# Deductive Verification with Ghost Monitors

Martin Clochard, Claude Marché, Andrei Paskevich

# Deductive Verification with Ghost Monitors

Martin Clochard[1], Claude Marché[2,3], and Andrei Paskevich[3,2]

[1] ETH Zürich, Switzerland
[2] Inria, Université Paris-Saclay, F-91120 Palaiseau, France
[3] LRI, Université Paris-Sud & CNRS, F-91405 Orsay, France

**Abstract.** 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. 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.
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. We introduce one such framework, with an original extension that allows us to specify and prove fine-grained properties about infinite behaviors of target programs. The proof of correctness of our approach relies on a particular flavor of transfinite games. This proof is formalized and verified using the Why3 tool.

**Keywords:** deductive verification, unstructured programs, ghost code, games, Hoare logic, predicate transformers, infinite behaviors

## 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 optimisation techniques may alter the control flow making the original loop invariants inadequate. Finally, compilation into an unstructured language produces code which is difficult to verify directly.

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 the desired algorithmic structure, allowing us to place right invariants in 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 acts as 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 mean 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 can not 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 more clear 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 [10], which evaluates to 91 when $n \leq 100$, and to $n - 10$ otherwise. To prove that function `mcc91` satisfies this specification is a matter of milliseconds using any modern automated program verifier, backed up by an SMT solver. The tricky 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 quite harder: it requires discovering a more complex variant and a non-trivial loop invariant.

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 must break loops.

This 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):

$$\begin{aligned}
pc = 0 &\rightarrow pc' = 1 \wedge e' = 1 \wedge n' = n \\
pc = 1 \wedge n > 100 &\rightarrow pc' = 2 \wedge e' = e - 1 \wedge n' = n - 10 \\
pc = 1 \wedge n \leq 100 &\rightarrow pc' = 1 \wedge e' = e + 1 \wedge n' = n + 11 \\
pc = 2 \wedge e = 0 &\rightarrow pc' = 3 \wedge e' = e \wedge n' = n \\
pc = 2 \wedge e \neq 0 &\rightarrow pc' = 1 \wedge e' = e \wedge n' = n
\end{aligned}$$

We attach this specification (with an added precondition $0 \leq pc \leq 2$) to a procedure named `CONT`, and define the ghost monitor as shown in Fig. 1. In the monitor code, the variables that refer to the state of the target C program are written in italic. This program, modulo a few syntactic adjustments, is automatically proved by Why3.

The control flow of `mcc91_monitor` follows that of `mcc91`. However, the actual computations are performed as side effects of `CONT`, 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 `CONT`, we can conclude that the iterative C program does indeed compute the 91 function.

*Contributions and outline.* The main 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

```
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' = if n > 100 then n - 10 else 91 }  (* n' = mcc91 n *)
= if n > 100 then
    CONT();              (* no recursive calls, move to breakpoint 2 *)
  else begin
    CONT();              (* update n and e, move to breakpoint 1 *)
    mcc91_monitor();     (* inner mcc91 call, stop at breakpoint 2 *)
    CONT();              (* 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' = if n > 100 then n - 10 else 91 }  (* n' = mcc91 n *)
= CONT();                (* initialize e, move to breakpoint 1 *)
  mcc91_monitor();       (* compute mcc91 once, stop at breakpoint 2 *)
  CONT()                 (* exit the while loop *)
```

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

discovery of appropriate contracts and invariants, as shown in Section 2 on the example of Schorr-Waite algorithm [12]. 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 [6]), 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 `CONT` operation. This specification can be generated mechanically, for example, by means of symbolic execution of the target code between the chosen breakpoints. Note that this is a separate effort, not related to the implementation or verification of the monitor.

What is more, a ghost monitor does not even need to be executable. In this paper, we propose a language for ghost monitors that allows us to specify and prove properties of *infinite executions* of the target program, after a transfinite number of calls to `CONT`.

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 `CONT` operations, one for each target,

and make it follow the execution of both, in an appropriate cadence. In presence of non-determinism, the choices made by one target may be transferred to the other (as arguments given to `CONT`). Again, we can prove that verifying the monitor program ensures the corresponding properties of the targets. We show our method applied to the proof of relational properties between potentially non-terminating programs in Section 3.

The theoretical foundation of our approach uses the framework of games, as presented in Section 4. This framework is well suited for reasoning about infinite executions. It also lets us adopt a generalized approach with respect to the interpretation of non-determinism. 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 a given program or programs. We show in Section 4.2 that our approach can be used to formally prove at least three kinds of properties in presence of non-determinism: program correctness, existence of a specific behavior, and, finally, simulation between two programs.

Finally in Section 5, we provide a language and a weakest pre-condition calculus for ghost monitors. This language supports arbitrary recursion, continuations, and fine-grained specification and proof of non-terminating behaviors, both in the target and in the monitor.

Due to lack of space we do not give in this paper the proofs of our theorems. These are partly available in a research report [4] and in the first author's PhD thesis [3]. In order to provide stronger guarantees about our results, we mechanized the Hoare logic underlying our language for ghost monitors using the Why3 tool. The development is available online at `http://toccata.lri.fr/gallery/hoare_logic_and_games.en.html`

## 2 Extended Example: Schorr-Waite 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 graph traversal algorithm [12], which is a landmark example for evaluating proof methods dealing with pointer aliasing [2]. To our knowledge, Leino was the first to propose a proof using only automated theorem provers [8].

Let us consider the implementation of this algorithm written in a low-level pointer-manipulating C code, given in Fig. 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

```
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.

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* [2] to represent heap memory.

Proving directly that the low-level code meets this specification requires quite complex loop invariants [8]. With the ghost monitor approach, we write a new ghost code following the standard structure of a recursive depth-first traversal. As for the McCarthy function, we assume first a `CONT` procedure is defined, with the break points set as shown in the code. Our monitor is then written as in Fig. 4, annotated with the usual pre- and post-conditions expected for a classical depth-first traversal. This annotated code is proved with automated solvers (see http://toccata.lri.fr/gallery/schorr_waite_with_ghost_monitor.en.html). Unlike [8], we do not need complex loop invariants to make explicit the hidden notion of backtracking stack behind Schorr-Waite algorithm.

Among the existing proofs of Schorr-Waite algorithm, the closest to ours might be the one proposed by Yang [13], 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 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 post-conditions to the monitor, are considerably simpler than in [8].

```
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.

## 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:

```
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 ()
```

```
(* 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
    CONT (); (* push *)
    recursive_monitor new_gray; (* traverse the left child *)
    CONT (); (* swing *)
    recursive_monitor new_gray; (* traverse the right child *)
    CONT ()  (* pop *)
  in
  CONT (); (* initialize *)
  recursive_monitor ∅;
  CONT ()  (* exit *)
```

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

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 `CONT` procedure for each program. Instead of representing the input streams explicitly, as two global variables in the monitor, from which the `CONT` procedures would take the next character returned by `getchar`, we opt for a non-deterministic semantics of `getchar`, in order to

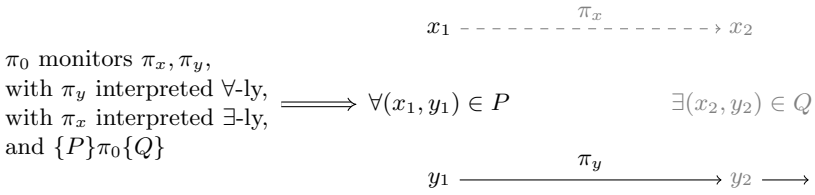


**Fig. 5.** Transfer property yielding a simulation.

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 `CONT` perform non-deterministic choices whenever the target program does. In order to ensure that `(C)` and `(M)` read the same input values, the choices of one program must be transferred to the other. We achieve this by adding arguments to one of the `CONT` operations, giving to the ghost monitor 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. Formally, the transfer property, illustrated in Fig. 5, states that for every

- ghost monitor $\pi_0$ for target programs $\pi_x$ and $\pi_y$, where non-determinism is interpreted as demonic (universal) for $\pi_y$ and as angelic (existential) for $\pi_x$,
- total correctness contract $\{P\}\pi_0\{Q\}$ (valid Hoare triple),
- pair of states $x_1, y_1$ for $\pi_x$ and $\pi_y$ that satisfy $P$,
- maximal (potentially infinite) execution trace of $\pi_y$ starting from $y_1$,

there exists a pair of states $(x_2, y_2)$ satisfying $Q$ together with a partial execution trace of $\pi_x$ from $x_1$ to $x_2$ such that $y_2$ belongs to the chosen execution trace of $\pi_y$.

When $P$ (resp. $Q$) expresses that both states are initial (resp. final) and equivalent to each other, this simulation result means an inclusion of behaviors. By proving inclusion 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 Fig. 6. As these breakpoints break all execution cycles, we can obtain proper specifications for $\texttt{CONT}_C$ and $\texttt{CONT}_M$, shown in Fig. 7.

```
/* 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.

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

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

**Fig. 7.** Specifications of $\texttt{CONT}_C$ and $\texttt{CONT}_M$.

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 `CONT` procedure receives an argument *toRead* and returns a return value *read* which represents the character just 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 ghost monitor to control the non-deterministic choices.

Equipped with these specifications, 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 Fig. 8. The core of the monitor is the recursive function `sync`, which relates:

```
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 CONT *)
  CONT_C (ign); CONT_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 CONT_M (CONT_C (ign)) else CONT_C (CONT_M (ign));
    if pc_M = 1 then begin sync (); CONT_M (ign); sync () end
  in
  sync (); CONT_C (ign); CONT_M (ign)
```

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

- on one hand, iterations of program `(C)` until the counter has decreased;
- on the other hand, one full recursive call of `scan` in program `(M)`.

While this monitor is automatically proved using Why3 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 simplest way to recover total correctness for the ghost monitor is to assume that the program with demonic non-determinism terminates, for example using a variable representing execution fuel. While this would easily lead to a terminating monitor, we would not obtain the right equivalence 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 only finitely many inputs, while `(M)` cannot.

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 `CONT` an infinite number of times *and then* proceed to an exit point, covered by the monitor's post-condition. In terms of the transfer property in Fig. 5, this means that we transfer an execution trace running up to a limit program state located after an infinite sequence of steps.

To be able to proceed with that plan, we first need a way to actually represent that 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 matches perfectly 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. However, this in general has no meaning for regular variables. In order to treat these values properly, we augment the state with time counters $t_C$ and $t_M$ which increase at each execution step. Using a proper ordering (see Section 4.2) on states augmented with counters, we can discard the meaningless values in infinite states. By convention, we also set $pc_C$ and $pc_M$ to $\infty$ after an infinite execution.

Let us now focus on the verification mechanism. First, we replace decreasing termination measures with increasing progression measures. Instead of representing a termination argument, progression measures serve as constraints over potential divergence behavior. For our running example, we need to enforce that time counters increase between recursive calls to make sure that whenever `sync` diverges, so does the target program. Similarly, we need to enforce that input log grows to rule out behaviors in which a program stops performing input. Note that we do not impose any condition on the progression ordering itself.

Second, we equip recursive definitions with divergence handlers. Divergence handlers are given by an extra `diverges` clause after a recursive definition. This clause introduces a statement intended to handle any form of divergence issuing from such 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 follow the progression clause, 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 post-condition of this particular call. 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

```
let monitor ()
  requires { pc_C = 0 ∧ pc_M = 0 ∧ in_M = in_C }
  requires { angel_C ≠ angel_M }
  ensures  { (pc'_C = 2 ∧ pc'_M = 3) ∨ (pc'_C = ∞ ∧ pc'_M = ∞) }
  ensures  { in'_M = in'_C }
= let ign = '(' in
  CONT_C (ign); CONT_M (ign);
  let rec sync () : unit
    requires { pc_C = 1 ∧ pc_M = 1 ∧ in_M = in_C ∧ n_C ≥ 0 }
    progress { (t_C) × (t_M) × (|in_C|) } (* t_C, t_M, |in_C| all increase *)
    ensures  { (pc'_C = 1 ∧ pc'_M = 2) ∨ (pc'_C = ∞ ∧ pc'_M = ∞) }
    ensures  { pc'_C ≠ ∞ → stack'_M = stack_M ∧ n'_C = n_C − 1 }
    ensures  { in'_M = in'_C }
    = if angel_M then CONT_M (CONT_C (ign)) else CONT_C (CONT_M (ign));
      if pc_M = 1 then begin
        sync ();
        if pc_M ≠ ∞ then begin CONT_M (ign); sync () end
      end
    diverges (unbounded_stack) -> choose_any(unbounded_stack)
  in
  sync (); if pc_C ≠ ∞ then begin CONT_C (ign); CONT_M (ign) end
```

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

executing even after divergence has been caught (potentially infinitely many times through local recursive functions).

In practice, we should also change iterative statements, but the changes would be similar, since loops are essentially calls to anonymous tail-recursive functions. We can now use this framework to turn our earlier ghost monitor into a ghost monitor that proves the equivalence of both programs, taking into account non-terminating behavior. We show the monitor adapted to divergence handling in Fig. 9. We add the progression clause proposed earlier, as well as a divergence handler which merely returns at any level. To account for the extra behaviors, we also add a few tests after calls to `sync` to treat infinite states appropriately. While we gave a short body for the divergence handler, in practice we need a few assertions (manual proof steps) to recover all the desired post-conditions in a fully automated manner.

So far, we did not explain how the call stack leading to divergence was specified in the divergence handler. There are several choices, and not all of them are equivalent from the perspective of mechanized proofs, nor are they equally powerful. In Section 5, 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 remove alternative post-condition corresponding to divergence from recursive calls. Third, the fact that this alternative post-condition is only linked to the bottom of the recursion stack makes it easier to link this post-condition to the initial parameters and states. 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. Therefore we focus on the more general construction below.

## 4 Unifying Transfer Properties through Games

We base our verification approach on the formal model of *games*, which can be thought as the natural generalization of transition systems in the 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 angel. In particular, this turns ghost monitors into tools to prove the existence of such winning strategy. Indeed, in this framework the transfer property now map valid contracts for ghost monitors to winning strategies in the underlying game.

The purpose of the game framework is to unify and generalize the different kinds of transfer properties we have seen so far. Indeed, total correctness amounts to the existence of a strategy in a game whose state is the program state, and where the demon performs non-deterministic choices. Similarly, simulation properties can be expressed as existence of winning strategies in a game corresponding to the execution of two programs, where the demon makes choices for one program and the angel makes choices for the other. We can also express other properties, e.g., by having the angel making all choices for a single program. Then the existence of a winning strategy reduces to the existence of a program behavior from an initial state to a final state satisfying some properties.

In this section, we first present our formal model of games, before demonstrating the embedding of several transfer properties over transition systems as existence of winning strategies in the corresponding games. Note that transition systems abstract small-step operational semantics of programs, which we require for the ghost monitor approach.

### 4.1 Games

As we solely focus on tools to prove the existence of such winning strategies for the angel, we define games asymmetrically with respect to both players. The domain of the game corresponds to angelic states, while demonic states are implicitly defined as sets of angelic states. 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. Here, the outer set represents the set of angelic choices, while the inner set represents the set of demonic choices. These definitions match the behavior of the `CONT` routines of Section 3. Angelic choice selects the arguments given to that routine, which from the verification perspective amounts to select a post-condition (a set of angelic states), while demonic choice selects the effective updates performed by that routine as it wishes as long as it respects the behavior specified in the post-condition (non-deterministic choice of a behavior in the set).

In order to handle the case of non-terminating programs with observable effects, we equip our games with an order, so that the state only grows during a play of the game. We then represent the overall infinite execution of a program by the least upper bound of a play, which is typically the sequence of all observable effects. We impose that the order is chain-complete so that the requested least upper bound always exists.

**Definition 1 (Chain-Complete Order, Game).** *A partially ordered set $(O, \leq)$ is* chain-complete *if for all non-empty $X \subseteq O$ totally ordered by $\leq$, $X$ admits a least upper bound. A* game *$\mathbb{G} = (G, \preccurlyeq, \Delta)$ is a chain-complete partially ordered set $(G, \preccurlyeq)$ equipped with a function $\Delta$ 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 $\Delta$, respectively, the* domain*, the* order*, and the* transitions *of the game.*

Note that the current state of a play may contain not enough 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. Due to our treatment of infinite behaviors, we have not been able to prove that such memory could be systematically eliminated (nor to prove that it is strictly necessary). Thus we introduce the notion of *history* to represent the complete memory of a play.

**Definition 2 (History, Prefix Order).** *Given a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$, a* history *of $\mathbb{G}$ is a non-empty subset of $G$ for which the restriction of $\preccurlyeq$ is a total order, and which admits a maximum for that order. Given two histories $H_1, H_2$ for a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$, 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 $H_1$.*

We then introduce the notion of *victory invariant*, a notion closely related to the notion of winning strategy. A victory invariant is essentially a set of histories

that the angel has the ability to keep following from a source state until a state from the winning set is reached. This is in fact equivalent to the notion of winning strategy with non-deterministic choices.

**Definition 3 (Victory Invariant).** *Given a game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ *and two subsets* $P, Q$ *of* $G$*, a* victory invariant *of* $\mathbb{G}$ *for* $P, Q$ *is a set* $I$ *of histories of* $\mathbb{G}$ *such that*

*(i) for all* $x \in P$*,* $\{x\} \in I$*,*
*(ii) for all* $H \in I$ *such that* $H \cap Q = \emptyset$*, there exists* $X \in \Delta(\max H)$ *such that for all* $x \in X$*,* $H \cup \{x\} \in I$*,*
*(iii) for every non-empty* $\mathcal{H} \subseteq I$ *that is totally ordered by the prefix order,* $\bigcup \mathcal{H} \cup \{\sup \mathcal{H}\} \in I$*.*

The three propagation rules of victory invariants tell us exactly how to build a winning strategy, except for some choices in rule (ii). Rule (i) means that such a strategy can be built when starting from any element of $P$. Rule (ii) means that when the current history is $H$, either the angel has already won the game or a (not necessarily unique) winning decision can be made at that point. Finally, rule (iii) means that if a history grows indefinitely, it continues at the least upper bound. Note that building a play while respecting the invariant guarantees that after some ordinal number of iterations, the winning set $Q$ will be attained. This is implied by the fact that the history strictly increases until $Q$ is reached.

We prefer using victory invariants rather than strategies for two reasons. First, leftover choices in condition (ii) make victory invariants more permissive and more practical to perform proofs. Second, victory invariants are quite similar in nature to loop invariants, which makes them more directly related to program logics than winning strategies.

### 4.2 Linking Games and Transition Systems

We show how to connect transition systems and games, in order to recover more intuitive transfer properties talking about the operational semantics of the target programs. We consider several translations from transition systems to games. The details of the translation differ depending on the actual transfer property. We show here how to translate three different properties as existence of victory invariants: program correctness, existence of behaviors, and simulation.

Note that bare transition systems do not come with any way to describe infinite behaviors except for the complete trace, which may be exceedingly precise. We equip transition systems with observation records to model abstractly the trace of observable effects of the program, which is the natural tool to describe the behavior of a non-terminating program. We then use the least upper bound of observation records along an execution to represent the "state" at infinity.

**Definition 4 (Transition System with Observation Records).** *A* transition system with observation records *is a tuple* $\mathcal{S} = (S, \rightarrow, p, O \preccurlyeq)$ *where*

– $(S, \rightarrow)$ *is a transition system,*

– $(O, \preccurlyeq)$ *is a chain-complete order, representing the observation records,*
– $p$ *is a monotonic map from* $(S, \to^*)$ *to* $(O, \preccurlyeq)$ *that extracts records.*

We now define the two translations corresponding respectively to the angelic (existential) and demonic (universal) interpretation of non-determinism. These two translations share the same support. To represent finite states, we pair the transition system state with a counter to represent effective transition. For infinite states, we use an observation record.

**Definition 5 (Timed support).** *The* timed support *associated to a transition system with observation records* $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$ *is the ordered set* $(G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}})$ *defined by*

– $G_{\mathcal{S}} = \{(n, x) \in (\mathbb{N} \cup \{\infty\}) \times (S \cup O) \mid (n = \infty \wedge x \in O) \vee (n \neq \infty \wedge x \in S)\}$,
– $\forall n \in \mathbb{N}, x \in S, y \in O.\ (n, x) \prec_{\mathcal{S}} (\infty, y) \Leftrightarrow p(x) \preccurlyeq y$,
– $\forall n, m \in \mathbb{N}, x, y \in S.\ (n, x) \prec_{\mathcal{S}} (m, y) \Leftrightarrow n < m \wedge p(x) \prec p(y)$,
– $\forall x, y \in O.\ (\infty, x) \prec_{\mathcal{S}} (\infty, y) \Leftrightarrow x \prec y$.

**Definition 6 (Existential game).** *The* existential game *associated to a transition system with observation records* $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$ *is defined as the game* $\mathcal{G}_{\mathcal{S},\exists} = (G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}}, \Delta_{\mathcal{S},\exists})$ *with transitions defined by*

$$\Delta_{\mathcal{S},\exists}((n, x)) = \begin{cases} \{\{(n+1, y)\} \mid x \to y\} & \text{when } n \in \mathbb{N}, \\ \emptyset & \text{when } n = \infty. \end{cases}$$

**Definition 7 (Universal game).** *The* universal game *associated to a transition system with observation records* $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$ *is defined as the game* $\mathcal{G}_{\mathcal{S},\forall} = (G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}}, \Delta_{\mathcal{S},\forall})$ *with transitions defined by*

$$\Delta_{\mathcal{S},\forall}((n, x)) = \begin{cases} \{\{(n+1, y) \mid x \to y\}\} - \{\emptyset\} & \text{when } n \in \mathbb{N}, \\ \emptyset & \text{when } n = \infty. \end{cases}$$

These definitions are valid only because the timed support is chain-complete, which is a routine consequence of chain-completeness of observations. We remove $\emptyset$ from the universal transition because when it occurs, it corresponds to a stuck state. In particular, the angel should not be able to progress on such states. Moreover, this enforces that universal and existential games coincide for deterministic transition systems, i.e., the interpretation of non-determinism does not matter in this case.

We prove several results linking the existence of victory invariants in these games to properties about the original transition system. For existential games, this represents the existence of a behavior. For universal games, this means that a certain set of states is unavoidable (possibly after infinitely many steps). If there are no infinite states in the target set, this corresponds to total correctness.

**Theorem 1.** *For all* $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$*, for all* $P, Q \subseteq S, Q_\infty \subseteq O$*, there is a victory invariant relative to* $\mathbb{G}_{\mathcal{S},\exists}$ *for* $\mathbb{N} \times P, \mathbb{N} \times Q \cup \{\infty\} \cup Q_\infty$ *if and only if for all* $s_0 \in P$*, one of the following two conditions is true:*

*(i)* *there exists a finite transition sequence* $s_0 \to \ldots \to s_n$ *of* $\mathcal{S}$ *such that* $s_n \in Q$,
*(ii)* *there exists an infinite sequence* $s_0 \to \ldots \to s_n \to \ldots$ *of* $\mathcal{S}$ *such that the least upper bound of observation records over the sequence belongs to* $Q_\infty$.

*Proof.* See [4] (Lemma 4.5, pages 25-26).

**Theorem 2.** *For all* $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$, *for all* $P, Q \subseteq S, Q_\infty \subseteq O$, *there is a victory invariant relative to* $\mathbb{G}_{\mathcal{S},\forall}$ *for* $\mathcal{N} \times P, \mathbb{N} \times Q \cup \{\infty\} \times Q_\infty$ *if and only if for all potentially infinite transition sequences* $s_0 \to \ldots \to s_n(\to \ldots)^?$ *of* $\mathcal{S}$ *such that* $s_0 \in P$, *one of the following conditions is true:*

*(i)* *there exists an element* $s_i$ *in the sequence such that* $s_i \in Q$,
*(ii)* *the sequence is infinite, and the least upper bound of observation records along the sequence is in* $Q_\infty$,
*(iii)* *the sequence is finite with last element* $s_n$, *and there exists* $x \in S$ *such that* $s_n \to x$.

*Proof.* See [4] (Lemma 4.6, page 26).

Using a product construction, we propose a third translation for simulation. Note that the final result allows arbitrary connections between finite and infinite behaviors of respective transition systems.

**Definition 8.** *Given two transition systems with observation records* $\mathcal{S}_1, \mathcal{S}_2$, *the simulation game from* $\mathcal{S}_1$ *to* $\mathcal{S}_2$ *is the game*

$$\mathbb{G}_{\mathcal{S}_1 \to \mathcal{S}_2} = (G_{\mathcal{S}_1} \times G_{\mathcal{S}_2}, \preccurlyeq_{\mathcal{S}_1} \times \preccurlyeq_{\mathcal{S}_2}, \Delta_{\mathcal{S}_1 \to \mathcal{S}_2})$$

*with transitions defined by:*

$$\Delta_{\mathcal{S}_1 \to \mathcal{S}_2}((x, y)) = \{\{x\} \times Y \mid Y \in \Delta_{\mathcal{S}_2,\exists}(y)\} \cup \{X \times \{y\} \mid X \in \Delta_{\mathcal{S}_1,\forall}(x)\}.$$

**Theorem 3.** *For all transition systems* $\mathcal{S}_1 = (S_1, \to_1, p_1, O_1, \preccurlyeq_1)$ *and* $\mathcal{S}_2 = (S_2, \to_2, p_2, O_2, \preccurlyeq_2)$, *for all set* $P \subseteq S_1 \times S_2$, *set family* $(Q_{a,b})_{a,b \in \{true,false\}}$ *such that* $Q_{a,b} \subseteq (a \ ? \ S_1 : O_1) \times (b \ ? \ S_2 : O_2)$, *there exists a victory invariant for the set pair made of the pre-condition* $\mathbb{N} \times P$ *and of the post-condition*

$$\{((n, x), (m, y)) \in G_{\mathcal{S}_1} \times G_{\mathcal{S}_2} \mid (x, y) \in Q_{n=\infty, m=\infty}\}$$

*if and only if for all* $t_0 \in S_2$, *for all* $s_0 \to \ldots \to s_n(\to \ldots)^?$ *maximal transition sequence of* $\mathcal{S}_1$ *such that* $(s_0, t_0) \in P$, *there exists* $t_0 \to \ldots \to t_n(\to \ldots)^?$ *a transition sequence of* $\mathcal{S}_2$ *such that:*

*(i)* *the transition sequence from* $\mathcal{S}_2$ *ends at* $t_n$, *and there exists* $i$ *such that* $(s_i, t_i) \in Q_{false,false}$,
*(ii)* *the transition sequence from* $\mathcal{S}_2$ *ends at* $t_n$, *the transition sequence from* $\mathcal{S}_1$ *is infinite with least upper bound of observation* $s_\infty$, *and* $(s_\infty, t_n) \in Q_{true,false}$,
*(iii)* *the transition sequence from* $\mathcal{S}_2$ *is infinite with least upper bound of observation* $t_\infty$, *and there exists* $i$ *such that* $(s_i, t_\infty) \in Q_{false,true}$,

$$
\begin{array}{lll}
pgm & ::= & fun(expr) \\
& | & \mathsf{let}\ var = pgm\ \mathsf{in}\ pgm \\
& | & \mathsf{switch}\ case^*\ \mathsf{end\text{-}switch} \\
& | & \mathsf{kont}\ fun\ \mathsf{in}\ pgm\ \mathsf{end\text{-}kont} \\
& | & \mathsf{rec}\ fun(var : typ) : contract\ \mathsf{progress}(expr) \\
& & = pgm\ \mathsf{diverges}(var) : pgm \\
& & \mathsf{in}\ pgm \\
case & ::= & \mathsf{case}\ var : typ.\ formula\ \mathsf{in}\ pgm \\
contract & ::= & \langle formula \hookrightarrow var : typ.\ formula \rangle
\end{array}
$$

**Fig. 10.** Syntax of $\mathcal{W}_G$.

*(iv) the transition sequences are both infinite with respective least upper bound of observations $s_\infty, t_\infty$, and $(s_\infty, t_\infty) \in Q_{true,true}$.*

*Proof.* See [4] (Lemma 4.8, page 27).

Note that Theorem 3 admits a straightforward generalization to arbitrary number of transition systems in universal and existential positions. In particular, for a single transition system, this degenerates to Theorems 2 and 1. We also obtain other equivalences: for two universal transition systems, existence of victory invariants translates to deterministic correlation of behaviors.

## 5 Ghost Monitors for Games

We now define the language $\mathcal{W}_G$ that we use to establish victory invariant on games $\mathbb{G} = (G, \preccurlyeq, \Delta)$. This is a mostly functional programming language, where functions are first-order and equipped with contracts. The only non-functional feature is a global variable $\texttt{now} \in G$, which represents the current state of the game. We only permit update of this variable through the use of predefined procedure `CONT` which performs one transition of the game. Procedure `CONT` takes as argument an element $X$ of $\Delta(\texttt{now})$ and updates the global variable `now` to a non-deterministically chosen element of $X$.

### 5.1 Syntax of $\mathcal{W}_G$

The syntax is given in Fig. 10. We deliberately keep the language as small as possible. We exploit the fact that most usual programming constructions (like loops, conditional, assertions and others) with their usual verification rules can be derived as sugar to focus on the core constructions. For example, the missing meta-logic expression programs $e$ can be encoded via continuation and call, as $\mathsf{kont}\ k\ \mathsf{in}\ k(e)\ \mathsf{end\text{-}kont}$. The continuation mechanism can be used to smoothly simulate constructions like `break`, `continue` or `return`.

In the language definitions, *expr* and $formula$ refer respectively to arbitrary mathematical expressions and formulas in some meta-logic language, whose exact

$$\frac{\Gamma, \texttt{now} : G \vdash e : A}{\Sigma, f : A \to B | \Gamma \vdash \texttt{f}(e) : B} \qquad \frac{\Sigma|\Gamma \vdash \pi_0 : A \qquad \Sigma|\Gamma, x : A \vdash \pi_1 : B}{\vdash \mathsf{let}\ x = \pi_0\ \mathsf{in}\ \pi_1 : B}$$

$$\frac{\Sigma|\Gamma, x : A \vdash \pi : B \qquad \Gamma, x : A, \texttt{now} : G \vdash \varphi : \{\top, \bot\}}{\Sigma|\Gamma \vdash \mathsf{case}\ x : A.\ \varphi\ \mathsf{in}\ \pi : B}$$

$$\frac{\Sigma, k : B \to \emptyset | \Gamma \vdash \pi : B}{\Sigma|\Gamma \vdash \mathsf{kont}\ k\ \mathsf{in}\ \pi\ \mathsf{end\text{-}kont} : B} \qquad \frac{\forall i \in [\![1; n]\!]\ \Sigma|\Gamma \vdash c_i : B}{\Sigma|\Gamma \vdash \mathsf{switch}\ c_1 \ldots c_n\ \mathsf{end\text{-}switch} : B}$$

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

**Fig. 11.** Typing rules for $\mathcal{W}_G$.

nature is irrelevant here. These expressions can access any immutable variable, as well as the global variable `now`. Formulas in contract post-conditions can also access the special variable `old`, which refers to the state of the global variable `now` before the execution of a function.

Syntactic categories *var*, *fun* and *typ* correspond respectively to immutable variable names, function names, and variable types. For $\mathcal{W}_G$, variable types are given by statically known (non-dependent) sets. As functions of $\mathcal{W}_G$ are necessarily first-order, function types are simply given by $A \to B$, where $A$ is the type of the parameter and $B$ is the type of the return value. We define the typing system for language $\mathcal{W}_G$ by a standard relation $\Sigma|\Gamma \vdash \pi : B$ defined in Fig. 11, where $\Gamma$ (resp. $\Sigma$) is the context of variables (resp. functions) with their types, and $B$ is the derived type for the program. This typing relation is subject to a similarly-structured typing relation for the meta-logic that we do not present here.

There is one unusual construction in our verification language: the recursion, which introduces a recursively-defined function equipped with a handler to process the result of diverging behavior. The progress clause gives the progression order $r$, with respect to which the game state and function parameters should increase at recursive call. Recursion behaves in a special way in case of infinite execution, as presented in Section 3. If the recursion stack grows indefinitely up to a limit state, the divergence handler is called with that recursion stack, which is the totally ordered set of call parameter/state pairs of uncompleted recursive calls. Then, control flow returns at the end of the recursive call indicated by the result of the handler, with the return value given by the handler. Note that the divergence handler itself is allowed to make recursive calls as long as the pa-

rameter/state pair increases further, which may lead to the divergence handler calling itself recursively with an even larger recursion stack.

### 5.2 Predicate Transformer Semantics for $\mathcal{W}_G$

We give a predicate transformer semantics for $\mathcal{W}_G$, using weakest pre-condition transformers in the style of Dijkstra [5]. Intuitively, the transformer of a program $\pi$ for a winning set $Q$ gives the largest set $P$ such that $\pi$ proves the existence of a victory invariant for $P, Q$.

Due to the presence of continuations as well as defined objects, we parameterize the weakest pre-condition transformer by contextual elements. We add an assignation parameter, which gives the values of variables, and a specification context parameter, which gives the specification of functions.

**Definition 9 (Assignation, Specification Context).** *An $\Gamma$-*assignation *for $\mathcal{W}_G$ is a mapping $\sigma$ from variables names $x : A$ from $\Gamma$ to values in the set $A$. A $\Sigma$-*specification context *for $\mathcal{W}_G$ is a mapping $\Phi$ from function names $\mathtt{f} : A \to B$ from $\Sigma$ to pairs $P \subseteq A \times G, Q \subseteq A \times G \times B \times G$. Intuitively, the pair $P, Q = \Phi(\mathtt{f})$ represents the* contract *for procedure $\mathtt{f}$ within specification context $\Phi$.*

We define the semantics of $\mathcal{W}_G$ as a mapping $\mathcal{T}[\Phi, \sigma]\,(\pi, Q)$, which takes as argument a $\Sigma$-specification context $\Phi$, an $\Gamma$-assignation $\sigma$, a well-typed program $\pi$ with type derivation $\Sigma | \Gamma \vdash \pi : B$, and a set $Q \subseteq B \times G$ (the post-condition); and returns a subset of $G$ (the weakest pre-condition). The expressions and formulas within the program are interpreted by the mean of a standard interpretation function $[\![\cdots]\!]_{\cdots}$, which given an expression or a formula and an assignation of its variables, computes the value of the expression as an element of its type, or the satisfaction of the formula as a truth value.

We also define a predicate $\mathcal{C}[(\Phi_x)_{x \in G}, \sigma]\,(\pi : \langle \varphi_0 \hookrightarrow y : B.\ \varphi_1 \rangle)$ corresponding to the notion of contract satisfaction, which we use for recursion. We give the full definition in Fig. 12.

Technically, the weakest predicate semantics should be defined from the complete type derivation (or a program tagged with all types) in order to get type information. For simplicity, we only write $\pi$ as parameter and assume that the typing context is known.

The definition of the backward predicate transformer semantics for $\mathcal{W}_G$ is mostly the standard definition of weakest pre-conditions, except for two cases: continuation and recursion. We define the predicate transformer for continuation introduction by adapting in a natural way the typing scheme of the `call-with-current-continuation` operator. For introduction of recursive functions, we perform the following modifications to the traditional transformer for total correctness:

– We replace the traditional well-founded decreasing termination measure by a progression order, which must increase along recursive calls, including the ones in the divergence handler.

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

$$\mathcal{T}[\Phi,\sigma]\,(\mathsf{let}\ x = \pi_0\ \mathsf{in}\ \pi_1, Q) \triangleq$$
$$\mathcal{T}[\Phi,\sigma]\,(\pi_0, \{(a,y) \in A \times G \mid y \in \mathcal{T}[\Phi,\sigma[x \leftarrow a]]\,(\pi_1,Q)\})$$
$$(\text{with } x : A)$$

$$\mathcal{T}[\Phi,\sigma]\,(\mathsf{kont}\ \mathtt{f}\ \mathsf{in}\ \pi\ \mathsf{end\text{-}kont}, Q) \triangleq \mathcal{T}[\Phi[\mathtt{f} \leftarrow (Q,\emptyset)],\sigma]\,(\pi,Q)$$

$$\mathcal{T}[\Phi,\sigma]\,\big(\mathsf{case}\ x : A.\ \varphi\ \mathsf{in}\ \pi, Q\big) \triangleq$$
$$\bigcap_{a\in A} \{x \in G \mid [\![\varphi]\!]_{\sigma[x\leftarrow a,\mathtt{now}\leftarrow x]} \Rightarrow x \in \mathcal{T}[\Phi,\sigma[x \leftarrow a]]\,(\pi,Q)\}$$

$$Grd[\sigma](\mathsf{case}\ x : A.\ \varphi\ \mathsf{in}\ \pi) \triangleq \bigcup_{a\in A} \{x \in G \mid [\![\varphi]\!]_{\sigma[x\leftarrow a,\mathtt{now}\leftarrow x]}\}$$

$$\mathcal{T}[\Phi,\sigma]\,(\mathsf{switch}\ c_1 \ldots c_n\ \mathsf{end\text{-}switch}, Q) \triangleq \Big(\bigcup_{i\in[\![1;n]\!]} Grd[\sigma](c_i)\Big) \cap \bigcap_{i\in[\![1;n]\!]} \mathcal{T}[\Phi,\sigma]\,(c_i,Q)$$

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

$$z \in \mathcal{T}[\Phi,\sigma]\left(\begin{pmatrix}\mathsf{rec}\ \mathtt{f}(x) : \langle\varphi_0 \hookrightarrow y : B.\ \varphi_1\rangle\ \ \mathsf{progress}(r)\\ = \pi_0\ \mathsf{diverges}(S) : \pi_1\\ \mathsf{in}\ \pi_2\end{pmatrix}, Q\right)$$

if and only if the following conditions (i)-(iv) hold:

(i) $z \in \mathcal{T}[\Phi^{rec}_{\emptyset},\sigma]\,(\pi_2,Q)$

(ii) $[\![r]\!]_\sigma$ is a strict order

(iii) $\forall a \in A.\ \mathcal{C}[(\Phi^{rec}_{\{a,x\}})_{x\in G},\sigma[x \leftarrow a]]\,(\pi_0 : \langle\varphi_0 \hookrightarrow y : B.\ \varphi_1\rangle)$

(iv) for all $H \in \mathcal{P}(A \times G)$ such that:

(a) $H$ is totally ordered by relation $[\![r]\!]_\sigma$

(b) $H$ is inhabited and does not have a maximum

(c) $\forall (a,x) \in H.\ [\![\varphi_0]\!]_{\sigma[x\leftarrow a,\mathtt{now}\leftarrow x]}$

then $\sup_{(a,x)\in H} x \in \mathcal{T}[\Phi^{rec}_H,\sigma[S \leftarrow H]]\,\big(\pi_1, Q^{lim}_H\big)$

where $\Phi^{rec}_H = \Phi[\mathtt{f} \leftarrow (P^{rec}_H, Q^{cont})]$

and $P^{rec}_H = \{(a,x) \in A \times G \mid [\![\varphi_0]\!]_{\sigma[x\leftarrow a,\mathtt{now}\leftarrow x]} \wedge \forall (a_0,x_0) \in H.\ (a_0,x_0)[\![r]\!]_\sigma [\![e]\!]_{\sigma[x\leftarrow a,\mathtt{now}\leftarrow x]}\}$

and $Q^{cont} = \{(a,x,b,y) \in A \times G \times B \times G \mid [\![\varphi_1]\!]_{\sigma[x\leftarrow a,y\leftarrow b,\mathtt{now}\leftarrow y,\mathtt{old}\leftarrow x]}\}$

and $Q^{lim}_H = \{((a,x,b),y) \in (A \times G \times B) \times G \mid (a,x) \in H \wedge [\![\varphi_1]\!]_{\sigma[x\leftarrow A,y\leftarrow b,\mathtt{now}\leftarrow y,\mathtt{old}\leftarrow x]}\}$

**Fig. 12.** Predicate transformer semantics for $\mathcal{W}_G$.

– We add condition (iv) for the divergence handling code $\pi_1$. This condition states that $\pi_1$ terminates correctly given any limit recursion stack controlled by the pre-condition and the progression relation of the program.

### 5.3 From Weakest Pre-Conditions to Victory Invariants

We now state the main correctness result for our verification language, as a transfer property from language $\mathcal{W}_G$ to games. Essentially, we claim that weakest pre-conditions for $\mathcal{W}_G$ induce the existence of victory invariants.

**Definition 10.** *We define the* transition contract $(P_{\mathbb{G}}, Q_{\mathbb{G}})$ *associated to game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ *as*

$$P_{\mathbb{G}} = \{(X, x) \in \mathcal{P}(G) \times G \mid X \in \Delta(x)\}$$
$$Q_{\mathbb{G}} = \{(X, x, 0, y) \in \mathcal{P}(G) \times G \times \{0\} \times G \mid y \in X\}$$

*We define the* base specification context $\Phi_{\mathbb{G}}$ *associated to game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ *as the single-element specification context mapping* `CONT` *of type* $\mathcal{P}(G) \to \{0\}$ *to the transition contract* $(P_{\mathbb{G}}, Q_{\mathbb{G}})$ *of* $\mathbb{G}$.

**Theorem 4.** *For all games* $\mathbb{G} = (G, \preccurlyeq, \Delta)$*, all well-typed programs* $\texttt{CONT} : \mathcal{P}(G) \to \{0\} | \Gamma \vdash \pi : \{0\}$*, all subsets* $Q$ *of* $G$*, and all* $\Gamma$*-assignations* $\sigma$*, there is a victory invariant for* $\mathcal{T}[\Phi_{\mathbb{G}}, \sigma](\pi, \{0\} \times Q), Q$.

We derive from Theorem 4 an immediate corollary, which concludes the existence of a victory invariant for a pair of sets definable in the meta-logic as long as we can find a program satisfying the corresponding contract.

**Corollary 1.** *For all games* $\mathbb{G} = (G, \preccurlyeq, \Delta)$*, all well-typed programs* $\texttt{CONT} : \mathcal{P}(G) \to \{0\} | \Gamma \vdash \pi : \{0\}$ *of* $\mathcal{W}_G$*, all formula pairs* $\varphi_0, \varphi_1$ *well-typed in context* $\Gamma, \texttt{now} : G$*, and all* $\Gamma$*-assignations* $\sigma$*, if the contract satisfaction property* $\mathcal{C}[(\Phi_{\mathbb{G}})_{x \in G}, \sigma](\pi : \langle \varphi_0 \hookrightarrow u : \{0\}.\ \varphi_1 \rangle)$ *holds, then there is a victory invariant for the set pair* $\{x \in G \mid [\![\varphi_0]\!]_{\sigma[\texttt{now} \leftarrow x]}\}, \{x \in G \mid [\![\varphi_1]\!]_{\sigma[\texttt{now} \leftarrow x]}\}$.

In order to prove Theorem 4, we need to state a stronger version that generalizes it over both specification contexts and output types.

**Definition 11.** *A contract* $P \subseteq A \times G, Q \subseteq A \times G \times B \times G$ *is said to be* valid *with respect to game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ *if for all* $(a, x) \in A \times G$*, there exists a victory invariant for the set pair* $\{x\}, \{y \in G \mid \exists b \in B.\ (a, x, b, y) \in Q\}$*. By extension, a* $\Sigma$*-specification context* $\Phi$ *is* valid *with respect to* $\mathbb{G}$ *if all its contracts are valid with respect to* $\mathbb{G}$.

**Theorem 5.** *For all games* $\mathbb{G}$*, all well-typed programs* $\Sigma | \Gamma \vdash \pi : B$ *of* $\mathcal{W}_G$*, all* $\Sigma$*-specification contexts* $\Phi$ *valid with respect to* $\mathbb{G}$*, all subsets* $Q$ *of* $B \times G$*, and all assignations* $\sigma$*, there is a victory invariant for the pair* $\mathcal{T}[\Phi, \sigma](\pi, Q), \{y \in G \mid \exists b \in B.\ (b, y) \in Q\}$.

The proof is detailed in [4] (Theorem 3.13, page 18). Theorem 4 is an immediate consequence of Theorem 5. Let us mention an interesting artifact of the proof: in order to prove correction of the continuation/recursive construction, the proof proceeds through games with extra transitions representing the additional function calls. In the case of recursion, this produces games with arbitrary interleaving of angelic and demonic non-determinism. In particular, these games cannot keep the special structure inherited from transition systems of Section 4.2, which justifies *a posteriori* the generic game framework.

### 5.4 Relative Completeness

For verification, we only need our proof methodology to be sound. However, it is also relatively complete, in the sense that if we can prove directly the existence of a victory invariant for a definable set pair in the meta-logic, then we can also prove its existence by finding a program of $\mathcal{W}_G$ which satisfy the corresponding contract, via Corollary 1. The proof amounts to create a program that follows the explicit victory invariant. We need to make a few implicit hypotheses about the meta-logic for the proof to work out, notably that it can talk about victory invariants of the considered game, and express the notion in a manner compatible with our definition. We also need to be able to express a few set operators.

**Lemma 1.** *For all games $\mathbb{G} = (G, \preccurlyeq, \Delta)$, all well-typed formulas pair $\varphi_0, \varphi_1$ well-typed in context $\Gamma, \mathtt{now} : G$, and all $\Gamma$-assignations $\sigma$, if the existence of a victory invariant for the set pair $\{x \in G \mid [\![\varphi_0]\!]_{\sigma,\mathtt{now}\leftarrow x}\}, \{x \in G \mid [\![\varphi_1]\!]_{\sigma,\mathtt{now}\leftarrow x}\}$ is provable in the meta-logic, then there exists a well-typed program $\mathtt{CONT} : \mathcal{P}(G) \to \{0\} | \Gamma \vdash \pi : \{0\}$ such that $\mathcal{C}[(\Phi_{\mathbb{G}})_{x\in G}, \sigma]\,(\pi : \langle \varphi_0 \hookrightarrow u : \{0\}.\ \varphi_1 \rangle)$ holds.*

*Proof.* See [4] (Lemma 3.19, pages 23-24).

## 6 Related Work 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 [1]. A related Hoare logic is studied by Mamouras in 2016 [9]. 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 [11], 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. Note that step-indexed logics like Iris [7] support non-terminating recursion by the mean of the Löb rule, as well as arbitrary higher-order programs, but drop any support for total correctness, while we support fine-grained specification and proof for non-terminating recursion, but limited to first-order.

*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 [7] should pose no trouble. Another direction is to generalize our approach to a higher-order setting. This is reasonably easily if we enforce a restriction 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 closure themselves passed to recursive calls. 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, it is not clear that it is usable for that purpose in practice. In particular, our auxiliary language has no support for parallel program composition.

## References


1. Back, R.J.R., von Wright, J.: Duality in specification languages: a lattice-theoretical approach. Acta Informatica **27**(7), 583–625 (Jul 1990). 10.1007/BF00259469
2. Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction. pp. 102–126 (2000)
3. Clochard, M.: Méthodes et outils pour la spécification et la preuve de propriétés difficiles de programmes séquentiels. Thèse de doctorat, Université Paris-Saclay (Mar 2018), https://tel.archives-ouvertes.fr/tel-01787689
4. Clochard, M., Paskevich, A., Marché, C.: Deductive verification via ghost debugging. Research Report 9219, Inria (2018), https://hal.inria.fr/hal-01907894
5. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**, 453–457 (August 1975). 10.1145/360933.360975
6. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods in System Design **48**(3), 152–174 (2016). 10.1007/s10703-016-0243-x
7. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J.H., Dreyer, D., Birkedal, L.: The essence of higher-order concurrent separation logic. In: 26th European Symposium on Programming Languages and Systems. Lecture Notes in Computer Science, vol. 10201, pp. 696–723. Springer (2017). 10.1007/978-3-662-54434-1_26
8. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR-16. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
9. Mamouras, K.: Synthesis of strategies using the Hoare logic of angelic and demonic nondeterminism. Logical Methods in Computer Science **12**(3) (2016). 10.2168/LMCS-12(3:6)2016
10. Manna, Z., McCarthy, J.: Properties of programs and partial function logic. In: Machine Intelligence. vol. 5 (1970)
11. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 568–582. Springer (2007)
12. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Communications of the ACM **10**, 501–506 (1967)
13. Yang, H.: Relational separation logic. Theoretical Computer Science **375**(1), 308–334 (2007)