Deductive Verification via the Debug Adapter Protocol

Gidon Ernst Johannes Blau
Ludwig-Maximilians-Universität
Munich, Germany
gidon.ernst@lmu.de

Toby Murray
University of Melbourne
Melbourne, Australia
toby.murray@unimelb.edu.au

We propose a conceptual integration of deductive program verification into existing user interfaces for software debugging. This integration is well-represented in the "Debug Adapter Protocol", a widely-used and generic technology to integrate debugging of programs into development environments. Commands like step-forward and step-in are backed by steps of a symbolic structural operational semantics, and the different paths through a program are readily represented by multiple running threads of the debug target inside the user interface. Thus, existing IDEs can be leveraged for deductive verification debugging with relatively little effort. We have implemented this scheme for SecC, an auto-active program verifier for C, and discuss its integration into Visual Studio Code.

1 Introduction

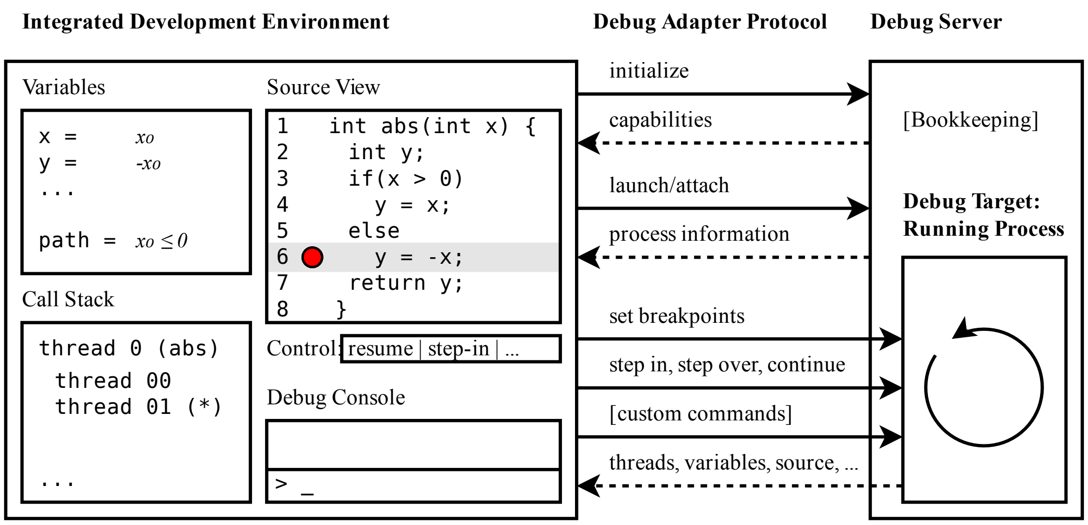
Deductive verification of programs with respect to strong requirements relies on human proof engineering effort. The user has to provide the primary correctness specifications (e.g. procedure contracts), as well as auxiliary annotations (e.g. loop invariants), key lemmas, and other proof hints. This is much facilitated by modern Integrated Development Environments (IDEs) for formal methods tools and by advances in verification technology. Over the recent years, the term "push-button" has been coined, suggesting perhaps that proof automation is nowadays good enough to not burden the user with internal details. However, proofs for code that is already correctly annotated are fundamentally different from the typical trial an error to find these. The ability to dig into the causes of a verification failure is not just nice-to-have—it is crucial have access to as much information as possible.

Related Work: To that end, state-of-the-art IDEs for formal development offer different features: Dafny [13], for example, highlights those annotations which cannot be proven, and the Boogie Verification Debugger [12] gives structured access to concrete counterexamples. In VeriFast [11], one can inspect the symbolic state and a tree representation of the paths explored. Why3 [4] shows the generated verification conditions in a nice and structured way and offers interactive as well as automatic proof steps. In contrast, general purpose interactive theorem provers like Isabelle/PIDE [19], KIV [6], Rodin [18], and PVS [14] (to name a few with a sophisticated user interface), tend to expose proof internals in detail. The latter paper [14] nicely compares some popular formal IDEs and their features.

The approaches mentioned are rather different from the experience of traditional, concrete debugging of programs in IDEs like Eclipse, IntelliJ, or Visual Studio Code, where the main features are breakpoints, single stepping, and inspection of data at runtime. Recently, loose coupling between IDEs and language-specific toolchains has become popular, based on the Language Server Protocol (LSP)[1] and the Debug Adapter Protocol (DAP).[2] LSP is used in several of the above mentioned formal IDEs. For VDM a debugger for concrete model executions has been developed [17] using the DAP. The KeY Symbolic Execution Debugger [9] is a feature-rich tool for Java verification built on the Eclipse platform.

[1] https://microsoft.github.io/language-server-protocol
[2] https://microsoft.github.io/debug-adapter-protocol

Submitted to:
F-IDE 2021



```
x =     x0
y =     -x0
...

path =  x0 ≤ 0
```

```
1   int abs(int x) {
2     int y;
3     if(x > 0)
4       y = x;
5     else
6       y = -x;
7     return y;
8   }
```

```
thread 0 (abs)
  thread 00
  thread 01 (*)

...
```

Figure 1: Schematics of the Debug View inside an IDE.

Contribution: In this paper, we propose and describe an integration of deductive program verification into general purpose IDEs on top of the DAP, which can be used to interactively navigate to specific parts of a proof. The first contribution is an abstract characterization of how that integration works conceptually (Sec. 2). As the second contribution, we briefly describe an ongoing implementation effort of this scheme for SecC, an autoactive verifier for correctness and security of C programs (Sec. 3) inside Visual Studio Code. Our conclusion is that implementing debugging for an existing verification tool that is based on symbolic execution is relatively easy and straight-forward when using the DAP.

2 Conceptual Model of a Debug Server

In this section we develop a conceptual model of a debug server. The server exposes certain operations to the IDE and maintains the state of running processes that are being debugged (the debug targets). Fig. 1 depicts the integration between the IDE (as the client, on the left) and a debug server (on the right) in terms of the graphical front-end of the IDE and the messages exchanged between the two components. Typically, the debugging perspective of an IDE shows the program's source code and the breakpoints within. Moreover, there is a view that shows a current state of the program, in terms of runtime values of variables (top-left), possibly organized according to the structure of lexical scopes.

In Fig. 1 we anticipate a debug target that is executed symbolically, such that the program variables `x` and `y` are assigned symbolic expression over logical variables. The example program, a function `abs` to compute the absolute value of parameter `x`, has been halted at a breakpoint in line 6, consequentially, variable `y` is set to $-x_0$ wrt. logical variable x_0 capturing the initial value of `x`. The synthetic variable `path` shows the current path condition, a formula that captures the constraints for reaching the respective code location, here the negated test of the conditional. Below, the two possible branches through function `abs` are represented as execution thread with identifiers `00` and `01`, that are subsumed by a parent thread `0`. The views in the window on the left are populated from data that is requested by the IDE from the server.

The simple, semi-formal model below omits inessential details of realistic languages (e.g., memory, see [2, 5, 8, 11] here for details). The key point is that the ideas presented here translate to any formal system that can be described by a symbolic structural operational semantics [15].

Representation of States and Execution Steps. The state associated with a debug target primarily consists of a *configuration* c, that describes the progress of the execution. Configurations bundle up programs p, defined over the set of program variables X, with *symbolic assignment* $s\colon X \to E$ from program variables to logical expressions E and a *path condition* φ as a logical formula that describes the branches taken so far. Here, we capture execution using three kinds of configurations:

$$\text{Configurations } c ::= \big(s,\varphi,p_1;\ldots;p_k\big) \mid \big(c_1 \parallel \cdots \parallel c_n\big) \mid \big(\varphi \wedge \neg\psi, \lightning\big)$$

Sequential compositions execute programs $p_1;\ldots;p_k$ from symbolic assignment s and path condition φ, where for $k=0$ this execution branch terminates. *Parallel compositions* of configurations capture multiple branches that arise e.g. from conditionals. *Proof obligations*, marked by $\lightning$, verify that condition ψ holds for the path condition φ, where $\varphi \wedge \neg\psi$ **unsat** denotes that the obligation discharges successfully. In practice, such proof obligations would be annotated with additional contextual information.

Symbolic execution proceeds by unwinding a small-step relation $c \xrightarrow{\sigma} c'$ between configurations according to a *schedule* σ, which reflects the user's choice where to step next. Recursively, the first entry in the schedule, for $1 \le i \le n$, resolves which part of a parallel configuration performs the next step:

$$\big(c_1 \parallel \cdots c_i \cdots \parallel c_n\big) \xrightarrow{\langle i\rangle\cdot\sigma} \big(c_1 \parallel \cdots c_i' \cdots \parallel c_n\big) \qquad \text{if} \qquad c_i \xrightarrow{\sigma} c_i'$$

In the base case, execution of sequential composition steps with the empty schedule $\sigma = \langle\rangle$. Nondeterminism is captured by producing multiple branches, for example:

$$\begin{aligned}
\big(s,\varphi,\textbf{assume } e;\ldots\big) &\xrightarrow{\langle\rangle} \big(s,\varphi \wedge s(e),\ldots\big)\\
\big(s,\varphi,\textbf{assert } e;\ldots\big) &\xrightarrow{\langle\rangle} \big(s,\varphi \wedge s(e),\ldots\big) \parallel \big(\varphi \wedge \neg s(e),\lightning\big)\\
\big(s,\varphi,x := e;\ldots\big) &\xrightarrow{\langle\rangle} \big(s[x \mapsto s(e)],\varphi,\ldots\big)\\
\big(s,\varphi,\textbf{if } e \textbf{ then } p_1 \textbf{ else } p_2;\ldots\big) &\xrightarrow{\langle\rangle} \big(s,\varphi \wedge s(e),p_1;\ldots\big) \parallel \big(s,\varphi \wedge \neg s(e),p_2;\ldots\big)
\end{aligned}$$

where $s(e)$ denotes evaluation of program expression e in symbolic state s. Of course, configurations with an unsatisfiable path condition or empty sequential compositions can be soundly dropped from their surrounding parallel context (which our implementation does eagerly), e.g., the proof obligation from an **assert** when $s(e)$ follows from φ, or either of the branches of an **if** in case it is unreachable. Loops can be unwound interactively (not discussed here) or summarized by invariants I as shown

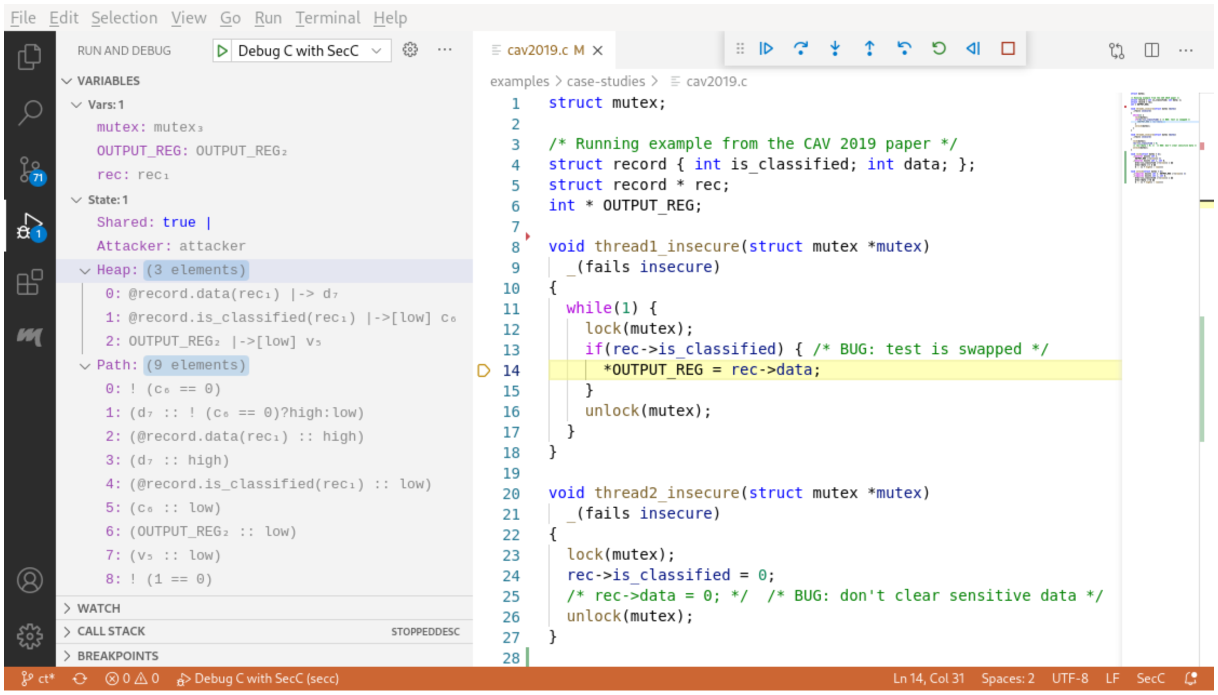
$$\big(s,\varphi,\textbf{while } e \textbf{ do } p;\ldots\big) \xrightarrow{\langle\rangle} \big(s,\varphi \wedge \neg s(I);\lightning\big) \parallel \big(\hat{s},\varphi \wedge \hat{s}(e) \wedge \hat{s}(I),p;\textbf{assert } I\big) \parallel \big(\hat{s},\varphi \wedge \neg\hat{s}(e) \wedge \hat{s}(I);\ldots\big)$$

where the latter produces three successor configurations, 1) to prove invariant I initially, 2) to preserve I over an arbitrary iteration, where $\hat{s} = s[x_1 \mapsto x_1',\ldots,x_n \mapsto x_n']$ introduces fresh logical variables $x_1',\ldots,x_n'$ for the program variables modified in loop body p, and 3) to continue with the code after the loop.

Initialization. The initial symbolic configurations for an entire translation unit is exemplified below. A C file has a list of global variables g_i that are to be initialized in sequence, which we represent as a program g such that $g \equiv \{g_1 := e_1;\ldots;g_n := e_n\}$. A top-level procedure declaration of the form $f(y_1,\ldots,y_m)\ \{q\}$, with precondition/postcondition pair P,Q, parameters y_j, and implementation q can be mapped onto a configuration $c_f = \big(s,\mathit{true},\textbf{assume } P;q;\textbf{assert } Q\big)$ where $s_f = [g_i \mapsto x_i, y_j \mapsto y_j,\ldots]$ initializes all these variables in the respective scopes to fresh logical ones. The verification of the `main` procedure may additionally assume that the globals were just initialized, which can be represented as $c_{\texttt{main}} = \big(s_{\texttt{main}},\mathit{true},g;q\big)$ where the body q of `main` is prefixed by the sequence g of global initializers.

Dispatching Requests in the Debug Server. We outline how to realize the operations that implement the main requests issued by the IDE in reference to a top-level configuration C. In addition, we track a set B of breakpoints, which are program locations, and we denote the location of a program by $loc(p)$. In the following, we denote by $\sigma(C) = \big(s, \varphi, \dots\big)$ the sub-configuration triggered by σ, which is necessarily a sequential one in our simple model.

- The request `Launch`(file) for program stored in *file* produces the initial configuration as a big parallel composition $C := \big(c_{f_1} \parallel \cdots \parallel c_{f_n} \parallel c_{\texttt{main}}\big)$ where the respective sub-configurations are constructed wrt. the procedure declarations and globals as outlined above.
- The request `GetThreads` returns the currently running "threads", which are those parts of parallel compositions that can be stepped. This information can be represented in terms of possible schedules for the next step, i.e., the set `GetThreads` returns $\{\sigma \mid \exists\, c'.\ C \xrightarrow{\sigma} c'\}$. As suggested in Fig. 1, the hierarchical structure of these identifiers can be exploited for more complex operations, such as stepping all branches that share a common prefix schedule.
- The request `GetVariables`(σ) returns the variable assignment s stored in the sequential configuration $\sigma(C)$ as remarked above. In our implementation for SecC, we use this request to add synthetic variables, such as $\texttt{path} \mapsto \varphi$ where φ is the path constraint of configuration $\sigma(C)$ for that thread, and we additionally include a representation of the symbolic heap and information about the attacker level for security proofs.
- The request `SetBreakpoints`(b) simply sets $B := b$ to the set of specified breakpoints. The protocol knows not only source breakpoints but also function breakpoints (triggered by calls), and exception breakpoints (when an exception is thrown), which we have not used so far.
- The requests `Next`(σ) and `StepIn`(σ) execute a single transition of a given thread σ according to the rules for $C \xrightarrow{\sigma} c'$, where c' is taken as the next state, $C := c'$. The difference between these two commands in a concrete execution is that the first proceeds over function calls in one atomic step, whereas the second jumps into functions. This behavior can be mirrored in a modular deductive verifier, where `Next` dispatches such a call using a given function contract, whereas `StepIn` inlines the call and disregards such contracts, similarly for proving loops with invariants or unfolding a finite number of iterations Our implementation supports `Next` only so far, but e.g. KIV and KeY support both interactions in their respective GUIs.
- The request `Continue`(σ) executes multiple transition of a given thread σ until the corresponding configuration $\sigma(C) = \big(s, \varphi, p; \dots\big)$ with $loc(p) \in B$, i.e., the program execution has reached a breakpoint, or until $\sigma(C) = \big(s, \varphi\big)$ is final with no residual program.
- The request `StepBack`(σ) undoes the latest corresponding transition (or sequence of transitions) of a particular sub-configuration. This can be realized by keeping a history $c_0, c_1, \dots$ of previous top-level configurations, which in practice is facilitated by the fact that often tools are implemented in functional languages and do not use destructive modification of states.
- The request `Evaluate`(σ, e) is issued to inspect the value of arbitrary expressions e within a state. The response consists of evaluation $s(e)$ wrt. the logical variables from $\sigma(C) = \big(s, \varphi, \dots\big)$. Since the format of the result is just a string, further information can be computed with the help of a solver and included in the response, such as whether $\varphi \implies s(e)$ holds when e is boolean, or concrete values for e and its free variables if $\varphi \wedge s(e)$ **sat**.


File Edit Selection View Go Run Terminal Help
RUN AND DEBUG
Debug C with SecC
cav2019.c M
examples > case-studies > cav2019.c
WATCH
CALL STACK
STOPPEDDESC
BREAKPOINTS
ct*
⊗ 0 ⚠ 0
Debug C with SecC (secc)
Ln 14, Col 31 Spaces: 2 UTF-8 LF SecC

```
VARIABLES
  Vars: 1
    mutex: mutex₃
    OUTPUT_REG: OUTPUT_REG₂
    rec: rec₁
  State: 1
    Shared: true |
    Attacker: attacker
    Heap: (3 elements)
      0: @record.data(rec₁) |-> d₇
      1: @record.is_classified(rec₁) |->[low] c₆
      2: OUTPUT_REG₂ |->[low] v₅
    Path: (9 elements)
      0: ! (c₆ == 0)
      1: (d₇ :: ! (c₆ == 0)?high:low)
      2: (@record.data(rec₁) :: high)
      3: (d₇ :: high)
      4: (@record.is_classified(rec₁) :: low)
      5: (c₆ :: low)
      6: (OUTPUT_REG₂ :: low)
      7: (v₅ :: low)
      8: ! (1 == 0)
```

```c
struct mutex;

/* Running example from the CAV 2019 paper */
struct record { int is_classified; int data; };
struct record * rec;
int * OUTPUT_REG;

void thread1_insecure(struct mutex *mutex)
  _(fails insecure)
{
  while(1) {
    lock(mutex);
    if(rec->is_classified) { /* BUG: test is swapped */
      *OUTPUT_REG = rec->data;
    }
    unlock(mutex);
  }
}

void thread2_insecure(struct mutex *mutex)
  _(fails insecure)
{
  lock(mutex);
  rec->is_classified = 0;
  /* rec->data = 0; */  /* BUG: don't clear sensitive data */
  unlock(mutex);
}

```

Figure 2: Debugging an insecure concurrent program in VSCode with SecC.

3 Debug Server Implementation for SecC

SecC[3] is an autoactive verifier programs for functional correctness and security of C programs. It is built around Security Concurrent Separation Logic (SecCSL) [5], which can express value-dependent security properties of concurrent heap-manipulating programs. The tool is currently used to verify a variety of small case studies. Internally, SecC is based on a symbolic execution engine for Separation Logic [2], which is similar to that of VeriFast (the latter is described nicely in [11]). Thus, SecC lends itself to the approach outlined in Sec. 2.

SecC is implemented in the Scala programming language,[4] which runs on the Java Virtual Machine so that we can rely on the mature library `lsp4j`,[5] which fully abstracts the DAP (and also the Language Server Protocol, LSP) in Java. Creating a debug server with `lsp4j` simply amounts to implementing a particular Java interface whose operations correspond to DAP requests, all protocol data structures are available as Java classes, too. Within the client, here Visual Studio Code, some additional effort has to be spent to register the respective language extension, and to provide the necessary hooks that spin up the debug server with an appropriate configuration.

The integration was developed as a VS Code extension in a Master thesis project by the second author over the course of roughly six months, resulting in about 1000 LoC of Scala for the server and 400 LoC of TypeScript for the extension, albeit getting a working initial version for a toy language was a matter of a few days. In addition to debugging, the VS Code extension provides syntax highlighting, verify-on-save, a debug console that lets one inspect the current state and evaluate expressions, and a

[3] `https://covern.org/secc`
[4] `https://bitbucket.org/covern/secc/`
[5] `https://github.com/eclipse/lsp4j`

graphical view of the symbolic execution tree. Note that these extended features cannot be realized via the DAP alone but require LSP functionality as well as the extension facilities inside VS Code. The extension is currently available in binary form (file `secc-0.2.0.vsix` on Bitbucket) and will be released as open source soon.

A screenshot of the debugging perspective is shown in Fig. 2, for an example from [5, Sec. 2] that has a defect. The symbolic store s appears under "Vars" and the synthetic variables, including φ as "Path" under "State" on the left; in addition there is a list of symbolic heap chunks describing the memory. Stepping line 14 with the controls shown at the top-right subsequently leads to a verification failure. Informally, the code writes a secret value to a public memory location `OUTPUT_REG`. This can be recognized from the data shown as follows: Value d_7 stored in `rec->data` (item 1 under Heap) is classified information (item 3: $d_7 :: \texttt{high}$ under "Path"), whereas the memory location `OUTPUT_REG` is public (item 2: $\texttt{OUTPUT\_REG}_2 \mapsto [\texttt{low}] \ldots$ under "Heap").

While we have not done a systematic evaluation or larger case study in this new SecC IDE yet, the integration was useful for the second challenge of the VerifyThis 2021 competition [7]. During the competition, it was helpful to investigate the symbolic states while developing the correctness proof interactively, for example to determine some subtle arithmetic constraints, or to debug the unfolding/folding of memory predicates.

4 Discussion & Conclusion

We have shown a concept to embed symbolic execution engines into existing IDEs for interactive debugging via the established Debug Adapter Protocol (Sec. 2) Our implementation for the autoactive verifier SecC proved to be straight-forward and low effort (Sec. 3).

The approach inherits as a limitation the exponential path explosion from the nondeterministic execution when branches are not joined. This is a problem with many conditionals in sequence, but we have not yet been impeded by this limitation. At a conceptual level, it is not entirely clear how to remedy the approach proposed here with ideas that defer splitting up branches to the SMT solver as it is done in Boogie [1] and generally in Horn clause verifiers [3]. Our approach can nevertheless complement such ideas, e.g., by stepping selectively only that thread corresponding to a particular procedure or branch of interest to investigate precisely those proof obligations that fail with the more efficient encoding of [1].

The vanilla formulation of Sec. 2 repeatedly traverses the tree of configurations down to the currently executing leaf and rebuilds it on the way back. Zippers [10] avoid this (probably perceived) inefficiency, and initial experiments with changing the implementation suggests that Zippers lead to quite elegant code, too. This idea has indeed been followed before [16].

Overall, we think that the proposed approach is general and flexible enough, to be used to retrofit existing verification tools and languages with a symbolic interactive debugger. By relying on existing infrastructure, such an undertaking is well within the reach of short-term projects. By relying on established interaction paradigms, the approach brings software development practice and program verification a step closer together. For future work we would like to investigate how to integrate concrete symbolic and concrete debugging techniques, and we plan to conduct a larger case study inside the SecC IDE to evaluate the benefits of the proposed approach in practice.

Acknowledgement. We thank the reviewers for their suggestions to improve the presentation.

References

[1] Mike Barnett & K Rustan M Leino (2005): *Weakest-precondition of unstructured programs*. In: *Proc. of Program Analysis for Roftware Tools and Engineering (PASTE)*, pp. 82–87.

[2] Josh Berdine, Cristiano Calcagno & Peter W O'Hearn (2005): *Symbolic execution with Separation Logic*. In: *Proc. of Asian Symposium on Programming Languages and Systems (APLAS)*, Springer, pp. 52–68.

[3] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan & Andrey Rybalchenko (2015): *Horn clause solvers for program verification*. In: *Fields of Logic and Computation II*, Springer, pp. 24–51.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond & Andrei Paskevich (2011): *The Why3 platform*. Technical Report, LRI, CNRS & Univ. Paris-Sud & INRIA Saclay.

[5] G. Ernst & T. Murray (2019): *SecCSL: Security Concurrent Separation Logic*. In: *Proc. of Computer Aided Verification (CAV)*, *LNCS* 11562, Springer, pp. 208–230.

[6] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg & W. Reif (2015): *KIV—Overview and VerifyThis competition*. *Software Tools for Technology Transfer (STTT)* 17(6), pp. 677–694.

[7] Gidon Ernst, Marieke Huisman, Wojciech Mostowski & Mattias Ulbrich (2019): *VerifyThis–verification competition with a human factor*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 176–195.

[8] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun & Philippa Gardner (2020): *Gillian, part i: a multi-language platform for symbolic execution*. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 927–942.

[9] Martin Hentschel, Reiner Hähnle & Richard Bubel (2016): *The interactive verification debugger: Effective understanding of interactive proof attempts*. In: *Proc. of Automated Software Engineering (ASE)*, pp. 846–851.

[10] Gérard Huet (1997): *The zipper*. *Journal of functional programming* 7(5), pp. 549–554.

[11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx & Frank Piessens (2011): *VeriFast: A powerful, sound, predictable, fast verifier for C and Java*. In: *Proc. of NASA Formal Methods (NFM)*, Springer, pp. 41–55.

[12] Claire Le Goues, K Rustan M Leino & Michał Moskal (2011): *The Boogie Verification Debugger*. In: *Proc. of Software Engineering and Formal Methods (SEFM)*, Springer, pp. 407–414.

[13] K. Rustan M. Leino & Valentin Wüstholz (2014): *The Dafny Integrated Development Environment*. In: *Proc. of Formal Integrated Development Environment (F-IDE)*, *EPTCS* 149, pp. 3–15.

[14] Paolo Masci & César A. Muñoz (2019): *An Integrated Development Environment for the Prototype Verification System*. In: *Proc. of Formal Integrated Development Environment (F-IDE)*, *EPTCS* 310, pp. 35–49.

[15] Gordon D Plotkin (2004): *The origins of structural operational semantics*. *The Journal of Logic and Algebraic Programming* 60, pp. 3–15.

[16] Norman Ramsey & Joao Dias (2006): *An applicative control-flow graph based on Huet's zipper*. *Electronic Notes in Theoretical Computer Science* 148(2), pp. 105–126.

[17] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo & Peter Gorm Larsen (2021): *Visual Studio Code VDM Support*. In: *Proc. of Overture Workshop*, p. 35.

[18] Laurent Voisin & Jean-Raymond Abrial (2014): *The rodin platform has turned ten*. In: *Proc. of Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)*, Springer, pp. 1–8.

[19] Makarius Wenzel (2018): *Isabelle/PIDE after 10 years of development*. In: *Proc. of User Interfaces for Theorem Provers (UITP)*.