((\Diamond, (t^\bullet, t^\circ)), (y, (t^\bullet y, t^\circ))) \mid t^\bullet y \in G_{\mathcal{S}^\bullet} \wedge t^\circ \in G_{\mathcal{S}^\circ}\}$$

and NEXT° to the contract

$$\{(y, (t^\bullet, t^\circ)) \mid t^\bullet \in G_{\mathcal{S}^\bullet} \wedge t^\circ y \in G_{\mathcal{S}^\circ}\},\ \{((y, (t^\bullet, t^\circ)), (\Diamond, (t^\bullet, t^\circ y))) \mid t^\bullet \in G_{\mathcal{S}^\bullet} \wedge t^\circ y \in G_{\mathcal{S}^\circ}\}.$$

Let π be a well-typed program with type derivation $\Sigma \mid \Gamma \vdash \pi : \mathbb{U}$. Let σ be a Γ-valuation and $Q \subseteq G_{\mathcal{S}^\bullet} \times G_{\mathcal{S}^\circ}$, a postcondition set. Suppose that $\mathrm{WP}\langle\Phi, \sigma\rangle(\pi, \mathbb{U} \times Q)$ is not empty and let $(t_0^\bullet, t_0^\circ)$ be a pair of traces in this set. Let $t^\bullet$ be an arbitrary maximal trace in $G_{\mathcal{S}^\bullet}$ that extends $t_0^\bullet$, that is, $t_0^\bullet \preccurlyeq_{\mathcal{S}^\bullet} t^\bullet$ and $\forall y. t^\bullet y \notin G_{\mathcal{S}^\bullet}$. Then there exists a pair of traces $(t_1^\bullet, t_1^\circ)$ such that $t_0^\bullet \preccurlyeq_{\mathcal{S}^\bullet} t_1^\bullet \preccurlyeq_{\mathcal{S}^\bullet} t^\bullet$, $t_0^\circ \preccurlyeq_{\mathcal{S}^\circ} t_1^\circ$, and $(t_1^\bullet, t_1^\circ) \in Q$.

Here, $\mathcal{S}^\bullet$ represents the demonic target program and $\mathcal{S}^\circ$, the angelic one. Notice that Theorem 4.7 admits a straightforward generalization to an arbitrary number of transition systems in demonic and angelic modes. In particular, for a single transition system, this reduces to Theorems 4.5 and 4.6. For two transition systems in demonic mode, the axiomatic semantics of $\mathcal{M}$ translates to the systematic correlation of behaviors, which includes determinism properties.

Another slight generalization is shown in the example in Section 3, where the choice of the demonic or angelic role is not fixed in the monitor but determined by a special flag added to the target program state.

5 UNIFYING TRANSFER PROPERTIES USING GAMES

In order to model the various transfer properties in a single framework, we use the formal model of *games*, which can be thought as a natural generalization of transition systems in presence of both angelic and demonic non-determinism. In this setting, the validity of a Hoare triple corresponds to the existence of a winning strategy for the angelic player. In particular, this turns ghost monitors into tools to prove the existence of such a winning strategy. Indeed, the generalized transfer property, which we state in this section, translates valid contracts for monitors to winning strategies in the underlying game. For games that encode transition systems presented above, this general result implies Theorems 4.5-4.7.

5.1 Games and Strategies

As we solely focus on tools to prove the existence of a winning strategy for the angelic player, our definition of games is asymmetric between the two players. The domain of the game corresponds to angelic states, while a demonic state is implicitly defined as a set of angelic states. This set represents the angelic states that can be chosen by the demon after the angel has played his turn. We define the transition function according to this intuition, as a function mapping angelic states to sets of such implicit demonic states, hence sets of sets of angelic states. The angel chooses the demonic state S in the outer set, and the demon chooses the next angelic state in S.

These definitions match the behavior of the NEXT routines of Section 3. Angelic choice (controlled by the monitor) selects the argument passed to NEXT, while demonic choice (not controlled by the monitor) selects the effective updates performed by the target program. From the verification perspective, the postcondition formula of NEXT represents a family of postcondition sets (sets of outcome states), parameterized by the parameter of NEXT. Thus, by fixing this parameter, the angel selects a particular postcondition set, and the demon chooses, non-deterministically, a particular behavior within that postcondition.

As required earlier for $\mathcal{M}$, we also equip our games with an appropriate order on states, so the state only grows during a play, and the limit states correspond to the least upper bounds.

Definition 5.1 (Game). A *game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ is a partially ordered set $(G, \preccurlyeq)$ such that every non-empty chain in G admits a least upper bound, equipped with a function Δ from G to $\mathcal{P}(\mathcal{P}(G))$ such that

$$\forall x \in G.\, \forall X \in \Delta(x).\, \forall y \in X.\, x \prec y.$$

We call G, $\preccurlyeq$, and Δ, respectively, the *domain*, the *order*, and the *transitions* of the game.

Note that the current state of a play may contain insufficient information for a winning strategy to drive the play in the right direction. Indeed, a monitor keeping track of auxiliary data naturally translates into a strategy that uses memory. Thus we introduce the notion of *history* to represent the complete memory of a play.

Definition 5.2 (History, Prefix Order). Given a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$, a *history* of $\mathbb{G}$ is a non-empty chain in G that admits a maximum. Given two histories H_1, H_2, we say that H_1 is a *prefix* of H_2, if $H_1 \subseteq H_2$ and for all $x \in H_2 \backslash H_1$, x is an upper bound of H_1.

One might wonder why we define histories via totally ordered sets instead of regular natural-indexed sequences, which we used for traces in Section 4.3. While a single target program cannot indeed continue after an infinite amount of steps, a game may need to encode several target programs combined in order to verify relational properties (cf. Section 3). Since our framework does not require the target programs to advance in lock-step, it is perfectly possible for one target program to diverge (i.e., to reach a limit state), while another target program is still able to advance, leading to plays of ordinal length up to $\omega \cdot k$, where k is the number of target programs.

Definition 5.3 (Strategies). Given a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$, an *existential strategy* for $\mathbb{G}$ is a function $\sigma_\exists$ that maps histories of $\mathbb{G}$ to the subsets of G. It makes a *valid choice* at H if $\sigma_\exists(H) \in \Delta(\max H)$. A *universal strategy* for $\mathbb{G}$ is a function $\sigma_\forall$ that maps subsets of G to the elements of G. It makes a *valid choice* at X if $\sigma_\forall(X) \in X$.

While existential strategies (employed by the angelic player) have access to the full history of the play, universal strategies (employed by the demonic player) get to choose based only on the current demonic state. This asymmetry comes from the fact that we only care about the existence of winning strategies for the existential side. Indeed, while memory may be necessary to create a winning strategy, it is not necessary to prevent a particular strategy from winning. In this situation, we may assume that the demonic player has complete knowledge both of the existential strategy and of the initial state before picking his own universal strategy. Together with the current state and knowledge of his past choices, the demonic player can rebuild the complete history of the play, and choose accordingly.

Definition 5.4 (Play). Given a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$, an existential/universal strategy pair $(\sigma_\exists, \sigma_\forall)$, and an element $x \in G$, the *play from x for* $(\sigma_\exists, \sigma_\forall)$ is the smallest set of histories $\mathcal{P}$ such that:

(a) $\{x\} \in \mathcal{P}$,

(b) for all $H \in \mathcal{P}$, if $\sigma_\exists$ makes a valid choice at H, and $\sigma_\forall$ makes a valid choice at $\sigma_\exists(H)$, then $H \cup \{\sigma_\forall(\sigma_\exists(H))\} \in \mathcal{P}$,
(c) for all non-empty $\mathcal{H} \subseteq \mathcal{P}$ totally ordered by the prefix order, $\bigcup \mathcal{H} \cup \{\sup(\bigcup \mathcal{H})\} \in \mathcal{P}$.

The three propagation rules above correspond precisely to the rules to build the history of a play incrementally, including the divergent behaviors: initialization, regular step, and limit step.

Definition 5.5 (Winning strategy). An existential strategy $\sigma_\exists$ for a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$ is *winning* for an initial set $P \subseteq G$ and a target set $Q \subseteq G$, if for any universal strategy $\sigma_\forall$, every play for $(\sigma_\exists, \sigma_\forall)$ starting within P contains a history H such that $\max H \in Q$, or such that $\sigma_\exists(H)$ is a valid choice but $\sigma_\forall(\sigma_\exists(H))$ is not.

We cannot only consider strategies that always make valid choices, since a particular game may put either of the players in a situation where they have no valid choice: no suitable transition for the angelic player or an empty set of angelic states to choose from for the demonic one. Thus we have to include the invalid choices by the demon as a winning condition for the angel.

5.2 Encoding Transition Systems as Games

We show how the transition systems from Section 4.3 can be encoded as games, so that the properties over traces in Theorems 4.5-4.7 can be uniformly expressed in terms of winning strategies.

Definition 5.6 (Universal game). The *universal game* for a transition system $\mathcal{S} = (S, \rightarrow)$ is the game $\mathcal{G}_{\mathcal{S},\forall} = (G_\mathcal{S}, \preccurlyeq_\mathcal{S}, \Delta_{\mathcal{S},\forall})$ with transitions defined by

$$\Delta_{\mathcal{S},\forall}(t) \;=\; \{\, \{\, ty \mid ty \in G_\mathcal{S} \,\} \,\} \setminus \{\, \emptyset \,\}.$$

In this game, the angelic player always gets at most one choice, and the demonic player may choose any admissible extension for the current trace. Moreover, if the current trace does not admit an extension (e.g., when it is infinite), the angelic player cannot make his move at all: eliminating $\emptyset$ from the set of demonic states corresponds to the precondition $\exists y.ty \in G_\mathcal{S}$ of `NEXT` in Theorem 4.5.

Definition 5.7 (Existential game). The *existential game* for a transition system $\mathcal{S} = (S, \rightarrow)$ is the game $\mathcal{G}_{\mathcal{S},\exists} = (G_\mathcal{S}, \preccurlyeq_\mathcal{S}, \Delta_{\mathcal{S},\exists})$ with transitions defined by

$$\Delta_{\mathcal{S},\exists}(t) \;=\; \{\, \{ty\} \mid ty \in G_\mathcal{S} \,\}.$$

Here, the choice of the next transition belongs to the angelic player, and the move of the demonic player is enforced by the angel's choice.

Definition 5.8 (Simulation game). Given two transition systems $\mathcal{S}^\bullet, \mathcal{S}^\circ$, the *simulation game* from $\mathcal{S}^\bullet$ to $\mathcal{S}^\circ$ is the game $\mathcal{G}_{\mathcal{S}^\bullet \to \mathcal{S}^\circ} = (G_{\mathcal{S}^\bullet} \times G_{\mathcal{S}^\circ}, \preccurlyeq_{\mathcal{S}^\bullet} \times \preccurlyeq_{\mathcal{S}^\circ}, \Delta_{\mathcal{S}^\bullet \to \mathcal{S}^\circ})$ with transitions defined by

$$\Delta_{\mathcal{S}^\bullet \to \mathcal{S}^\circ}((t^\bullet, t^\circ)) \;=\; \{\, X \times \{t^\circ\} \mid X \in \Delta_{\mathcal{S}^\bullet,\forall}(t^\bullet) \,\} \;\cup\; \{\, \{t^\bullet\} \times Y \mid Y \in \Delta_{\mathcal{S}^\circ,\exists}(t^\circ) \,\}.$$

Note that the transitions of the simulation game correspond to the union of possible transitions represented by the two `NEXT` functions in Theorem 4.7: the angelic player, i.e., the ghost monitor itself, decides which one of the two target programs moves to the next breakpoint.

The following lemmas link the properties over traces in Theorems 4.5-4.7 to the winning strategies in these three games.

Lemma 5.9. *For all $\mathcal{S} = (S, \rightarrow)$ and $P, Q \subseteq G_\mathcal{S}$, there exists a winning strategy for game $\mathcal{G}_{\mathcal{S},\forall}$ for initial set P and target set Q if and only if, for any maximal trace $t \in G_\mathcal{S}$ that extends a trace $t_0 \in P$, there exists a trace $t_1 \in G_\mathcal{S}$ such that $t_0 \preccurlyeq_\mathcal{S} t_1 \preccurlyeq_\mathcal{S} t$ and $t_1 \in Q$.*

Lemma 5.10. *For all $\mathcal{S} = (S, \rightarrow)$ and $P, Q \subseteq G_{\mathcal{S}}$, there exists a winning strategy for game $\mathcal{G}_{\mathcal{S},\exists}$ for initial set P and target set Q if and only if, for all $t_0 \in P$, there exists a trace $t_1 \in G_{\mathcal{S}}$ such that $t_0 \preccurlyeq_{\mathcal{S}} t_1$ and $t_1 \in Q$.*

Lemma 5.11. *For all transition systems $\mathcal{S}^\bullet = (S^\bullet, \rightarrow^\bullet)$, $\mathcal{S}^\circ = (S^\circ, \rightarrow^\circ)$ and all $P, Q \subseteq G_{\mathcal{S}^\bullet} \times G_{\mathcal{S}^\circ}$, there exists a winning strategy for the simulation game $\mathcal{G}_{\mathcal{S}^\bullet \rightarrow \mathcal{S}^\circ}$ for initial set P and target set Q if and only if, for every pair of traces $(t_0^\bullet, t_0^\circ) \in P$ and every maximal trace $t^\bullet \in G_{\mathcal{S}^\bullet}$ that extends $t_0^\bullet$, there exists a pair of traces $(t_1^\bullet, t_1^\circ)$ such that $t_0^\bullet \preccurlyeq_{\mathcal{S}^\bullet} t_1^\bullet \preccurlyeq_{\mathcal{S}^\bullet} t^\bullet$, $t_0^\circ \preccurlyeq_{\mathcal{S}^\circ} t_1^\circ$, and $(t_1^\bullet, t_1^\circ) \in Q$.*

5.3 From Weakest Preconditions to Winning Strategies

We now state the main soundness result for our verification language, as a transfer property from language $\mathcal{M}$ to games. Essentially, we claim that weakest preconditions for $\mathcal{M}$ induce the existence of winning strategies.

Definition 5.12. A contract (P, Q) where $P \subseteq A \times G$ and $Q \subseteq (A \times G) \times (B \times G)$, is said to be *valid* with respect to game $\mathbb{G} = (G, \preccurlyeq, \Delta)$ if for all $(a, x) \in P$, there exists a winning strategy for the initial set $\{x\}$ and the target set $\{\, y \in G \mid \exists b \in B.\, ((a, x), (b, y)) \in Q \,\}$. By extension, a Σ-specification context Φ is *valid* with respect to $\mathbb{G}$ if all contracts in Φ are valid with respect to $\mathbb{G}$.

Theorem 5.13. *Consider a game $\mathbb{G}$, a well-typed $\mathcal{M}$ program $\Sigma \mid \Gamma \vdash \pi : B$, a Γ-valuation σ, and a Σ-specification context Φ valid with respect to $\mathbb{G}$. Then for all $Q \subseteq B \times G$, there exists a winning strategy for the initial set $\mathrm{WP}\langle \Phi, \sigma \rangle(\pi, Q)$ and the target set $\{\, y \in G \mid \exists b \in B.\, (b, y) \in Q \,\}$.*

The proof is presented in the technical report [Clochard et al. 2018b]. One peculiar feature of the proof is that it handles recursive definitions and continuation bindings by constructing new games with extra transitions representing the additional function calls. In the case of recursion, this produces games with an arbitrary interleaving of angelic and demonic non-determinism. These games do not retain the simple structure inherited from transition systems (as shown in Section 5.2), which justifies *a posteriori* our choice of the generic game-based framework.

In order to derive Theorems 4.5-4.7 as corollaries of Theorem 5.13, we only need to show that the contracts of functions `NEXT` are indeed valid with respect to a game that appropriately encodes the original transition systems, as shown in Definitions 5.6-5.8. In fact, the functions `NEXT` from Section 4.3 can all be seen as special cases (and even implemented as wrappers) of a single generic transition function, described by the following definition.

Definition 5.14. We define the *transition contract* $(P_{\mathbb{G}}, Q_{\mathbb{G}})$ associated to game $\mathbb{G} = (G, \preccurlyeq, \Delta)$ as

$$P_{\mathbb{G}} = \{\, (X, x) \in \mathcal{P}(G) \times G \mid X \in \Delta(x) \,\}$$
$$Q_{\mathbb{G}} = \{\, ((X, x), (\Diamond, y)) \in \mathcal{P}(G) \times G \times \mathbb{U} \times G \mid y \in X \,\}.$$

We define the *base specification context* $\Phi_{\mathbb{G}}$ as a single-element specification context that binds function `NEXT` of type $\mathcal{P}(G) \rightarrow \mathbb{U}$ to the transition contract $(P_{\mathbb{G}}, Q_{\mathbb{G}})$.

When a ghost monitor calls the function `NEXT` from $\Phi_{\mathbb{G}}$, it selects a particular demonic state (a set of possible behaviors for the demon to choose from) to pass to `NEXT` as an argument. This selection determines, in particular, which target program is going to be executed and whatever choices the monitor can make for that target program.

Lemma 5.15. *Given a game $\mathbb{G}$, the base specification context $\Phi_{\mathbb{G}}$ is valid with respect to $\mathbb{G}$.*

Theorem 5.16. *Consider a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$, a well-typed $\mathcal{M}$ program π with type derivation $\texttt{NEXT} : \mathcal{P}(G) \rightarrow \mathbb{U} \mid \Gamma \vdash \pi : \mathbb{U}$, and a Γ-valuation σ. Then for all $Q \subseteq G$, there exists a winning strategy for the initial set $\mathrm{WP}\langle \Phi_{\mathbb{G}}, \sigma \rangle(\pi, \mathbb{U} \times Q)$ and the target set Q.*

Finally, we derive from Theorem 5.16 an immediate corollary that states the existence of a winning strategy relative to a specification written as a pair of logical formulas, provided that we can write a ghost monitor that satisfies this specification.

Corollary 5.17. *Consider a game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$*, a well-typed* $\mathcal{M}$ *program* π *with type derivation* $\mathsf{NEXT} : \mathcal{P}(G) \to \mathbb{U} \,|\, \Gamma \vdash \pi : \mathbb{U}$*, and a* Γ*-valuation* σ*. Consider a pair of logical formulas* φ_0*,* φ_1 *such that* $\Gamma, \mathsf{now} : G \vdash_\circ \varphi_0$ *and* $\Gamma, \mathsf{now} : G, \mathtt{old} : G \vdash_\circ \varphi_1$*. Suppose that the program correctness property* $\mathrm{PC}\langle(\Phi_{\mathbb{G}})_{x\in G}, \sigma\rangle(\pi : \langle \varphi_0 \hookrightarrow u : \mathbb{U}.\ \varphi_1\rangle)$ *holds. Then for all* $x \in G$ *such that* $[\![\varphi_0]\!]_{\sigma[\mathsf{now}\leftarrow x]}$*, there exists a winning strategy for the initial set* $\{x\}$ *and the target set* $\{\, y \in G \mid [\![\varphi_1]\!]_{\sigma[\mathsf{now}\leftarrow y, \mathtt{old}\leftarrow x]} \,\}$*.*

5.4 Relative Completeness

For the purposes of program verification, we only need our methodology to be sound. It turns out, however, that it is also relatively complete, in the sense that if we can prove the existence of a winning strategy for given initial and target sets in the meta-logic, then we can also prove its existence by finding a ghost monitor that satisfies the corresponding contract. The proof amounts to creating a program in $\mathcal{M}$ that follows the explicit winning strategy. This requires a few hypotheses about the meta-logic, notably that it can reason about winning strategies for the considered game in a manner compatible with our definition.

Lemma 5.18. *Consider a game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$*, a typing context* Γ*, a* Γ*-valuation* σ*, and two logical formulas* φ_0*,* φ_1 *such that* $\Gamma, \mathsf{now} : G \vdash_\circ \varphi_0$ *and* $\Gamma, \mathsf{now} : G \vdash_\circ \varphi_1$*. Suppose that the existence of a winning strategy for the initial set* $\{\, x \in G \mid [\![\varphi_0]\!]_{\sigma[\mathsf{now}\leftarrow x]} \,\}$ *and the target set* $\{\, y \in G \mid [\![\varphi_1]\!]_{\sigma[\mathsf{now}\leftarrow y]} \,\}$ *is provable in the meta-logic. Then there is an* $\mathcal{M}$ *program* π *such that* $\mathsf{NEXT} : \mathcal{P}(G) \to \mathbb{U} \,|\, \Gamma \vdash \pi : \mathbb{U}$ *and the program correctness property* $\mathrm{PC}\langle(\Phi_{\mathbb{G}})_{x\in G}, \sigma\rangle(\pi : \langle \varphi_0 \hookrightarrow u : \mathbb{U}.\ \varphi_1\rangle)$ *holds.*

6 RELATED AND FUTURE WORK

Hoare logic, games, and dual non-determinism. Games were introduced as a model for weakest precondition transformers by Back and von Wright in 1990 [Back and Wright 1990]. A related Hoare logic is studied by Mamouras in 2016 [Mamouras 2016]. In both cases, the goal is to model and verify programming languages containing dual non-determinism. Our work differs in the sense that games are used as a model of small-step semantics, and that the nature of non-determinism is only chosen to obtain the transfer property we are interested in. In particular, we may obtain games with both types of non-determinism when considering relational properties, and also internally since our proof of soundness constructs such hybrid games for the recursion rule.

Hoare logics for machine code. Hoare logics defined for machine code in proof assistants are typically defined by proof rules which are mostly unrelated to the syntax of the underlying program. Such logics are closely related to our work, as proof rules match the syntactic constructs of ghost monitors. Moreover, automation by tactics may simulate a weakest precondition calculus. A typical example is the work of Myreen [Myreen and Gordon 2007], which corresponds to a `While` language extended by well-founded recursion for the auxiliary language. Note that well-founded recursion is inherently tied to the meta-logic induction. In contrast, we support non-terminating recursion and proof of diverging behaviors.

Step-indexed logics like Iris [Krebbers et al. 2017] support non-terminating recursion by the means of Löb's rule, as well as arbitrary higher-order programs. However, this rule, in its standard form, precludes us from proving total correctness. There are variants of Iris [Tassarotti et al. 2017] that remove Löb's rule or restrict its application in order to handle termination proofs. There are also works that allow us to prove the program complexity bounds in Iris using time credits [Mével et al. 2019]. Such proof entails termination, of course, but proving complexity is generally a more

difficult task and is not always feasible. It seems that no consensus has been reached so far on how to combine proof of total correctness with proof of diverging programs in the same step-indexed framework.

Non-standard rules for Hoare logic. Some basic cases of our approach can be covered by alternative proof rules for Hoare logic. Within the context of separation logic, Tuerk proposed an alternative postcondition-based proof rule for while loops [Tuerk 2010], with the intent of making specification and proofs simpler by exploiting local reasoning within loops. From the perspective of our work, this proof rule can be directly derived by defining a tail-recursive ghost monitor for the while loop.

Relational properties of programs. Traditional approaches consist in considering a product of target programs. This requires matching their control structure [Banerjee et al. 2016], possibly modulo various program transformations [Kiefer et al. 2018] (loop unrolling, function inlining) and refinement [Barthe et al. 2011]. Our approach improves on that by using an external monitor program that controls the execution of the target programs, keeping their states in sync even if their control structure is completely different.

The approach proposed by Ulbrich [Ulbrich 2013] is more similar to ours in that it does not make assumptions over the control structures of the target programs. The framework of *Unstructured Dynamic Logic* (UDL) used for program specification and verification in this work is sufficiently expressive to encode the control structure of a simple monitor, allowing essentially the same kind of proof as we do here. While this may lead one to think that there is a Curry-Howard correspondence between monitors and UDL proofs, not all monitors can be conveniently expressed in this approach. Of course, one cannot embed general monitors that use divergence handlers within UDL. Also, loop rules for UDL require that all target programs progress during an iteration. In contrast, a single iteration for our monitors may result in no changes for some (or any) target programs. We expect that a similar issue would arise when adding recursion to UDL.

Future work. Our approach can be extended in several directions. A first direction is towards separation logic. We believe that adding a separation logic layer *a posteriori* in a similar fashion as the first-order fragment of Iris should not be difficult. Another direction is to generalize our approach to a higher-order setting. This is reasonably easy if we enforce certain restrictions of the higher-order features: either by limiting the rank of the allowed functions, or by limiting the usage of recursive calls so that they do not occur under closures that are passed to recursive calls as arguments. It is an open question whether it is possible to remove both limitations. A last possible direction is to consider concurrency: while our approach obviously supports sequential consistency in target programs, it is not clear that it is usable for this purpose in practice. In particular, our auxiliary language has no support for parallel program composition.

One application we have already started to investigate is verified compilation [Clochard 2018b, Chapter 6]. The idea is that a compiler can generate, alongside each compiled fragment, a ghost monitor that proves the simulation property between the source code and the compiled code. We expect our approach to be particularly effective in the presence of compiler optimizations that modify the control structure of the source code, e.g., loop permutations. In this way, we can make verification of compilers more amenable to automated proof, in comparison to existing projects that mostly rely on interactive proof assistants.

At this stage, constructing a ghost monitor is a manual, creative process: just as creative as instrumenting a program with assertions and invariants sufficient to complete its proof. A major line of further research would be combining methods for invariant generation and program synthesis to mechanize the discovery of ghost monitors.

ACKNOWLEDGMENTS

We gratefully thank the reviewers for their detailed and constructive feedback on the preliminary versions of this paper. We also thank Robbert Krebbers for useful comments and remarks, in particular on the related work on Iris.

REFERENCES

Ralph-Johan Back and Joakim von Wright. 1990. Duality in specification languages: a lattice-theoretical approach. *Acta Informatica* 27, 7 (July 1990), 583–625. https://doi.org/10.1007/BF00259469

Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 2016. Relational Logic with Framing and Hypotheses. In *Foundations of Software Technology and Theoretical Computer Science (Leibniz International Proceedings in Informatics)*, Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen (Eds.), Vol. 65. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 11:1–11:16.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *International Conference on Formal Methods (Lecture Notes in Computer Science)*, Vol. 6664. Springer, 200–214.

Richard Bornat. 2000. Proving Pointer Programs in Hoare Logic. In *Mathematics of Program Construction*. 102–126.

Martin Clochard. 2018a. Hoare Logic and Games. Formal development, http://toccata.lri.fr/gallery/hoare_logic_and_games.en.html.

Martin Clochard. 2018b. *Méthodes et outils pour la spécification et la preuve de propriétés difficiles de programmes séquentiels*. Thèse de Doctorat. Université Paris-Saclay. https://tel.archives-ouvertes.fr/tel-01787689.

Martin Clochard, Jean-Christophe Filliâtre, and Claude Marché. 2018a. Variations on the McCarthy's 91 Function. Formal development, http://toccata.lri.fr/gallery/mccarthy.fr.html.

Martin Clochard and Claude Marché. 2018. Schorr-Waite Algorithm proved using a Ghost Monitor. Formal development, http://toccata.lri.fr/gallery/schorr_waite_with_ghost_monitor.en.html.

Martin Clochard, Andrei Paskevich, and Claude Marché. 2018b. *Deductive Verification via Ghost Debugging*. Research Report 9219. Inria. https://hal.inria.fr/hal-01907894.

Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18 (August 1975), 453–457. Issue 8. https://doi.org/10.1145/360933.360975

Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The Spirit of Ghost Code. *Formal Methods in System Design* 48, 3 (2016), 152–174. https://doi.org/10.1007/s10703-016-0243-x https://hal.archives-ouvertes.fr/hal-01396864v1.

Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. 2018. Relational Program Reasoning Using Compiler IR. *Journal of Automated Reasoning* 60, 3 (March 2018), 337–363. https://doi.org/10.1007/s10817-017-9433-5

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *26th European Symposium on Programming Languages and Systems (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16 (Lecture Notes in Computer Science)*, Vol. 6355. Springer, 348–370.

Konstantinos Mamouras. 2016. Synthesis of Strategies Using the Hoare Logic of Angelic and Demonic Nondeterminism. *Logical Methods in Computer Science* 12, 3 (2016). https://doi.org/10.2168/LMCS-12(3:6)2016

Zohar Manna and John McCarthy. 1970. Properties of programs and partial function logic. 5 (1970), 79–98.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *European Symposium on Programming (LNCS)*, Luís Caires (Ed.), Vol. 11423. Springer, 3–29.

Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *Tools and Algorithms for the Construction and Analysis of Systems*, Orna Grumberg and Michael Huth (Eds.). Springer, 568–582.

Herbert Schorr and William M. Waite. 1967. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM* 10 (1967), 501–506.

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *European Symposium on Programming (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 909–936.

Thomas Tuerk. 2010. Local Reasoning about While-Loops. VS-Theory Workshop, 3rd Int. Conf. on Verified Software: Theories, Tools and Experiments. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.226.4205.

Mattias Ulbrich. 2013. *Dynamic Logic for an Intermediate Language Verification, Interaction and Refinement*. Ph.D. Dissertation. Karlsruhe Institute of Technology. http://nbn-resolving.org/urn:nbn:de:swb:90-411691

Hongseok Yang. 2007. Relational Separation Logic. *Theoretical Computer Science* 375, 1 (2007), 308–334.