Program specification paradigms and the SpecifyThis contest

Dmitry Kondratyev

A.P. Ershov Institute of Informatics Systems
Siberian Branch of the Russian Academy of Sciences

VerifyThis contest

Program verification contest

VerifyThis contest: since 2011

Huisman M., Monti R., Ulbrich M., Weigl A. The VerifyThis Collaborative Long Term Challenge. Lecture Notes in Computer Science. 2020. Volume 12345. pp. 246–260. DOI: `https://doi.org/10.1007/978-3-030-64354-6_10`

SpecifyThis

Program specification contest

Was first held in 2022

Organized by VerifyThis community

VerifyThis and SpecifyThis are based on different approaches: SpecifyThis is not contest of solutions of predefined problems.

SpecifyThis is a contest of researches about different approaches to defining software specifications.

SpecifyThis is a special track at ISoLA 2022 conference.

1 review paper and 7 research papers about SpecifyThis have been published in ISoLA 2022 proceedings.

Program specification paradigms

"Programming paradigms" concept has been introduced in the Turing Award lecture of Robert W. Floyd, entitled "The Paradigms of Programming" (1978).

"Program specification paradigms" concept has been introduced in the paper "SpecifyThis – Bridging Gaps Between Program Specification Paradigms" resulted from SpecifyThis contest (2022).

Ahrendt W., Herber P., Huisman M., Ulbrich M. SpecifyThis – Bridging Gaps Between Program Specification Paradigms. Lecture Notes in Computer Science. 2022. Volume 13701. pp. 3–6.
DOI: `https://doi.org/10.1007/978-3-031-19849-6_1`

Paper "SpecifyThis – Bridging Gaps Between Program Specification Paradigms" is a review of SpecifyThis contest.

Review of paper from SpecifyThis contest

Review of the paper "Deductive Verification Based Abstraction for Software Model Checking" from SpecifyThis contest (2022).

Amilon J., Lidström C., Gurov D. Deductive Verification Based Abstraction for Software Model Checking. Lecture Notes in Computer Science. 2022. Volume 13701. pp. 7–28. DOI: `https://doi.org/10.1007/978-3-031-19849-6_2`

Previous approach of model checking for big programs: translation whole program to TLA$^+$ specification and model checking obtained specification.

Architecture of a lot of big programs may be considered as reactive event-driven at high level and many proactive procedures at low-level.

Review of paper "Deductive Verification Based Abstraction for Software Model Checking"

New approach from the paper "Deductive Verification Based Abstraction for Software Model Checking": deductive verification of many proactive procedures in big program and using contracts of these procedures instead of program code to generate TLA^+ specification.

Specification obtained by new approach application contains less states than specification obtained by direct translation of whole program. Thus, specification obtained by new approach may be model checked faster.

Two examples from the paper "Deductive Verification Based Abstraction for Software Model Checking".

Two experiments from the paper "Deductive Verification Based Abstraction for Software Model Checking"

First experiment is simple file open-close example.

Full model has been obtained by translation whole program to TLA^+ specification.

Full model contains 45574 unique states.

Verification time in the case of full model is 51 seconds.

Abstract model has been obtained by deductive verification of program procedures and translation their contracts to TLA^+ specification.

Abstract model contains 841 unique states.

Verification time in the case of abstract model is 5 seconds.

Second experiment from the paper "Deductive Verification Based Abstraction for Software Model Checking"

Second experiment is a real software module taken from the automotive industry.

Full model has been obtained by translation whole program to TLA$^+$ specification.

Full model contains 46265 unique states.

Verification time in the case of full model is 12 seconds.

Abstract model has been obtained by deductive verification of program procedures and translation of their contracts to TLA$^+$ specification.

Abstract model contains 4552 unique states.

Verification time in the case of abstract model is 10 seconds.

Problems of the approach from the paper "Deductive Verification Based Abstraction for Software Model Checking"

Authors have not considered complexity of defining contracts of program procedures. Authors have not considered methods that can help to solve loop invariant problem in some cases.

Authors use WP plugin of Frama-C system for deductive verification. But authors have not considered complexity of deductive verification.

Nevertheless, authors have presented very interesting combination of deductive verification and model checking.

Problem: annotating standrard libraries has not been covered by SpecifyThis contest

Possible solution:

The C-lightVer system: annotating C standard library

Promsky A.V. C program verification: Verification condition explanation and standard library. Automatic Control and Computer Sciences. 2012. Volume 46. Issue 7. pp. 394–401. DOI: `https://doi.org/10.3103/S0146411612070127`

stdio.h: fopen

```
/*@ requires valid_string(filename) && valid_string(mode);
    assigns \nothing;
    behavior update:
        assumes mode == "r+b";
        ensures \result == res(filename);
    behavior truncate_or_update:
        assumes mode == "w+b";
        ensures \result == rew(filename);
    ...
    behavior failure:
        ensures \result == NULL;
    complete behaviors update, truncate_or_update, ...,
                            failure;
*/
FILE *fopen(const char restrict *filename,
            const char restrict *mode);
```

stdio.h: fread

```
/*@ requires \valid(stream) && size >= 0 && nmemb >= 0;
    assigns *ptr, stream;
    behavior wrong_read:
        assumes size == 0 || nmemb == 0;
        ensures \result == 0 && stream == \old(stream);
    behavior good_read:
        assumes size > 0 || nmemb > 0;
        ensures \exists int n;
        n == \max(length(\old(stream))/size, nmemb) &&
        stream == get(\old(stream), n, size) &&
        \forall 0 <= i < n; *ptr[i] == buf(stream)[i] &&
        \result == n;
    complete behaviors wrong_read, good_read;
*/
size_t fread(void * restrict ptr, size_t size,
                size_t nmemb, FILE * restrict stream);
```

stdio.h: EOF

```
/*@ axiomatic EOF {
        logic FILE* f;
        axiom eof-1: eof(left(f)) ==>
                             buf(left(f)) == \omega;
        axiom eof-2: eof(f) <==> right(f) == empty_file;
      ...
*/
```

Copying file

```
#include <stdio.h>

/*@ requires \nothing;
    assigns from, to, Buffer;
    ensures \value == 1 ||
        \value == 0 && (file(from) == file(to)) && eof(from) && eof(to);
*/
int main(){
    FILE *from, *to;
    char Buffer;

    if ((from = fopen("example.txt", "r+b")) == NULL) return 1;
    if ((to = fopen("example.bak", "w+b")) == NULL) return 1;

    /*@ invariant left(from) == file(to) &&
        (eof(from) ==> fbuf(from) == \omega) */
    while(fread(&Buffer, 1, 1, from) != 0)
        fwrite(&Buffer, 1, 1, to);
    return 0;
}
```

Problem: nature of program specification paradigms

Two approaches of defining program specifications:

- Set theoretic approach.
- Executable specifications.

Example

Verification benchmark "Java program verification challenge"

`negate_first` program from this benchmark

Implementation of `negate_first` in C programming language

```
void negate_first(int n, int* a) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] < 0) {a[i] = -a[i]; break;}}}
```

Problems illustrated by this program:

1. Array update
2. Possible `break` execution

Example: set theoretic approach

$$\begin{aligned}
\text{pre} \;:\;& \exists \mathit{old} : \texttt{int[]}.\texttt{a} \neq \text{null} \wedge \\
&\qquad\qquad\quad \texttt{a} = \mathit{old})
\end{aligned}$$

$$\begin{aligned}
\text{post:}\;& \forall i.\ (0 \leq i \leq \texttt{Length} \Longrightarrow \\
&\quad (\mathit{old}[i] < 0 \wedge \\
&\quad (\forall j.\ 0 \leq j < i \Rightarrow \mathit{old}[j]) \geq 0)) \Rightarrow \\
&\qquad\qquad\qquad a[i] = -\mathit{old}[i] \wedge \\
&\quad \mathit{old}[i] \geq 0 \Rightarrow a[i] = \mathit{old}[i]
\end{aligned}$$

$$\begin{aligned}
\text{inv} \;:\;& 0 \leq \texttt{i} \leq \texttt{Length} \wedge \\
&(\forall j.\ 0 \leq j < \texttt{i} \Rightarrow \\
&\quad (\,a[j] \geq 0 \wedge \\
&\quad a[j] = \mathit{old}[j]
\end{aligned}$$

Example: executable specifications

`negate_first`: precondition:

$$(a_0 = a) \wedge (0 < n) \wedge (n \leq length(a_0))$$

`negate_first`: postcondition:

$$(\neg found\_negative(n,\ a_0) \rightarrow a = a_0) \wedge \\ (found\_negative(n,\ a_0) \rightarrow a = update(a_0, \\ count\_index(n,\ a_0), -a_0[count\_index(n,\ a_0)]))$$

found_negative predicate checks presence of a negative element in the array. *count-index* function computes index of the first negative element in the case of its presence in the array.

Recursive function may be generated instead of loop invariant in the case of finite iteration:
Kondratyev D.A., Maryasov I.V., Nepomniaschy V.A. The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination. Automatic Control and Computer Sciences. 2019. Volume 53. Issue 7. pp. 653–662. DOI:
`https://doi.org/10.3103/S0146411619070101`

References

1. Ahrendt W., Herber P., Huisman M., Ulbrich M. SpecifyThis – Bridging Gaps Between Program Specification Paradigms. Lecture Notes in Computer Science. 2022. Volume 13701. pp. 3–6. DOI: `https://doi.org/10.1007/978-3-031-19849-6_1`
2. Amilon J., Lidström C., Gurov D. Deductive Verification Based Abstraction for Software Model Checking. Lecture Notes in Computer Science. 2022. Volume 13701. pp. 7–28. DOI: `https://doi.org/10.1007/978-3-031-19849-6_2`
3. Huisman M., Monti R., Ulbrich M., Weigl A. The VerifyThis Collaborative Long Term Challenge. Lecture Notes in Computer Science. 2020. Volume 12345. pp. 246–260. DOI: `https://doi.org/10.1007/978-3-030-64354-6_10`
4. Promsky A.V. C program verification: Verification condition explanation and standard library. Automatic Control and Computer Sciences. 2012. Volume 46. Issue 7. pp. 394–401. DOI: `https://doi.org/10.3103/S0146411612070127`
5. Kondratyev D.A., Maryasov I.V., Nepomniaschy V.A. The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination. Automatic Control and Computer Sciences. 2019. Volume 53. Issue 7. pp. 653–662. DOI: `https://doi.org/10.3103/S0146411619070101`

Program specification paradigms and the SpecifyThis contest

Dmitry Kondratyev

A.P. Ershov Institute of Informatics Systems
Siberian Branch of the Russian Academy of Sciences