
ROPAS

Research On Program Analysis System
National Creative Research Initiative Center
Korea Advanced Institute of Science and Technology

ROPAS MEMO
2000-7
October 4, 2000

Program Logics Made Easy

Nikolay V. Shilov and Kwangkeun Yi
Department of Computer Science
KAIST
{shilov, kwang}@ropas.kaist.ac.kr

October 4, 2000

Abstract

In spite of the importance of Formal Methods for development of a reliable hard- and software this domain is not well acquainted to non-professionals. In particular, many students consider Formal Methods either too poor for their pure mathematics, either too pure for their poor mathematics. We suppose that a deficit of a popular papers on Formal Methods is the main reason for this ignorance. In the paper we would like to present in a popular (but mathematically sound) form a Program Logics tributary creek of a powerful stream called Formal Methods. From the application viewpoint, the paper is oriented on *model checking* of program logics in finite models. The basic ideas, definitions and theorems are illustrated by game examples usually presented as puzzles. Only some knowledge of propositional calculus, elementary set theory and theory of binary relations is prerequested.

1 Story Began

1.1 A Hard Puzzle

Once upon a time a program committee for a regional middle school contest on mathematics discussed problems for a forthcoming competition. The committee consists of a professor, several holders of Ph.D. degree and a couple of Ph.D. students. All were experienced participants or organizers of mathematical contests on the regional and national level, several had successful international experience. A thunder-storm was unexpected, but it came. Suddenly (when a set of problems was just complete) one of the youngest suggested another problem to be included. It was the following puzzle about a false coin among valid ones:

> A set of coins consists of 14 valid and 1 false coin. All valid coins have one and the same weight while the false coin has a different weight. One of the valid coins is marked while all other coins (including the false one) are unmarked (i.e. it is known that the unique marked coin is valid). It is required to identify the false coin with use of a balance not more then 3 times.

"If it is known that the false coin is heavier then the valid one then the problem is relevant for 11-13 years old school-children" the professor said[1]. "Yes, sir!" immediately replied one of Doctors of Philosophy and then suggested: "If it is known that the false coin is lighter then the

[0]This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.
[1]Professors are for making *absolutely correct* while *absolutely unuseful* remarks.

valid one then the puzzle is relevant for children also!"[2] "Sorry, but it is not known whether the false coin is heavier or lighter then the valid one" interrupted them the youngster and added: "To the best of my knowledge it is really a hard problem. I did not solve it during the national contest several years ago and I do not know a solution till now". "In this case it would be better not to include the puzzle into the problem list for the forthcoming contest" the professor summarized.

Problem 1 *Solve the original puzzle.*

One of the committee members was a computer scientist specializing in program logics and their applications. He agreed to exclude the puzzle from the problem list, since he did not solve it also. Nevertheless he was concerned and decided to try his luck. The day passed without real progress while the coins and balance became his nightmare... Next morning the computer scientist decided to overcome the trouble and adopted the the following plan with two concurrent approaches for finding a solution:

- human-aided,
- computer-aided.

The computer-aided approach and its relation to program logics are topics of next sections of the paper. As far as concerns the human-oriented approach, then it was designed very simple but had unexpected implications. The puzzle was offered (with a special bonus for the first solution[3]) for students and faculty of the Mathematics Department. But the first who solved the puzzle was the wife of the computer scientist[4]. She spent approximately 3 hours for it tonight. A week later the computer scientist got several correct (and very similar) solutions from students and found that a ratio of a number of students which succeeded to a number of students who tried to solve is 1/10, and all lucky students spent around 3 hours also solving the puzzle. Then several weeks later 2 professors solved the puzzle correctly also...

The University where the computer scientist is employed is situated in a cozy scientific town[5] not in a political, neither an industrial nor a financial center. So there was no surprise that several months later a street retail book dealer began to offer a special deal for textbooks: if a customer can find in one hour a strategy how to identify a unique false among 39 coins with using a marked valid coin[6] and a balance 4 times at most, then the customer get his money back; if the customer can find a strategy in one day then he get 50% of his money back.

Problem 2 *Solve the retailer puzzle.*

1.2 Put It for Programming

Let us turn to computer-aided approach and realize how both puzzles can be put for programming. In both cases the problem is not to find a false coin but to find *a strategy* how to identify this coin. A natural model for a strategy is a program which choice the next step with respect to the information available after previous steps. The difference between puzzles is

- a number of coins N under question,
- a limit of balancing K.

[2] Do you guess that the doctor was a research fellow of the professor?
[3] The bonus was a photocopy of 100$.
[4] So the bonus left at home.
[5] Can you guess the name of the town and where is it?
[6] Valid coins have one and the same weight while the false coin has a different weight.

In both puzzles a marked valid coin is given but in principle it is possible to consider

- a number M of marked valid coins

as another variable.

After above preliminaries let us to consider the following programming problem as a natural formalization of the puzzles:

• Write a program with 3 inputs

- a number of coins N under question,
- a number M of marked valid coins,
- a limit of balancing K

which outputs either `impossible` either another executable interactive program `ALPHA` (in the same language) with respect to existence of a strategy to identify a unique false coin among N coins with use of M marked valid coins and balancing K times at most. Your initial program should output `impossible` iff there is no such strategy. Otherwise it should output the program `ALPHA` which implement a strategy in the following settings.

All $(N + M)$ coins are enumerated by consequent numbers from 1 to $(N + M)$, all marked valid coins are enumerated by initial numbers from 1 up to M and enumeration is fixed[7].

Each interactive session with `ALPHA` begins from user choice of a number of the false coin and weather it is lighter or heavier. The user can choose any number between $(M + 1)$ and $(N + M)$ and must fix the number in the mind till the end of the session.

Then the session consists of a series of rounds and general amount of rounds in the session can not exceed K. On each round i $(1 \leq i \leq K)$ the program `ALPHA` output two disjoint subsets of numbers of coins to be placed on the left and the right pans of the balance and the request `?` for user reply. The user in his/her turn replies by $<$, $=$ or $>$ in accordance with his initial choice of the number of the false coin and its weight.

The session finishes with the the final output of `ALPHA` – `false coin number is` – followed by the number of the false coin.

Here the programming problem is over. Since the problem is to write a program which produce another program then we would like to refer to the first program as the *metaprogram* and to the problem as the *metaprogram problem* respectively. For the first time the problem was designed and offered by the computer scientist for training university students for 1999 regional ACM Collegiate Programming Contest [1].

Problem 3 *Solve the metaprogram problem.*

Let us illustrate the metaprogram problem[8] by examples of inputs and outputs presented in PASCAL. A triple 5, 1 and 2 is an example of possible inputs for a metaprogram. The semantics of this particular input is a request for a strategy which can identify a unique false coin among five coins $(N = 5)$ under question with use of a special marked additional valid coin $(M = 1)$ and balancing coins at most twice $(K = 2)$. All $6 = (N + M)$ coins are enumerated by consequent numbers from 1 to 6, and the unique marked valid coin got the number 1. The following PASCAL program `ALPHA` (presented on **Fig. 1**) is a possible corresponding output of the metaprogram. Below is a summary of four sessions with this program `ALPHA`:

[7] $M = 0$ is acceptable also.

[8] In accordance with the style of the ACM contests.

```
program ALPHA
var R: '<','=','>';
begin
writeln(1,2); writeln(3,4); writeln('?'); readln(R);
if R='='
   then
     begin
     writeln(1); writeln(5); writeln('?'); readln(R);
     if R='='
        then writeln('false coin number is 6')
        else writeln('false coin number is 5')
     end
   else
     if R='<'
        then
          begin
          writeln(3); writeln(4); writeln('?'); readln(R);
          if R='='
             then writeln('false coin number is 2')
             else if R='<'
                     then writeln('false coin number is 4')
                     else writeln('false coin number is 3')
          end
        else
          begin
          writeln(3); writeln(4); writeln('?'); readln(R);
          if R='='
             then writeln('false coin number is 2')
             else if R='<'
                     then writeln('false coin number is 3')
                     else writeln('false coin number is 4')
          end
end.
```

Figure 1: A PASCAL program which identifies a unique false coin among five coins with aid of a special marked valid coin and balancing coins at most twice.

user:	2^{nd} is heavier	3^{rd} is lighter	4^{th} is heavier	5^{th} is lighter
prog:	`{1,2} {3,4}`	`{1,2} {3,4}`	`{1,2} {3,4}`	`{1,2} {3,4}`
user:	`>`	`>`	`<`	`=`
prog:	`{3} {4}`	`{3} {4}`	`{3} {4}`	`{1} {5}`
user:	`=`	`<`	`<`	`>`
prog:	`2`	`3`	`4`	`5`

9, 1 and 2 is another example of possible inputs for a metaprogram. The semantics of this input is a request for a strategy which can identify a unique false coin among nine coins under question with use of a special marked additional valid coin and balancing coins at most twice. The output of the metaprogram for this particular input is `impossible`.

2 Games with Dynamic Logic

2.1 Game Interpretation

The above examples of sessions naturally lead to a game interpretation for both puzzles and metaprogram problem.

• Let M and N be non-negative integer parameters and let $(N + M)$ coins be enumerated by consequent numbers from 1 to $(N + M)$. Coins with numbers in $[1..M]$ are valid while there is a unique false among coins with numbers in $[(M + 1)..(M + N)]$. The *GAME(N,M)* of two players *user* and *prog* consists of a series of rounds. On each round a move of the *prog* is a pair of disjoint subsets (with equal cardinalities) of $[1..(M + N)]$. A possible move of the *user* is either $<$, $=$ or $>$, but his reply must be *consistent* with all previous rounds in the following sense: some number in $[1..(M + N)]$ and weight of a unique false coin satisfy all constraints induced on the current and previous rounds. The *prog* wins the *GAME(N,M)* as soon as a *unique* number in $[1..(M + N)]$ satisfies all constraints induced during the game.

Now problems 1-3 can be reformulated as follows.

1. Find a 3-rounds at most winning strategy for *prog* in the *GAME(14,1)*.
2. Find a 4-rounds at most winning strategy for *prog* in the *GAME(39,1)*.
3. Write a metaprogram which for all $N \geq 1$, $K \geq 0$ and $M \geq 0$ generates (iff possible) a K-rounds at most wining strategy for *prog* in the *GAME(N,M)*.

The above game interpretation is still too complicated for analysis[9]. So let us introduce and analyze another simpler game first.

•On the eve of New 2000 Year Alice and Bob played the *millennium game*. Positions in the game were dates of 2000 and 2001 years. The *initial position* was a random date of the year 2000. Then Alice and Bob made moves in their turn: Alice, Bob, Alice, Bob, etc. Available moves were one and the same for both Alice and Bob: if a current position is a *date* then *the next calendar date* and *the same day of the next month* are possible next positions. A player won the game iff his/her counterpart was the first who launched the year 2001.

Problem 4 *Find sets of all initial positions where (a)Alice and, respectively, (b)Bob had winning strategy. What is the union of these sets?*

A mathematical model for the millennium game is quite obvious. It is a special labeled graph $G_{2000/2001}$. Nodes of this graph correspond to game positions — dates of years 2000

[9]In particular, it is not clear what are game positions.

and 2001. All dates of the year 2001 are marked by *fail* while all other dates are unmarked. Edges of the graph correspond to possible moves and are marked by *move*. We would like to consider *fail* and *move* as variables for collections of nodes and sets of edges and call them propositional and action variables respectively. The model fixed interpretation (i.e. values) of these variables in the manner described above.

2.2 Elementary Propositional Dynamic Logic

Let $\{true, false\}$ be boolean constants, Prp and Act be disjoint finite alphabets of *propositional* and *action* variable respectively. (In the previous section they are $\{fail\}$ and $\{move\}$.)

The syntax of the classical propositional logic consists of *formulae* and is constructed from propositional variables and boolean connectives $\neg$ (*negation*), $\wedge$ (*conjunction*) and $\vee$ (*disjunction*) in accordance to standard rules:

1. all propositional variables and boolean constants are formulae;
2. if ϕ is a formula then $(\neg\phi)$ is a formula;
3. if ϕ and ψ are formulae then $(\phi \wedge \psi)$ a formula,
4. if ϕ and ψ are formulae then $(\phi \vee \psi)$ a formula.

Elementary Propositional Dynamic Logic (EPDL)[2] has additional features for constructing formulae — modalities which are associated with action variables:

5. if a is an action variable and ϕ is a formula then $([a]\phi)$ is a formula[10],
6. if a is an action variable and ϕ is a formula then $(< a > \phi)$ is a formula[11].

We would like to use several standard abbreviations $\rightarrow$ and $\leftrightarrow$ in the usual manner: if ϕ and ψ are formulae then $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$ are abbreviations for formulae $(\neg\phi) \vee \psi)$ and $((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi))$ respectively. Then we would like to avoid extra parenthesis and use a standard priorities for connectives and modalities: $\neg$, $<>$, $[\,]$, $\wedge$, $\vee$, $\rightarrow, \leftrightarrow$. We also would like to use a *meta-symbol* $\equiv$ for *syntactical equality*.

Problem 5 *Are formulae of EPDL a context-free language?*

The semantics of EPDL is defined in models, which are called *Labeled Transition Systems* by computer scientists and *Kripke Structures* by mathematicians. A model M is a pair (D_M, I_M) where the *domain* D_M is a nonempty set, while the *interpretation* I_M is a pair of special mappings (P_M, R_M). Elements of the domain D_M are called *states*. The interpretation maps propositional variables into sets of states and action variables into binary relations on states:

$$P_M : Prp \rightarrow \mathcal{P}(D_M) \quad , \quad R_M : Act \rightarrow \mathcal{P}(D_M \times D_M)$$

where $\mathcal{P}$ is a power-set operation. We write $I_M(p)$ and $I_M(a)$ instead of $P_M(p)$ and $R_M(a)$ frequently. Whenever it is implicit that p and a are propositional and action variables respectively.

Models can be considered as labeled graphs with nodes and edges marked by sets of propositional and action variables respectively. For a model $M = (D_M, I_M)$ nodes of the corresponding graph are states of D_M. In this graph a node $s \in D_M$ is marked by a propositional variable

[10] which is read as "*box a* ϕ" or "*after a always* ϕ"
[11] which is read as "*diamond a* ϕ" or "*after a sometimes* ϕ"

$p \in Prp$ iff $s \in I_M(p)$. A pair of nodes $(s_1, s_2) \in D_M \times D_M$ is an edge of the graph iff $(s_1, s_2) \in I_M(a)$ for some action variable $a \in Act$; in the last case the edge (s_1, s_2) is marked by this action variable a. Vice versa, labeled graphs with nodes and edges marked by sets of propositional and action variables respectively can be considered as models also. In this setting the graph $G_{2000/2001}$ of the millennium game is really a model for EPDL, i.e. Labeled Transition System or Kripke Structure if you like.

For every model $M = (D_M, I_M)$ the *validity* relation $\models_M$ between states and formulae can be defined inductively with respect to the structure of formulae:

1. for every state $s \models_M true$ and not $s \models_M false$;
   for all state s and propositional variables p: $s \models_M p$ iff $s \in I_M(p)$;
2. for all state s and formula ϕ: $s \models_M (\neg\phi)$ iff it is not the case $s \models_M \phi$;
3. for all state s, formulae ϕ and ψ: $s \models_M (\phi \wedge \psi)$ iff $s \models_M \phi$ and $s \models_M \psi$;
4. for all state s, formulae ϕ and ψ: $s \models_M (\phi \vee \psi)$ iff $s \models_M \phi$ or $s \models_M \psi$;
5. for all state s, action variable a and formulae ϕ:
   $s \models_M (< a > \phi)$ iff $(s, s') \in I_M(a)$ and $s' \models_M \phi$ for same state s' ;
6. for all state s, action variable a and formulae ϕ:
   $s \models_M ([a]\phi)$ iff $(s, s') \in I_M(a)$ implies $s' \models_M \phi$ for every state s'.

Moreover, an experienced mathematician can remark that EPDL is just a polymodal variant the classical and basic modal logic **K** [13]

2.3 Finite Games in EPDL

First let us illustrate the above definition by several examples in the model $G_{2000/2001}$. The formula $fail$ is valid in those states where the game is lost. Then the formula $[move]fail$ is valid in those states from which all possible moves lead to the lost game. Hence the formula $\neg fail \wedge [move]fail$ is valid in the states where the game is not over but all possible moves lead to the lost game. Consequently, the formula $< move > (\neg fail \wedge [move]fail)$ is valid iff there is a move after which the game is not lost while then all possible moves always lead to the lost game. Finally we get: the formula

$$\neg fail \wedge < move > (\neg fail \wedge [move]fail)$$

is valid in those states where the game is not over, where exists a move after which the game is not lost while then all possible moves always lead to the lost game. So the last EPDL formula is valid in those states of $G_{2000/2001}$ (i.e. dates of years 2000 and 2001) where Alice has a 1-round wining strategy where Bob loses the game[12]. So it is natural to denote this formula by win_1. A 1-round wining strategy for Alice is just a move to a position where Bob always loses the game after his move, i.e. to a position where the formula $\neg fail \wedge [move]fail$ is valid. It becomes quite clear from the above arguments that the following formula

$$\neg fail \wedge < move > \big(\neg fail \wedge [move](fail \vee win_1)\big)$$

is valid in those states of $G_{2000/2001}$ where Alice has a wining strategy with 2-rounds at most. So it is natural to denote this formula by win_2. A 2-round at most wining strategy for Alice is

[12] Alice has all odd moves while Bob has all even moves.

just a move to a position where the formula $\neg fail \wedge [move](fail \vee win_1)$ is valid. Let us define formulae win_i for all $i \geq 1$ similarly to win_1 and win_2: for every $i \geq 1$ let win_{i+1} be

$$\neg fail \wedge < move > \big(\neg fail \wedge [move](fail \vee win_i)\big).$$

Let win_0 be $false$ in addition. After the above discussion about win_1 and win_2 it becomes quite simple to prove by induction the following assertion.

Assertion 1 *For every $i \geq 1$ the formula win_i is valid in those states of $G_{2000/2001}$ where Alice has a wining strategy with i-rounds at most. A first step of a corresponding i-rounds at most wining strategy for Alice is just a move to a position where the formula $\neg fail \wedge [move](fail \vee win_{i-1})$ is valid.*

Then let us introduce a new propositional variable win and interpret it to be valid in those dates of years 2000 and 2001 where Alice has a wining strategy, i.e. as $\bigcup_{i\geq 1}\{\ s\ :\ s \models_{G_{2000/2001}} win_i\ \}$. Let us refer to this new model a *special extended model* $G_{2000/2001}$.

Assertion 2 *The formula $win \leftrightarrow \neg fail \wedge < move > \big(\neg fail \wedge [move](fail \vee win)\big)$ is valid in all states of the special extended model $G_{2000/2001}$.*

In general, a *finite game of two plays* A and B is tuple $(P,\ M_A,\ M_B,\ F)$ where

- P is a nonempty finite set of *positions*,
- $M_A, M_B \subseteq P \times P$ are *(possible) moves* of A and B,
- $F \subseteq P$ is a set of *final positions*.

A *session* of the game is a finite sequence of positions $s_0,\ ...\ s_n$ $(n > 0)$ where all even pairs are moves of one player (ex., all $(s_{2i},\ s_{2i+1}) \in M_A$) while all odd pairs are moves of another player (ex., all $(s_{2i+1},\ s_{2i+2}) \in M_B$). A pair of consequentive moves of two players in a session comprises three consequentive positions (ex., $(s_{2i}$,s_{2i+1},$s_{2(i+1)})$ is called a *round*. A player *loses* a session iff after a move of the player the session enters a final position for the first time. A player *wins* a session iff another player loses the session. A *strategy* of a player is a subset of the player's possible moves. A *winning strategy* for a player is a strategy of the player which always leads to the player's win: the player wins every session which he/she begins and in which he/she implement this strategy instead of all possible moves. The millennium game is just an example of a finite game.

Finite games of two players can be presented as labeled graphs easily. Nodes correspond to game positions, all nodes which correspond to final positions are marked by $fail$ while all other nodes are unmarked. Edges of these graphs correspond to possible moves of players and are marked by $move_A$ and $move_B$ respectively. Let us denote by $G_{(P,M_A,M_B,F)}$ the labeled graph corresponding to a game (P, M_A, M_B, F). We would like to consider $fail$, $move_A$ and $move_B$ as a propositional and action variables respectively. In this setting the graphs of finite games become models for formulae of EPDL. The following proposition is just a generalization of the above assertions 1 and 2.

Proposition 1 *Let (P, M_A, M_B, F) be a finite game of two players, a formula WIN_0 be $false$ and for every $i \geq 1$ let WIN_{i+1} be a formula $\neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee WIN_i)\big)$. Let win be a new propositional variable and let extend the model $G_{(P,M_A,M_B,F)}$ by interpretation of win as $\bigcup_{i\geq 1}\{s : s \models_{G_{(P,M_A,M_B,F)}} WIN_i\}$.*

- *For every $i \geq 0$ the formula WIN_i is valid in those states of $G_{(P,M_A,M_B,F)}$ where the player A has a wining strategy with i-rounds at most.*

- *For every $i \geq 0$ a first step of a corresponding i-rounds at most wining strategy for the player A consists in a move to a position where $\neg fail \wedge [move_B](fail \vee WIN_{i-1})$ is valid.*
- *The formula $win \leftrightarrow \neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee win)\big)$ is valid in all states of the extended model $G_{(P,M_A,M_B,F)}$.*

2.4 Puzzle Corner

Now we are ready to discuss a mathematical *concrete model* for the *GAME(N,M)* ($N \geq 1$, $M \geq 0$). Positions in this parameterized game are tuples (U, L, H, V, Q) where

- U is a set of coin numbers in $[(M+1)..(M+N)]$ which are under question but which *were not* tested against other coins;
- L is a set of coin numbers in $[(M+1)..(M+N)]$ which are under question but which *were* tested against other coins and turned to be *lighter*;
- H is a set of coin numbers in $[(M+1)..(M+N)]$ which are under question but which *were* tested against other coins and turned to be *heavier*;
- V is a set of coin numbers in $[1..(N+M)]$ which are known to be *valid*;
- Q is a *balancing query*, i.e. a pair of disjoint subsets of $[1..(N+M)]$ of equal cardinality.

Three constraints are natural:

1. U, L, H and V are disjoint,
2. $U \cup L \cup H \cup V \;=\; [1..(N+M)]$,
3. $U \;\cup L \;\cup H \;\neq\; \emptyset$.

In addition we can claim that

4. $U \neq \emptyset$ iff $L \cup H \;=\; \emptyset$

since a unique false is among untested coins iff all previous balancings gave equal weights, and

5. if $Q = (S_1, S_2)$ then either $S_1 \cap V = \emptyset$ either $S_2 \cap V = \emptyset$

since it is not reasonable to add extra valid coins on both pans of a balance. A possible move of the player *prog* is a query for balancing two sets of coins, i.e. pair of positions

$$\begin{array}{c}(U,\ L,\ H,\ V,\ (\emptyset,\emptyset)\\ \big|\\ move_{prog}\\ \downarrow\\ (U,\ L,\ H,\ V,\ (S_1,S_2))\end{array}$$

where S_1 and S_2 are disjoint subsets of $[1..(N+M)]$ with equal cardinalities. A possible move of the player *user* is a reply $<$, $=$ or $>$ for a query which causes position change

$$\begin{array}{c}(U,\ L,\ H,\ V,\ (S_1,S_2))\\ \big|\\ move_{user}\\ \downarrow\\ (U',\ L',\ H',\ V',\ (\emptyset,\emptyset)).\end{array}$$

in accordance with the query and a reply: if $S_1 = U_1 \cup L_1 \cup H_1 \cup V_1$ and $S_2 = U_2 \cup L_2 \cup H_2 \cup V_2$ respectively with $U_1, U_2 \subseteq U$, $L_1, L_2 \subseteq L$, $H_1, H_2 \subseteq H$, $V_1, V_2 \subseteq V$, then

$$U' = \begin{cases} \emptyset \text{ if the reply is } <, \\ (U \setminus (U_1 \cup U_2)) \text{ if the reply is } =, \\ \emptyset \text{ if the reply is } >, \end{cases}$$

$$L' = \begin{cases} (L_1 \cup U_1) \text{ if the reply is } <, \\ (L \setminus (L_1 \cup L_2)) \text{ if the reply is } =, \\ (L_2 \cup U_2) \text{ if the reply is } >, \end{cases}$$

$$H' = \begin{cases} (H_2 \cup U_2) \text{ if the reply is } <, \\ (H \setminus (H_1 \cup H_2)) \text{ if the reply is } =, \\ (H_1 \cup U_1) \text{ if the reply is } >, \end{cases}$$

$$V' = [1..(N+M)] \setminus (U' \cup L' \cup H').$$

A final position is a position $(U,\ L,\ H,\ V,\ (\emptyset, \emptyset))$ where $|U| + |L| + |H| = 1$.

We suppose that positions, moves of the player *prog* and final positions are modeled in an obvious way and additional comments are not required while some auxiliary intuition on moves of the player *user* are essential. Since $U \neq \emptyset$ iff $L \cup H = \emptyset$ then there are two disjoint cases: $U = \emptyset$ XOR $L \cup H = \emptyset$. Let us consider the first one only since the second is similar. In this case $U_1 = U_2 = U' = \emptyset$. Then

$$L' = \begin{cases} L_1 \text{ if the reply is } <, \text{ since in this case a false coin is} \\ \quad \text{either in } L_1 \text{ and is lighter either it is in } H_2 \text{ and is heavier;} \\ (L \setminus (L_1 \cup L_2)) \text{ if the reply is } =, \text{ since in this case a false coin is} \\ \quad \text{neither in } L_1 \text{ or } L_2 \text{ neither it is in } H_1 \text{ or } H_2; \\ L_2 \text{ if the reply is } >, \text{ since in this case a false coin is} \\ \quad \text{either in } L_2 \text{ and is lighter either it is in } H_1 \text{ and is heavier;} \end{cases}$$

$$H' = \begin{cases} H_2 \text{ if the reply is } <, \text{ since in this case a false coin is} \\ \quad \text{either in } L_1 \text{ and is lighter either it is in } H_2 \text{ and is heavier;} \\ (H \setminus (H_1 \cup H_2)) \text{ if the reply is } =, \text{ since in this case a false coin is} \\ \quad \text{neither in } L_1 \text{ or } L_2 \text{ neither it is in } H_1 \text{ or } H_2; \\ H_1 \text{ if the reply is } >, \text{ since in this case a false coin is} \\ \quad \text{either in } L_2 \text{ and is lighter either it is in } H_1 \text{ and is heavier.} \end{cases}$$

The above model is quite good from mathematical viewpoint but too large from viewpoint of the computer scientist since amounts of possible positions and possible moves is an exponential function of N. Really, for all possible U, L and H the number of enable queries ranges from 1 (at least one query $(\emptyset, \emptyset)$ is enable) up to at least

$$Q(N) = \sum_{i=0}^{i=[\frac{N}{2}]} (C_i^N \times C_i^{N-i})$$

where all C_k^l are binomial coefficients. Since there are 2^N possible values for U and 3^N possible values for disjoint L and H, then the total amount of positions is

$$(2^N + 3^N) \leq P(N) \leq (2^N + 3^N) \times Q(N).$$

The amount of possible moves of the player *prog* is $P(N)$ too, while the amount of possible moves of the player *user* is bounded by the same number and triple $P(N)$. So the total amount of possible moves is

$$2 \times P(N) \leq M(N) \leq 4 \times P(N).$$

In particular, a concrete model $GAME(14,1)$ for the *GAME(14,1)* (i.e. the original puzzle) is on the edge of abilities of modern personal computers.

3 Model Checking and Abstraction

3.1 Model Checking

Model checking is a testing a model against a formula. Let us be more concrete. Models for logics which are under consideration in the paper are Labeled Transitions Systems or Kripke Structures where states of a system are presented as nodes of a graph but transitions between states are presented as labeled edges. The *global (model) checking* problem consists in a calculation of *the set of all states* of a model where a formula is valid, while the *local (model) checking* consists in testing the validity of a formula in a state of a model. Thus the corresponding model checking algorithms as well as their implementations (called *model checkers*) can be characterized by there inputs and outputs as follows:

global checking

inputs: a model and a formula;

output: a set of all states of the model where the formula is valid;

local checking

inputs: a model, a formula, and a state;

output: a boolean value of the formula in the state of the model.

We are especially interested in model checking problem for finite models, i.e. models with finite domains. For these models both model checking problems are algorithmically equivalent:

- for global checking just check locally all states and then collect states where a formula is valid,
- for local checking just check globally and check whether a state is in the validity set of a formula.

Of course, the above reduction of global checking to local one leads to changes of time complexity: the global checking complexity is less then or equal to the local checking complexity multiplied by amount of states. We would like to concentrate on global model checking only since this complexity difference is not important for logics presented in the paper. More important topic are parameters used for measuring this complexity. If $M = (D_M, (R_M, P_M))$ is a finite model then let d_M, r_M and m_M be amount of states in D_M, amount of edges in R_M and $(d_M + r_M)$ respectively. If a model M is implicit then we would like to use these parameters without subscripts, i.e. just d, r and m. If ϕ is a formula then let f_ϕ be amount of variables, connectives and modalities instances in ϕ. If a formula ϕ is implicit then we would like to use this parameter f_ϕ without subscript, i.e. just f.

Proposition 2 *Model checking problem of EPDL formulae in finite models is decidable with time complexity* $O(m \times f)$

As follows from the **Proposition 1**, if identify players A and B with *prog* and *user* then model checking of the formulae WIN_3 and WIN_4 in models $GAME(14,1)$ and $GAME(39,1)$ answers whether is it always possible to identify the *number* of a unique false coin among

coins with numbers [2..15] and [2..40] balancing 3 and 4 times at most respectively. In both case it is enough to check whether these formulae are valid in "initial" positions of these models $([2..15], \emptyset, \emptyset, \{1\})$ and $([2..40], \emptyset, \emptyset, \{1\})$. As far as concern corresponding identification strategies, it is sufficient to model check formulae $\neg fail \wedge [move_B] fail$, $\neg fail \wedge [move_B](fail \vee WIN_1)$, $\neg fail \wedge [move](fail \vee WIN_2)$ in both $GAME(14, 1)$ and $GAME(39, 1)$ and then the formula $\neg fail \wedge [move](fail \vee WIN_2)$ in $GAME(39, 1)$ only. A complexity of the model checker (**Proposition 2**) is linear on the size of a model and the size of a formula, so "Plug and play"! — Sorry, the first model is too large for modern computer while the second is just a huge! So a problem how to put both puzzles for programming remains...

3.2 Abstraction in Puzzle Corner

A hint how to put both puzzles for programming and solve the metaprogram problem is quite easy: to consider *amounts of coins* instead of *coin numbers*. This idea is natural: when somebody is solving puzzles he/she operates in terms of amounts of coins of different kinds[13] not in terms of their numbers!

Now we are ready to discuss a new mathematical *abstract model game(N,M)* for the *GAME(N,M)* ($N \geq 1$, $M \geq 0$). Positions in this parameterized game are tuples (u, l, h, v, q) where

- u is an amount of coins in $[1..N]$ which are under question but which *were not* tested against other coins;
- l is an amount of coins in $[1..N]$ which are under question but which *were* tested against other coins and turned to be *lighter*;
- h is an amount of coins in $[1..N]$ which are under question but which *were* tested against other coins and turned to be *heavier*;
- v is an amount of coins in $[1..(N + M)]$ which are known to be *valid*;
- q is a *balancing query*, i.e. a pair of quadruples $((u_1, l_1, h_1, v_1), (u_2, l_2, h_2, v_2))$ of numbers of $[1..(N + M)]$.

Five constraints are closely relate to constraints 1-5 for the $GAME(N, M)$:

1. $u + l + h \leq N$,
2. $u + l + h + v = N + M$,
3. $u + l + h \geq 1$,
4. $u \neq 0$ iff $l + h = 0$,
5. $v_1 = 0$ or $v_2 = 0$.

Additional but natural constraints should be imposed for queries (since we can borrow coins for weighing from available untested, lighter, heavier and valid ones):

6. $u_1 + u_2 \leq u$,
7. $l_1 + l_2 \leq l$,
8. $h_1 + h_2 \leq h$,

[13] A coin can be *untested*, *lighter*, *heavier* or *valid*.

9. $v_1 + v_2 \leq v$,
10. $u_1 + l_1 + h_1 + v_1 \ = \ u_2 + l_2 + h_2 + v_2$.

A possible move of the player *prog* is a query for balancing two sets of coins, i.e. pair of positions

$$\begin{array}{c}(u,\ l,\ h,\ v,\ ((0,0,0,0),(0,0,0,0))) \\ \Big| \\ move_{prog} \\ \downarrow \\ (u,\ l,\ h,\ v,\ ((u_1,l_1,h_1,v_1),(u_2,l_2,h_2,v_2))).\end{array}$$

A possible move of the player *user* is a reply $<$, $=$ or $>$ for a query which causes position change

$$\begin{array}{c}(u,\ l,\ h,\ v,\ ((u_1,l_1,h_1,v_1),(u_2,l_2,h_2,v_2))) \\ \Big| \\ move_{user} \\ \downarrow \\ (u',\ l',\ h',\ v',\ ((0,0,0,0),(0,0,0,0))).\end{array}$$

in accordance with the query and a reply:

$$u' \ = \ \begin{cases} 0 \text{ if the reply is } < , \\ (u - (u_1 + u_2)) \text{ if the reply is } = , \\ 0 \text{ if the reply is } > , \end{cases}$$

$$l' \ = \ \begin{cases} (l_1 + u_1) \text{ if the reply is } < , \\ (l - (l_1 + l_2)) \text{ if the reply is } = , \\ (l_2 + u_2) \text{ if the reply is } > , \end{cases}$$

$$h' \ = \ \begin{cases} (h_2 + u_2) \text{ if the reply is } < , \\ (h - (h_1 + h_2)) \text{ if the reply is } = , \\ (h_1 + u_1) \text{ if the reply is } > , \end{cases}$$

$$v' \ = \ ((N + M) - (u' + l' + h')).$$

A final position is a position $(u,\ u,\ h,\ v,\ ((0,0,0,0),(0,0,0,0)))$ where $u + l + h = 1$. Thus a mew model $game(N, M)$ is over.

How big/small is the abstract model? For every possible u $(0 \leq u \leq N)$ there are

$$\sum_{i=0}^{i=u}(u - i) \ = \ \frac{(u+1) \times u}{2}$$

enable queries (u_1, u_2) at most. So there are at most

$$\sum_{u=0}^{u=N} \frac{(u+1) \times u}{2} \ = \ \frac{N^3 + 3N^2 + 2N}{6} \ \leq \ \frac{(N+1)^3}{6}$$

positions $(u, 0, 0, ((u_1, 0, 0, v_1), (u_2, 0, 0, v_2)))$. For every possible l $(0 \leq l \leq N)$ there are

$$\sum_{i=0}^{i=l}(l - i) \ = \ \frac{(l+1) \times l}{2}$$

possible values for (l_1, l_2) at most. So there are at most

$$\sum_{l=0}^{l=N} \frac{(l+1)\times l}{2} \;=\; \frac{N^3+3N^2+2N}{6} \;\leq\; \frac{(N+1)^3}{6}$$

consistent triples (l, l_1, l_2). Absolutely similar calculation gives the same upper bound for amount of consistent triples (h, h_1, h_2). So an upper bound for amount of positions $(0, l, h, ((0, l_1, h_1, v_1), (0, l_2, h_2, v_2$ is

$$\frac{(N+1)^6}{36}.$$

Finally, just similar to the modal $GAME(N, M)$, the overall amount of possible positions in the $game(N, M)$ is

$$p(N) \;\leq \frac{(N+1)^6}{36} + \frac{(N+1)^3}{6} \;\leq\; \frac{(N+1)^6}{30}$$

while the overall amount of possible moves in the game is

$$2 \times p(N) \;\leq\; m(N) \;\leq 4 \times p(N).$$

3.3 Checking and Abstraction

Again, if the player A is identified with *prog* and the player B with *user* then model checking of the formulae WIN_3 and WIN_4 in models $game(14, 1)$ and $game(39, 1)$ answers whether is it always possible to identify a unique false coin[14] in the original and in the retailer puzzles. In both cases it is enough to check whether these formulae are valid in "initial" positions of these models $(14, 0, 0, 1, ((0, 0, 0, 0), (0, 0, 0, 0)))$ and, respectively, $(39, 0, 0, 1, ((0, 0, 0, 0), (0, 0, 0, 0)))$. As far as concern corresponding identification strategies it is sufficient to model check formulae $\neg fail \wedge [move_B] fail$, $\neg fail \wedge [move_B](fail \vee WIN_1)$, $\neg fail \wedge [move](fail \vee WIN_2)$ in both $game(14, 1)$ and $game(39, 1)$ and then the formula $\neg fail \wedge [move](fail \vee WIN_2)$ in $game(39, 1)$ only. In these cases the strategies operates in terms of amounts of coins of different kind to be put on pans of the balance, but not in terms of coin numbers. It seems reasonable to try "plug and play" model checking since sizes of both models $game(14, 1)$ and $game(39, 1)$ are small in comparison with sizes of $GAME(14, 1)$ and $GAME(39, 1)$. But it does not lead to a solution of the metaprogram problem... For solving this problem we would like to understand better relations between the *abstract* model $game(N, M)$ and the *concrete* model $GAME(N, M)$ in general and how to restore winning strategies for $GAME(N, M)$ from winning strategies for $game(N, M)$ in particular. For it let us consider carefully a mathematical nature of the *abstraction*.

In general, let Φ be a set of formulae, $M_1 = (I_1, D_1)$ and $M_2 = (I_2, D_2)$ be two models, and $g : D_1 \to D_2$ be a mapping. The model M_2 is called an *abstraction* of the model M_1 with respect to formulae Φ iff[15] for all formula $\phi \in \Phi$ and state $s \in D_1$ the following holds: $s \models_1 \phi \Leftrightarrow g(s) \models_2 \phi$. In particular let us consider models $Game(N, M)$ and $game(N, M)$ $(N \geq 1, M \geq 0)$ and define a *counting* mapping $count \;:\; D_{GAME(N,M)} \to D_{game(N,M)}$ as follows:

$$count \;:\; (U, L, H, V, (S_1, S_2)) \;\mapsto (|U|, |L|, |H|, q)$$

where q is

$$((|S_1 \cap U|, |S_1 \cap L|, |S_1 \cap H|, |S_1 \cap V|), (|S_2 \cap U|, |S_2 \cap L|, |S_2 \cap H|, |S_2 \cap V|)).$$

This counting mapping can be component-withe extended on pairs of positions.

[14] But not the number of this coin!

[15] g is called an *abstraction mapping* in this case.

Assertion 3 *For all $N \geq 1$ and $M \geq 0$ the counting mapping is a homomorphism of a labeled graph $GAME(N, M)$ onto another labeled graph $game(N, M)$ with the following property for every position (U, L, H, Q) in the $GAME(N, M)$:*

1. *count maps all moves of a player which begins in the position onto moves of the same player in the $game(N, M)$ which begins in $count(U, L, H, Q)$;*
2. *count maps all moves of a player which finishes in the position onto moves of the same player in the $game(N, M)$ which finishes in $count(U, L, H, Q)$.*

The following assertion is an immediate consequence of the above one.

Assertion 4 *For all $N \geq 1$ and $M \geq 0$ the $game(N, M)$ is an abstraction of the $GAME(N, M)$ with respect to formulae of EPDL written with use of the unique propositional variable $fail$ and two action variables $move_A$ and $move_B$ only.*

3.4 Toward Stronger Logic

As follows from the assertions 3 and 4, for all $N \geq 1$, $M \geq 0$, $i \geq 1$ and position (U, L, H, V, Q) of the $GAME(N, M)$ the following holds: $(U, L, H, V, Q) \models_{GAME(N,M)} WIN_i$ iff $count(U, L, H, V, Q) \models_{game(N,M)} WIN_i$. As far as concern winning strategies, a move

$$count(U, L, H, V, Q) \rightarrow (u, l, h, v, ((u_1, l_1, h_1, v_1), (u_2, l_2, h_2, v_2)))$$

of the player *prog* is the first move of a winning strategy with i-rounds at most in the $game(N, M)$ iff a move

$$(U, L, H, V, Q) \rightarrow (U, L, H, V, ((U_1 \cup L_1 \cup H_1 \cup V_1), (U_2 \cup L_2 \cup H_2 \cup V_2)))$$

of the player *prog* is the first move of a winning strategy with i-rounds at most in the $GAME(N, M)$ where

- U_1 and U_2 are the first u'_1 and the next u'_2 consequentive elements of U,
- L_1 and L_2 are the first l'_1 and the next l'_2 consequentive elements of L,
- H_1 and H_2 are the first h'_1 and the next h'_2 consequentive elements of H,
- V_1 and V_2 are the first v'_1 and the next v'_2 consequentive elements of V.

Hence a high-level design of a solution of the original and retailer puzzles can looks like follows:

1. to model check formulae $\neg fail \wedge [move_B]fail$, $\neg fail \wedge [move_B](fail \vee WIN_1)$, $\neg fail \wedge [move](fail \vee WIN_2)$ in both models $game(14, 1)$ and $game(39, 1)$, and $\neg fail \wedge [move](fail \vee WIN_2)$ in $game(39, 1)$ only;
2. define 3- and 4-rounds at most winning strategy in $game(14, 1)$ and, respectively, $game(39, 1)$ for the player *prog* as moves (if possible) to positions where at least one of these formulae (i.e. their disjunction) is valid,
3. restore winning strategies for $GAME(14, 1)$ and $GAME(39, 1)$ from their scratches in the models $game(14, 1)$ and $game(39, 1)$.

Nevertheless several disadvantages of the above approach remain. First, we can not solve the metaprogram problem on base of the acquainted mathematical background! Second, the formulae are too large for handling them. Really, the formula $\neg fail \wedge [move](fail \vee WIN_2)$ is unfolding as

$$\Big\{\neg f \wedge [u]\Big\{f \vee \Big(\neg f \wedge < p > \Big(\neg f \wedge [u]\Big(f \vee \big(\neg f \wedge < p > (\neg f \wedge [u]f)\big)\Big)\Big)\Big)\Big\}\Big\}$$

where f, p and u are abbreviations for $fail$, $move_A$ and $move_B$ respectively[16] Third, it is not clear how to express a property that there is a winning strategy. Really the property can be expressed by an *infinite* disjunction

$$WIN_0 \vee WIN_1 \vee WIN_2 \vee WIN_3 \vee WIN_4 \vee ... = \bigvee_{i \geq 0} WIN_i$$

but this expression is *illegal* formula in EPDL. The following arguments proves formally that EPDL is too weak for expressing it. Let us consider all non-positive integers as a domain and interpret $fail$ to be valid on 0 only while $move_A$ and $move_B$ to be interpreted as the successor function $+1$ on negatives. Let us denote the resulting model by NEG. Then let us define an *action nesting an* for EPDL formulae by induction on formulae structure:

1. $an(fail) = an(true) = an(false) = 0$,
2. $an(\neg\phi) = an(\phi)$,
3. $an(\phi \wedge \psi) = \max\{an(\phi), an(\psi)\}$,
4. $an(\phi \vee \psi) = \max\{an(\phi), an(\psi)\}$,
5. $an([move_A]\phi) = an([move_B]\phi) = 1 + an(\phi)$,
6. $an(< move_A > \phi) = an(< move_B > \phi) = 1 + an(\phi)$.

In this setting, for every EPDL formula ϕ, for all $k > an(\phi)$ and $l > an(\phi)$ the following can be trivially proved by induction on formulae structure: $(-k) \models_{NEG} \phi \Leftrightarrow (-l) \models_{NEG} \phi$. So for every formula of EPDL there exists a non-positive number prior to which the formula is a boolean constant, while the infinite disjunction $\bigvee_{i \geq 0} WIN_i$ is valid in all even negative integers.

So it seems reasonable to have another logic with stronger expressive power. For it we are going to describe below a so-called *μ-Calculus* [3] as an extension of the EPDL. Both syntax and semantics of this logic are more complicated then EPDL ones.

4 Propositional μ-Calculus

4.1 μ-Calculus Syntax

Let us extend the syntax of EPDL by two new features:

7. if p is a propositional variable and ϕ is a formula then $(\mu p.\phi)$ is a formula[17],

[16] Let us remind that we identify the player *prog* with A and the player *user* with B.

[17] which is read as "*mu p ϕ*" or "*the least fixpoint p of ϕ*"

8. if p is a propositional variable and ϕ is a formula then $(\nu p.\phi)$ is a formula[18].

Informally speaking $\mu p.\phi$ is an "abbreviation" for an infinite disjunction

$$false \ \vee \ \phi_p(false) \ \vee \ \phi_p(\phi_p(false)) \ \vee \ \phi_p(\phi_p(\phi_p(false))) \ \vee \ ... \ = \ \bigvee_{i\geq 0} \phi_p^i(false)$$

while $\nu p.\phi$ is an "abbreviation" for another infinite conjunction

$$true \ \wedge \ \phi_p(true) \ \wedge \ \phi_p(\phi_p(true)) \ \wedge \ \phi_p(\phi_p(\phi_p(true))) \ \wedge \ ... \ = \ \bigwedge_{i\geq 0} \phi^i(true),$$

where

- $\phi_p(\psi)$ is a result of substitution of a formula ψ on places of p in ϕ,
- $\phi_p^0(const)$ is a boolean constant $const \in \{false, true\}$,
- $\phi_p^{i+1}(const)$ is $\phi_p(\phi_p^i(const))$ for $i \geq 0$.

In particular, if ϕ is a formula

$$\neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee win)\big)$$

where win is a new propositional variable, then the formula WIN_0 is just $\phi_{win}(false)$, the formula WIN_1 is equivalent to

$$\phi_{win}^1(false) \ \equiv \ \neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee false)\big)$$

since WIN_1 is defined as $\neg fail \wedge < move_A > (\neg fail \wedge [move_B] fail)$, while WIN_{i+1} $(i \geq 1)$ is equivalent to

$$\phi_{win}^{i+1}(false) \ \equiv \ \neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee \phi_{win}^i(false))\big)$$

since it is defined as $\neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee WIN_i)\big)$. Finally, the infinite disjunction $\bigvee_{i\geq 0} WIN_i$ should be equivalent to

$$\mu \ win.\phi \ \equiv \ \mu \ win.\Big(\neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee win)\big)\Big).$$

Let us denote the last formula by WIN.

The above definition of formulae is too wide. We would like to impose two context-sensitive restrictions on the set formula for getting formulae of the μ-Calculus[19]. The restrictions are listed below.

- In formulae $(\mu p.\phi)$ and $(\nu p.\phi)$ the *range* of μp and νp is the formula ϕ and all instances of the variable p are called *bounded* in $(\mu p.\phi)$ and $(\nu p.\phi)$. An instance of a variable in a formula is called *free* iff it is not bounded. *No propositional variable can have free and bounded instances in a sub-formula of a formula.*
- In a formula $(\neg\phi)$ the *range* of negation is the formula ϕ. An instant of a propositional variable in a formula is called *positive/negative* iff it is located in an even/odd number of negation ranges. *No bounded instance of a propositional variable can be negative.*

[18] which is read as "*nu p* ϕ" or "*the greatest fixpoint p of* ϕ"

[19] The first is for syntactical convenience while the second is essential.

These restrictions finish the definition of the syntax of the μ-Calculus formulae. As usual we would like to avoid extra parenthesis and extend a standard list of priorities: $\neg$, $<>$, $[\,]$, μ, , ν, $\wedge$, $\vee$, $\rightarrow, \leftrightarrow$.

Let us illustrate how the above restrictions work. The first restriction prohibits the formula $p \wedge \nu q.\big([a]q \wedge \mu q.(p \vee [a]q)\big)$ to be a formula of the μ-Calculus since a propositional variable q has free and bounded instances in a sub-formula $[a]q \wedge \mu q.(p \vee [a]q)$. In contrast the following formula $FAIR_1$

$$p \wedge \nu q.\big([a]q \wedge \mu r.(p \vee [a]r)\big)$$

is a correct formula of the μ-Calculus. The second restriction prohibits the formula $\nu q.\big(p \wedge [a]\mu r.(\neg q \vee [a]r)\big)$ to be a formula of the μ-Calculus since a propositional variable q has a bounded negative instance. In contrast the following formula $FAIR_2$

$$\nu q.\big(p \wedge [a]\mu r.(q \vee [a]r)\big)$$

is a correct formula of the μ-Calculus. The above formula WIN is also a correct formula of the μ-Calculus. This formula play a special role in metaprogram problem as is shown below. Finally let us remark that the above restrictions do not prevent formulae of EPDL to be formulae of μ-Calculus since EPDL formulae do not contain bounded instances of variables.

Problem 6 *Are formulae of μ-Calculus a context-free language?*

4.2 μ-Calculus Semantics

The semantics of μ-Calculus is defined in the same models as EPDL (i.e. Labeled Transition Systems or Kripke Structures) in terms of sets of states where formulae are valid. For every model $M = (D_M, I_M)$ let us denote by $M(formula)$ a set of all states of the model where a $formula$ is valid. The first 6 closes of the definition deal with EPDL features:

1. for boolean constants $M(true) = D_M$ and $M(false) = \emptyset$;
   for every propositional variable p, $M(p) = I_M(p)$;
2. for every formula ϕ, $M(\neg\phi) = D_M \setminus M(\phi)$;
3. for all formulae ϕ and ψ, $M(\phi \wedge \psi) = M(\phi) \cap M(\psi)$;
4. for all formulae ϕ and ψ, $M(\phi \vee \psi) = M(\phi) \cup M(\psi)$;
5. for all action variable a and formula ϕ, $M(< a > \phi) =$
   $\{s \in D_M : (s, s') \in I_M(a)$ and $s' \in M(\phi)$ for same state $s' \in D_M\}$;
6. for all action variable a and formula ϕ, $M([a]\phi) =$
   $\{s \in D_M : (s, s') \in I_M(a)$ implies $s' \in M(\phi)$ for every state $s' \in D_M\}$.

As far as concern new features μ and ν then let us define their semantics for *finite* models only since it is the case of interest of the paper and the major domain for model checking applications:

7. for every formula ϕ, $M(\mu p.\ \phi) =$
   $$\bigcup\{\ L_i \subseteq D_M\ :\ L_0 = \emptyset \text{ and } L_{i+1} = M_{L_i/p}(\phi) \text{ for all } i \geq 0\ \},$$
8. for every formula ϕ, $M(\nu p.\ \phi) =$
   $$\bigcap\{\ K_i \subseteq D_M\ :\ K_0 = D_M \text{ and } K_{i+1} = M_{K_i/p}(\phi) \text{ for all } i \geq 0\ \},$$

where $M_{S/p}$ is a model which differs from M by interpretation of p as S (i.e. $D_{M_{S/p}} = D_M$, $I_{M_{S/p}}(a) = I_M(a)$ for every action variable a, $I_{M_{S/p}}(q) = I_M(q)$ for every propositional variable q other then p, while $I_{M_{S/p}}(p) = S$). Finally let us define the *validity* relation $\models'_M$ for all formula ϕ and state s in a natural way: $s \models'_M \phi$ iff $s \in M(\phi)$. Let us remark that we can use in the framework of the μ-Calculus the same notation $\models_M$ as in a framework of EPDL since the following holds:

Proposition 3 *The μ-Calculus is a conservative extension of EPDL, that is $s \models'_M \phi$ iff $s \models_M \phi$ for all EPDL formula ϕ, model M and state s.*

Let us also to compare the informal semantics of fixpoints with the formal one. Let M be a finite model and ϕ be a formula. We have

$$\begin{array}{rl} \phi_p^0(false) \equiv false & L_0 = \emptyset \\ \ldots & \ldots \\ \phi_p^{i+1}(false) \equiv \phi_p(\phi_p^i(false)) & L_{i+1} = M_{L_i/p}(\phi) \\ \ldots & \ldots \end{array}$$

Hence $M(\phi_p^i(false)) = L_i$. Then we have

$$\begin{array}{rl} K_0 = D_M & true \equiv \phi_p^0(true) \\ \ldots & \ldots \\ K_{i+1} = M_{K_i/p}(\phi) & \phi_p^{i+1}(true) \equiv \phi_p(\phi_p^{i+1}(true) \\ \ldots & \ldots \end{array}$$

Hence $M(\phi_p^i(true)) = K_i$ for every $i \geq 0$. So in finite models $\mu p.\ \phi(p)$ is really an "abbreviation" for an infinite disjunction $\bigvee_{i\geq 0} \phi_p^i(false)$ while $\nu p.\ \phi(p)$ is an "abbreviation" for an infinite conjunction $\bigwedge_{i\geq 0} \phi^i(true)$. In particular the formula WIN is really equivalent to the infinite disjunction $\bigvee_{i\geq 0} WIN_i$. Hence, this formula of the μ-Calculus is not equivalent to any formula of EPDL. Simultaneously all formulae of EPDL are formulae of the μ-Calculus. These arguments prove the next proposition.

Proposition 4 *The μ-Calculus is more expressive then EPDL.*

The above formula WIN is not the unique formula of interest. Let us consider $FAIR_1$ for example. Where a sub-formula $\phi \equiv \mu r.(p \vee [a]r)$ of this formula is valid in a model? – Just in those states where every infinite a-path eventually leads to p. Really, $\phi_r^0(false) \equiv false$ is always invalid, $\phi_r^1(false) \equiv (p \vee [a]false)$ is valid iff p holds or a does not halts, $\phi_r^2(false) \equiv (p \vee [a]\phi_r^1(false)$ is valid iff p holds or a does not halts after at most 1-step of a,... $\phi_r^{i+1}(false) \equiv (p \vee [a]\phi_r^i(false)$ is valid iff p holds or a does not halts after at most i-steps of a, etc. Where a sub-formula $\nu q.([a]q \wedge \phi)$ of $FAIR_1$ is valid in a model? – Just in those states where every a-path always leads to ϕ. Really, let us consider a formula $\psi \equiv \nu q.([a]q \wedge r)$ and remark that $\psi_r(\phi)$ is $\nu q.([a]q \wedge \phi)$. In this case $\psi_q^0(true) \equiv true$ is always valid, $\psi_q^1(true) \equiv ([a]true \wedge r)$ is equivalent to r, i.e. is valid iff r holds, $\psi_q^2(true) \equiv ([a]\psi_q^1(true) \wedge r)$ is valid iff r holds now and after 1-step of a,... $\psi_q^{i+1}(true) \equiv ([a]\phi_q^i(true) \wedge r)$ is valid iff r always holds during at most i-steps of a, etc. So, in general, $\nu q.([a]q \wedge r)$ is valid iff every a-path always leads to r, and, in particular, $\nu q.([a]q \wedge \phi)$ is valid in those states of a model where every a-path always leads to ϕ. But ϕ itself is valid in those where every infinite a-path eventually leads to p. So $\nu q.([a]q \wedge \phi) \equiv \nu q.\big([a]q \wedge \mu r.(p \vee [a]r)\big)$ is valid in a state of a model iff every infinite a-path infinitely often visits states where p holds. An infinite sequence is said to be *fair* with respect to a property iff the property holds for an infinite amount of elements of the

sequence. For example, a scheduler of CPU time among several permanent jobs is fair with respect to a concrete job iff it schedules this job for CPU infinitely often. Since $FAIR_1$ is a conjunction of p with the above formula, then $FAIR_1$ holds in a state of a model iff p holds and every infinite a-path is fair with respect to p.

Problem 7 *Prove that formulae $FAIR_1$ and $FAIR_2$ are equivalent.*

5 Model Checking for μ-Calculus: Metaprogram and far Beyond...

5.1 Toward Metaprogram Problem

Now we are ready to solve the metaprogram problem but we are not going to present a solution of the problem in the paper. In contrast we would like to discuss cornerstones and a detailed design of a solution while a solution is left for readers.

Let $N \geq 1$, $M \geq 0$ and $K \geq 0$ be numbers. In accordance with the assertion 4, the model $game(N, M)$ is an abstraction of the model $GAME(N, M)$ with respect to formulae of EPDL and *count* is a corresponding abstraction mapping. But $L_i = \ game(N, M)(WIN_i)$ for every $i \geq 1$, where $L_0 = \emptyset$ and

$$L_{j+1} = \ game(N, M)_{L_i/win}\Big(\neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee win)\big)\Big)$$

for every $j \geq 0$. As follows from the **Proposition 1**, $game(N, M)(WIN_i)$ comprises all positions where *prog* has a i-rounds at most strategy for identification of a false coin. Hence, a criterion which decides whether there exists a K-times balancing (at most) strategy for identification a unique false coin among N coins under question with aid of M valid coins is to check whether the initial position $(N, 0, 0, M, ((0, 0, 0, 0), (0, 0, 0, 0)))$ is in L_K.

Fig. 2 presents a body of PASCAL program `BETA` which implements the above criterion for input values of N, M, K. In this program variables `N`, `M` and `K` are for the initial amounts of coins under question, of valid coins and of balancing. A variable for a number of a current round is `i`. Variables for current amounts of untested, lighter and heavier coins are `u`, `l` and `h` respectively. Variables `u1`, `u2`, `l1`, `l2`, `h1` and `h2` are for amounts of untested, lighter and heavier coins in a query, i.e. for different pans in a current balancing. Finally, variables `preLu`, `nexLu` are 1-dimension and variables `preLlh`, `nexLlh` are 2-dimensional boolean arrays with indexes range [0..N]. An array `*Lu` is for presenting a set of positions $(u, 0, 0, ((0, 0, 0, 0), (0, 0, 0, 0)))$ while another array `*Llh` is for presenting a set of positions $(0, l, h, ((0, 0, 0, 0), (0, 0, 0, 0)))$. A pair of arrays `preLu` and `preLlh` is for presenting a set of positions L_i while another pair of arrays `nexLu` and `nexLlh` is for presenting a set of positions L_{i+1}.

Finally let us discuss a detailed design of a metaprogram. Let `GAMMA(`N`,`M`, K)` be be a program which works just the same as `BETA`, but it treats `N`, `M` and `K` as constants (in particular it does not input their values) with fixed values N, M, K and, (in addition) saves successful moves of the player *prog*: for every $0 \leq i < K$ and every *position* $\in L_{i+1}$ the program saves a *query* during calculation of L_{i+1} iff it is the first time when all replies on this query of the other player *user* lead to positions in L_i. Let `DELTA(`N`,`M`,`K`)` be another program which initially works like `GAMMA(`N`,`M`,`K`)` and as soon as `GAMMA(`N`,`M`,`K`)` halts it begins to work in terms of numbers of coins (i.e. positions of the $GAME(N, M)$) as follows. `DELTA(`N`,`M`,`K`)` adopts $([(M + 1)..(N + M)], \emptyset, \emptyset, [1..M], (\emptyset, \emptyset))$ as the initial current position and then executes the following loop until it reaches a final current position $(U, L, H, V, (\emptyset, \emptyset))$ with $|U \cup L \cup H| = 1$:

1. apply a mapping *count* to a current *position* in the $GAME(N, M)$;

```
for u:=0 to N do preLu[u]:=FALSE;
for l:=0 to N do for h:=0 to N do preLlh[l,h]:=FALSE;
WriteLn('Input N'); ReadLn(N);
WriteLn('Input M'); ReadLn(M);
WriteLn('Input K'); ReadLn(K);
for i:=1 to K do
    begin
    for u:=0 to N do nexLu[u]:=FALSE;
    for l:=0 to N do for h:=0 to N do nexLlh[l,h]:=FALSE;
    begin
    for u:=0 to N do
        for u1:=0 to u do
            for u2:=0 to (u-u1) do
                if (M+N)-u >= abs(u1-u2)
                   then nexLu[u] := preLu[u] OR (u=1) OR
                        (((u1+u2=1) OR preLlh[u1,u2]) AND
                         ((u-(u1+u2)=1) OR preLu[u-(u1+u2)]) AND
                         ((u1+u2=1) OR preLlh[u2,u1]));
    for l:= 0 to N do
        for h:=0 to (N-l) do
            for l1:=0 to l do
                for l2:=0 to (l-l1) do
                    for h1:=0 to h do
                        for h2:=0 to (h-h1) do
                            if (N+M)-(l-h) >=abs((l1+h1)-(l2+h2))
                               then nexLlh[u,h] :=
                                    nexLlh[l,h] OR (l+h=1) OR
                                 (((l1+h2=1) OR preLlh[l1,h2]) AND
                                  ((l-(l1+l2)+h-(h1+h2)=1) OR
                                  preLlh[l-(l1+l2),h-(h1+h2)]) AND
                                  ((l1+h2=1) OR preLlh[l2,h1]));
    for u:=1 to N do preLu[u]:=nexLu[u];
    for l:=0 to N do for h:=0 to N do preLlh[l,h]:=nexLlh[l,h];
    end;
if preLu[N] then writeLn('possible') else writeLn('impossible')
```

Figure 2: A body of a PASCAL program which checks whether is it possible to identify a unique false coin among N coins under question with aid of M valid coins and balancing then K times at most.

2. get a successful *query* which is calculated by `GAMMA(`N`,`M`,`K`)` and corresponds to the position $count(position)$ in the $game(N, M)$;

3. define (in accordance with the assertion 3) a corresponding *move* of the player *prog* in the $GAME(N, M)$ and output it for the user;

4. define the *next_position* with respect to the user reply in the $game(N, M)$;

5. restore a corresponding *new_position* in the $GAME(N, M)$ (in accordance with the assertion 3) and adopt it as the next current position.

After that `DELTA(`N`,`M`,`K`)` outputs the unique number in the $U \cup L \cup H$ and halts.

Assertion 5 *For all given and fixed values N, M and K* `DELTA(`N`,`M`,`K`)` *is a strategy of identification of a number of a unique false coin among N coins with numbers $(M+1)..(N+M)]$ with use of M valid coins and balancing K times at most iff such strategy exists.*

Do you now guess how a solution of the metaprogram problem looks like? It looks like the program `BETA` but in contrast to `BETA` it has two special features. The metaprogram declares an auxiliary string constant `DELTA` which is a text of the program `DELTA(`N`,`M`,`K`)` without values for constants `N`, `M` and `K`. Then the metaprogram has another last operator then `BETA`: `writeLn('possible')` in the last operator of `BETA`

```
if preLu[N] then writeLn('possible') else writeLn('impossible')
```

is replaced in the metaprogram by a subprogram which inserts symbolically values of N, M, K as values for constants `N`, `M`, `K` into string constant `DELTA` and output the string.

5.2 μ-Calculus Semantics Again

Semantics of formulae as well as the semantics of propositional variables are sets of states. It gives us a new opportunity to consider formulae as functions which map interpretations of propositional variables into sets where formulae are valid in corresponding interpretations. Let us illustrate this new approach to the μ-Calculus semantics by a game example. Let (P, M_A, M_B, F) be a finite game of two players while $M = G_{(P,M_A,M_B,F)}$ be a corresponding model. Let ϕ be a formula $\neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee win)\big)$. An interpretation P of propositional variables of the formula is a pair $(P(fail), P(win))$ of sets of positions. Let T be a "standard" interpretation for $fail$ as in the $G_{(P,M_A,M_B,F)}$, let $S_0 = \emptyset$ and for every $i \geq 1$ let S_i be a set of all positions where the player A has a wining strategy with i-rounds at most. For every $i \geq 0$ let $P_i = (T, S_i)$. Since $S_i = G_{(P,M_A,M_B,F)}(WIN_i)$ (**Proposition 1**) and

$$WIN_{i+1} \equiv \neg fail \wedge < move_A > \big(\neg fail \wedge [move_B](fail \vee WIN_i)\big)$$

for every $i \geq 1$ then $M_{S_i/win}(\phi) \mapsto S_{i+1}$ for every $i \geq 0$. For every $i \geq 1$ a natural inclusion $S_i \subseteq S_{i+1}$ holds, since a i-rounds at most winning strategy is automatically a $(i+1)$-rounds at most winning strategy. Let us summarize it all as follows:

argument S:	$\emptyset$	$\subseteq$	S_1	$\subseteq$	S_2	$\subseteq$	...	S_i	$\subseteq$	...
$\lambda S.\big(M_{S/win}(\phi)\big)$:	$\downarrow$		$\downarrow$		$\downarrow$			$\downarrow$		
result $M_{S/win}(\phi)$:	S_1	$\subseteq$	S_2	$\subseteq$	S_3	$\subseteq$	...	S_{i+1}	$\subseteq$	...

As follows from the table, the mapping $\lambda S.\big(M_{S/win}(\phi)\big)$ non-decries monotonically on $\{S_i : i \geq 0\}$. This mapping has another important *fixpoint* property: if $S = \bigcup_{i \geq 0} S_i$ then (**Proposition**

1), S is a *fixed point* of $M(\phi)$, i.e. $M_{S/win}(\phi) = S$. By the way, informally speaking the above equality is very natural: if the player A is in a position where he/she has a winning strategy then he/she has a move prior to and after which the game is not lost, but after which every move of another player B leads to a position where the game is lost or A has a winning strategy.

Is the above monotonicity an accidental property of special formulae in special models? – Not at all! It is a basic property of the μ-Calculus formulae.

Proposition 5 *For all model M, sets of states $S' \subseteq S''$, propositional variable p and formula ϕ*

- *if p has not negative instances in the ϕ then $M_{S'/p}(\phi) \subseteq M_{S''/p}(\phi)$,*
- *if p has not positive instances in the ϕ then $M_{S''/p}(\phi) \subseteq M_{S'/p}(\phi)$.*

The *monotonicity* property has very important semantical implications. In particular it leads to a fixpoint characterization of semantics of μ and ν. (A particular example of this characterization is discussed above.)

Proposition 6 *For all propositional variable p, formula ϕ of the μ-Calculus without negative and bounded instances of p, and model $M = (D_M, (R_M, P_M))$ $M(\mu p.\phi)$ and $M(\nu p.\phi)$ are the least and the greatest fixpoints with respect to inclusion of a monotone non-decreasing function*

$$\big(\lambda S \subseteq D_M.\ M_{S/p}(\phi)\big)\ : \mathcal{P}(D_M)\ \longrightarrow\ \mathcal{P}(D_M)$$

which maps each $S \subseteq D_M$ to $M_{S/p}(\phi) \subseteq D_M$.

5.3 Model Checking the μ-Calculus

Let us return to the millennium game which Alice and Bob played on the the Eve of the New Year 2000. In this game (in contrast to the metaprogram problem) we are interested in a set of positions where a winning strategy exists in principle, i.e. in states of the model $G_{2000/2001}$ where the formula[20] $WIN' \ \equiv\ \mu\ win.\ \Big(\neg fail \wedge < move > \big(\neg fail \wedge [move](fail \vee win)\big)\Big)$ holds. It is a typical model checking problem but for the μ-Calculus this time. The semantics of the μ-Calculus defined in the section 4.2 is (in principle) a model checking algorithm for the μ-Calculus in finite models. Let us first remind parameters used for measuring model checking complexity and then formulate a statement about complexity of this model checking algorithm.

If $M = (D_M, (R_M, P_M))$ is a finite model then d_M and r_M are amounts of states in D_M and edges in R_M, while $m = (d_M + r_M)$. If a model M is implicit then we would like to use these parameters without subscripts, i.e. just d, r and m. If ϕ is a formula then f_ϕ is an amount of variables, connectives, modalities, μ and ν instances in ϕ. The last measure can be generalized as follows: if ϕ is a formula and *set* is a collection of variables, connectives, modalities, μ and ν, then $f_\phi(set)$ is an amount of instances of elements of *set* in ϕ. So f_ϕ is just $f_\phi(Act \cup Prp \cup \{\neg, \wedge, \vee, <>, [\,], \mu, \nu\})$. But we are particular concerned by the amount of instances of μ and ν in the formula ϕ, i.e. by $(f\mu\nu)_\phi = \ f_\phi(\{\mu, \nu\})$. If a formula ϕ is implicit then we would like to use this parameter f_ϕ without subscript, i.e. just f.

Proposition 7 *Model checking problem for the μ-Calculus in finite models is decidable with an upper time bound $O(m \times f \times d^{(f\mu\nu)})$.*

[20] In the above the formula WIN uses two different action variables $move_A$ and $move_B$. In this section WIN' is a variant of the formula with $move_A \equiv move_B \equiv move$.

In particular, a computer-aided solution for the problem 4 for the millennium game, becomes just technical: implement the above model checking algorithm for the μ-Calculus, code the model $G_{2000/2001}$ and then "plug and play", i.e. model check the formula WIN'. Moreover, it is possible to generalize (or parameterize) the problem: just consider another New Year Eve then 2000, i.e. 1999, 1998, etc. Let us denote corresponding models by $G_{1999/2001}$, $G_{1998/2001}$ and so on, or, in general, $G_{N/2001}$. An upper bound for time complexity of finding all positions where Alice had a winning strategy in the $G_{N/2001}$ is $O\big((2001-N)^3\big)$ in accordance with the **Proposition 7**. But a constant coefficient in the above $O(...)$ is too large: something around $2,000,000,000$. The main reason is the factor $d^{(f\mu\nu)}$ which in this case gives around 88% of the above multiplicative coefficient[21]. A careful analysis of the computational complexity of the model checking algorithm sketched in the proof of the **Proposition 7** can lead to a better upper bound $O(m \times f \times d^n)$ where n is a new parameter called *nesting depth*. Informally speaking a nesting depth of a formula is a maximal amount of nesting μ and ν in the formula. But nevertheless the improved upper bound remains exponential... In particular, an improved upper bound for finding all positions where Alice had a winning strategy in the $G_{N/2001}$ is $O\big((2001-N)^2\big)$ A multiplicative coefficient in $O(...)$ is now 365-times smaller then in the above: something around $6,000,000$ only!

5.4 Algorithmic Problems for the μ-Calculus

A problem how to leap a complexity gap between a polynomial model checking EPDL and an exponentially hard model checking the μ-Calculus in finite models is quite natural. Unfortunately, best known model checking algorithms for the μ-Calculus and finite models are exponential. For example, a time bound of Faster Model Checking Algorithm (FMC-algorithm) [4] is roughly

$$O(m \times f) \times \Big(\frac{m \times f}{a}\Big)^{a-1}$$

where an alternating depth a of a formula is a maximal amount of alternating nesting μ and ν with respect to the syntactical dependences and formally is defined by induction. A formal definition is out of scope of the paper due to space limitations. We would like to point out only that the alternating depth is always less then or equal to the nesting depth for every formula: $a_\phi \leq n_\phi$. In particular, the a complexity of model checking the formula WIN in a model $G_{N/2001}$ is $O(2001-N)$ at all! As far as concerns lower bounds for the model checking problem for the μ-Calculus in finite models then it is known that the problem is in $\mathcal{NP} \bigcap co-\mathcal{NP}$ [5], i.e. it is not more complicated then checking formulae of the propositional calculus to be a tautology and a satisfiable formula. Due to this reason it seems to be very hard to prove an exponential lower bound for the model checking problem for the μ-Calculus in finite models. Since it is not known whether the problem is complete in $\mathcal{NP}$ then it seems to be more realistic to try to find a polynomial model checking algorithm for the μ-Calculus in finite models. At least several expressive fragments of the μ-Calculus which have polynomial hard model checking problem for finite models have been identified [5, 6]. As follows from the upper bound for the FMC-algorithm, formulae with a bounded alternating nesting depth form a fragment of this kind.

Problem 8 *(a) Describe new fragments of the μ-Calculus with a polynomial model checking in finite models. (b) Prove a polynomial upper or an exponential lower time bound for model checking the μ-Calculus in finite models.*

[21] Let us say that a the first factor gives $p^a_{ab} = \frac{|a-b|}{a} \times 100\%$ to the product $(a \times b)$, while the second factor gives $p^b_{ab} = \frac{|a-b|}{b} \times 100\%$. Of course it is nonsense to measure how much gives each factor to their product. In particular $p^a_{ab} + p^b_{ab} \neq 100\%$!

Decidability is another important algorithmic problem. The problem is how to check whether a given formula of the μ-Calculus is valid in *all* models. It is known that it is possible to check the validity not in *all* models but in *all finite* models only due to a so-called *finite-model property* of the μ-Calculus formulae: a formula is satisfiable[22] in a model iff it is satisfiable in a *finite* model [7, 10]. But this reduction does not make the problem to be trivial! Moreover, the reduction itself is just a corollary of the decidability of the μ-Calculus with an exponential upper bound. In principle, an exponential decidability for this logic can be proved by means of an automata-theoretic technique [7, 8]. This and other impressive applications of the automata-theoretic technique lead the program logic community to the opinion [9] that the automata-theoretic approach is the unique paradigm for proving decidability for complicated propositional program logics. In spite of this opinion, the successful applications of another technique called Program Scheme Technique (PST) were reported several times. This technique [10] is a powerful approach for proving decidability of program logics. It is completely self-contained, automata-free technique yielding one-exponential upper time bounds. A revised version of the Program Scheme Technique for decidability of the μ-Calculus and a role of model checking of the formula WIN in this framework is a topic for forthcoming paper [11].

A more complicated algorithmic problem for μ-Calculus is axiomatization in general and how to axiomatize μ-Calculus on based on PST in particular. In this context we would like to remark that in the paper [3] a natural sound axiomatization for μ-Calculus was proposed, but the completeness of the axiomatization was proved for a fragment of this logic only[23]. The completeness problem for μ-Calculus was an open problem during 10 years. Finally it was solved by I. Walukiewicz in 1993 [9, 12] on base of game theory and theory of automata on infinite trees. Nevertheless the completeness proof is very complicated and any simplification suggestions are welcome!

Problem 9 *A complete axiomatization of the μ-Calculus made easy.*

6 Program Logics at All

6.1 Where Are "Program Logics"?!

Really, where they are? All above is about Elementary Propositional Dynamic Logic and the μ-Calculus! Moreover, an experienced mathematician can remark that EPDL is just a polymodal variant the classical and basic modal logic **K** [13] as well as the μ-Calculus is just a polymodal variant of the μ**K**, i.e. **K** extended by fixpoints. Really, in terms of EPDL, **K** is a variant of EPDL with a unique action variable. Since in this case a name of this variable is not important then it is possible to omit the variable in formulae and write $\Box$ and $\Diamond$ instead of [...] and $< ... >$ respectively. This "new" modalities are read "*box*" or "*always*" and "*diamond*" or "*sometimes*". In particular, the formulae win_i $(i \geq 0)$ for positions in the millennium game where Alice had a i-round at most winning strategy, — all these formula are formulae of **K** and can be rewritten in $\Box$ and $\Diamond$ notation as $win_0 \equiv false$ and $win_{i+1} \equiv \neg fail \wedge \Diamond\big(\neg fail \wedge \Box(fail \vee win_i)\big)$ for every $i \geq 1$. In this notation the following formula $\mu\, win.\, \Big(\neg fail \wedge \Diamond\big(\neg fail \wedge \Box(fail \vee win)\big)\Big)$ of the μ**K** characterizes the set of all game positions where Alice had a winning strategy. So it is really reasonable to consider EPDL as a polymodal variant the modal logic **K** and the

[22] A formula is said to be *satisfiable* in a model iff it is valid on a state of the model.

[23] Some mathematicians use the term *calculus* for systems defined *syntactically* by axioms, rewriting rules, etc., not *semantically* like *logics* or *algebras*, i.e. by validity in models, equalities, etc. For example, λ-Calculus is defined syntactically by means of equational axiomatization or in terms of α- and β-reductions, while the First-Order Logic is usually defined semantically in terms of validity of formulae in models. From a viewpoint of this *conventional* terminology, μ-Calculus in this paper is rather a logic then a calculus.

μ-Calculus as a polymodal variant of the $\mu\mathbf{K}$. Why do we write about them under the title "*Program logics* made easy"? And why do we give non-mathematical names for them?

The answers are quite simple. *Program logics* are modal logics used in soft- and hardware verification and specification for reasoning about *programs*. In 1980-ies program logics comprised

- *dynamic logics* [14, 16],
- *temporal logics* [15, 17],

and their extensions by means of fixpoints. A more recent addition to the family of program logics is *logic of knowledge* [18]. The utility of this logic for this application is that it provides a language that formalizes constructs capturing notions that are used informally in reasoning about multi-agent systems when a pure dynamic/temporal approach is not very convenient. The "given names" of program logics are sometimes traditional and closely related to their mathematical names[24], sometimes they are invented by their parents with respect to their intuition about verification and specification application domain[25]. Situation with "given names" is quite similar to the situation with a generic name of models for program logics: some researchers prefer a mathematical name *Kripke Structures* while other prefer the application-oriented name *Labeled Transition System*. What is better? – Up to you!

6.2 Why We Should Know Program Logics?

The role of Formal Methods in the development of computer hard- and software increases since systems become more complex and require more efforts for their specification and verification. A logical approach to the verification and specification comprises of the following choices:

- a specification language for properties presentation,
- a formal proving technique for specified properties.

Specification languages which are in use for presentation of properties rage from propositional to high-order logics while a proving technique is either model-checking (a semantical approach) or deductive reasoning (a syntactical approach). In spite of the importance of the logical approach for development of a reliable hard- and software this research domain is not well acquainted to non-professionals. In particular, many undergraduate students of departments which are closely related to further progress of computer hard- and software (i.e. pure/applied mathematics and electric/electronic engineering) consider Formal Methods in general to be out of scope of their interests, since they (Formal Methods) are

- either too poor for their pure mathematics,
- either too pure for their poor mathematics.

We are especially concerned by this disappointing not-well motivated attitude and suppose that a deficit of a popular papers on this topic is the main reason for this obscurantism. In this particular paper we would like to present in a popular (but mathematically sound) form a Program Logics tributary creek of a powerful stream called Formal Methods.

It is possible (in principle) to construct a complete first-order axiomatization for each finite model and then try to prove a desired property (semi)automatically by means of any available logical framework [19, 20, 21]. But this purely deductive approach is sometimes

[24] Ex., *temporal* is a program logic, while *tense* is a basic one.
[25] Ex., *dynamic* is a program logic, while **K** is a basic one.

not practical for complexity reasons. Let us consider a finite model of a moderate size with approximately $100,000$ states. If it has a "clear" structure then it is reasonable to try to "catch" the model on the whole by means of a sound axiomatization and then to try to prove a desired property in a (semi)automatic style. But if a model has a "vague" structure which can be generated automatically (e.g. all possible configurations of a "small" distributed system) then it is reasonable to apply an automatic model checker to the generated system and a desired property, presented as a formula of a program logic. In this case decidability and complexity issues of model checking for a particular logic arise. A choice of an efficient model checking algorithm and an implementation problem follow. Efficiency issues become more important as soon as model checking is applied to huge models with, say, 10^{100} states, since large sets representation problem arises.

6.3 Concluding Remarks

We would like to recommend some further reading on mathematical theory of program logics[26]:

first [18]

$$\underbrace{[14] \quad [15] \quad [16] \quad [17]}_{\text{in any order.}}$$

For those who are interested in applications of program logics some reading is recommended below also.

Temporal logic have been shown to provide a convenient framework for specifying and reasoning about properties of a broad class of systems which can be presented or simulated by computer programs. A. Pnueli was the first who proposed to use temporal logic for reasoning about programs [23]. His approach for specification of concurrent and reactive systems is now well developed [24] as well as a manual deductive methodology for proving special properties [25]. This approach consists in proving properties of a program from a set of axioms that describe the behavior of the individual statements and problem-oriented inductive proof principles. Since it is a deductive approach where proofs are constructed by hand, the technique is often difficult to automate and use in practice.

Part of the reason for further success of temporal logic is based on automatic model checking of specifications expressed on propositional level temporal logics for finite state systems [26]. Branching temporal logic CTL and polynomial model checking algorithms were developed as a new mathematical background for a new verification methodology for finite state systems by E.M. Clarke and E.A. Emerson, J.-P. Queille and J. Sifacis in the early 1980-ies. An improved model checking algorithm for CTL was implemented in the EMC model checker which were able to treat models with up to 100,000 states.

At fall of 1980-ies model checking researchers encouraged by polynomial complexity of model checking for CTL in finite models, and success of model checking verification experiments for systems of a moderate size had moved on further research topics, such that model checking for more expressive program logics (like the μ-calculus) in finite huge (10^{20} states and far beyond) and infinite models. As far as concerns a handling of huge finite models then an advantage of Ordered Binary Decision Diagrams (OBDD) [27] was realized in 1987-92 [28]. OBDDs provides a canonical form for boolean formulas that is often more compact then conjunctive or disjunctive normal form, and very efficient dynamic algorithms have been developed and implemented for manipulating them. The most popular modern model checker SMV was implemented by combining CTL model checking algorithm with symbolic representation of

[26] especially for those who have not a special logical background

finite models. The most recent versions of SMV for UNIX, Linux and Windows95 are free available for download [29].

A complete survey of program logics was out of scope of the paper. In contrast the paper introduces basic notions related to model checking problem for a powerful program logic called μ-Calculus, and presents a survey of other algorithmic problems for this logic. The basic ideas, definitions and theorems are illustrated by game examples usually presented as puzzles. All formal statements presented in the paper are of two kinds: the first one (*assertions*) are about concrete examples while the second one (*propositions*) are about some general facts.

References

[1] *ACM International Collegiate Programming Contest.* http://acm.baylor.edu/acmicpc/default.htm

[2] D. Harel, *First-Order Dynamic Logic.* Lecture Notes in Computer Science, v.68, 1979.

[3] D. Kozen, *Results on the Propositional Mu-Calculus.* Theoretical Computer Science, v.27, n.3, 1983, p.333-354.

[4] R. Cleaveland and M. Klain and B. Steffen, *Faster Model-Checking for Mu-Calculus.* Lecture Notes in Computer Science, v.663, 1993, p.410-422.

[5] E.A. Emerson and C.S. Jutla and A.P. Sistla, *On model-checking for fragments of Mu-Calculus.* Lecture Notes in Computer Science, v.697, 1993, p.385-396.

[6] S.A. Berezine and N.V. Shilov, *An approach to effective model-checking of real-time finite-state machines in Mu-Calculus.* Lecture Notes in Computer Science, v.813, 1994, p.47-55.

[7] R.S. Streett and E.A. Emerson, *An Automata Theoretic Decision Procedure for the Propositional Mu-Calculus.* Information and Computation, v.81, n.3, 1989, p.249-264.

[8] M.Y. Vardi, *Reasoning about the past with two-way automata'.* LNCS, v.1443, 1998, p.628-641.

[9] I. Walukiewicz, *A Complete Deduction System for the μ - Calculus.* Doctoral Thesis, Warsaw, 1993.

[10] N.V. Shilov, *Program schemata vs. automata for decidability of program logics.* Theoretical Computer Science, v.175, n.1, 1997, p.15-27.

[11] N.V. Shilov, *How to decide a complicated program logic without automata.* Manuscript (submitted to 25^{th} International Symposium on Mathematical Foundations of Computer Science).

[12] I. Walukiewicz, *On completeness of the μ-calculus.* IEEE Computer Society Press, Proc. of 8-th Ann. IEEE Symposium on Logic in Computer Science, 1993, p.136-146.

[13] R.A. Bull and K. Segerberg, *Basic Modal Logic.* Handbook of Philosophical Logic, v.II, Reidel Publishing Company, 1984 (1-st ed.), Kluwer Academic Publishers, 1994 (2-nd ed.), p.1-88.

[14] D. Harel, *Dynamic Logic.* Handbook of Philosophical Logic, v.II, Reidel Publishing Company, 1984 (1-st ed.), Kluwer Academic Publishers, 1994 (2-nd ed.), p.497-604.

[15] C. Stirling, *Modal and Temporal Logics.* Handbook of Logic in Computer Science, v.2, Claredon Press, 1992, p.477-563.

[16] D. Kozen and J. Tiuryn, *Logics of Programs.* Handbook of Theoretical Computer Science, v.B, Elsilver and The MIT Press, 1990, p.789-840.

[17] E.A. Emerson, *Temporal and Modal Logic.* Handbook of Theoretical Computer Science, v.B, Elsilver and The MIT Press, 1990, p.995-1072.

[18] R. Fagin and J.Y. Halpern and Y. Moses and M.Y. Vardi, Reasoning about Knowledge. MIT Press, 1995.

[19] R.S. Boyer and J.S. Moor, *A Computational Logic.* Academic Press, 1979.

[20] L.S. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987.

[21] J. Crow and S. Owre and J. Rushby and N. Shankar and M. Srivas, *A tutorial introduction to PVS.* http://www.csl.sri.com/sri-csl-fm.html

[22] E.M. Clarke and E.A. Emerson, *Design and Synthesis of synchronization skeletons using Branching Time Temporal Logic.* Lecture Notes in Computer Science, v.131, 1982, p.52-71.

[23] A. Pnueli, *Temporal Logic of Programs.* Theoretical Computer Science, v.13, n.1, 1981, p.45-60.

[24] Z. Manna and A. Pnueli, The temporal logic of Reactive and Concurrent Systems. Springer-Verlag, 1991.

[25] Z. Manna and A. Pnueli, Temporal verification of reactive systems: safety. Springer-Verlag, 1995.

[26] E.M. Clarke and O. Grumberg and D. Peled, Model Checking. MIT Press, 1999.

[27] R.E. Bryant, *Graph-Based algorithms for boolean function manipulation.* IEEE Trans. on Comp., v.35, n.6, 1986, p.677-691.

[28] K.L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem. Ph.D. Thesis, Carnegie Mellon University, 1992.

[29] http:/www-cad.eecs.berkley.edu/∼kenmcmil/smv/