ELSEVIER

Information Processing Letters 53 (1995) 165-172

Information Processing Letters

Games and winning strategies

R.J.R. Back [1], J. von Wright [*,1]

Åbo Akademi University, 20520 Turku, Finland

Abstract

Two-person games are modeled as specifications in a language with angelic and demonic nondeterminism, and methods of program verification and transformation are used to reason about games. That a given strategy is winning can be proved using a variant of the traditional loop correctness rule. Furthermore, an implementation of the winning strategy can be derived using equivalence transformations.

Keywords: Formal semantics; Program correctness; Programming calculi

1. Introduction

Recently, specification languages have been proposed containing two modes of nondeterminism: demonic and angelic nondeterminism. In a predicate transformer framework, both kinds of nondeterminism are easily accommodated, and angelic nondeterminism has been shown to be useful in various ways [3,5,10].

Execution of a command where both angelic and demonic nondeterminism is present can be described as a game, in which the angel tries to guide the execution to a successful end and the demon tries to prevent this [4,7]. Similar connections between games and reactive systems are discussed in [1,9], where results from game theory and descriptive set theory are used to shed light on properties of concurrent systems.

In this paper, we show that it is possible to go in the other direction; we use program verification and program transformation methods to reason about two-person games. In particular, we show how a variation on the classical loop correctness rule [6] can be used to prove the existence of winning strategies in a game. We also indicate how program transformations in the style of [2] can be used to produce a description of the winning strategy as an implementable program.

Our aim is modest - we do not claim that our method can replace traditional reasoning about two-person games. However, it points out a close connection between program verification and reasoning about games.

We work in a total correctness framework. Commands are described as predicate transformers, with the intuition that $S\ q$ (i.e., the weakest precondition for command S with respect to postcondition q) holds in state σ if and only if execution of S in initial state σ is guaranteed to establish q. This intuition has to be extended to allow *angelic nondeterminism*; an angelically nondeterministic command is guaranteed to establish postcondition q if at least one possible computation terminates with q holding (under demonic nondeterminism,

* Corresponding author.

[1] Email: {backrj,jwright}@abo.fi.

all possible computations must terminate with q holding). Our specification language is based on that in [3], to which we refer for more details. A novelty in this paper is the *unguarded iteration*. Combined with the notions of *win* (miracle) and *loss* (abortion), the unguarded iteration is used to represent the repetition of moves in a game.

Notation and proof style. We assume that the reader is familiar with basic facts about lattices, monotonic functions and fixpoints. We use λ-abstraction to describe functions and write function application as juxtaposition, so $f\ x$ is function f applied to argument x. We use $\equiv$ for boolean equality and $\bot$ and $\top$ for the truth values.

We write proofs in a calculational style. In proofs, we permit steps that are justified by properties of functions and quantifiers. This is very useful, e.g., in fixpoint reasoning. We also use the calculational proof format for proof outlines, with hints in the comments.

2. A specification language

A *state* is a tuple of values, e.g., $(2,0,\top)$. Each position in the tuple is a *state component* (also called a program variable).

A *state predicate* is a boolean function on states. State predicates are easily expressed using λ-notation. For example, the state predicate $(\lambda(x,y,b).\ b \wedge (x<y))$ holds in state $(0,1,\top)$ but not in state $(2,0,\bot)$.

By the rules of α-equivalence, the names of the bound variables in a state predicate are arbitrary. However, within a given context we usually give names to the bound variables in a consistent way. In this way we introduce names corresponding to *program variables*. For example, by writing $(\lambda(x,y,b).\ b \wedge (x<y))$, we indicate that the three state components are named x, y and b.

State predicates are ordered by strength (pointwise extension from the booleans); we write $p \leqslant q$ for $(\forall\sigma.\ p\ \sigma \Rightarrow q\ \sigma)$. Negation, conjunction, disjunction and implication of predicates are defined by lifting. For example, $p \wedge q$ is the predicate $(\lambda\sigma.\ p\ \sigma \wedge q\ \sigma)$. We write false for the everywhere false predicate; $\mathsf{false} = (\lambda\sigma.\ \bot)$. The everywhere true predicate true is defined similarly. The state predicates over a fixed state space form a complete boolean lattice.

A *state relation* is a binary state predicate, i.e., a mapping from (initial) states to (final) states to booleans. We let metavariables P, Q and R range over state relations, p, q and r over state predicates and σ over states.

2.1. A command notation

A *predicate transformer* is a function from state predicates to state predicates. We identify commands with their semantic functions, so every command is at the same time a monotonic predicate transformer. We let metavariable S range over commands. When $S\ q\ \sigma \equiv \top$, we say that command S *establishes postcondition* q in state σ. If $p \leqslant S\ q$ holds, then execution of S in any initial state satisfying p is guaranteed to establish postcondition q.

Basic commands. Basic commands in our language are the *angelic update* $\{R\}$ and the *demonic update* $[R]$. Here R is any state relation, and the weakest precondition semantics of the update commands are as follows:

$$\begin{aligned} \{R\}\ q\ \sigma &\equiv (\exists\sigma'.\ R\ \sigma\ \sigma' \wedge q\ \sigma') \\ [R]\ q\ \sigma &\equiv (\forall\sigma'.\ R\ \sigma\ \sigma' \Rightarrow q\ \sigma') \end{aligned}$$

This semantics reflects the following intuition. Executed in initial state σ, the angelic update $\{R\}$ chooses a final state σ' among those satisfying $R\ \sigma\ \sigma'$. The choice is *angelic*, which means that the choice is made so that postcondition q is established, if possible. The demonic update makes a demonic choice between the possible final states, avoiding to establish q, if possible.

If the initial state is such that there is no possible final state, then the angelic update is aborting. Intuitively, this is the situation where the angel has no choice available. In our game interpretation, we look at computations from the angel's point of view, calling an aborting computation a *loss*. Dually, the demonic update leads to a miraculous computation if the demon has no final states to choose between. We call such a computation a *win*. A winning computation establishes any postcondition, even false. Alternatively, we can consider a win to be a deadlocked computation.

As an example, consider state relation $R = (\lambda x\ x'.\ x' < x)$, where the state has only one component, which ranges over the natural numbers. Demonic update $[R]$ decreases the value of the state component by an unspecified amount. If the value is initially 0, then $[R]$ is a win.

Assert and guard. A special case of the angelic update is the *assert command* $\{p\}$, which has next-state relation $(\lambda\sigma\ \sigma'.\ p\ \sigma \wedge (\sigma' = \sigma))$. If state predicate p holds, then the state is left unchanged, otherwise this command leads to a loss. Dually the *guard command* $[p]$ leaves the state unchanged if predicate p holds in the state; otherwise it leads to a win. The semantics of the assert and guard commands are easily computed:

$$\{p\}\ q = p \wedge q \qquad [p]\ q = p \Rightarrow q$$

Sequential composition. Sequential composition has its usual meaning:

$$(S_1; S_2)\ q = S_1(S_2\ q)$$

Execution of a sequential composition of two or more updates can be interpreted as a game between the angel and the demon. The angel chooses final values in each angelic update and the demon chooses values in each demonic update. The compound command S establishes postcondition q if the angel can make choices in such a way that the final state satisfies q, regardless of how the demon makes its choices. Thus S establishes q if the angel has a *strategy* that guarantees a final state satisfying q.

2.2. Recursion and unguarded iteration

The recursive command construct $(\mu X.\ T)$ is defined whenever T is an expression built from commands and command variable X. Then $(\lambda X.\ T)$ is a monotonic function on commands, and $(\mu X.\ T)$ is its least fixpoint.

Iteration commands are usually defined using *guarded* recursion, i.e., they have a termination condition which is evaluated before each iteration. We define the *unguarded iteration command* repeat S forever as the following least fixpoint:

$$\text{repeat } S \text{ forever} = (\mu X.\ S; X).$$

Intuitively, repeat S forever is executed by recursive unfolding, i.e., by repeatedly executing the body S. Such an execution can go on forever, but it can also end in a loss (if an empty angelic choice occurs in the body) or a win (empty demonic choice).

Semantically, execution of repeat S forever can lead to only two things: either no postcondition, not even true, is established (an infinite run or a loss) or else any postcondition, even false, is established (a win). This is because the unguarded iteration can never terminate in a proper state.

Since commands are monotonic functions on the complete lattice of predicates, every command S has a least fixpoint. The following theorem shows that μS characterises those states from which repeated execution of S leads to a win.

Theorem 1. *Assume that S is a command. Then*

$$(\text{repeat } S \text{ forever})\, q \;=\; \mu S$$

for arbitrary predicate q.

Proof. We prove this theorem using basic facts about fixpoints. An alternative proof applies the μ-fusion theorem given in [8]. We have

$$\begin{array}{ll}
 & \mu S \leqslant (\mu X.\ S; X)q \\
\Leftarrow & \{\text{least fixpoint property}\} \\
 & S((\mu X.\ S; X)q) \leqslant (\mu X.\ S; X)q \\
\equiv & \{\text{definition of sequential composition}\} \\
 & (S; (\mu X.\ S; X))q \leqslant (\mu X.\ S; X)q \\
\equiv & \{\text{folding least fixpoint}\} \\
 & \top
\end{array}$$

for arbitrary predicate q, and furthermore

$$\begin{array}{ll}
 & (\forall q.\ (\mu X.\ S; X)q \leqslant \mu S) \\
\equiv & \{\text{pointwise order}\} \\
 & (\mu X.\ S; X) \leqslant (\lambda q.\ \mu S) \\
\Leftarrow & \{\text{least fixpoint property}\} \\
 & S; (\lambda q.\ \mu S) \leqslant (\lambda q.\ \mu S) \\
\equiv & \{\text{pointwise order}\} \\
 & (\forall q.\ (S; (\lambda q.\ \mu S))q \leqslant \mu S) \\
\equiv & \{\text{definition of sequential composition}\} \\
 & S(\mu S) \leqslant \mu S \\
\equiv & \{\text{folding least fixpoint}\} \\
 & \top
\end{array}$$

so equality follows since the order $\leqslant$ is antisymmetric. □

We will also use the following result, which says that if command S can re-establish a precondition while decreasing a termination function, then repeated execution of S will lead to a win.

Theorem 2. *Assume that f is a monotonic function over predicates, p is a predicate, t is a state function ranging over some well-founded set W, and*

$$p \wedge (\lambda\sigma.\, t\,\sigma \leqslant w) \;\leqslant\; f(p \wedge (\lambda\sigma.\, t\,\sigma < w))$$

for all $w \in W$. Then $p \leqslant \mu f$.

Proof. Set $p_w = p \wedge (\lambda\sigma.\ t\ \sigma \leqslant w)$, for all $w \in W$. We first prove $(\forall w \in W.\ p_w \leqslant \mu f)$ by well-founded induction on w. The induction hypothesis is $(\forall v < w.\ p_v \leqslant \mu f)$. We have

$$\begin{array}{ll} & p \wedge (\lambda\sigma.\, t\,\sigma \leqslant w) \\ \leqslant & \{\text{assumption}\} \\ & f(p \wedge (\lambda\sigma.\, t\,\sigma < w)) \\ = & \{\text{see below}\} \\ & f(p \wedge (\lambda\sigma.\, \exists v < w.\, t\,\sigma \leqslant v)) \\ = & \{\text{pointwise order}\} \\ & f\Big(\bigvee_{v<w} (p \wedge (\lambda\sigma.\, t\,\sigma \leqslant v))\Big) \\ \leqslant & \{\text{induction assumption}\} \\ & f(\mu f) \\ = & \{\text{folding least fixpoint}\} \\ & \mu f \end{array}$$

where the second step uses the trivial fact that in an ordered set,

$$x < w \equiv (\exists v < w.\, x \leqslant v)$$

Now,

$$p = \Big(\bigvee_{w \in W} p_w\Big) \leqslant \mu f$$

which finishes the proof. □

3. Modelling games

A *game* is played on a board (the state) between two opponents: Player (or the angel) and Opponent (or the demon). The choices of Player can be coded as an angelic update command $\{P\}$ on the state. Similarly, the choices of Opponent can be coded as a demonic update command $[Q]$ on the state. In the simple case when the two players alternate making moves, the game can be represented as the iteration

$$\mathcal{G} \quad = \quad \text{repeat } \{P\}; [Q] \text{ forever}$$

if Player moves first.

We can also have more complicated commands in the body, if the rules of the game are more intricate. We may also add guards and assertions, to indicate points at which it is decided that the game is won by either player.

Example. As an example, consider a simple version of the game of Nim, where the players alternate removing one or two matches from a pile. Player is the first to move, and the player who removes the last match loses.

The state has only one variable x (the number of matches) and to simplify things, we let x range over the natural numbers with the subtraction rule $0 - 1 = 0$ ("monus" rather than "minus").

We introduce the following abbreviations: *check* is the state predicate $(\lambda x.\, x > 0)$ and *Move* is the state relation $(\lambda x\, x'.\, x - 2 \leqslant x' < x)$. Then the game can be expressed in the following simple form:

$$\mathcal{G} \quad = \quad \text{repeat } [\mathit{check}]; \{\mathit{Move}\}; \{\mathit{check}\}; [\mathit{Move}] \text{ forever.}$$

The guard $[\mathit{check}]$ is a check to see if Player has already won. If not, then Player makes a move, setting the final value of x to either $x-2$ or $x-1$. After this, we have the dual situation. The assertion $\{\mathit{check}\}$ is a check to see if Opponent has won. If not, then Opponent moves according to the demonic update.

4. Proof of winning strategies

We have shown how games can be represented as unguarded iterations. The question is now: what can we prove about a game? If the game repeat S forever establishes postcondition false, then Player can make choices in the angelic updates in such a way that there is a win, regardless of what choices Opponent makes in the demonic updates. This justifies the following definition: we say that *Player has a winning strategy for the game* repeat S forever *under precondition* p if

$$p \leqslant (\text{repeat } S \text{ forever}) \text{ false}.$$

Using the results from Section 2, we get a rule for proving the existence of winning strategies.

Theorem 3. *Player has a winning strategy for game repeat S forever under precondition p, provided the following two conditions hold for some predicate I (the invariant) and some total state function t ranging over some well-founded set W:*

$$p \leqslant I, \tag{1}$$

$$I \wedge (\lambda\sigma.\, t\,\sigma \leqslant w) \leqslant S(I \wedge (\lambda\sigma.\, t\,\sigma < w)) \quad \textit{for all } w \in W. \tag{2}$$

Proof.

$$\begin{array}{ll}
 & p \leqslant (\text{repeat } S \text{ forever}) \text{ false} \\
\equiv & \{\text{Theorem 1}\} \\
 & p \leqslant \mu S \\
\Leftarrow & \{\text{assumption (1)}\} \\
 & I \leqslant \mu S \\
\Leftarrow & \{\text{Theorem 2}\} \\
 & (\forall w \in W.\, I \wedge (\lambda\sigma.\, t\,\sigma \leqslant w) \leqslant S(I \wedge (\lambda\sigma.\, t\,\sigma < w))) \\
\equiv & \{\text{assumption (2)}\} \\
 & \top \quad \square
\end{array}$$

Note that repeat S forever is not a do-loop with a true guard (do true $\rightarrow$ S od), since the body of a do-loop can never be miraculous. Thus Theorem 3 is not a special case of the traditional rule for proving total correctness of loops, though it is similar to that rule.

5. Example: Nim

We now return to the example game of Nim. The global state has a single component x, ranging over natural numbers. It is well known that to win this game, it is necessary always to make the number x of matches satisfy the condition $x \bmod 3 = 1$. The strategy consists of always removing so many matches that $x \bmod 3 = 1$ holds.

5.1. Existence of a winning strategy

Assume that precondition p is $x \bmod 3 \neq 1$ (since Player moves first, this is necessary to guarantee that Player can actually establish $x \bmod 3 = 1$ in the first move). Invariant I is simply p and termination function t

is $(\lambda x.\ x)$. This means that condition (1) is trivially satisfied and we only have to prove condition (2), i.e., that

$$(\lambda x.\ x \bmod 3 \neq 1 \wedge x \leqslant n) \ \leqslant\ S(\lambda x.\ x \bmod 3 \neq 1 \wedge x < n) \tag{3}$$

holds for arbitrary natural number n, where $S = [\mathit{check}]; \{\mathit{Move}\}; \{\mathit{check}\}; [\mathit{Move}]$.

We prove (3) by calculating $S\ I$, in two parts. First, we find the intermediate condition

$$\begin{aligned}
& (\{\mathit{check}\}; [\mathit{Move}])(\lambda x.\ x \bmod 3 \neq 1 \wedge x < n) \\
= \quad & \{\text{definitions}\} \\
& (\lambda x.\ x > 0 \wedge (\forall x'.\ x-2 \leqslant x' < x \Rightarrow x \bmod 3 \neq 1 \wedge x' < n)) \\
= \quad & \{\text{finite quantification}\} \\
& (\lambda x.\ x > 0 \wedge (x-2) \bmod 3 \neq 1 \wedge x-2 < n \wedge (x-1) \bmod 3 \neq 1 \wedge x-1 < n) \\
= \quad & \{\text{arithmetic}\} \\
& (\lambda x.\ x \bmod 3 = 1 \wedge x \leqslant n)
\end{aligned}$$

This is the condition that Player should establish on every move.

Continuing, we find the precondition

$$\begin{aligned}
& ([\mathit{check}]; \{\mathit{Move}\})(\lambda x.\ x \bmod 3 = 1 \wedge x \leqslant n) \\
= \quad & \{\text{definitions}\} \\
& (\lambda x.\ x = 0 \vee (\exists x'.\ x-2 \leqslant x' < x \wedge x \bmod 3 = 1 \wedge x' \leqslant n)) \\
= \quad & \{\text{finite quantification}\} \\
& (\lambda x.\ x = 0 \vee ((x-2) \bmod 3 = 1 \wedge x-2 \leqslant n) \vee ((x-1) \bmod 3 = 1 \wedge x-1 \leqslant n)) \\
\geqslant \quad & \{\text{arithmetic, pointwise order}\} \\
& (\lambda x.\ x \bmod 3 \neq 1 \wedge x \leqslant n+1)
\end{aligned}$$

Since this condition is implied by $(\lambda x.\ x \bmod 3 \neq 1 \wedge x \leqslant n)$, we have shown that (3) holds, i.e., that the game has a winning strategy.

5.2. Extracting a Nim-playing program

The above calculation shows that there exists a winning strategy for Player in the game of Nim. However, the angelic nondeterminism in Player's move means that we do not have an implementation of the strategy. We shall now describe a method for finding an implementation, using program transformations. Because of the space limitation, we only outline the method.

In order to get an implementation (i.e., a program that plays Nim), we apply equivalence preserving transformations to Player's component $\{\mathit{Move}\}$ of the game expression $\mathcal{G}$, in such a way that no angelic nondeterminism remains.

First, we make a case split according to precondition p:

$$\mathcal{G} \ = \ \{p\}; \mathcal{G} \vee \{\neg p\}; \mathcal{G}$$

where $\vee$ is an angelic choice operator (there is no nondeterminism involved in this case; the right hand side is an if-then-else command).

Now it is possible to use the fact that p is an invariant and the calculations in Section 5.1 to replace Player's $\{\mathit{Move}\}$ while preserving the value of $\{p\}; \mathcal{G}$. We replace $\{\mathit{Move}\}$ by the command $[\mathit{Move2}]$, where $\mathit{Move2}$ is the state relation

$$(\lambda x\, x'.\ (x \bmod 3 \neq 2 \wedge x' = x-2)) \vee (x \bmod 3 \neq 0 \wedge x' = x-1)$$

As a result, we have the following equality:

$$\{p\}; \mathcal{G} \quad = \quad \{p\}; \text{repeat } [\mathit{check}]; [\mathit{Move2}]; \{\mathit{check}\}; [\mathit{Move}] \text{ forever}$$

The command $[\mathit{Move2}]$ can be written using Dijkstra's nondeterministic conditional as

$$\text{if } x \bmod 3 \neq 2 \rightarrow x := x-2 \ [] \ x \bmod 3 \neq 0 \rightarrow x := x-1 \text{ fi} \tag{4}$$

Replacing Player's component by applying equivalence transformations can be seen as implementing a strategy. Thus, (4) shows how Player should play Nim under precondition p: if $x \bmod 3 \neq 2$, then remove two matches and if $x \bmod 3 \neq 0$, then remove one match. This strategy does not specify Player's choice completely if $x \bmod 3 = 1$. In fact, no winning strategy exists under this precondition (this can be proved formally using the idea described below).

6. Concluding remarks

In those initial states where no winning strategy exists, our method cannot suggest strategies that would be useful against an imperfect opponent. A finer semantics could be of some use (e.g., preferring a loss after a longer game to a loss after a shorter game), but this would hardly lead to smart strategies.

The methods we have outlined in this paper can also be used to show the nonexistence of winning strategies. This can be done by showing the existence of a winning strategy for the dual game, i.e., the game with the roles of Player and Opponent reversed and with a greatest fixpoint semantics. For dual games, condition (2) simply becomes $I \leqslant S\, I$, i.e., no termination function is needed.

Acknowledgments

We would like to thank Roland Backhouse, David Gries and the anonymous referees for their helpful comments.

References

[1] M. Abadi, L. Lamport and P. Wolper, Realizable and unrealizable specifications of reactive systems, in: G. Ausiello et al., eds., *Proc. 16th ICALP*, Stresa, Italy (Springer, Berlin, 1989) 1-17.

[2] R.J.R. Back, A calculus of refinements for program derivations, *Acta Inform.* **25** (1988) 593-624.

[3] R.J.R. Back and J. von Wright, Refinement calculus, part I: Sequential programs, in: *REX Workshop for Refinement of Distributed Systems*, Nijmegen, The Netherlands, Lecture Notes in Computer Science **430** (Springer, Berlin, 1989).

[4] R.J.R. Back and J. von Wright, Duality in specification languages: A lattice-theoretical approach, *Acta Inform.* **27** (1990) 583-625.

[5] P.H. Gardiner and C.C. Morgan, Data refinement of predicate transformers, *Theoret. Comput. Sci.* **87** (1) (1991) 143-162.

[6] D. Gries, *The Science of Programming* (Springer, New York, 1981).

[7] W.H. Hesselink, Nondeterminism and recursion via stacks and games, *Theoret. Comput. Sci.* **124** (1994) 273-295.

[8] Mathematics of Program Construction Group (Eindhoven University of Technology), Fixed-point calculus, *Inform. Process. Lett.* **53** (3) (1995) 131-136, this issue.

[9] Y.N. Moschowakis, A model of concurrency with fair merge and full recursion, *Inform. and Comput.* (1991) 114-171.

[10] N. Ward and I.J. Hayes, Applications of angelic nondeterminism, in: P.A.C. Bailes, ed., *Proc. 6th Australian Software Engineering Conf.*, Sydney, Australia (1991) 391-404.