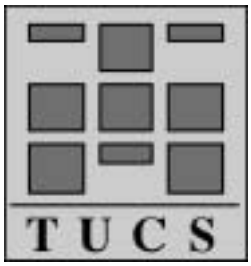
Correctness and Refinement of Dually Nondeterministic Programs

Orieta Celiku
Joakim von Wright



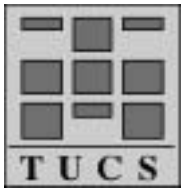
Turku Centre for Computer Science

TUCS Technical Reports

No 516, March 2003

Correctness and Refinement of Dually Nondeterministic Programs

Orieta Celiku
Joakim von Wright



Turku Centre for Computer Science
TUCS Technical Report No 516
March 2003
ISBN 952-12-1139-3
ISSN 1239-1891

Abstract

In this paper we extend different reasoning methods from traditional (demonic) programs to programs with both demonic and angelic nondeterminism. In particular, we discuss correctness proofs, and refinement of programs while reducing angelic nondeterminism (into demonic nondeterminism or determinism). As expected, reducing angelic nondeterminism is generally not a refinement; however, when context is taken into consideration, it can result in refinement. We also show how correctness proofs can be used to implement a winning strategy for the angel (when such a strategy exists).

TUCS Laboratory
Learning and Reasoning Laboratory

1 Introduction

Nondeterminism in programs appears in two dual forms: *demonic* and *angelic*. Intuitively, demonic nondeterminism is resolved so as to avoid establishing the postcondition, while angelic choices are made in such a way that the postcondition is established, if possible.

Of these two forms, demonic nondeterminism has been studied and used more in program reasoning. The main reason for this is that demonic nondeterminism models underspecification, which leaves implementation decisions open and makes abstraction possible. Moreover, the move from more abstract to more concrete specifications goes well with the notion of refinement, in the sense that the less demonic nondeterminism is present in a specification, the more refined and implementable the specification is. Refinement, in turn, has been studied extensively in the refinement calculus [2, 5, 14].

Angelic nondeterminism, on the other hand, is more "controversial": increasing angelic nondeterminism makes a specification more refined, but less implementable. Another drawback of angelic nondeterminism is that the *angel* is assumed always to know how to resolve the choices in the best possible way. Nevertheless, angelic choice is important for more than its algebraic role as *join* operation in the predicate transformer lattice.

Demonic and angelic nondeterminism in combination can model system-user situations, with the demon modeling the system side and the angel the user side. The two types of nondeterminism are also useful when modeling interaction in game-like situations, where a number of agents try to achieve (potentially conflicting) goals by taking turns in making choices [5, 6]. The agent or coalition whose goal we are focusing on is modeled as the angel and the associated nondeterminism is angelic; the remaining agents are collectively considered as the demon.

Correctness of system-user specifications is analyzed with respect to whether the user can carry out his or her desired actions, and refinement makes things better from the user's point of view by potentially increasing the user's choices or decreasing the system's arbitrariness. In the case of games, however, it is more useful to transform the program so as to eliminate "bad" moves from angel's choices. One reason is that the rules of the game are usually laid down in advance, so increasing choices would be illegal. Another reason is that in practice the role of the angel will be played by an imperfect agent, so the existence of a winning strategy will generally not be enough – what we need is an implementation of the winning strategy [5].

In this paper, we describe two methods for reducing angelic nondeterminism (into demonic nondeterminism or determinism) systematically. One consists of extracting a nonangelic program that implements a winning strategy

for the angel (when such a strategy exists) while proving program correctness. We introduce a new correctness rule for angelic choice which makes such extraction possible. The other method is a special form of refinement in context, where the context consists of the precondition and postcondition of the program and information is propagated via guards. We introduce rules for propagating guards over angelic statements, and rules that structurally change angelic constructs into demonic ones. Note that taking the context into consideration is crucial since reducing angelic nondeterminism is generally not a refinement.

We start with some background information in Sec. 2. Sec. 3 discusses correctness proofs and introduces a correctness rule for angelic choice. We discuss what refinement of programs with both angelic and demonic nondeterminism means in Sec. 4. We also recall what refinement in context is, and extend the rules for guard propagation in order to be able to propagate guards over angelic statements. In Sec. 5 we discuss two ways of reducing angelic nondeterminism: one based on refinement in context, and the other on proving correctness. We illustrate the methods with two examples in Sec. 6. Related work and conclusions are discussed in Sec. 7. Proofs for some of the results are shown in App. A.

2 Background

Our formal framework is that of the refinement calculus [5]. The underlying logic is higher-order logic. The application of function f to argument x is written as $f.x$. Proofs are generally written as structured derivations [5].

The *program state* is of polymorphic type Σ. *State predicates* are boolean functions on states ($\Sigma \to \mathsf{Bool}$). *State relations* are binary state predicates ($\Sigma \to \Gamma \to \mathsf{Bool}$) relating (initial) states to (final) states. *Predicate transformers* are functions from state predicates to state predicates ($(\Gamma \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool})$).

The order relation $\subseteq$, negation $\neg$, conjunction $\cap$, disjunction $\cup$, and implication $\Rightarrow$ on predicates, are defined by pointwise extension of the order and corresponding operations on booleans. Similarly, the order $\sqsubseteq$, and operations $\sqcap$ and $\sqcup$ on predicate transformers, are lifted from those on predicates $(\subseteq, \cap, \cup)$.

Predicate transformers map *preconditions* to *postconditions*, the intended interpretation being that of a *weakest precondition* [8]. This means that if q is a postcondition and σ a state, then $S.q$ holds in σ if and only if execution of a program statement modeled by S from initial state σ is guaranteed to terminate in a final state where q holds.

$$
\begin{array}{rcll}
\mathsf{skip}.q & = & q & (skip) \\
\mathsf{abort}.q & = & \mathsf{false} & (abort) \\
\mathsf{magic}.q & = & \mathsf{true} & (magic) \\
(x := e).q & = & q[x := e] & (deterministic\ update) \\
\{g\}.q & = & g \cap q & (assertion) \\
[g].q & = & \neg g \cup q & (guard) \\
(S_1; S_2).q & = & S_1.(S_2.q) & (sequence) \\
(\mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}).q & = & g \cap S_1.q \cup \neg g \cap S_2.q & (conditional) \\
(S_1 \sqcup S_2).q & = & S_1.q \cup S_2.q & (angelic\ choice) \\
(S_1 \sqcap S_2).q & = & S_1.q \cap S_2.q & (demonic\ choice) \\
\{x := x' \mid b\}.q & = & (\exists x' \bullet b \wedge q[x := x']) & (angelic\ update) \\
[x := x' \mid b].q & = & (\forall x' \bullet b \Rightarrow q[x := x']) & (demonic\ update)
\end{array}
$$

Figure 1: Weakest Precondition for Basic Program Statements

Fig. 1 shows basic program statements modeled by predicate transformers. Demonic choice is seen as the choice we have no control over, thus both statements should establish the postcondition for the choice to do so. Angelic choice is interpreted as the choice we can affect, hence, if any of the statements establishes the postcondition, so does angelic choice. Demonic and angelic updates, in which $x := x' \mid b$ describes a state relation that leaves all variables except x unchanged, are interpreted similarly to the choices. A mix of angelic and demonic nondeterminism can be interpreted as a game; it establishes the postcondition q if the angel can make its choices in such a way that q is reached, regardless of how the demon makes its choices.

Predicates form a complete boolean lattice, and so do predicate transformers. In the latter lattice, abort is bottom, magic top, demonic choice $\sqcap$ meet, angelic choice $\sqcup$ join, and refinement $\sqsubseteq$ the order.

The loop construct is defined in the usual way, as the least fixpoint of the unfolding function:

$$\mathsf{do}\ g \rightarrow S\ \mathsf{od}\ =\ (\mu X \bullet \mathsf{if}\ g\ \mathsf{then}\ S; X\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}) \tag{1}$$

We will also need the notion of the *dual* of a predicate transformer:

$$S^{\circ}.q\ =\ \neg S.(\neg q) \tag{2}$$

Fig. 2 lists the dualities that hold between program statements modeled by predicate transformers. The properties can also be read in reverse since dualization is an involution ($(S^{\circ})^{\circ}\ =\ S$).

A program statement S is said to be *monotonic* if:

$$(p \subseteq q)\ \Rightarrow\ (S.p \subseteq S.q)$$

$$\mathsf{skip}^\circ = \mathsf{skip} \quad (3) \qquad \{p\}^\circ = [p] \quad (7)$$
$$\mathsf{magic}^\circ = \mathsf{abort} \quad (4) \qquad \{x := x' \mid b\}^\circ = [x := x' \mid b] \quad (8)$$
$$(x := e)^\circ = (x := e) \quad (5) \qquad (S_1 \sqcup S_2)^\circ = S_1 \sqcap S_2 \quad (9)$$
$$(S_1; S_2)^\circ = S_1^\circ; S_2^\circ \quad (6) \qquad S_1 \sqsubseteq S_2 \equiv S_2^\circ \sqsubseteq S_1^\circ \quad (10)$$

Figure 2: Duality Properties

Furthermore, S is *conjunctive* if, for an arbitrary nonempty collection of predicates, we have :

$$S.(\cap i \mid i \in I \bullet q_i) = (\cap i \mid i \in I \bullet S.q_i)$$

Dually, S is *disjunctive* if:

$$S.(\cup i \mid i \in I \bullet q_i) = (\cup i \mid i \in I \bullet S.q_i)$$

All statements introduced above satisfy monotonicity. Statements that do not contain angelic nondeterminism also satisfy conjunctivity, while statements that do not contain demonic nondeterminism satisfy disjunctivity.

3 Correctness

In this section we recall facts about program correctness, with special focus on correctness of angelic update and angelic choice. We also introduce a new rule for proving correctness of angelic choice.

A program statement S is *totally correct* with respect to precondition p and postcondition q if any execution of S from an initial state where p holds terminates in a final state where q holds:

$$p \; \{\!|\, S \,|\!\} \; q \equiv p \subseteq S.q \quad (11)$$

With S interpreted as a game, correctness means that the angel has a winning strategy for goal q in any initial state that satisfies p [5].

Hoare-logic [11] style rules (Fig. 3) can be used to reason about program correctness. For most statements, the rules follow directly from the definitions (Fig. 1). Loop correctness is proved by proving that invariant I is established initially, preserved by each iteration, and is strong enough to establish the postcondition upon loop termination. Loop termination is proved by proving that the variant t is decreased at each iteration.

$$\frac{p \subseteq q}{p \;\{\!|\, \mathsf{skip} \,|\!\}\; q} \quad (12) \qquad \frac{p \;\{\!|\, S_1 \,|\!\}\; r \qquad r \;\{\!|\, S_2 \,|\!\}\; q}{p \;\{\!|\, S_1; S_2 \,|\!\}\; q} \quad (16)$$

$$\frac{p \subseteq q[x := e]}{p \;\{\!|\, x := e \,|\!\}\; q} \quad (13) \qquad \frac{g \cap p \;\{\!|\, S_1 \,|\!\}\; q \qquad \neg g \cap p \;\{\!|\, S_2 \,|\!\}\; q}{p \;\{\!|\, \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \,|\!\}\; q} \quad (17)$$

$$\frac{p \subseteq g \cap q}{p \;\{\!|\, \{g\} \,|\!\}\; q} \quad (14) \qquad \frac{p \;\{\!|\, S_1 \,|\!\}\; q \qquad p \;\{\!|\, S_2 \,|\!\}\; q}{p \;\{\!|\, S_1 \sqcap S_2 \,|\!\}\; q} \quad (18)$$

$$\frac{p \cap g \subseteq q}{p \;\{\!|\, [g] \,|\!\}\; q} \quad (15) \qquad \frac{p \subseteq (\forall x' \bullet b \Rightarrow q[x := x'])}{p \;\{\!|\, [x := x' \mid b] \,|\!\}\; q} \quad (19)$$

$$\frac{p \subseteq I \qquad g \cap I \cap (t = n) \;\{\!|\, S \,|\!\}\; I \cap t < n \qquad \neg g \cap I \subseteq q}{p \;\{\!|\, \mathsf{do}\ g \to S\ \mathsf{od} \,|\!\}\; q} \quad (20)$$

Figure 3: Hoare Logic for Total Correctness

The rule for angelic assignment is also straightforward:

$$\frac{p \subseteq (\exists x' \bullet b \wedge q[x := x'])}{p \;\{\!|\, \{x := x' \mid b\} \,|\!\}\; q} \quad (21)$$

A witness for the existentially quantified condition describes a winning strategy for the angel, since it describes one of the alternatives contained in b that is going to lead to a win for the angel.

The two obvious correctness rules [5] for angelic choice are:

$$\frac{p \;\{\!|\, S_1 \,|\!\}\; q}{p \;\{\!|\, S_1 \sqcup S_2 \,|\!\}\; q} \quad (22) \qquad \frac{p \;\{\!|\, S_2 \,|\!\}\; q}{p \;\{\!|\, S_1 \sqcup S_2 \,|\!\}\; q} \quad (23)$$

and their interpretation, as expected, is that if one of the statements is correct with respect to the pre- and postcondition then the angelic choice statement is also correct.

These rules are too strong and as such not always helpful. Consider the following example:

$$(x = 0 \vee x = 1) \;\{\!|\, x := x + 1 \sqcup \mathsf{skip} \,|\!\}\; x = 1$$

In order to prove this assertion we need to prove one of the following assertions:

$$(x = 0 \vee x = 1) \;\{\!|\, x := x + 1 \,|\!\}\; x = 1 \qquad (x = 0 \vee x = 1) \;\{\!|\, \mathsf{skip} \,|\!\}\; x = 1$$

Neither of them is true although the initial correctness assertion is. We can get further if we reason as follows: if initially $x = 0$ then the first assignment

establishes the postcondition, whereas if $x = 1$ the second assignment does the job.

We introduce the following rule, which enables the above line of reasoning:

$$\frac{r \cap p \ \{\!|\, S_1 \,|\!\}\ q \qquad \neg r \cap p \ \{\!|\, S_2 \,|\!\}\ q}{p \ \{\!|\, S_1 \sqcup S_2 \,|\!\}\ q} \tag{24}$$

The predicate r describes a winning strategy for the angel by partitioning the precondition into two parts depending on which of the choice statements can establish the postcondition. Note that for $r = \mathsf{true}$ ($r = \mathsf{false}$) we get as special case Rule 22 (Rule 23).

The existence of a partitioning predicate is guaranteed when the angel has a winning strategy:

$$p \ \{\!|\, S_1 \sqcup S_2 \,|\!\}\ q \ \equiv\ (\exists r \bullet (r \cap p \ \{\!|\, S_1 \,|\!\}\ q) \wedge (\neg r \cap p \ \{\!|\, S_2 \,|\!\}\ q)) \tag{25}$$

Discovering such a predicate can be reduced to calculating weakest preconditions. More concretely, $S_1.q$ is always a valid partitioning predicate, since the second statement need be correct only when the first one fails to.

In the example above, we use $r = (x = 0)$; the proof obligations, which are trivially true, are:

$$x = 0 \ \{\!|\, x := x + 1 \,|\!\}\ x = 1 \qquad\qquad x = 1 \ \{\!|\, \mathsf{skip} \,|\!\}\ x = 1$$

Rule 24 is hinted at in [5], where the correctness rule given for (the arbitrary) angelic choice:

$$p \ \{\!|\, (\sqcup i \in I \bullet S_i) \,|\!\}\ q \ \equiv\ (\forall \sigma \bullet p.\sigma \Rightarrow (\exists i \in I \bullet \{\sigma\} \ \{\!|\, S_i \,|\!\}\ q)) \tag{26}$$

permits the chosen alternative S_i to depend on the present state σ.

4 Refinement

We recall that a statement S *is refined* by S' if S' is correct with respect to postcondition q whenever S is:

$$S \sqsubseteq S' \ \equiv\ (\forall q \bullet S.q \subseteq S'.q) \tag{27}$$

If we refine only the demonic parts of a program (and do not introduce angelic nondeterminism as a result), then traditional interpretations work: the original specification leaves decisions open to be made during refinement or implementation.

If we refine angelic parts, then we can introduce new capabilities. This may make completely new behaviors possible, but since they are under the control of the angel, they need not appear in an execution. Refinement may introduce chaotic changes, for example, adding a new "menu alternative" containing arbitrary computations into an angelic choice. This kind of refinement is applicable in client-programmer situations: system functionality can be added in a stepwise manner and presented as a collection of choices to the user. Increasing angelic nondeterminism is a well understood refinement.

On the other hand, we could require that no new angelic nondeterminism be added, and that the angelic parts of the program may only be changed under equivalence (into more deterministic constructs). This would apply to game-like situations where the rules of the game are laid down in advance, and players are not permitted any new moves. Refinement then aims at transforming the angelic parts so that no angelic constructs remain, that is, no clairvoyance is required.

This means that we are looking for transformations of the form $\{x := x' \mid b\} \sqsubseteq \{x := x' \mid b'\}$ where $b' \subseteq b$ (i.e., b' has less alternatives than b), or $\{x := x' \mid b\} \sqsubseteq [x := x' \mid b'']$ (where again $b'' \subseteq b$), which are generally not valid. These problems are also described in [5] and solved in an ad hoc way (for games where the angelic nondeterminism appears in certain locations). Here we give methods that do not depend on where the angelic nondeterminism is located.

4.1 Context Information

As hinted above, in many situations, requiring that a statement S be refined by a statement S' is too restrictive. If S occurs in a statement C (written as $C[S]$) we may be interested only in replacing S by S' so that $C[S] \sqsubseteq C[S']$. In fact, if $S \sqsubseteq S'$ then $C[S] \sqsubseteq C[S']$ since all statements are monotonic. However, the latter refinement can be true even if $S \sqsubseteq S'$ does not hold. For example:

$$\{x = 1\}; x := x + 1 \sqsubseteq \{x = 1\}; x := 2$$

even though $x := 2$ is not a refinement of $x := x + 1$.

The refinement of S by S' while taking context C into consideration is known as *refinement in context* [3, 5, 14]. There are two general methods for performing such refinements systematically.

The *assertion method*: introduce information which can safely be assumed (from preconditions and forward propagation) to get $\{p\}; S$, then use information p to refine S, and finally remove the assertion (using the rule $\{p\} \sqsubseteq \mathsf{skip}$).

The *guard method*: introduce information which is assumed to be true (using the rule $\mathsf{skip} \sqsubseteq [p]$) to get $[p]; S$, then use information p to refine S, and finally remove the guard by backward propagation.

4.2 Guard Propagation

We will be adapting the guard method with the purpose of getting a method for reducing angelic nondeterminism (Sec. 5), so here we concentrate on how guards are propagated. Fig. 4 shows the rules for guard propagation for demonic programs [12] extended with rules for handling angelic statements.

$$\mathsf{skip}; [q] = [q]; \mathsf{skip} \tag{28}$$
$$[p]; [q] = [p \cap q] \tag{29}$$
$$\{p\}; [q] = [\neg p \cup q]; \{p\} \tag{30}$$
$$(x := e); [q] = [q[x := e]]; (x := e) \tag{31}$$
$$\mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}; [q] = \mathsf{if}\ g\ \mathsf{then}\ S_1; [q]\ \mathsf{else}\ S_2; [q]\ \mathsf{fi} \tag{32}$$
$$\mathsf{if}\ g\ \mathsf{then}\ [p]; S_1\ \mathsf{else}\ [q]; S_2\ \mathsf{fi} \sqsubseteq [g \cap p \cup \neg g \cap q]; \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \tag{33}$$
$$[x := x' \mid b]; [q] \sqsubseteq [\forall x' \bullet b \Rightarrow q[x := x']]; [x := x' \mid b] \tag{34}$$
$$[p]; S_1 \sqcap [q]; S_2 \sqsubseteq [p \cup q]; (S_1 \sqcap S_2) \tag{35}$$
$$(S_1 \sqcap S_2); [q] = S_1; [q] \sqcap S_2; [q] \tag{36}$$
$$\{x := x' \mid b\}; [q] = [\forall x' \bullet b \Rightarrow q[x := x']]; \{x := x' \mid b\} \tag{37}$$
$$[p]; S_1 \sqcup [q]; S_2 = [p \cap q]; (S_1 \sqcup S_2) \tag{38}$$
$$(S_1 \sqcup S_2); [q] = S_1; [q] \sqcup S_2; [q] \tag{39}$$

Figure 4: Rules for Guard Propagation

A good part of the rules for angelic and deterministic statements are instances of the following more general rule:

$$\mathsf{disjunctive}\ S \ \vdash\ S; [q] \ =\ [S^{\circ}.q]; S \tag{40}$$

This rule can be further generalized as follows:

$$S^{\circ}.q \subseteq p \ \vdash\ S; [q] \ =\ [p]; S; [q] \tag{41}$$

Note that S need not be disjunctive; monotonicity is sufficient. The cost of this generalization is that the generated guard p no longer carries the maximum possible amount of information – we cannot discard the "postguard" $[q]$. For this reason, it might seem like applying the rule repeatedly to pull a guard over a sequence of statements will lead to the introduction of a trace of guards. That is, a situation like:

$$S_1; S_2; [q] \;=\; S_1; [p']; S_2; [q] \;=\; [p]; S_1; [p']; S_2; [q]$$

However, by using $(S_1; S_2)^\circ.q$ as p or using Rule 41 from right to left, we can avoid extra intermediate guards.

A corollary of Rule 41 is:

$$S^\circ.q \subseteq p \;\vdash\; [p]; S \sqsubseteq S; [q] \tag{42}$$

Rule 41 can also be used to pull guards over loops. The idea is that $S^\circ.q \subseteq p$ is the same as $\neg p \subseteq S.(\neg q)$. So when S is a loop, we compute some p' such that $p' \subseteq S.(\neg q)$ and then set p to $\neg p'$. Although the usefulness of such a propagation might seem doubtful, we will show in Sec. 5.2 that it is not.

5 Implementing Angelic Nondeterminism

By "implementation" we will mean reducing angelic nondeterminism into demonic nondeterminism or determinism. This kind of transformation can be thought of as implementation in the following sense: angelic specifications abstract away from the details of how the winning strategy is to be implemented by assuming that the angel knows how to make the right moves; demonic and deterministic specifications that achieve the same goal are more concrete because they contain only right moves.

As already noted, implementing angelic nondeterminism is usually not a refinement, but can be so with respect to certain contexts. The most natural context to consider is the precondition and postcondition of a program. After all, we are interested in those choices that will guarantee that the postcondition is established, starting from an initial state satisfying the precondition.

We will use this fact, which characterizes correctness on the statement level:

$$p \;\{\!|\, S \,|\!\}\; q \;\equiv\; [p]; S; [\neg q] = \mathsf{magic} \tag{43}$$

The intention is to start from $p \;\{\!|\, S \,|\!\}\; q$ and find a more deterministic S' such that $p \;\{\!|\, S \,|\!\}\; q = p \;\{\!|\, S' \,|\!\}\; q$ (we call this the *correctness-based method*); alternatively, we can start from $[p]; S; [\neg q]$ and transform S into S' so that $[p]; S; [\neg q] = [p]; S'; [\neg q]$ (the *refinement-based method*).

5.1 Correctness-Based Method

The crux of this method is describing a winning strategy while proving correctness for a program S that contains angelic nondeterminism. In a separate step we construct a nonangelic program S' that implements the winning strategy.

$$\begin{array}{ccccccc} rr & \Leftarrow \cdots \Leftarrow & rr' & \Leftarrow \cdots \Leftarrow & \mathsf{T} \\ \Downarrow & & \Downarrow & & \\ p \;\{\!|\, S \,|\!\}\; q & & p \;\{\!|\, S' \,|\!\}\; q & & \end{array}$$

So, while trying to reduce $p \;\{\!|\, S \,|\!\}\; q$ to *true*, we prove facts rr', which can also imply that S' is correct with respect to p and q.

Example

$$\begin{array}{ll} & (x = 0 \vee x = 1) \;\{\!|\, x := x + 1 \sqcup \mathsf{skip} \,|\!\}\; x = 1 \\ \Leftarrow & \{\text{angelic choice, Rule 24, } r = (x = 0)\} \\ & \boxed{(x = 0 \;\{\!|\, x := x + 1 \,|\!\}\; x = 1) \wedge (x = 1 \;\{\!|\, \mathsf{skip} \,|\!\}\; x = 1)} \\ \Leftarrow & \{\text{assignment, Rule 13; skip, Rule 12}\} \\ & \mathsf{T} \end{array}$$

Think of the framed assertion as rr' above. Now we proceed as follows:

$$\begin{array}{ll} & \mathsf{T} \\ \Rightarrow & \{\text{first derivation}\} \\ & (x = 0 \;\{\!|\, x := x + 1 \,|\!\}\; x = 1) \wedge (x = 1 \;\{\!|\, \mathsf{skip} \,|\!\}\; x = 1) \\ \Rightarrow & \{\text{conditional, Rule 17, } g = (x = 0)\} \\ & (x = 0 \vee x = 1) \;\{\!|\, \mathsf{if}\ x = 0\ \mathsf{then}\ x := x + 1\ \mathsf{else\ skip\ fi} \,|\!\}\; x = 1 \end{array}$$

We have transformed the angelic choice into a conditional statement showing when each of the choices is to be executed. Note how the intermediate predicate r became the guard of the conditional. This result is not all that surprising, since the correctness rules for angelic choice and the conditional (Rule 24 and Rule 17) are similar:

$$\frac{r \cap p \;\{\!|\, S_1 \,|\!\}\; q \qquad \neg r \cap p \;\{\!|\, S_2 \,|\!\}\; q}{p \;\{\!|\, S_1 \sqcup S_2 \,|\!\}\; q} \qquad \frac{g \cap p \;\{\!|\, S_1 \,|\!\}\; q \qquad \neg g \cap p \;\{\!|\, S_2 \,|\!\}\; q}{p \;\{\!|\, \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \,|\!\}\; q}$$

Once we have found a working r we have also found how to implement angelic choice.

Example

$$
\begin{array}{ll}
 & x = 0 \ \{\!|\ \{x := x' \mid x' > x\}\ |\!\}\ x < 3 \\
\Leftarrow & \{\text{angelic update, Rule 21}\} \\
 & (x = 0) \Rightarrow (\exists x' \bullet x' > x \wedge x' < 3) \\
\Leftarrow & \{\text{eliminate existential quantifier, witness } x' = x + 1\} \\
 & \boxed{(x = 0) \Rightarrow (x + 1 < 3)} \\
= & \{\text{one-point rule, arithmetic}\} \\
 & \mathsf{T}
\end{array}
$$

The witness for the existential quantifier gives rise to an assignment:

$$
\begin{array}{ll}
 & \mathsf{T} \\
= & \{\text{above derivation}\} \\
 & (x = 0) \Rightarrow (x + 1 < 3) \\
\Rightarrow & \{\text{assignment, Rule 13, } (x + 1 < 3) = ((x < 3)[x := x + 1])\} \\
 & x = 0 \ \{\!|\ x := x + 1\ |\!\}\ x < 3
\end{array}
$$

In fact, the witness for the existential quantifier always translates into an assignment that implements angelic update.

5.2 Refinement-Based Method

This method consists of using guard propagation and some structural rules to transform a statement S (of form $S_1; \cdots; S_i; \cdots; S_n$, where S_i is angelic) into S' in which there is no angelic nondeterminism.

$$
\begin{array}{ll}
 & [p]; S_1; \cdots; S_i; \cdots; S_n; [\neg q] \\
= & \{\text{guard propagation, use equality rules}\} \\
 & [p]; S_1; \cdots; S_i; [\neg q']; \cdots; S_n; [\neg q] \\
\sqsubseteq & \{\text{introduce a suitable } p' \text{ (for example } S_i.q')\} \\
 & [p]; S_1; \cdots; [p']; S_i; [\neg q']; \cdots; S_n; [\neg q] \\
\sqsubseteq & \{\text{apply structural rules}\} \\
 & [p]; S_1; \cdots; [p']; S_i'; [\neg q']; \cdots; S_n; [\neg q] \\
= & \{\text{reverse guard propagation}\} \\
 & [p]; S_1; \cdots; [p']; S_i'; \cdots; S_n; [\neg q] \\
\sqsubseteq & \{\text{backward propagation of } p' \text{ should make it disappear}\} \\
 & [p]; S_1; \cdots; S_i'; \cdots; S_n; [\neg q]
\end{array}
$$

As a result, we should get $[p]; S; [\neg q] \sqsubseteq [p]; S'; [\neg q]$ and if, indeed, the angel has a winning strategy in S ($[p]; S; [\neg q] = \mathsf{magic}$, Theorem 43), then it does in S' too ($[p]; S'; [\neg q] = \mathsf{magic}$ since magic is top element).

5.2.1 Loops

Two additional concerns arise when dealing with loops: propagating guards meaningfully over them, and considering what context should be used when implementing angelic nondeterminism in their bodies.

The first concern was already discussed in Sec. 4.2: when faced with $S; [q]$ where S is a loop, we try to find a predicate p such that $p \subseteq S.(\neg q)$. From Rule 41, we get: $S; [q] = [\neg p]; S; [q]$. However, finding the (weakest) precondition of loop S with respect to the potentially strange predicate $\neg q$ can at best prove difficult.

But we now have more information: the guard to be propagated will, in fact, be the negation of some intermediate predicate that should be established once the loop terminates (since we start with the propagation of the negation of the postcondition). This means that the predicate p we are looking for could as well be the loop invariant (since the loop invariant is a loop precondition, although not necessarily the weakest).

Regarding the second concern, when implementing the angelic parts of a loop $\mathsf{do}\ g \to S\ \mathsf{od}$ it is sufficient to refine:

$$[g \cap I \cap (t = n)]; S; [\neg(I \cap t < n)]$$

where I is loop invariant and t termination function. Intuitively, the reason is that the loop effect, expressed by the invariant, remains the same, so if the initial loop is good enough for our purposes then the refined one will also do. This derivation schema explains things more formally:

$$\begin{array}{ll} & p \ \{\!|\ S_1; \mathsf{do}\ g \to S\ \mathsf{od}; S_2\ |\!\}\ q \\ \Leftarrow & \{\text{sequential composition, Rule 16, } r_1 \text{ and } r_2 \text{ intermediate assertions}\} \\ & (p \ \{\!|\ S_1\ |\!\}\ r_1) \wedge (r_1 \ \{\!|\ \mathsf{do}\ g \to S\ \mathsf{od}\ |\!\}\ r_2) \wedge (r_2 \ \{\!|\ S_2\ |\!\}\ q) \\ \Leftarrow & \{\text{focus on second conjunct}\} \\ & \quad \bullet\ \ r_1 \ \{\!|\ \mathsf{do}\ g \to S\ \mathsf{od}\ |\!\}\ r_2 \\ & \quad \Leftarrow\ \{\text{loop, Rule 20, invariant } I, \text{ variant } t\} \\ & \quad (r_1 \subseteq I) \wedge \boxed{(g \cap I \cap (t = n) \ \{\!|\ S\ |\!\}\ I \cap t < n)} \wedge (\neg g \cap I \subseteq r_2) \end{array}$$

Now it is clearer that the transformation of the loop body S into S' while keeping the same invariant and variant, ultimately affects only the framed assertion. So if the initial program is proved correct, and:

$$[g \cap I \cap (t = n)]; S; [\neg(I \cap t < n)] = [g \cap I \cap (t = n)]; S'; [\neg(I \cap t < n)]$$

then we can substitute S' for S in the loop body.

5.2.2 Structural Rules

We are aiming at rules that change angelic constructs into demonic (or deterministic) ones. The operational idea is that we are happy with any of the alternatives (from the angelic ones) that establish the postcondition, so if there are several such we can decide later which one to implement; we can let the demon resolve the acceptable part of angelic nondeterminism.

Angelic choice. One of the results that follow from Theorem 43 is:

$$\begin{array}{ll} & p \subseteq (S_1 \sqcup S_2).q \\ \vdash & [p]; (S_1 \sqcup S_2); [\neg q] \;=\; [p]; \mathsf{if}\ S_1.q \to S_1 \;[\!]\; S_2.q \to S_2\ \mathsf{fi}; [\neg q] \end{array} \quad (44)$$

where Dijkstra's *guarded conditional* is defined as:

$$\mathsf{if}\ g_1 \to S_1 \;[\!]\; g_2 \to S_2\ \mathsf{fi} \;=\; \{g_1 \cup g_2\}; ([g_1]; S_1 \sqcap [g_2]; S_2) \quad (45)$$

When both guards are true, either of the conditional branches can be executed. This means that in the states where both choice statements are correct with respect to the postcondition we can let the demon choose which one to execute.

Angelic update. The rule for implementing angelic update is:

$$\begin{array}{ll} & p \subseteq \{x := x' \mid b\}.q \\ \vdash & [p]; \{x := x' \mid b\}; [\neg q] \;=\; [p]; [x := x' \mid b \wedge q[x := x']]; [\neg q] \end{array} \quad (46)$$

The demon is given choices according to b but only among those satisfying q. It is simple to see that the right-hand side is always magic, but the assumption that the angel has a winning strategy makes the left-hand side magic too.

Example Let us transform $\{x := x' \mid x' > x\}$ for precondition $x = 0$ and postcondition $0 \le x \le 2$.

$$\begin{array}{ll} & [x = 0]; \{x := x' \mid x' > x\}; [\neg(0 \le x \le 2)] \\ = & \{\text{Rule 46}\} \\ & \quad \bullet\ (x = 0) \Rightarrow (\exists x' \bullet (x' > x) \wedge (0 \le x' \le 2)) \\ & \quad \Leftarrow\ \{\text{one point rule, witness } 1\} \\ & \quad\ \ \mathsf{T} \\ \cdots & [x = 0]; [x := x' \mid (x' > x) \wedge (0 \le x' \le 2)]; [\neg(0 \le x \le 2)] \end{array}$$

$$
\begin{array}{ll}
\sqsubseteq & \{\text{use context assumption}\} \\
& [x = 0]; [x := x' \mid (x' > 0) \wedge (0 \leq x' \leq 2)]; [\neg(0 \leq x \leq 2)] \\
= & \{\text{simplification}\} \\
& [x = 0]; (x := 1 \sqcap x := 2); [\neg(0 \leq x \leq 2)]
\end{array}
$$

The result shows that setting x to either 1 or 2 is valid and that in fact these are the only valid options.

6 Examples

Sec. 6.1 describes a two-player game, and shows an implementation of the winning strategy for the first player. Nim is taken from Back and von Wright [5]. Sec. 6.2 describes the synthesis of a discrete controller, an example taken from Asarin et al [1].

6.1 Nim

In Nim two players take turns in removing one or two matches from a pile (of x matches). The player to remove the last match loses the game. We side with the first player, hence the first player is interpreted as the angel and the other player as the demon.

The angel can win the game, provided that initially $x \bmod 3 \neq 1$, by making sure that $x \bmod 3 = 1$ after its turn. We have proved this assertion:

$$
\begin{array}{l}
\mathsf{var}\ x : \mathsf{num}. \\
\neg(x \bmod 3 = 1)\ \{\!| \\
\mathsf{do}\ /\!*\ \mathsf{invariant}\ \neg(x \bmod 3 = 1)\ \mathsf{variant}\ x\ *\!/ \\
\mathsf{T} \rightarrow [0 < x]; (x := x - 1 \sqcup x := x - 2); \\
\qquad \{0 < x\}; (x := x - 1 \sqcap x := x - 2) \\
\mathsf{od} \\
|\!\}\ \mathsf{T}
\end{array}
$$

The guard $[0 < x]$ is interpreted as: check if there are matches before the angel takes the turn and if not the game is over and the angel has won. Similarly, the demon wins if the assertion $\{0 < x\}$ fails.

Now we implement the angelic part of the loop body:

$$
\begin{array}{l}
[\neg(x \bmod 3 = 1) \wedge (x = n)]; \\
[0 < x]; (x := x - 1 \sqcup x := x - 2);
\end{array}
$$

$\{0 < x\}; (x := x - 1 \sqcap x := x - 2);$
$[(x \bmod 3 = 1) \vee x \geq n]$

$=$ {merge guards}
$[\neg(x \bmod 3 = 1) \wedge (x = n) \wedge 0 < x];$
$(x := x - 1 \sqcup x := x - 2); \{0 < x\};$
$(x := x - 1 \sqcap x := x - 2);$
$[(x \bmod 3 = 1) \vee x \geq n]$

$=$ {pull guard over demonic choice, Rule 41, $((S_1 \sqcap S_2)^\circ = S_1 \sqcup S_2)$}
$[\neg(x \bmod 3 = 1) \wedge (x = n) \wedge 0 < x];$
$(x := x - 1 \sqcup x := x - 2); \{0 < x\};$
$[((x - 1) \bmod 3 = 1) \vee (x - 1 \geq n) \vee$
$((x - 2) \bmod 3 = 2) \vee (x - 2 \geq n)];$
$(x := x - 1 \sqcap x := x - 2);$
$[(x \bmod 3 = 1) \vee x \geq n]$

$=$ {pull guard over assertion, Rule 30}
$[\neg(x \bmod 3 = 1) \wedge 0 < x];$
$(x := x - 1 \sqcup x := x - 2);$
$[(x = 0) \vee ((x - 1) \bmod 3 = 1) \vee (x - 1 \geq n) \vee$
$((x - 2) \bmod 3 = 2) \vee (x - 2 \geq n)];$
$\{0 < x\}; (x := x - 1 \sqcap x := x - 2);$
$[(x \bmod 3 = 1) \vee x \geq n]$

$=$ {focus}

- $[\neg(x \bmod 3 = 1) \wedge 0 < x]$
  $(x := x - 1 \sqcup x := x - 2);$
  $[(x = 0) \vee ((x - 1) \bmod 3 = 1) \vee (x - 1 \geq n) \vee$
  $((x - 2) \bmod 3 = 2) \vee (x - 2 \geq n)]$

  $=$ {replace angelic choice by conditional, Rule 44}

$$
\begin{array}{lll}
 & \bullet & (\neg(x \bmod 3 = 1) \wedge (x = n) \wedge 0 < x) \ \{\!| \\
 & & x := x - 1 \sqcup x := x - 2 \\
 & & |\!\} \ (0 < x \wedge \neg((x-1) \bmod 3 = 1) \wedge (x - 1 < n) \wedge \\
 & & \quad \neg((x-2) \bmod 3 = 1) \wedge (x - 2 < n)) \\
 & = & \{\text{simplify postcondition}\} \\
 & & (\neg(x \bmod 3 = 1) \wedge (x = n) \wedge 0 < x) \ \{\!| \\
 & & x := x - 1 \sqcup x := x - 2 \\
 & & |\!\} \ ((x \bmod 3 = 1) \wedge (x - 1 < n)) \\
 & \Leftarrow & \{\text{correctness reasoning}\} \\
 & & \mathsf{T} \\
\cdots & & [\neg(x \bmod 3 = 1) \wedge (x = n) \wedge 0 < x]; \\
 & & \mathsf{if} \ (x := x - 1).((x \bmod 3 = 1) \wedge (x - 1 < n)) \rightarrow x := x - 1 \\
 & & [\!] \ (x := x - 2).((x \bmod 3 = 1) \wedge (x - 1 < n)) \rightarrow x := x - 2 \ \mathsf{fi}; \\
 & & [(x = 0) \vee ((x - 1) \bmod 3 = 1) \vee (x - 1 \geq n) \vee \\
 & & ((x - 2) \bmod 3 = 2) \vee (x - 2 \geq n)] \\
 & = & \{\text{simplify conditional guards}\} \\
 & & [\neg(x \bmod 3 = 1) \wedge (x = n) \wedge 0 < x]; \\
 & & \mathsf{if} \ (x \bmod 3 = 2) \rightarrow x := x - 1 \\
 & & [\!] \ (x \bmod 3 = 0) \rightarrow x := x - 2 \ \mathsf{fi}; \\
 & & [(x = 0) \vee ((x - 1) \bmod 3 = 1) \vee (x - 1 \geq n) \vee \\
 & & ((x - 2) \bmod 3 = 2) \vee (x - 2 \geq n)] \\
\cdots & & [\neg(x \bmod 3 = 1) \wedge (x = n) \wedge 0 < x]; \\
 & & \mathsf{if} \ (x \bmod 3 = 2) \rightarrow x := x - 1 \\
 & & [\!] \ (x \bmod 3 = 0) \rightarrow x := x - 2 \ \mathsf{fi}; \\
 & & [(x = 0) \vee ((x - 1) \bmod 3 = 1) \vee (x - 1 \geq n) \vee \\
 & & ((x - 2) \bmod 3 = 2) \vee (x - 2 \geq n)]; \\
 & & \{0 < x\}; (x := x - 1 \sqcap x := x - 2); \\
 & & [(x \bmod 3 = 1) \vee (x \geq n)] \\
= & & \{\text{split guards, push guards backwards (reverse initial steps)}\} \\
 & & [\neg(x \bmod 3 = 1) \wedge (x = n)]; [0 < x]; \\
 & & \mathsf{if} \ (x \bmod 3 = 2) \rightarrow x := x - 1 \\
 & & [\!] \ (x \bmod 3 = 0) \rightarrow x := x - 2 \ \mathsf{fi}; \\
 & & \{0 < x\}; (x := x - 1 \sqcap x := x - 2); \\
 & & [(x \bmod 3 = 1) \vee (x \geq n)]
\end{array}
$$

When simplifying the guards we used the fact that x was assumed positive before the conditional, and also that $x = n$. Note also that we use *monus* when subtracting natural numbers: $(x < y) \Rightarrow (x - y = 0)$.

The implementation shows that the angel needs to establish $x \bmod 3 = 1$

after its turn. The guarded conditional statement can be replaced by a standard conditional statement since the guards cannot be true at the same time, and the context assumption $\neg(x \bmod 3) = 1$ limits the (normally three) options ($x \bmod 3$ is 0 or 1 or 2) to the two mentioned in the guards.

6.2 A Discrete Scheduler

This section describes the synthesis of a discrete controller. The example is a simplified version of the discrete scheduler taken from Asarin et al. [1]. Their system, which is defined as an automaton, is synthesized algorithmically by searching the state space and pruning out bad states. We, on the other hand, start with an angelic scheduler, prove its correctness with respect to the desired behavior, and then implement the angelic nondeterminism according to this behavior.

Suppose we have two identical processes that can be either waiting (w_i) or idle ($\neg w_i$). The processes can be idle as long as they wish, and they can also generate a request (move to waiting). They can move from waiting to idle whenever the scheduler gives them permission by flagging the variable g_i. We want the scheduler to behave so that no process will wait more than two time units, and that mutual exclusion is satisfied, that is, either g_1 or g_2 is disabled.

Since we are interested in the behavior of the scheduler, its choices are modeled as angelic. We start with the most liberal scheduler:

$$\begin{array}{lcl} \mathit{Schedule} & = & g_1, g_2 := \mathsf{T}, \mathsf{T} \sqcup g_1, g_2 := \mathsf{T}, \mathsf{F} \sqcup \\ & & g_1, g_2 := \mathsf{F}, \mathsf{T} \sqcup g_1, g_2 := \mathsf{F}, \mathsf{F} \end{array}$$

To model the waiting time we use a variable c_i for each of the processes. c_i ranges over natural numbers.

$$\begin{array}{lcl} \mathit{Process}_i & = & \mathsf{if}\ \neg w_i\ \mathsf{then\ skip} \sqcap w_i := T; c_i := 0\ \mathsf{else} \\ & & \mathsf{if}\ g_i\ \mathsf{then}\ w_i := F\ \mathsf{else}\ c_i := c_i + 1\ \mathsf{fi\ fi} \end{array}$$

The system is described by the following specification:

$$\mathsf{do}\ \mathsf{T} \rightarrow \mathit{Schedule}; \mathit{Process}_1; \mathit{Process}_2\ \mathsf{od}$$

Note that although the processes run in parallel, they can be modeled as running sequentially since they do not share any variables.

The desired behavior for the scheduler is an *invariance property* – it should be preserved by each loop iteration. The property is formalized as:

$$\begin{array}{lcl} \mathit{inv} & = & (w_1 \Rightarrow c_1 < 2) \wedge (w_2 \Rightarrow c_2 < 2) \wedge \\ & & (w_1 \wedge w_2 \Rightarrow \neg(c_1 = 1 \wedge c_2 = 1)) \wedge \neg(g_1 \wedge g_2) \end{array}$$

The requirement that *inv* should be preserved is captured by this correctness assertion, which should be proved true:

$$inv \;\{\![\; Schedule; Process_1; Process_2 \;]\!\}\; inv \tag{47}$$

The intermediate assertion (the precondition of $Process_1; Process_2$ with respect to the postcondition), is easily calculated:

$$\begin{aligned} inter \;=\; & g_1 \wedge \neg g_2 \wedge (w_2 \wedge c_2 = 0 \vee \neg w_2) \vee \neg g_1 \wedge g_2 \wedge (w_1 \wedge c_1 = 0 \vee \neg w_1) \vee \\ & \neg g_1 \wedge \neg g_2 \wedge (w_1 \wedge \neg w_2 \wedge c_1 = 0 \vee \neg w_1 \wedge w_2 \wedge c_2 = 0 \vee \neg w_1 \wedge \neg w_2) \end{aligned}$$

We now prove (47).

$inv \;\{\![\; Schedule; Process_1; Process_2 \;]\!\}\; inv$

$\Leftarrow$ {sequential composition, Rule 16}

$(inv \;\{\![\; Schedule \;]\!\}\; inter) \wedge (inter \;\{\![\; Process_1; Process_2 \;]\!\}\; inv)$

$\Leftarrow$ {focus on second conjunct}

- $\boxed{inter \;\{\![\; Process_1; Process_2 \;]\!\}\; inv}$

  $\equiv$ {*inter* is weakest precondition with respect to *inv*}

  T

$\cdots$ $inv \;\{\![\; Schedule \;]\!\}\; inter$

$\equiv$ {expand *Schedule*}

$inv \;\{\![\; g_1, g_2 := \mathsf{T}, \mathsf{T} \sqcup$

$\quad g_1, g_2 := \mathsf{T}, \mathsf{F} \sqcup g_1, g_2 := \mathsf{F}, \mathsf{T} \sqcup g_1, g_2 := \mathsf{F}, \mathsf{F} \;]\!\}\; inter$

$\Leftarrow$ {angelic choice, Rule 23 (none of the *inter* clauses has $g_1 \wedge g_2$)}

$inv \;\{\![\; g_1, g_2 := \mathsf{T}, \mathsf{F} \sqcup g_1, g_2 := \mathsf{F}, \mathsf{T} \sqcup g_1, g_2 := \mathsf{F}, \mathsf{F} \;]\!\}\; inter$

$\Leftarrow$ {angelic choice, Rule 24, $r = (w_2 \wedge c_2 = 0 \vee \neg w_2)$}

$(inv \wedge (w_2 \wedge c_2 = 0 \vee \neg w_2) \;\{\![\; g_1, g_2 := \mathsf{T}, \mathsf{F} \;]\!\}\; inter) \wedge$

$(inv \wedge \neg(w_2 \wedge c_2 = 0 \vee \neg w_2) \;\{\![\; g_1, g_2 := \mathsf{F}, \mathsf{T} \sqcup g_1, g_2 := \mathsf{F}, \mathsf{F} \;]\!\}\; inter)$

$\Leftarrow$ {focus on first conjunct}

- $\boxed{inv \wedge (w_2 \wedge c_2 = 0 \vee \neg w_2) \;\{\![\; g_1, g_2 := \mathsf{T}, \mathsf{F} \;]\!\}\; inter}$

  $\Leftarrow$ {assignment, Rule 13}

  $(inv \wedge (w_2 \wedge c_2 = 0 \vee \neg w_2)) \Rightarrow (w_2 \wedge c_2 = 0 \vee \neg w_2)$

  $\equiv$ {logic}

  T

$\cdots$ $inv \wedge \neg(w_2 \wedge c_2 = 0 \vee \neg w_2) \;\{\![\; g_1, g_2 := \mathsf{F}, \mathsf{T} \sqcup g_1, g_2 := \mathsf{F}, \mathsf{F} \;]\!\}\; inter$

$\Leftarrow$ {angelic choice, Rule 23}

$\boxed{inv \wedge \neg(w_2 \wedge c_2 = 0 \vee \neg w_2) \;\{\![\; g_1, g_2 := \mathsf{F}, \mathsf{T} \;]\!\}\; inter}$

$\equiv$ {expand *inv*, logic simplification, assignment, Rule 13}
$((\neg w_1 \vee w_1 \wedge c_1 = 0) \wedge w_2 \wedge (c_2 = 1) \wedge \neg(g_1 \wedge g_2)) \Rightarrow$
$(w_1 \wedge c_1 = 0 \vee \neg w_1)$
$\equiv$ {logic}
T

We have proved that, if cooperative, the scheduler has the required behavior. However, the scheduler cannot realistically be expected to be cooperative, so we would also like to implement the angelic part with a deterministic statement. While proving correctness, we have extracted (the framed) facts that will be used in the implementation.

T
$\Rightarrow$ {from the above derivation}
$(inv \wedge (w_2 \wedge c_2 = 0 \vee \neg w_2)\ \{\![\, g_1, g_2 := \mathsf{T}, \mathsf{F} \,]\!\}\ inter) \wedge$
$(inv \wedge \neg(w_2 \wedge c_2 = 0 \vee \neg w_2)\ \{\![\, g_1, g_2 := \mathsf{F}, \mathsf{T} \,]\!\}\ inter) \wedge$
$(inter\ \{\![\, Process_1; Process_2 \,]\!\}\ inv)$
$\Rightarrow$ {focus on the first two conjuncts}
- $(inv \wedge (w_2 \wedge c_2 = 0 \vee \neg w_2)\ \{\![\, g_1, g_2 := \mathsf{T}, \mathsf{F} \,]\!\}\ inter) \wedge$
  $(inv \wedge \neg(w_2 \wedge c_2 = 0 \vee \neg w_2)\ \{\![\, g_1, g_2 := \mathsf{F}, \mathsf{T} \,]\!\}\ inter) \wedge$
  $\Rightarrow$ {conditional, Rule 17}
  $inv\ \{\![\, \mathsf{if}\ (w_2 \wedge c_2 = 0 \vee \neg w_2)\ \mathsf{then}\ g_1, g_2 := \mathsf{T}, \mathsf{F}$
  $\mathsf{else}\ g_1, g_2 := \mathsf{F}, \mathsf{T}\ \mathsf{fi} \,]\!\}\ inter$

$\cdots$ $(inv\ \{\![\, \mathsf{if}\ (w_2 \wedge c_2 = 0 \vee \neg w_2)\ \mathsf{then}\ g_1, g_2 := \mathsf{T}, \mathsf{F}$
$\mathsf{else}\ g_1, g_2 := \mathsf{F}, \mathsf{T}\ \mathsf{fi} \,]\!\}\ inter) \wedge$
$(inter\ \{\![\, Process_1; Process_2 \,]\!\}\ inv)$
$\Rightarrow$ {sequential composition, Rule 16}
$inv\ \{\![$
$\mathsf{if}\ (w_2 \wedge c_2 = 0 \vee \neg w_2)\ \mathsf{then}\ g_1, g_2 := \mathsf{T}, \mathsf{F}\ \mathsf{else}\ g_1, g_2 := \mathsf{F}, \mathsf{T}\ \mathsf{fi};$
$Process_1; Process_2$
$\,]\!\}\ inv$

Now the scheduler is implemented so that the first process is enabled for as long as the second one has not generated a request or waited more than one time unit. This strategy might cause the first process to be enabled unnecessarily – when it has not asked to; however this does not violate the behavior we asked from the scheduler. If we expect such concerns to arise, we should try to avoid eliminating moves that could in some cases conform to

the behavior (unlike we did when eliminating $g_1, g_2 := F, F$ from the moves). For example, since we have proved $inv \{\!|\, Schedule \,|\!\} inter$, we can use Rule 44 (generalized for more than two choices) and get the winning strategy for the scheduler implemented as:

$$\begin{array}{l}
\mathsf{if}\ (w_2 \wedge c_2 = 0 \vee \neg w_2) \rightarrow g_1, g_2 := \mathsf{T}, \mathsf{F} \\
[\!]\ (w_1 \wedge c_1 = 0 \vee \neg w_1) \rightarrow g_1, g_2 := \mathsf{F}, \mathsf{T} \\
[\!]\ (w_1 \wedge \neg w_2 \wedge c_1 = 0 \vee \neg w_1 \wedge w_2 \wedge c_2 = 0 \vee \neg w_1 \wedge \neg w_2) \rightarrow g_1, g_2 := \mathsf{F}, \mathsf{F} \\
\mathsf{fi}
\end{array}$$

Obviously $g_1, g_2 := \mathsf{T}, \mathsf{T}$ does not feature here since its weakest precondition with respect to *inter* is false – the guard for that branch is false. Also note that when both processes are idle, all the guards of the conditional are enabled, so the scheduler is free to choose which of the branches to execute. This implementation corresponds more closely to the algorithmically synthesized version of the scheduler [1], since it leaves all the valid states in the picture.

7 Related Work and Conclusions

The notion of angelic nondeterminism goes back to the theory of nondeterministic automata and Floyd's nondeterministic programs [9]. Floyd talks about nondeterministic programs that have free will rather than being random, and are "in part governed ... by final causes ... for the sake of which their effects are carried out"; moreover, these programs are seen as aiming at "achievement of success and avoidance of failure" [9]. Floyd expresses backtracking algorithms with constructs that do not refer to the details needed for implementing backtracking. Floyd-like constructs can be described using guarded conditionals and miracles [13, 15], but also angelic nondeterminism [20], the subtle difference in the result being that an angelic specification can "look ahead" and avoid nontermination [19].

In a weakest precondition predicate transformer setting, angelic choice was independently introduced by Gardiner and Morgan [10] (as a "conjunction operator"), and Back and von Wright [4]. The algebraic appeal is that angelic nondeterminism is dual to the demonic, and the specification language is complete with both types of nondeterminism in it: every construct in the language is a weakest precondition, and every weakest precondition corresponds to a construct in the language. A direct application of the algebraic properties of angelic constructs is in program inversion, as shown in [17]. Another benefit of having both types of angelic nondeterminism is that the data refinement relation, which is often a different relation from the

normal refinement relation, can be defined in terms of the normal refinement [18].

However, as described in the introduction, it is when modeling interaction that angelic nondeterminism becomes indispensable. Back and von Wright [5] also describe how to extract a winning strategy in the form of a nonangelic program for certain games. These games are loops each iteration of which starts with an angelic statement (a move from our favorite player), and then contains only demonic statements. The given refinement rules then make sure that a nonempty subset of choices that lead to a win become demonic.

Ward [19] has used ideas from the classical refinement calculus to develop a refinement calculus for a functional ("expression") specification language with both angelic and demonic nondeterminism. As in [20], it is shown that angelic constructs from this language can be used to model Floyd-like constructs for backtracking purposes. Ward also talks about restricting angelic choices using information from the context or, in his own words, making the choices more educated. For this purpose he gives a refinement rule, which in our setting would correspond to merging an angelic update with an assertion.

In an Object-Z and CSP context (where system components are specified as Object-Z classes and combined with CSP operators), Smith and Derrick [16] use angelic nondeterminism ("an angelic model of outputs") to model cooperative communication between components. They also show how structural refinements can be used to move from this more abstract view to a more implementation-oriented view, in which the actual agreement mechanism is made explicit.

The common denominator of these works and ours is the realization that angelic nondeterminism abstracts away from the details of how clairvoyance, cooperation, or winning strategy is implemented. Demonic nondeterminism, on the other hand, has no pretensions of cooperation or will to establish postconditions, so it can be seen as standing between angelic nondeterminism and determinism in the abstractness scale.

We have used and extended correctness and refinement techniques from [5] in order to give two systematic methods for transforming programs with angelic nondeterminism into programs with nonangelic constructs. These transformations preserve correctness only with respect to specific postconditions – they are special cases of refinement in context. The main idea behind the structural changes themselves is that choices that establish the postcondition are virtually indistinguishable, so we could as well let the demon resolve the successful part of angelic nondeterminism (or pick one of the choices ourselves).

One of the methods relies on guard propagation for propagating information from the postcondition to the angelic statements to be implemented; the

other on extracting a winning strategy while proving correctness. However, whether it is guard propagation or figuring out an intermediate assertion while proving correctness, it all comes down to calculating weakest preconditions. This suggests that the two approaches not only produce similar results but also require similar efforts. At the same time, this means that precondition calculators (based on theorem provers) such as [7] can be used to automate this kind of refinement. In future work we intend to tailor [7] to perform implementations of angelic nondeterminism automatically.

References

[1] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, and S. Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*. Springer, 1995.

[2] R.-J. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.

[3] R.-J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[4] R.-J. Back and J. von Wright. Duality in specification languages: A lattice theoretical approach. *Acta Informatica*, 27(7):583–625, 1990.

[5] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[6] R.-J. Back and J. von Wright. Contracts, games, and refinement. *Information and Computation*, 157:25–45, 2000.

[7] O. Celiku and J. von Wright. Theorem prover support for precondition and correctness calculation. In *4th International Conference on Formal Engineering Methods*, volume 2495 of *Lecture Notes in Computer Science*, pages 299–310. Springer-Verlag, Oct. 2002.

[8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[9] R. W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, Oct. 1967.

[10] P. H. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[12] L. Laibinis and J. von Wright. Context handling in the refinement calculus framework. Technical Report 118, Turku Centre for Computer Science, Aug. 1997.

[13] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[14] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[15] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[16] G. Smith and J. Derrick. Abstract specification in Object-Z and CSP. In *4th International Conference on Formal Engineering Methods*, volume 2495 of *Lecture Notes in Computer Science*, pages 109–119. Springer-Verlag, Oct. 2002.

[17] J. von Wright. Program inversion in the refinement calculus. *Information Processing Letters*, 37(2):95–100, 1991.

[18] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31(2):105–135, 1994.

[19] N. Ward. *A Refinement Calculus for Nondeterministic Expressions*. PhD thesis, University of Queensland, 1994.

[20] N. Ward and I. J. Hayes. Applications of angelic nondeterminism. In P. A. C. Bailes, editor, *Proc. 6th Australian Software Engineering Conference*, pages 391–404. Sydney, Australia, 1991.

A Appendix

Proof (25)

$$
\begin{array}{ll}
 & (r \cap p) \subseteq S_1.q,\ (\neg r \cap p) \subseteq S_2.q \\
\vdash & p \\
= & \{r \cup \neg r = \mathsf{true}\}
\end{array}
$$

$$
\begin{array}{ll}
 & (r \cup \neg r) \cap p \\
= & \{\text{distributivity}\} \\
 & r \cap p \cup \neg r \cap p \\
\subseteq & \{\text{assumptions}\} \\
 & S_1.q \cup S_2.q
\end{array}
$$

The implication in the other direction follows directly by choosing $r = S_1.q$.

Proof (40)

$$
\begin{array}{ll}
 & \mathsf{disjunctive}\ S \\
\vdash & S;[q] = [S^\circ.q];S \\
\equiv & \{\text{definition of dual}\} \\
 & S;[q] = [\neg S.(\neg q)];S \\
\equiv & \{\text{equality on predicate transformers}\} \\
 & \forall r \bullet (S;[q]).r = ([\neg S.(\neg q)];S).r \\
\equiv & \{\text{definitions}\} \\
 & \forall r \bullet S.(\neg q \cup r) = S.(\neg q) \cup S.r \\
\equiv & \{\text{disjunctivity assumption}\} \\
 & \mathsf{T}
\end{array}
$$

Proof (41)

$$
\begin{array}{ll}
 & \mathsf{monotonic}\ S,\ S^\circ.q \subseteq p \\
\vdash & [p];S;[q] \sqsubseteq S;[q] \\
\equiv & \{\text{refinement definition}\} \\
 & \forall r \bullet ([p];S;[q]).r \subseteq (S;[q]).r \\
\equiv & \{\text{focus}\} \\
 & \quad \bullet\ ([p];S;[q]).r \\
 & \quad = \{\text{definitions}\} \\
 & \quad\ \neg p \cup S.(\neg q \cup r) \\
 & \quad \subseteq \{\text{assumption } S.q \subseteq p\} \\
 & \quad\ \neg S^\circ.q \cup S.(\neg q \cup r) \\
 & \quad = \{\text{definition of dual}\} \\
 & \quad\ S.(\neg q) \cup S.(\neg q \cup r)
\end{array}
$$

$$
\begin{array}{lll}
 & = & \{\text{monotonicity assumption}\} \\
 & & S.(\neg q \cup r) \\
 & = & \{\text{definitions}\} \\
 & & (S;[q]).r \\
\cdots & \mathsf{T} &
\end{array}
$$

The other direction is trivial from $\mathsf{skip} \sqsubseteq [p]$

Proof (43)

$$
\begin{array}{lll}
 & \mathsf{monotonic}\ S & \\
\vdash & p\ \{\![\ S\ ]\!\}\ q \equiv ([p];S;[\neg q] = \mathsf{magic}) & \\
\equiv & \{\text{correctness definition, equality on predicate transformers}\} & \\
 & (p \subseteq S.q) \equiv (\forall r \bullet ([p];S;[\neg q]).r = \mathsf{true}) & \\
\equiv & \{\text{definitions}\} & \\
 & (p \subseteq S.q) \equiv (\forall r \bullet \neg p \cup S.(q \cup r) = \mathsf{true}) & \\
\equiv & \{\text{definitions, logic}\} & \\
 & (p \subseteq S.q) \equiv (\forall r \bullet p \subseteq S.(q \cup r)) & \\
\equiv & \{\text{implication antisymmetric}\} & \\
 & \bullet\ (p \subseteq S.q) \Rightarrow (\forall r \bullet p \subseteq S.(q \cup r)) & \\
 & \equiv\ \{\text{focus on antecedant}\} & \\
 & \quad \bullet\ p \subseteq S.q & \\
 & \quad \equiv\ \{\text{add facts from monotonicity, using } \forall r \bullet q \subseteq q \cup r\} & \\
 & \quad\ (p \subseteq S.q) \wedge (\forall r \bullet S.q \subseteq S.(q \cup r)) & \\
 & \quad \equiv\ \{r \text{ not free in first conjunct}\} & \\
 & \quad\ \forall r \bullet (p \subseteq S.q) \wedge (S.q \subseteq S.(q \cup r)) & \\
 & \quad \Rightarrow\ \{\text{order on predicates transitive}\} & \\
 & \quad\ \forall r \bullet p \subseteq S.(q \cup r) & \\
 & \cdots\ \mathsf{T} & \\
 & \bullet\ (\forall r \bullet p \subseteq S.(q \cup r)) \Rightarrow (p \subseteq S.q) & \\
 & \equiv\ \{\text{focus on antecedant}\} & \\
 & \quad \bullet\ \forall r \bullet p \subseteq S.(q \cup r) & \\
 & \quad \Rightarrow\ \{\text{specialize for } q\} & \\
 & \quad\ p \subseteq S.q & \\
 & \cdots\ \mathsf{T} & \\
\cdots & \mathsf{T} &
\end{array}
$$

Proof (44)
From Theorem 43, the left-hand side is equal to magic; we show the right hand side is magic by proving correctness of the guarded conditional.

$$
\begin{array}{ll}
 & p \subseteq (S_1 \sqcup S_2).q \\
\vdash & p \subseteq (\mathsf{if}\ S_1.q \rightarrow S_1 \ [\!]\ S_2.q \rightarrow S_2\ \mathsf{fi}).q \\
= & \{\text{guarded conditional definition 45}\} \\
 & p \subseteq (\{S_1.q \cup S_2.q\}; ([S_1.q]; S_1 \sqcap [S_2.q]; S_2)).q \\
= & \{\text{definitions}\} \\
 & p \subseteq ((S_1.q \cup S_2.q) \cap (\neg S_1.q \cup S_1.q) \cap (\neg S_2.q \cup S_2.q)) \\
= & \{\text{simplification, definition of angelic choice}\} \\
 & p \subseteq (S_1 \sqcup S_2).q \\
= & \{\text{assumption}\} \\
 & \mathsf{T}
\end{array}
$$

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi

University of Turku
- **Department of Information Technology**
- **Department of Mathematics**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**