Gudmund Grov
Thomas Wies (eds.)

Contributions of

the 4th International Workshop on Invariant Generation (WING 2012)

A satellite event of IJCAR
Manchester, 2012

Preface

This report contains the contributions from the 4th International Workshop on Invariant Generation (WING), held on June 30 2012 in Manchester (UK) as a satellite event of IJCAR 2012.

The scope of the workshop is providing, debugging, and verifying auxiliary invariant annotations of programs - a key problem in program verification. This includes areas such as invariant generation techniques by Groebner bases, quantifier elimination, and algorithmic combinatorics, which can be used in conjunction with model checking, theorem proving, static analysis, and abstract interpretation.

The workshop consisted of 2 invited talks:

> Aditya Nori (Microsoft Research, India). *Specification Inference and Invariant Generation: A Machine Learning Perspective*
>
> Antoine Miné (CNRS & École normale supérieure, France). *Invited Astrée tutorial*

one full paper:

> Antoine Miné*. Abstract Domains for Bit-Level Machine Integer and Floating-point Operations*

and nine presentation-only abstracts - including work in progress, new ideas, tools under development and work by PhD students. We accepted presentations of papers submitted elsewhere. The following presentations were given:

> Friedrich Gretz, Joost-Pieter Katoen and Annabelle McIver*. Prinsys - A Software Tool for the Synthesis of Probabilistic Invariants*
>
> Alexei Iliasov*. Augmenting formal development with use case reasoning*
>
> Daniel Larraz, Enric Rodriguez-Carbonell and Albert Rubio. *SMT-Based Array Invariant Generation*
>
> Mengjun Li. *Formal Characterization and Verification of Loop Invariant Based on Finite Difference*
>
> Lamia Labed Jilani, Wided Ghardallou and Ali Mili. *Conclusive Proofs of While Loops Using Invariant Relations*
>
> Rajiv Murali and Andrew Ireland**.** *E-SPARK: Automated Generation of Verifiable Code from Formally Verified Designs*
>
> Pierre-Loic Garoche, Temesghen Kahsai and Cesare Tinelli. *Invariant Stream Generators using Automatic Abstract Transformers based on a Decidable Logic*.
>
> Ott Tinn. *Faster Automatic Test Case Generation*

This report contains the full paper and all presentation-only abstracts presented at the workshop.

The WING 2012 programme committee consisted of the following:

Clark Barrett (New York University, USA)
Nikolaj Bjorner (Microsoft Research, USA)
Gudmund Grov (Heriot-Watt University, UK)
Ashutosh Gupta (IST Austria)
Bart Jacobs (KUL, Belgium)
Moa Johansson (Chalmers, Sweden)
Laura Kovacs (TU Vienna, Austria)
David Monniaux (VERIMAG, France)
Enric Rodrguez Carbonell (UPC, Catalonia, Spain)
Helmut Veith (TU Vienna, Austria)
Thomas Wies (New York University, USA)

The workshop was sponsored by Formal Methods Europe (http://www.fmeurope.org) which covered the registration fees for the seven PhD students attending:

Parts of the workshop was run in conjunction with the workshop on Automated Theory eXploration (ATX). More information is available from the WING 2012 webpage:

http://cs.nyu.edu/acsys/wing2012/

Edinburgh & New York, September, 2012
Gudmund Grov
Thomas Wies

Inferring Loop Invariants Dynamically

Juan Pablo Galeotti and Andreas Zeller
{galeotti, zeller}@cs.uni-saarland.de

Saarland University – Computer Science, Saarbrücken, Germany

There is extensive literature on inferring loop invariants *statically* (i.e. without explicitly executing the program under analysis). We report on a new *dynamic* technique for inferring loop invariants based on the invariant detector Daikon [2]. Unlike InvGen [4], this new technique follows a counter example guided approach for refining candidate loop invariants. Let us consider the following annotated program for multiplying 16 bit integers in the left column:

```
_(requires 0<=x<65535)
_(requires 0<=y<65535)
_(ensures \result==x*y) {
  mult = i = 0;
  while (i<y) {
    mult+=x; i++;
  }
  return mult;
}
```

```
// Candidate Loop Invariants
#1  x one of { 1, 1316 }
#2  y one of { 1, 131 }
#3  i >= 0
...
#9  i <= y
#10  i == (mult / x)
#11  mult == (x * i)
```

Our approach starts by finding new test cases using the search-based test suite generator EvoSuite [3]. Then, the dynamic invariant detector collects 11 different loop invariant candidates (excerpt shown on the right), which we feed to the static verifier VCC [1].

Since the conjunction of all candidates under-approximates the loop invariant, the static verifier fails. Then, EvoSuite guides the generation of new test inputs using the static verifier's error model. The invariant detector synthesizes new candidates (ruling some of them out), which are fed to VCC. This refinement continues until VCC successfully verifies the program (using only candidates `#9` and `#11`).

The combination of test case generation and Daikon opens the potential for inferring loop invariants even for nontrivial programs. Current challenges include the static verification itself, as well as refining the candidate loop invariants.

The main challenge, however, will be to find appropriate *patterns* for the most recurrent loop invariants: Daikon itself is limited to at most three related variables, and we will have to expand the search space considerably. Finally, we are also looking for benchmarks such that we can compare against other existing automatic loop invariant detectors, such as InvGen [4].

References

1. Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., and Tobies S., *VCC: A Practical System for Verifying Concurrent C*. TPHOLs, 2009.
2. Ernst M., Cockrell J., Griswold W., and Notkin D. *Dynamically discovering likely program invariants to support program evolution*. IEEE TSE, 27(2), 2002.
3. Fraser G., and Arcuri A., *EvoSuite: Automatic Test Suite Generation for Object-Oriented Software*. ESEC/FSE, 2011.
4. Gupta A., Majumdar R., and Rybalchenko A., *From Tests To Proofs*. TACAS, 2009.

PRINSYS – a Software Tool for the Synthesis of Probabilistic Invariants ⋆

Friedrich Gretz[1,2], Joost-Pieter Katoen[1], and Annabelle McIver[2]

[1] RWTH Aachen University, Germany
[2] Macquarie University, Sydney, Australia

Abstract

We are interested in aiding correctness proofs for probabilistic programs, i.e. WHILE programs, enriched with a probabilistic choice operator "$[p]$" that executes the left alternative with probability p and the right alternative with $1-p$. There are tools for non-probabilistic programs that generate invariants for verification purposes [2, 1]. For probabilistic programs the existing tools rely on model checking and are limited to finite-state systems or do not allow parameters [6, 4, 3]. One tool that is based on abstract interpretation is mentioned in [7] but its merits cannot be assessed[3]. Our novel tool PRINSYS[4] is based on some of the ideas in [5] and aids the verification of *probabilistic programs with loops and real valued variables*. We can derive quantitative properties of a given program by reasoning over program annotations in the style of Hoare logic. Our annotations are generalised to take account of the quantitative properties of probabilistic programs. PRINSYS computes sound annotations for probabilistic while loops.

Our approach to generate invariants is constraint-based. Given a template, i.e. the shape of the desired invariant, and a loop we generate constraints in terms of inequalities between piecewise functions. These constraints are then transformed into universally quantified first-order formulas and solved using off the shelf tools. The result is the description of all invariant instances of the template that was given by the user. Using this semi-automatical approach iteratively, a user can construct an invariant that helps him to prove a property of the program.

In the workshop we would like to report about the latest ongoing work and show some use cases of PRINSYS.

References

1. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear Invariant Generation Using Non-linear Constraint Solving. In: CAV. pp. 420–432 (2003)

⋆ This work is supported by the EU FP7 project CARP, DFG research training group Algosyn, Australian Research Council DP1092464 and NWO visitor grant 040.11.302.

[3] ABSINTHE appears to be unmaintained and no documentation or download could be found as of this writing.

[4] Download the tool at: http://www-i2.informatik.rwth-aachen.de/prinsys/.

2. Gupta, A., Rybalchenko, A.: InvGen: An Efficient Invariant Generator. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 634–640. Springer (2009)
3. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Pass: Abstraction refinement for infinite probabilistic models. In: Esparza, J., Majumdar, R. (eds.) TACAS. Lecture Notes in Computer Science, vol. 6015, pp. 353–357. Springer (2010)
4. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic Reachability for Parametric Markov Models. STTT 13(1), 3–19 (2011)
5. Katoen, J.P., McIver, A., Meinicke, L., Morgan, C.: Linear-Invariant Generation for Probabilistic Programs. In: SAS, pp. 390–406 (2011)
6. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In: CAV. pp. 585–591 (2011)
7. Monniaux, D.: An Abstract Analysis of the Probabilistic Termination of Programs. In: Cousot, P. (ed.) SAS. LNCS, vol. 2126, pp. 111–126. Springer (2001)

Augmenting formal development with use case reasoning

(abstract)

Alexei Iliasov

Newcastle University, UK

State-based methods for correct-by-construction software development rely on a combination of safety constraints and refinement obligations to demonstrate design correctness. One prominent challenge, especially in an industrial setting, is ensuring that a design is adequate: requirements compliant and fit for purpose. The paper presents a technique for augmenting state-based, refinement-driven formal developments with reasoning about use case scenarios; in particular, it discusses a way for the derivation of formal verification conditions from a high-level, diagrammatic language of use cases, and the methodological role of use cases in a formal modelling process.

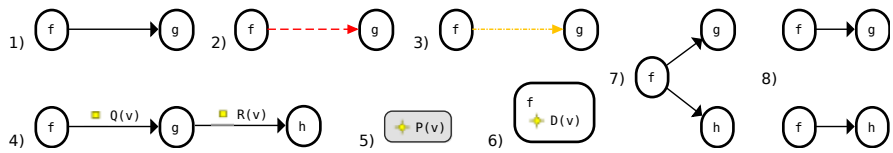
The approach to use case reasoning is based on our previous work on a graphical notation for expressing event ordering constraints [2, 1]. The extensions is realised as a plug in to the Event-B modelling tool set - the Rodin Platform [3] - and smoothly integrates into the Event-B modelling process. It provides a modelling environment for working with graph-like diagrams describing event ordering properties. In the simplest case, a node of such graph is an event of the associated Event-B machine; an edge is a statement about the relative properties of the connected nodes/events. There are three main edge kinds - **ena**, **dis** and **fis** - defined as relations over Event-B events.

$$\begin{aligned} U &= \{f \mapsto g \mid \varnothing \subset f \subseteq S \times S \wedge \varnothing \subset g \subseteq S \times S\} \\ \mathbf{ena} &= \{f \mapsto g \mid f \mapsto g \in U \wedge \mathrm{ran}(f) \subseteq \mathrm{dom}(g)\} \\ \mathbf{dis} &= \{f \mapsto g \mid f \mapsto g \in U \wedge \mathrm{ran}(f) \cap \mathrm{dom}(g) = \varnothing\} \\ \mathbf{fis} &= \{f \mapsto g \mid f \mapsto g \in U \wedge \mathrm{ran}(f) \cap \mathrm{dom}(g) \neq \varnothing\} \end{aligned}$$

where $f \subseteq S \times S$ is a relational model of an Event-B event (we treat an event as a next-state relation). These definitions are converted into *consistency* proof obligations. For instance, if in a use case graph there appears an **ena** edge connecting events b and h one would have to prove the following theorem (see [2] for a justification).

$$\forall v, v', p_b \cdot I(v) \wedge G_b(p_b, v) \wedge R_b(p_b, v, v') \Rightarrow \exists p_h \cdot G_h(p_h, v') \tag{1}$$

A use case diagram is only defined in an association with one Event-B model, it does not exist on its own. The use case plug in automatically generates all the relevant proof obligations. A change in a diagram or its Event-B model leads to the re-computation of all affected proof obligations. These proof obligations are dealt with, like all other proof obligation types, by a combination of automated provers and interactive proof. Like in the proofs of model consistency



1) f **ena** g
2) f **dis** g
3) f **fis** g
4) f **ena** $\mathrm{q}.g \wedge \mathrm{q}.g$ **ena** $\mathrm{r}.h$
$\mathrm{q} = \{v \mid Q(v)\}, \mathrm{r} = \{v \mid R(v)\}$

5) $skip(C), C = \{v \mid P(v)\}$
6) $f(C), C = \{v \mid D(v)\}$
7) f **ena** $g \vee f$ **ena** h
8) f **ena** $g \wedge f$ **ena** h

Fig. 1. A summary of the core use case notation and its interpretation.

and refinement, the feedback from an undischarged use case proof obligation may often be interpreted as a suggestion of a diagram change such as an additional assumptions or assertion - predicate annotations on graph edges that propagate properties along the graph structure. The example in the next section demonstrates how such annotations enable the proof of a non-trivial property.

The use case tool offers a rich visual notation. The basic element of a diagram is an event, visually depicted as a node (in Figure 1, f and g represent events). Event definition (its parameters, guard and action) is imported from the associated Event-B model. One special case of node is skip event, denoted by a grey node colour (Figure 1, 5). Event relations **ena**, **dis**, **fis** are represented by edges connecting nodes ((Figure 1, 1-3)). Depending on how a diagram is drawn, edges are said to be in *and* or *or* relation (Figure 1, 7-8). New events are derived from model events by strengthening their guards (a case of symmetric assumption and assertion) (Figure 1, 6). Edges may be annotated with constraining predicates inducing assertion and assumption derived events (Figure 1, 4). Not shown on Figure 1 are nodes for the initialisation event start (circle), implicit deadlock event stop (filled circle) and nodes for container elements such as loop (used in the coming example). To avoid visual clutter, the repeating parts of a diagram may be declared separately as diagram *aspects*[2].

References

1. Alexei Iliasov. Augmenting Event-B Specifications with Control Flow Information. In *NODES 2010*, May 2010.
2. Alexei Iliasov. Use case scenarios as verification conditions: Event-B/Flow approach. In *Proceedings of 3rd International Workshop on Software Engineering for Resilient Systems*, Septembre 2011.
3. The RODIN platform. Online at http://rodin-b-sharp.sourceforge.net/.

SMT-Based Array Invariant Generation*

Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio

Universitat Politècnica de Catalunya, Barcelona, Spain

Discovering loop invariants is an essential task for verifying the correctness of programs or computer systems in general. In this talk we present a technique for generating universally quantified loop invariants over array variables.

Namely, programs are assumed to consist of unnested loops and contain linear expressions in assignments, `if` and `while` conditions, as well as in array accesses. Now, let $\overline{a} = (A_1, \ldots, A_m)$ be the array variables of a program. Given a positive integer $k > 0$, our method generates invariants of the form

$$\forall \alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A_i [d_{ij}\alpha + \mathcal{E}_{ij}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \leq \ 0$$

where $\mathcal{C}$, $\mathcal{E}_{ij}$ and $\mathcal{B}$ are linear polynomials with integer coefficients over the scalar variables of the program $\overline{v} = (v_1, \ldots, v_n)$ and $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}$, for all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, k\}$. This family of properties is quite general and allows us to handle a wide variety of programs for which we can automatically generate non-trivial invariants.

Unlike previous approaches based on abstract interpretation or first-order theorem proving, our method builds upon the so-called *constraint-based* invariant generation approach. This method produces linear invariants, i.e., invariants expressed as linear inequalities over scalar variables, by transforming the problem of the existence of an inductive invariant for a loop into a satisfiability problem in propositional logic over non-linear arithmetic, thanks to Farkas' Lemma. Despite the potential of the method, its application has been limited so far due to the lack of good solvers for the obtained non-linear constraints.

However, recently significant progress has been made in SMT modulo the theory of non-linear arithmetic. In particular, the Barcelogic SMT solver has shown to be very effective on finding solutions in the presence of non-linear integer arithmetic. It can also combine integers and reals, which is very useful when handling the constraints generated by the constraint-based invariant generation approach.

Our techniques have been successfully implemented in the CppInv tool. By using the Barcelogic SMT solver as a back-end, it automatically generates inductive loop invariants (both linear scalar invariants as well as array invariants) for programs written in a subset of the C++ language. We believe that the combination of our tool with some static analysis to infer the set of potentially interesting invariants for proving some given property would be very useful in the automation of the verification process.

* This work has been partially supported by the Spanish MEC/MICINN under grant TIN 2010-68093-C02-01

Formal Characterization and Verification of Loop Invariant Based on Finite Difference

Mengjun Li

School of Computer Science, National University of Defense Technology, Changsha, China

Loop invariants play a major role in software verification. Dynamic approach provides ways to discover likely invariants rapidly. Since the likely loop invariant may not be real, the validity of the likely loop invariants need to be verified. In this paper, we present a formal characterization and a verification approach for equality loop invariants based on finite difference.

The following theorem gives a formal characterization of loop invariants, where $\overline{x} = (x_1, \cdots, x_n)$.

Theorem 1. *$E(\overline{x}) = 0$ is a loop invariant if and only if, for each transition $\tau_i (1 \leq i \leq m)$, $\Delta_{\tau_i} E(\overline{x})$ is 0 or $\Delta_{\tau_i} E(\overline{x}) = 0$ is also a loop invariant.*

Definition 1. *The finite difference tree (FDT) of a likely loop invariant $E(\overline{x}) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$ is defined as follows:*

(1)The root is $E(\overline{x})$ and the leaves are values 0;

(2)If the tree contains a non-leaf node $F(\overline{x})$, then $F(\overline{x})$ has m child nodes $\Delta_{\tau_1} F(\overline{x}), \cdots, \Delta_{\tau_m} F(\overline{x})$.

If the FDTs are infinite, theorem 1 can not be used to verify the validity of likely loop invariants. In the following, we presents a practical verification approach for loop invariants.

Definition 2. *Let T be a finite difference tree of a likely loop invariant $E(\overline{x}) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$, a node $F(\overline{x})$ in T is called a zero node if $F(\overline{x}) = \sum_{j=0}^{k} p_j(\overline{x}) F_j(\overline{x})$, where $F_j(\overline{x})(j = 0, \cdots, k))$ is the ancestor node of $F(\overline{x})$ and each $F_j(\overline{x})$ satisfies that $F_j(\overline{x_0}) = 0 (j = 0, \cdots, k)$, where $\overline{x_0}$ expresses the initial value of $\overline{x}$, and each $p_j(\overline{x})(j = 0, \cdots, k)$ is an arbitrary function over variables $x_1, \cdots, x_n$.*

Definition 3. *The decidable finite difference tree(DFDT) of a likely loop invariant $E(\overline{x}) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$ is the finite difference tree of $E(\overline{x}) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$ with zero nodes as leaves.*

Theorem 2. *If there exists a finite DFDT T of $E(\overline{x}) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$ and $E(\overline{x_0}) = 0$, then $E(\overline{x}) = 0$ is a loop invariant.*

The effectiveness of our verification approach have been demonstrated on those examples occurring in Laura Kovács's Ph.D.Thesis. We even can prove $f - n! = 0$ is a loop invariant of the program computing the greatest factorial less than or equal to a given N. Note that $f - n! = 0$ is not a polynomial loop invariant, to the best of our knowledge, our work is the first on verifing the validity of non-polynomial loop invariants.

Conclusive Proofs of While Loops Using Invariant Relations

Lamia Labed Jilani (ISG, Tunisia), Wided Ghardallou (FST, Tunisia), Ali Mili (NJIT, USA)

Traditional methods of verifying iterative programs rely on invariant assertions to prove partial correctness and on variant functions to prove termination. As such, they suffer from some weaknesses, which we characterize briefly and broadly as follows:

- If we attempt to prove the partial correctness of a while loop with respect to a (pre/post) specification and the proof fails, we have no simple way to tell whether the proof failed because the loop is incorrect or because the invariant assertion is inadequate.
- Even assuming we have determined that the invariant assertion is inadequate, we have no simple way to determine whether we need to adjust it by strengthening it or by weakening it: granted, if the invariant assertion does not imply the post condition or is not implied by the precondition, then the remedy is obvious; but if the invariant assertion does not meet the inductive step, it is not clear how it must be adjusted.
- When one uses variant functions to prove loop termination, one equates the termination of the loop with the condition that the number of iterations is finite, but fails to capture the situation where individual iterations of the loop body fail to terminate normally because they attempt an illegal operation (array reference out of bounds, pointer reference to null, etc).

In this paper, we present an alternative method to analyze loops, which appears to address most of the concerns raised above, and is based on the concept of invariant relations. Our approach can be characterized by the following premises.

- Any invariant relation can be converted into a necessary condition of termination; our definition of termination is comprehensive, in the sense that it encompasses the condition that the number of iterations is finite, as well as the condition that each individual iteration proceeds normally.
- Given an invariant relation, we can use it to generate a necessary condition of correctness of the loop with respect to a relational specification. This condition returns false whenever the invariant relation contradicts the candidate specification.
- Given an invariant relation, we can use it to generate a sufficient condition of correctness of the loop with respect to a relational specification. This condition returns true whenever the invariant relation subsumes the candidate specification.
- We have an automated tool that generates invariant relations from a static analysis of the source code of the loop, using a knowledge base that captures the necessary programming knowledge and domain knowledge that is required for an adequate analysis of the loop. Each invariant relation captures some functional property of the loop. If the tool runs out of invariant relations without capturing all the functional details of the loop, it gives an indication of how to complete the missing knowledge in the knowledge base.
- Using the capabilities discussed above, we generate an algorithm for proving the correctness of a loop with respect to a relational specification that extracts invariant relations until it finds one that contradicts the candidate specification (if the necessary condition is false) or one that subsumes the candidate specification (if the sufficient condition is true) or until it runs out of invariant relations (in which case it indicates what is missing from the knowledge base).

E-SPARK: Automated Generation of Verifiable Code from Formally Verified Designs

Rajiv Murali and Andrew Ireland

School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh, EH14 4AS, UK.
{rm339, A.Ireland}@hw.ac.uk

The safety-critical sectors are faced with conflicting demands of achieving both high assurance as well increasing the productivity of their development process. Auto-coders have been effective in many areas, but the need for high levels of assurance has prevented its use in safety critical applications such as avionics. Standards for safety critical systems require a constant degree of verification, and most commercially available auto-coders do not satisfy this requirement.

We propose an approach where the auto-coder takes a formal model and generates code along with annotations, i.e. information flow analysis and proof assertions. In this way design invariants can be reused at the code level in order to support formal verification. Specifically, we have targeted Event-B and the SPARK Approach. At the design level, Event-B provides formal modeling supported by a strong and extensible toolset called Rodin. On the implementation level, the SPARK Approach includes a range of static analysis tools, from data flow analysis to formal verification. We have developed an eclipse based plug-in called E-SPARK for the Rodin platform that supports the automatic generation of provably correct code. At this stage, E-SPARK has been designed for the sequential subset of Event-B, and has been tested successfully on a range of arithmetic, searching and sorting examples of algorithmic design. Future development of E-SPARK could aim to target Event-B models of concurrent systems.

Invariant Stream Generators using Automatic Abstract Transformers based on a Decidable Logic*

Pierre-Loïc Garoche[1,2], Temesghen Kahsai[2] and Cesare Tinelli[2]

[1] Onera, the French Aerospace Lab, France
[2] The University of Iowa

The use of formal analysis tools on system models or code often requires the availability of auxiliary invariants about the studied system. Abstract interpretation is currently one of the best approaches to discover useful invariants, especially numerical ones. However, its application is limited by two orthogonal issues: (i) developing an abstract interpretation is often non-trivial; each transfer function of the system has to be represented at the abstract level, depending on the abstract domain used; (ii) with precise but costly abstract domains, the information computed by the abstract interpreter can be used only once the a post fix point has been reached; something that may take a long time for very large system analysis or with delayed widening to improve precision.

In this work we try to address these issues by combining techniques from abstract interpretation and logic-based model checking. Specifically, we propose a general method for the automatic definition of abstract interpreters that compute numerical invariants of transition systems. We rely on the possibility of encoding the transition system in a decidable logic—such as those typically used by SMT-based model checkers—to compute transformers for an abstract interpreter *completely automatically*. Our method has the significant added benefit that the abstract interpreter can be instrumented to generate system invariants on the fly, during its iterative computation of a post fix point. A prototype implementation of the method provides initial evidence of the feasibility of our approach and the usefulness of its incremental invariant generation feature.

While motivated by practical issues (namely, the generation of auxiliary invariants for a k-induction model checker) the current work is more general and can be adapted to a wide variety of contexts. It only requires that the transition system semantics be expressible in a decidable logics with an efficient solver, such as SAT or SMT solvers, and that the elements of the chosen abstract domain be effectively representable in that logic. Such requirements are satisfied by a large number of abstract domains used in current practice. As a consequence, we believe that our approach could help considerably in expanding the reach of abstract interpretation techniques to a variety of target languages, as well as facilitate their integration with complementary techniques such as model checking ones.

* Work currently under submission at another conference.

Faster Automatic Test Case Generation

Ott Tinn
University of Edinburgh

This talk presents a new tool that extends an existing symbolic execution tool for C programs to partially support C++ and makes it faster at finding inputs that could crash the analysed programs. The speedup is achieved by adding a new transformation phase that aims to make bug finding faster without optimizing any relevant bugs away by trying to computes just the one bit per input that shows whether the program crashes on that input.

The transformation phase transforms the programs such that regular observable behaviour is removed if it is not needed (for example it is not necessary to produce output if it is not evaluated). Then it adds validation calls to make sure the compiler can not optimize away potentially dangerous operations such as memory accesses. After that optimizing transforms such as the following are applied:

- Validation removal: if a validation call is always preceded by another one that would catch at least the same set of bugs then it could be removed. The same can be done for calls that can never fail.
- Validation hoisting: move each validation call to the earliest point in the control flow graph (CFG) where it could be executed without changing the CFG or the set of inputs on which the program crashes.
- Loop removal: if a loop has no side effects and a known iteration count then it could be removed by replacing the induction variable with a symbolic variable that takes the same range. This makes the loop get executed with any of the values instead of all, but that is enough to find any potential crashes.

The sample was an extensive set of programs from a programming competition archive. The evaluation showed that on that class of C++ programs an input that triggers a crash can be found in less than five minutes for a significant number of cases and that the new system takes about half as much time as the base system if the time limit is between a minute and an hour per program. A bug was found in at most one hour in roughly half the cases where a program was known to crash[1] on some input.

In general similar bug finding systems should be usable for finding bugs in small programs with clearly defined input spaces and not too complicated logic. Thus these systems should be usable for finding bugs in students' assignments, other small programs, and parts of programs (e.g. a few classes).

The new transformation stage should also be applicable to other similar bug finding systems. Without the loop transformations it could even be used in systems without symbolic reasoning abilities, such as fuzzers.

[1] Some of the known crashes were undetectable by the systems so it is not clear how many detectable crashes were not found.

Abstract Domains for Bit-Level Machine Integer and Floating-point Operations

Antoine Miné*

CNRS & École Normale Supérieure
45, rue d'Ulm
75005 Paris, France
mine@di.ens.fr

Abstract

We present a few lightweight numeric abstract domains to analyze C programs that exploit the binary representation of numbers in computers, for instance to perform "compute-through-overflow" on machine integers, or to directly manipulate the exponent and mantissa of floating-point numbers. On integers, we propose an extension of intervals with a modular component, as well as a bitfield domain. On floating-point numbers, we propose a predicate domain to match, infer, and propagate selected expression patterns. These domains are simple, efficient, and extensible. We have included them into the Astrée and AstréeA static analyzers to supplement existing domains. Experimental results show that they can improve the analysis precision at a reasonable cost.

1 Introduction

Semantic-based static analysis is an invaluable tool to help ensuring the correctness of programs as it allows discovering program invariants at compile-time and fully automatically. Abstract interpretation [9] provides a systematic way to design static analyzers that are sound but approximate: they infer invariants which are not necessarily the tightest ones. A central concept is that of abstract domains, which consist of a set of program properties together with a computer representation and algorithms to compute sound approximations in the abstract of the effect of each language instruction. For instance, the interval domain [9] allows inferring variable bounds. Bound properties allow expressing the absence of many run-time errors (such as arithmetic and array overflows) but, due to approximations, the inferred bounds may not be sufficiently precise to imply the desired safety assertions (e.g., in the presence of loops). An effective static analyzer for run-time errors, such as Astrée [7], uses additional domains to infer local and loop invariants of a more complex form (e.g., octagons [20]) and derive tighter bounds.

Most numeric domains naturally abstract an ideal semantics based on perfect integers or rationals, while computers actually use binary numbers with a fixed number of digits. One solution is to adapt the domains to take into account hardware limitations: overflows are detected and treated as errors, while floating-point semantics is simulated by introducing rounding errors [17]. While this works well in many cases, it is not sufficient to analyze programs that perform overflows on purpose (expecting a wrap-around semantics) or that rely on the precise binary representation of numbers. The goal of this article is to propose a set of simple, lightweight numeric abstract domains that are aware of these aspects.

*This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France.

```
char add1(char x, char y) {
  return (char)
    ((unsigned char)x +
     (unsigned char)y);
}
```

(a)

```
char add2(char x, char y) {
  unsigned register r1,r2,r3;
  r1 = x; r2 = y;
  r3 = r1 + r2;
  return r3;
}
```

(b)

Figure 1: Integer "compute-through-overflow" examples. `char` are assumed to be signed.

```
union u { int i[2]; double d; };
double cast(int i) {
  union u x,y;
  x.i[0] = 0x43300000;
  y.i[0] = x.i[0];
  x.i[1] = 0x80000000;
  y.i[1] = i ^ x.i[1];
  return y.d - x.d;
}
```

(a)

```
double sqrt(double d) {
  double r;
  unsigned* p = (unsigned*)&d;
  int e = (*p & 0x7fe00000) >> 20;
  *p = (*p & 0x801fffff) | 0x3fe00000;
  r = ((c1*d+c2)*d+c3)*d+c4;
  *p = (e/2 + 511) << 20;
  p[1] = 0;
  return d * r;
}
```

(b)

Figure 2: Floating-point computations exploiting the IEEE binary representation. On the right, `c1` to `c4` are unspecified constant coefficients of a polynomial approximation. A 32-bit big-endian processor is assumed (e.g., PowerPC).

1.1 Motivating Examples

Figure 1.a presents a small C function that adds two signed bytes (`char`) by casting them to unsigned bytes before the addition and casting the result back to signed bytes. The function systematically triggers an overflow on negative arguments, which is detected by an analyzer such as Astrée. Additionally, on widespread architectures, the return value equals `x+y` due to wrap-around. This programming pattern is used in popular industrial code generators such as TargetLink [10] and known as "compute-through-overflow." An analysis not aware of wrap-around will either report dead code (if overflows are assumed to be fatal) or return $[-128, 127]$ (if overflows produce full-range results). Even a wrapping-aware interval analysis will return an imprecise interval for arguments crossing zero, e.g. $[-1, 0]$, as the first cast maps $\{-1, 0\}$ to $\{255, 0\}$ and intervals cannot represent non-convex properties (see Sec. 2.6).

A variant is shown in Fig. 1.b, where the casts are implicit and caused by copies between variables of different types. This pattern is used to ensure that arithmetic computations are performed in CPU registers only, using a pool of `register` variables with irrelevant signedness (i.e., register allocation is explicit and not entrusted to the compiler).

Figure 2.a presents a C function exploiting the binary representation of floating-point numbers based on the IEEE standard [13]. It implements a conversion from 32-bit integers to 64-bit floats by first constructing the float representation for $\texttt{x.d} = 2^{52} + 2^{31}$ and $\texttt{y.d} = 2^{52} + 2^{31} + \texttt{i}$ using integer operations and then computing $\texttt{y.d} - \texttt{x.d} = \texttt{i}$ as a float subtraction. This code is similar to the assembly code generated by compilers when targeting CPUs missing the conversion instruction (such as PowerPC). Some code generators choose to provide their own C implementation instead of relying on the compiler (for instance, to improve the traceability of

the assembly code). Figure 2.b exploits the binary encoding of floats to implement a square root: the argument is split into an exponent `e` and a mantissa in $[1, 4]$ (computed by masking the exponent in `d`); then the square root of the mantissa is evaluated through a polynomial, while the exponent is simply halved. In both examples, a sound analyzer not aware of the IEEE floating-point encoding will return the full float range.

These examples may seem disputable, yet they are representative of actual industrial codes (the examples have been modified for the sake of exposition). The underlying programming patterns are supported by many compilers and code generators. An industrial-strength static analyzer is expected to accept existing coding practices and handle them precisely.

1.2 Contribution

We introduce a refined concrete semantics taking bit manipulations into account, and present several abstractions to infer precise bounds for the codes in Figs. 1–2.

Section 2 focuses on integers with wrap-around: we propose an interval domain extended with a modular component and a bitfield domain abstracting each bit separately. Handling the `union` type and pointer cast from Fig. 2 requires a specific memory model, which is described in Sec. 3. Section 4 presents a bit-level float domain based on pattern matching enriched with predicate propagation. Section 5 presents experimental results using the Astrée and AstréeA static analyzers. Finally, Sec. 6 concludes.

The domains we present are very simple and lightweight; they have a limited expressiveness. They are intended to supplement, not replace, classic domains, when analyzing programs featuring bit-level manipulations. Moreover, they are often slight variations on existing domains [9, 15, 23, 19, 20]. We stress the fact that these domains and the change of concrete semantics they require have been incorporated into existing industrial analyzers, to enrich the class of programs they can analyze precisely, at low cost and with no precision regression on previously analyzed codes.

1.3 Related Work

The documentation [2] for the PolySpace analyzer suggests removing, prior to an analysis, all computes-through-overflows and provides a source filter based on regular expressions to do so. This solution is fragile and can miss casts (e.g., when they are not explicit, as in Fig. 1.b) or cause unsoundness (in case the pattern is too inclusive and transforms unrelated code parts), while the solution we propose is semantic-based.

Various domains supporting modular arithmetics have been proposed, such as simple [11] and affine congruences [12, 24]. Masdupuy introduced interval congruences [15] to analyze array indices; our modular intervals are a slightly simpler restriction and feature operators adapted to wrap-around. Simon and King propose a wrap-around operator for polyhedra [26]; in addition to being costly, it outputs convex polyhedra while our examples require the inference of non-convex invariants locally. Abstracting each bit of an integer separately is a natural idea that has been used, for instance, by Monniaux [23] and Regehr et al. [25]. Brauer et al. [8] propose a bit-blasting technique to design precise transfer functions for small blocks of integer operations, which can bypass the need for more expressive (e.g., disjunctive) local invariants.

We are not aware of any abstract domain able to handle bit-level operations on floating-point numbers. Unlike classic predicate abstraction [6], our floating-point predicate domain includes its own fast and ad-hoc (but limited) propagation algorithm instead of relying on an external generic tool.

$$\textit{int-type} ::= (\texttt{signed} \mid \texttt{unsigned})?\ (\texttt{char} \mid \texttt{short} \mid \texttt{int} \mid \texttt{long} \mid \texttt{long long})\ \ n? \qquad (n \in \mathbb{N}^*)$$

$$\textit{bitsize} : \textit{int-type} \to \mathbb{N}^*$$
$$\textit{signed} : \textit{int-type} \to \{\text{true}, \text{false}\}$$

$$\textit{range}(t) \stackrel{\text{def}}{=} \begin{cases} [0, 2^{\textit{bitsize}(t)} - 1] & \text{if } \neg \textit{signed}(t) \\ [-2^{\textit{bitsize}(t)-1}, 2^{\textit{bitsize}(t)-1} - 1] & \text{if } \textit{signed}(t) \end{cases}$$

Figure 3: Integer C types and their characteristics.

2 Integer Abstract Domains

2.1 Concrete Integer Semantics

In this section, we focus on integer computations. Before designing a static analysis, we need to provide a precise, mathematical definition of the semantics of programs. We base our semantics on the C standard [5], extended with hypotheses on the representation of data-types necessary to analyze the programs in Fig. 1.

The C language mixes operators based on mathematical integers (addition, etc.) and operators based on the binary representation of numbers (bit-wise operators, shifts). At the hardware level, however, all integer computations are performed in registers of fixed bit-size. Thus, one way to define the semantics is to break it down at the bit level (i.e., "bit-blasting" [8]). We choose another route and express the semantics using classic mathematical integers in $\mathbb{Z}$. Our semantics is higher-level than a bit-based one, which provides some advantages: on the concrete level, it makes the classic arithmetic C operations (`+`, `-`, `*`, `/`, `%`) straightforward to express; on the abstract level, it remains compatible with abstract domains expressed on perfect numbers (such as polyhedra). We show that this choice does not preclude the definition of bit-wise operators nor bit-aware domains.

2.2 Integer Types

Integer types in C come in different sizes and can be signed or unsigned. We present in Fig. 3 the type grammar for integers, *int-type*, including bitfields that can only appear in structures (when a bit size n is specified). The bit size and signedness of types are partly implementation-specific. We assume that they are specified by two maps: $\textit{bitsize} : \textit{int-type} \to \mathbb{N}^*$ and $\textit{signed} : \textit{int-type} \to \{\text{true}, \text{false}\}$. Moreover, we assume that unsigned integers are represented using a pure binary representation: $b_{n-1} \cdots b_0 \in \{0, 1\}^n$ represents $\sum_{i=0}^{n-1} 2^i b_i$, and signed integers use two's complement representation: $b_{n-1} \cdots b_0 \in \{0, 1\}^n$ represents $\sum_{i=0}^{n-2} 2^i b_i - 2^{n-1} b_{n-1}$. Although this is not required by the C standard, it is the case for all the popular architectures.[1] The *range* (i.e., the set of acceptable values) of each type is derived as in Fig. 3.

2.3 Integer Expressions

We consider here only a pure, side-effect-free, integer fragment of C expressions, as depicted in Fig. 4. To stay concise, we include only arithmetic and bit-wise operators and casts. Moreover, statements are reduced to assignments and assertions (which are sufficient to model programs as control-flow graphs). We perform a static transformation that makes all wrap-around effects

[1] The C standard allows some features that we do not handle: padding bits, trap representations, one's complement representations or sign-magnitude representations of negative numbers, and negative zeros.

$$
\begin{array}{rcll}
\mathit{expr} & ::= & n & \textit{(constant } n \in \mathbb{Z}\textit{)} \\
 & | & V & \textit{(variable } V \in \mathcal{V}\textit{)} \\
 & | & \texttt{(}\textit{int-type}\texttt{)}\ \mathit{expr} & \textit{(cast)} \\
 & | & \diamond\, \mathit{expr} & \textit{(unary operation, } \diamond \in \{\texttt{-}, \texttt{\sim}\}\textit{)} \\
 & | & \mathit{expr} \circ \mathit{expr} & \textit{(binary operation, } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&}, \texttt{|}, \texttt{\^{}}, \texttt{>>}, \texttt{<<}\}\textit{)} \\
\\
\mathit{stat} & ::= & V = \mathit{expr} & \textit{(assignment)} \\
 & | & \mathit{assert}(\mathit{expr}) & \textit{(assertion)}
\end{array}
$$

Figure 4: Fragment of integer C syntax.

$$
\begin{array}{l}
\tau : \mathit{expr} \to \mathit{int\text{-}type} \\
\quad \tau(n) \in \mathit{int\text{-}type} \quad \textit{(given)} \qquad \tau(V) \in \mathit{int\text{-}type} \quad \textit{(given)} \\
\quad \tau(\diamond\, e) \stackrel{\text{def}}{=} \mathit{promote}(\tau(e)) \qquad \tau(\texttt{(}t\texttt{)}\ e) \stackrel{\text{def}}{=} t \\
\quad \tau(e_1 \circ e_2) \stackrel{\text{def}}{=} \begin{cases} \mathit{lub}(\mathit{promote}(\tau(e_1)), \mathit{promote}(\tau(e_2))) & \text{if } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&}, \texttt{|}, \texttt{\^{}}\} \\ \mathit{promote}(\tau(e_1)) & \text{if } \circ \in \{\texttt{<<}, \texttt{>>}\} \end{cases}
\end{array}
$$

where:

$$
\mathit{promote}(t) \stackrel{\text{def}}{=} \begin{cases} \texttt{int} & \text{if } \mathit{rank}(t) < \mathit{rank}(\texttt{int}) \wedge \mathit{range}(t) \subseteq \mathit{range}(\texttt{int}) \\ \texttt{unsigned} & \text{else if } \mathit{rank}(t) < \mathit{rank}(\texttt{int}) \wedge \mathit{range}(t) \subseteq \mathit{range}(\texttt{unsigned}) \\ \mathit{promote}(t') & \text{if } t \text{ has bitfield type } t'\ n\text{, based on } t' \\ t & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
\mathit{lub}(t, t') \stackrel{\text{def}}{=} \\
\quad \text{when } \mathit{rank}(t) \geq \mathit{rank}(t')\text{:} \\
\qquad \begin{cases} t & \text{if } \mathit{signed}(t) = \mathit{signed}(t') \text{ or } \neg\mathit{signed}(t) \wedge \mathit{signed}(t') \\ & \text{or } \mathit{signed}(t) \wedge \neg\mathit{signed}(t') \wedge \mathit{range}(t') \subseteq \mathit{range}(t) \\ \texttt{unsigned}\ t & \text{if } \mathit{signed}(t) \wedge \neg\mathit{signed}(t') \wedge \mathit{range}(t') \not\subseteq \mathit{range}(t) \end{cases} \\
\quad \text{when } \mathit{rank}(t) < \mathit{rank}(t')\text{: } \mathit{lub}(t', t)
\end{array}
$$

$$
\begin{array}{llll}
\mathit{rank}(\texttt{char}) \stackrel{\text{def}}{=} 1 & \mathit{rank}(\texttt{short}) \stackrel{\text{def}}{=} 2 & \mathit{rank}(\texttt{int}) \stackrel{\text{def}}{=} 3 & \mathit{rank}(\texttt{long}) \stackrel{\text{def}}{=} 4 \\
\mathit{rank}(\texttt{long long}) \stackrel{\text{def}}{=} 5 & \mathit{rank}(\texttt{signed}\ t) \stackrel{\text{def}}{=} \mathit{rank}(\texttt{unsigned}\ t) \stackrel{\text{def}}{=} \mathit{rank}(t) & &
\end{array}
$$

Figure 5: Typing of integer expressions.

explicit in expressions by first typing sub-expressions and then inserting casts. These steps are performed in a front-end and generally not discussed, but we present them to highlight of few subtle points.

Typing. The type $\tau(e)$ of an expression e is inferred as in Fig. 5 based on the given type of variables and constants. Firstly, a promotion rule (*promote*) states that values of type t smaller than `int` (where the notion of "smaller" is defined by the *rank* function) are promoted to `int`, if `int` can represent all the values in type t, and to `unsigned` otherwise. Values of bitfield type $t\ n$ are promoted as their corresponding base type t. Secondly, for binary operators, the type of the result is inferred from that of both arguments (*lub*). Integer promotion causes values with the same binary representation but different types to behave differently. For instance,

$$
\begin{array}{lcl}
(\!| V |\!) & \stackrel{\text{def}}{=} & V \\
(\!| n |\!) & \stackrel{\text{def}}{=} & \texttt{(}\tau(n)\texttt{)}\ n \\
(\!| \texttt{(}t\texttt{)}\ e |\!) & \stackrel{\text{def}}{=} & \texttt{(}t\texttt{)}\ (\!| e |\!) \\
(\!| \diamond\ e |\!) & \stackrel{\text{def}}{=} & \text{let } t = \tau(\diamond e) \text{ in } \texttt{(}t\texttt{)}\ (\diamond\ \texttt{(}t\texttt{)}\ (\!| e |\!)) \\
(\!| e_1 \circ e_2 |\!) & \stackrel{\text{def}}{=} & \text{if } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&}, \texttt{|}, \texttt{\^{}}\} \text{ then:} \\
 & & \quad \text{let } t = \tau(e_1 \circ e_2) \text{ in } \texttt{(}t\texttt{)}\ (\texttt{(}t\texttt{)}\ (\!| e_1 |\!)\ \circ\ \texttt{(}t\texttt{)}\ (\!| e_2 |\!)) \\
 & & \text{if } \circ \in \{\texttt{<<}, \texttt{>>}\} \text{ then:} \\
 & & \quad \text{let } t = \tau(e_1) \text{ in } \texttt{(}t\texttt{)}\ (\texttt{(}t\texttt{)}\ (\!| e_1 |\!)\ \circ\ \texttt{(unsigned 5)}\ (\!| e_2 |\!)) \\
(\!| V = e |\!) & \stackrel{\text{def}}{=} & V = \texttt{(}\tau(V)\texttt{)}\ (\!| e |\!) \\
(\!| \mathit{assert}(e) |\!) & \stackrel{\text{def}}{=} & \mathit{assert}((\!| e |\!))
\end{array}
$$

Figure 6: Insertion of implicit casts.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{[\![\mathit{expr}]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{Z})} \\
[\![V]\!]\rho & \stackrel{\text{def}}{=} & \{\, \rho(V) \,\} \qquad\qquad [\![n]\!]\rho \stackrel{\text{def}}{=} \{\, n \,\} \\
[\![\texttt{(}t\texttt{)}\ e]\!]\rho & \stackrel{\text{def}}{=} & \{\, \mathit{wrap}(v, \mathit{range}(t)) \mid v \in [\![e]\!]\rho \,\} \qquad [\![\diamond\ e]\!]\rho \stackrel{\text{def}}{=} \{\, \diamond v \mid v \in [\![e]\!]\rho \,\} \\
[\![e_1 \circ e_2]\!]\rho & \stackrel{\text{def}}{=} & \{\, v_1 \circ v_2 \mid v_1 \in [\![e_1]\!]\rho \wedge v_2 \in [\![e_2]\!]\rho \wedge (v_2 \neq 0 \vee \circ \notin \{\, \texttt{/}, \texttt{\%} \,\}) \,\} \\
\multicolumn{3}{l}{\text{where } \mathit{wrap}(v, [\ell, h]) \stackrel{\text{def}}{=} \min \{\, v' \mid v' \geq \ell \wedge \exists k \in \mathbb{Z} : v = v' + k(h - \ell + 1) \,\}}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{[\![\mathit{stat}]\!] : \mathcal{E} \to \mathcal{P}(\mathcal{E})} \\
[\![V = e]\!]\rho & \stackrel{\text{def}}{=} & \{\, \rho[V \mapsto v] \mid v \in [\![e]\!]\rho \,\} \\
[\![\mathit{assert}(e)]\!]\rho & \stackrel{\text{def}}{=} & \{\, \rho \mid \exists v \in [\![e]\!]\rho : v \neq 0 \,\}
\end{array}
$$

Figure 7: Concrete semantics.

`unsigned char a = 255` and `signed char b = -1` have the same representation, but `a >> 1 = 127` while `b >> 1 = -1`. Integer promotion is said to be "value preserving", as opposed to "representation preserving" [4]. This rule comforts us in our decision to focus on the integer value of variables instead of their binary representation.

Cast introduction. The translation of expressions, $(\!| \cdot |\!)$, is presented in Fig. 6. Firstly, before applying an operator, its arguments are converted to the type of the result. This can lead to wrap-around effects. For instance, in `(int)-1 + (unsigned)1`, the left argument is converted to `unsigned`, which gives $2^{32} - 1$ on a 32-bit architecture. The case of bit shift operators is special; as shifting by an amount exceeding the bit-size of the result is undefined in the C standard, we model instead the behavior of intel 32-bit hardware: the right argument is masked to keep only the lower 5 bits (abusing bitfield types). Secondly, casts are introduced to ensure that the value of the result lies within the range of its inferred type. Finally, before storing a value into a variable, it is converted to the type of the variable.

2.4 Operator Semantics

After translation, the semantics of expressions can be defined in terms of integers, without any reference to C types. The arithmetic operators (`+`,`-`,`*`,`/`,`%`) have their classic meaning in $\mathbb{Z}$.[2] To

[2]Note that `/` rounds towards zero and that a `%` $b \stackrel{\text{def}}{=} a - (a/b)*b$.

define bit-wise operations (~, &, |, ^, <<, >>) on $\mathbb{Z}$, we first associate an (infinite) bit pattern to each integer in $\mathbb{Z}$. It is an element of the boolean algebra $B = (\{0,1\}^{\mathbb{N}}, \neg, \wedge, \vee)$ with pointwise negation $\neg$, logical and $\wedge$, and logical or $\vee$ operators. The pattern $p(x) \in B$ of an integer $x \in \mathbb{Z}$ is defined using an infinite two's complement representation:

$$p(x) \stackrel{\text{def}}{=} \begin{cases} p(x) = (b_i)_{i\in\mathbb{N}} & \text{where } b_i = \lfloor x/2^i \rfloor \mod 2, \text{ if } x \geq 0 \\ p(x) = (\neg b_i)_{i\in\mathbb{N}} & \text{where } (b_i)_{i\in\mathbb{N}} = p(-x-1), \text{ if } x < 0 \end{cases} \tag{1}$$

The elements in B are reminiscent of 2-adic integers, but we restrict ourselves to those representing regular integers. The function p is injective, and we note p^{-1} its inverse, which is only defined on sequences that are stable after a certain index ($\exists i : \forall j \geq i : b_j = b_i$). The bit-wise C operators are given a semantics in $\mathbb{Z}$, based on their natural semantics in B, as follows:

$$\begin{array}{rcl@{\qquad}rcl} \sim x & \stackrel{\text{def}}{=} & p^{-1}(\neg p(x)) = -x-1 & x \,\&\, y & \stackrel{\text{def}}{=} & p^{-1}(p(x) \wedge p(y)) \\ x \mid y & \stackrel{\text{def}}{=} & p^{-1}(p(x) \vee p(y)) & x \,\hat{}\, y & \stackrel{\text{def}}{=} & p^{-1}(p(x) \oplus p(y)) \\ x \ll y & \stackrel{\text{def}}{=} & \lfloor x \times 2^y \rfloor & x \gg y & \stackrel{\text{def}}{=} & \lfloor x \times 2^{-y} \rfloor \end{array} \tag{2}$$

where $\oplus$ is the exclusive or and $\lfloor \cdot \rfloor$ rounds towards $-\infty$. This semantics is compatible with that of existing arbitrary precision integer libraries [1, 22].

2.5 Expression Semantics

Given environments $\rho \in \mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{Z}$ associating integer values to variables, we can define the semantics $[\![\cdot]\!]$ of expressions and statements as shown in Fig. 7. The semantics of wrap-around is modeled by the *wrap* function. Our semantics is non-deterministic: expressions (resp. statements) return a (possibly empty) set of values (resp. environments). This is necessary to define the semantics of errors that halt the program (e.g., division by zero). Non-determinism is also useful to design analyses that must be sound with respect to several implementations at once. We could for instance relax our semantics and return a full range instead of the modular result in case of an overflow in signed arithmetics (as the result is undefined by the standard).

All it takes to adapt legacy domains to our new semantics is an abstraction of the *wrap* operator. A straightforward but coarse one would state that $\mathit{wrap}^\sharp(v, [\ell, h]) \stackrel{\text{def}}{=} \{v\}$ if $v \in [\ell, h]$, and $[\ell, h]$ otherwise (see also [26] for a more precise abstraction on polyhedra). In the following, we will introduce abstract domains specifically adapted to the wrap-around semantics.

2.6 Integer Interval Domain $\mathcal{D}_i^\sharp$

We recall very briefly the classic interval abstract domain [9]. It maps each variable to an interval of integers:

$$\mathcal{D}_i^\sharp \stackrel{\text{def}}{=} \{\, [\ell, h] \mid \ell, h \in \mathbb{Z} \cup \{\pm\infty\} \,\}$$

As it is a non-relational domain, the abstract semantics of expressions, and so, of assignments, can be defined by structural induction, replacing each operator $\circ$ on $\mathbb{Z}$ with an abstract version $\circ_i^\sharp$ on intervals. Abstract assertions are slightly more complicated and require backward operators; we refer the reader to [18, § 2.4.4] for details. We recall [9] that optimal abstract operators can be systematically designed with the help of a Galois connection (α, γ):

$$\begin{array}{rcl} [\ell_1, h_1] \circ_i^\sharp [\ell_2, h_2] & \stackrel{\text{def}}{=} & \alpha_i(\{\, v_1 \circ v_2 \mid v_1 \in \gamma_i([\ell_1, h_1]),\, v_2 \in \gamma_i([\ell_2, h_2]) \,\}) \\ \gamma_i([\ell, h]) & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{Z} \mid \ell \leq x \leq h \,\} \\ \alpha_i(X) & \stackrel{\text{def}}{=} & [\min X,\, \max X] \end{array}$$

$$
\begin{aligned}
&\texttt{-}^\sharp_m([\ell,h]+k\mathbb{Z}) \stackrel{\text{def}}{=} [-h,-\ell]+k\mathbb{Z}\\
&\texttt{\sim}^\sharp_m([\ell,h]+k\mathbb{Z}) \stackrel{\text{def}}{=} [-h-1,-\ell-1]+k\mathbb{Z}\\
&([\ell_1,h_1]+k_1\mathbb{Z}) \circ^\sharp_m ([\ell_2,h_2]+k_2\mathbb{Z}) \stackrel{\text{def}}{=}\\
&\qquad \begin{cases} ([\ell_1,h_1] \circ^\sharp_i [\ell_2,h_2]) + \gcd(k_1,k_2)\mathbb{Z} & \text{if } \circ \in \{\texttt{+},\texttt{-},\texttt{*},\cup,\triangledown\}\\ ([\ell_1,h_1] \circ^\sharp_i [\ell_2,h_2]) + 0\mathbb{Z} & \text{if } k_1=k_2=0,\ \circ \in \{\texttt{/},\texttt{\%},\texttt{\&},\texttt{|},\texttt{\^{}},\texttt{>>},\texttt{<<}\}\\ [-\infty,+\infty]+0\mathbb{Z} & \text{otherwise}\end{cases}\\
&\mathit{wrap}^\sharp_m([\ell,h]+k\mathbb{Z},[\ell',h']) \stackrel{\text{def}}{=}\\
&\qquad \text{let } k' = \gcd(k,\ h'-\ell'+1) \text{ in}\\
&\qquad \begin{cases} [\mathit{wrap}(\ell,[\ell',h']),\ \mathit{wrap}(h,[\ell',h'])]+0\mathbb{Z} & \text{if } (\ell'+k'\mathbb{Z})\cap[\ell+1,h]=\emptyset\\ [\ell,h]+k'\mathbb{Z} & \text{otherwise}\end{cases}
\end{aligned}
$$

where gcd is the greatest common divisor, and $\gcd(0,x)=\gcd(x,0)=x$.

Figure 8: Abstract operators in the modular interval domain $\mathcal{D}^\sharp_m$.

For instance, $[\ell_1,h_1] +^\sharp_i [\ell_2,h_2] \stackrel{\text{def}}{=} [\ell_1+\ell_2,h_1+h_2]$. Additionally, an abstraction $\cup^\sharp_i$ of the set union $\cup$, as well as a widening $\triangledown^\sharp_i$ [9] enforcing termination are required. We note that the optimal abstraction of *wrap* is:

$$
\begin{aligned}
&\mathit{wrap}^\sharp_i([\ell,h],[\ell',h']) =\\
&\quad \begin{cases} [\mathit{wrap}(\ell,[\ell',h']),\ \mathit{wrap}(h,[\ell',h'])] & \text{if } (\ell'+(h'-\ell'+1)\mathbb{Z})\cap[\ell+1,h]=\emptyset\\ [\ell',h'] & \text{otherwise}\end{cases}
\end{aligned}
$$

which returns the full interval $[\ell',h']$ when $[\ell,h]$ crosses a boundary in $\ell'+(h'-\ell'+1)\mathbb{Z}$. This is the case when $\{\, \mathit{wrap}(v,[\ell',h']) \mid v \in [\ell,h] \,\}$ is not convex.

Example. Consider the function `add1` in Fig. 1.a and assume that, in the abstract, the computed range for both arguments is $[-1,1]$. Then `(unsigned char)`$[-1,1]$ is abstracted as $[0,255]$. This is the best interval abstraction as, in the concrete, the value set is $\{0,1,255\}$. The following interval addition, which is computed in type `int` due to integer promotion, gives $[0,510]$. Once cast back to signed `char`, the interval result is the full range $[-128,127]$. This is much coarser than the concrete result, which is $[-2,2]$.

2.7 Modular Interval Domain $\mathcal{D}^\sharp_m$

We now propose a slight variation on the interval domain that corrects the precision loss in $\mathit{wrap}^\sharp_i$. It is defined as intervals with an extra modular component:

$$
\begin{aligned}
\mathcal{D}^\sharp_m &\stackrel{\text{def}}{=} \{\, [\ell,h]+k\mathbb{Z} \mid \ell,h \in \mathbb{Z}\cup\{\,\pm\infty\,\}, k\in\mathbb{N} \,\}\\
\gamma_m([\ell,h]+k\mathbb{Z}) &\stackrel{\text{def}}{=} \{\, x+ky \mid \ell \le x \le h,\ y\in\mathbb{Z} \,\}
\end{aligned}
$$

The domain was mentioned briefly in [14] but not defined formally. It is also similar to the interval congruences $\theta.[\ell,u]\langle m\rangle$ by Masdupuy [15], but with θ set to 1, which results in simpler abstract operations (abstract values with $\theta \neq 1$ are useful when modeling array accesses, as in [15], but not when modeling wrap-around).

Abstract operators $\circ^\sharp_m$ are defined in Fig. 8. There is no longer a best abstraction in general (e.g., $\{\,0,2,4\,\}$ could be abstracted as either $[0,4]+0\mathbb{Z}$ or $[0,0]+2\mathbb{Z}$ which are both minimal and

yet incomparable), which makes the design of operators with a guaranteed precision difficult. We designed $\texttt{~}^\sharp_m$, $\texttt{-}^\sharp_m$, $\texttt{+}^\sharp_m$, $\texttt{-}^\sharp_m$, and $\cup^\sharp_m$ based on optimal interval and simple congruence operators [11], using basic coset identities to infer modular information. The result may not be minimal (in the sense of minimizing $\gamma_m(x^\sharp \circ^\sharp_m y^\sharp)$) but suffices in practice. For other operators, we simply revert to classic intervals, discarding any modulo information.

We now focus our attention on $wrap^\sharp_m([\ell, h] + k\mathbb{Z}, [\ell', h'])$. Similarly to intervals, wrapping $[\ell, h]+k\mathbb{Z}$ results in a plain interval if no $[\ell, h]+ky$, $y \in \mathbb{Z}$ crosses a boundary in $\ell'+(h'-\ell'+1)\mathbb{Z}$, in which case the operation is exact. Otherwise, it returns the interval argument $[\ell, h]$ modulo both $h'-\ell'+1$ and k. This forgets that the result is bounded by $[\ell', h']$ but keeps an important information: the values ℓ and h. In practice, we maintain the range $[\ell', h']$ by performing a reduced product between $\mathcal{D}_m^\sharp$ and plain intervals $\mathcal{D}_i^\sharp$, which ensures that each operator (except the widening) is at least as precise as in an interval analysis.

Example. Consider again the function `add1` from Fig. 1.a, with abstract arguments $[-1, 1] + 0\mathbb{Z}$. Then, `e = (unsigned char)`[-1,1] is abstracted as $[-1, 1]+256\mathbb{Z}$. Thus, `e+e` gives $[-2, 2]+256\mathbb{Z}$. Finally, `(char)(e+e)` gives back the expected interval: $[-2, 2] + 0\mathbb{Z}$.

2.8 Bitfield Domain $\mathcal{D}_b^\sharp$

The interval domain $\mathcal{D}_i^\sharp$ is not very precise on bit-level operators. On the example of Fig. 2.b, (*range*(`int`) `& 0x801fffff`) `| 0x3fe00000` is abstracted as *range*(`int`), which does not capture the fact that some bits are fixed to 0 and others to 1. This issue can be solved by a simple domain that tracks the value of each bit independently. A similar domain was used by Monniaux when analyzing unsigned 32-bit integer computations in a device driver [23]. Our version, however, abstracts a $\mathbb{Z}-$based concrete semantics, making it independent from bit-size and signedness. The domain associates to each variable two integers, z and o, that represent the bit masks for bits that can be set respectively to 0 and to 1:

$$\begin{array}{rcl} \mathcal{D}_b^\sharp & \stackrel{\text{def}}{=} & \mathbb{Z} \times \mathbb{Z} \\ \gamma_b(z, o) & \stackrel{\text{def}}{=} & \{\, b \mid \forall i \geq 0 : (\neg p(b)_i) \wedge p(z)_i \text{ or } p(b)_i \wedge p(o)_i \,\} \\ \alpha_b(S) & \stackrel{\text{def}}{=} & (\vee \{\, \neg p(b) \mid b \in S \,\}, \vee \{\, p(b) \mid b \in S \,\}) \end{array}$$

where p is defined in (1). The optimal abstract operators can be derived through the Galois connection $\mathcal{P}(\mathbb{Z}) \underset{\alpha_b}{\overset{\gamma_b}{\leftrightarrows}} \mathbb{Z} \times \mathbb{Z}$. We present the most interesting ones in Fig. 9. They use the bit-wise operators on $\mathbb{Z}$ defined in (2). For bit shifts, we only handle the case where the right argument represents a positive singleton (i.e., it has the form $n_b^\sharp$ for some constant $n \geq 0$). Wrapping around an unsigned interval $[0, 2^n - 1]$ is handled by masking high bits. Wrapping around a signed interval $[-2^n, 2^n - 1]$ additionaly performs a sign extension. Our domain has infinite increasing chains (e.g., $X_n^\sharp = (-1, 2^n - 1)$), and so, requires a widening: $\triangledown_b^\sharp$ will set all the bits to 0 (resp. 1) if the mask for bits at 0 (resp. at 1) is not stable.

Efficiency. The three domains $\mathcal{D}_i^\sharp$, $\mathcal{D}_m^\sharp$, $\mathcal{D}_b^\sharp$ are non-relational, and so, very efficient. Using functional maps, even joins and widenings can be implemented with a sub-linear cost in practice [7, §III.H.1]. Each abstract operation costs only a few integer operations. Moreover, the values encountered during an analysis generally fit in a machine word. Our analyzer uses an arbitrary precision library able to exploit this fact to improve the performance [22].

$$
\begin{array}{rcl}
n_b^\sharp & \stackrel{\text{def}}{=} & (\sim n, n) \qquad (n \in \mathbb{Z}) \\
\sim_b^\sharp(z, o) & \stackrel{\text{def}}{=} & (o, z) \\
(z_1, o_1) \,\&_b^\sharp\, (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \mid z_2,\ o_1 \,\&\, o_2) \\
(z_1, o_1) \mid_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \,\&\, z_2,\ o_1 \mid o_2) \\
(z_1, o_1) \,\hat{}_b^\sharp\, (z_2, o_2) & \stackrel{\text{def}}{=} & ((z_1 \,\&\, z_2) \mid (o_1 \,\&\, o_2),\ (z_1 \,\&\, o_2) \mid (o_1 \,\&\, z_2)) \\
(z_1, o_1) \ll_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & ((z_1 \ll n) \mid ((1 \ll n) - 1),\ o_1 \ll n),\ \text{when } \exists n \geq 0 : (z_2, o_2) = n_b^\sharp \\
(z_1, o_1) \gg_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \gg n,\ o_1 \gg n),\ \text{when } \exists n \geq 0 : (z_2, o_2) = n_b^\sharp \\
(z_1, o_1) \cup_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \mid z_2,\ o_1 \mid o_2) \\
(z_1, o_1) \triangledown_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \triangledown z_2,\ o_1 \triangledown o_2) \text{ with } x \triangledown y \stackrel{\text{def}}{=} \text{ if } x = x \mid y \text{ then } x \text{ else } -1 \\
\mathit{wrap}_b^\sharp((z, o), [0, 2^n - 1]) & \stackrel{\text{def}}{=} & (z \mid (-2^n),\ o \,\&\, (2^n - 1)) \\
\mathit{wrap}_b^\sharp((z, o), [-2^n, 2^n - 1]) & \stackrel{\text{def}}{=} & ((z \,\&\, (2^n - 1)) \mid (-2^n z_n),\ (o \,\&\, (2^n - 1)) \mid (-2^n o_n))
\end{array}
$$

Figure 9: Abstract operators in the bitfield domain $\mathcal{D}_b^\sharp$. See also (2).

$$
\begin{array}{l}
\mathit{synt} : (\mathcal{L} \times \mathcal{P}(\mathcal{L})) \to \mathit{expr} \\
\mathit{synt}((V, o, t), C) \stackrel{\text{def}}{=} \\
\quad \begin{cases}
c & \text{if } c = (V, o, t) \in C \\
(t)\, c & \text{if } c = (V, o, t') \in C \wedge t, t' \in \mathit{int\text{-}type} \wedge \mathit{bitsize}(t) = \mathit{bitsize}(t') \\
\mathit{hi\text{-}word\text{-}of\text{-}dbl}(c) & \text{if } c = (V, o, t') \in C \wedge t \in \mathit{int\text{-}type} \wedge t' = \texttt{double} \wedge \mathit{bitsize}(t) = 32 \\
\mathit{dbl\text{-}of\text{-}word}(c_1, c_2) & \text{if } c_1 = (V, o, t') \in C \wedge c_2 = (V, o + 4, t') \in C \wedge t = \texttt{double} \wedge \\
 & \quad t' \in \mathit{int\text{-}type} \wedge \mathit{bitsize}(t') = 32 \\
\mathit{range}(t) & \text{otherwise}
\end{cases}
\end{array}
$$

Figure 10: Cell synthesis to handle physical casts. A big endian architecture is assumed.

3 Memory Abstract Domain

A prerequisite to analyze the programs in Fig. 2 is to detect and handle physical casts, that is, the re-interpretation of a portion of memory as representing an object of different type through the use of union types (Fig. 2.a) or pointer casts (Fig. 2.b).

In this section, we are no longer restricted to integers and consider arbitrary C expressions and types. Nevertheless our first step is to reduce expressions to the following simplified grammar:

$$
\begin{array}{lcl}
\mathit{expr} & ::= & n \mid \&V \mid \diamond\ \mathit{expr} \mid \mathit{expr} \circ \mathit{expr} \mid \texttt{(}\mathit{scalar\text{-}type}\texttt{)}\ \mathit{expr} \mid \texttt{*}_{\mathit{scalar\text{-}type}}\ \mathit{expr} \\
\mathit{stat} & ::= & (\texttt{*}_{\mathit{scalar\text{-}type}}\ \mathit{expr}) = \mathit{expr} \mid \mathit{assert}(\mathit{expr}) \\
\mathit{scalar\text{-}type} & ::= & \mathit{int\text{-}type} \mid \texttt{float} \mid \texttt{double} \mid \texttt{char *}
\end{array}
$$

Memory accesses (including field and array accesses, variable reads and modifications) are through typed dereferences $\texttt{*}_t\ e$, and pointer arithmetics is reduced to arithmetics on byte-based memory addresses. The translation to such expressions can be performed statically by a front-end. For instance, in Fig. 2.a, the statement `y.i[1] = i ^ x.i[1]` is translated into $\texttt{*}_{\texttt{int}}\ (\texttt{\&y} + 4) = (\texttt{*}_{\texttt{int}}\ \texttt{\&i})\ \texttt{\^{}}\ (\texttt{*}_{\texttt{int}}\ (\texttt{\&x} + 4))$. We do not detail this translation further.

A low-level semantics would model memory states as maps in $\{\, (V, i) \in \mathcal{V} \times \mathbb{N} \mid i < \mathit{bitsize}(\tau(V)) \,\} \to \{\, 0, 1 \,\}$ from bit positions to bit values. We consider instead a slightly higher-level concrete semantics: the memory is a collection of cells in $\mathcal{L} \stackrel{\text{def}}{=} \{\, (V, o, t) \in$

$\mathcal{V} \times \mathbb{N} \times$ *scalar-type* $\mid o + \textit{sizeof}(t) \leq \textit{sizeof}(V)\}$, where V is the variable the cell lives in, o is its offset in bytes from the start of V, and t is the cell type (integer, float, or pointer). A concrete environment ρ is a partial function from cells in $\mathcal{L}$ to values in $\mathbb{V}$, where $\mathbb{V}$ contains all the integer, float, and pointer values. Pointer values are represented as pairs composed of a variable name $V \in \mathcal{V}$ and a byte offset $o \in \mathbb{N}$ from the start of the variable, and written as $\&V + o$. Cells are added on-demand when writing into memory: an assignment $*_t\ e = e'$ creates a cell (V, o, t) for each pointer $\&V + o$ pointed to by e. Reading a cell (V, o, t) from a memory with cell set $C \subseteq \mathcal{L}$ returns $\rho(V, o, t)$ if $(V, o, t) \in C$. If $(V, o, t) \notin C$, we try to synthesize its value. An example synthesis function, *synt*, is presented in Fig. 10: it returns an expression over-approximating the value of a cell (V, o, t) using only cells in C. Firstly, if t is an integer type and there is an integer cell $(V, o, t') \in C$ with the same size but $t \neq t'$, its value is converted to t — i.e., a physical cast `*((int*)&V)` is treated as a regular cast `(int)V`. Secondly, selected bit-level manipulations of floats are detected and translated into expressions using two new operators: *hi-word-of-dbl* and *dbl-of-word*, that denote, respectively, extracting the most significant 32-bit integer word of a 64-bit double and building a 64-bit double from two words (this semantics is formalized in Sec. 4.1). Thirdly, if no synthesis can occur, the whole range of t is returned.

This concrete semantics is then abstracted in a straightforward way: a specific memory domain maintains a set $C \subseteq \mathcal{L}$ of current cells and delegates the abstraction of $\mathcal{P}(C \to \mathbb{V})$ to an underlying domain. Pointer values are abstracted by maintaining an explicit set of pointed-to variables $C \to \mathcal{P}(\mathcal{V})$ and delegating the abstraction of offsets to a numeric domain. Given an expression and an abstract environment, the memory domain resolves pointer dereferences, synthesizes cell values, and translates dynamically expressions into simple numeric expressions, similar to Fig. 4 extended with the operators introduced during synthesis. Such expressions can then be handled directly by numeric domains, as shown in Sec. 4.2.

This memory domain was introduced in [19] and we will not discuss it further. Note however that the synthesis was limited to integer cells in [19], while we extend it here to express some bit-level float manipulations.[3]

Example. In Fig. 2.a, writing into `x.i[0]` and `x.i[1]` creates the cells $c_1 = (\mathtt{x}, 0, \mathtt{int})$ and $c_2 = (\mathtt{x}, 4, \mathtt{int})$. Reading back `x.d` amounts to evaluating the expression *dbl-of-word*(c_1, c_2). In Fig. 2.b, the expression `(*p & 0x7fe00000) >> 20` is translated into `(`*hi-word-of-dbl*(c) `& 0x7fe00000) >> 20`, where the cell $c = (\mathtt{d}, 0, \mathtt{double})$ represents the variable `d`.

4 Floating-Point Abstract Domains

4.1 Concrete Bit-Level Floating-Point Semantics

We now consider the analysis of programs manipulating floating-point numbers. Due to the limited precision of computers, float arithmetics exhibits rounding errors. For many purposes, it is sufficient to model floats as reals, with rounding abstracted as a non-deterministic choice in an error interval. This permits the use of the classic abstract domains defined on rationals or reals (intervals, but also relational domains [17]) to model floating-point computations. However, this is not sufficient when the bit representation of numbers is exploited, as in Fig. 2.

We introduce a bit-level semantics based on the ubiquitous IEEE 754-1985 standard [13]. We focus here on double-precision numbers, which occupy 64 bits: a 1-bit sign s, a 11-bit

[3] Our analyzer (Sec. 5) extends this further to synthesize and decompose integers, bitfields, and 32-bit floats (not presented here for the sake of conciseness).

$$
\begin{array}{l}
\mathit{dbl} : \{0,1\}^{64} \to \mathbb{V} \\
\mathit{dbl}(s, e_{10}, \ldots e_0, m_0 \ldots, m_{51}) \stackrel{\text{def}}{=} \\
\quad \begin{cases}
(-1)^s \times (1 + \sum_{i=0}^{51} 2^{-i-1} m_i) \times 2^{(\sum_{i=0}^{10} 2^i e_i - 1023)} & \text{if } \sum_{i=0}^{10} 2^i e_i \notin \{\, 0, 2047 \,\} \\
(-1)^s \times (\sum_{i=0}^{51} 2^{-i-1} m_i) \times 2^{-1022} & \text{if } \forall i : e_i = 0 \\
(-1)^s \times \infty & \text{if } \forall i : e_i = 1 \wedge \forall j : m_j = 0 \\
\mathit{NaN} & \text{if } \forall i : e_i = 1 \wedge \exists j : m_j = 1
\end{cases} \\
[\![\, \mathit{dbl\text{-}of\text{-}word}(e_1, e_2) \,]\!]\rho \stackrel{\text{def}}{=} \{\, \mathit{dbl}(b^1_{31}, \ldots, b^1_0, b^2_{31}, \ldots, b^2_0) \mid \\
\qquad\qquad \forall j \in \{1,2\} : \sum_{i=0}^{31} 2^i b^j_i \in [\![\, \mathit{wrap}(e_j, [0, 2^{32} - 1]) \,]\!]\rho \,\} \\
[\![\, \mathit{hi\text{-}word\text{-}of\text{-}dbl}(e) \,]\!]\rho \stackrel{\text{def}}{=} \{\, \sum_{i=0}^{31} 2^i b_{i+32} \mid \exists b_0, \ldots, b_{31} : \mathit{dbl}(b_{63}, \ldots, b_0) \in [\![\, e \,]\!]\rho \,\}
\end{array}
$$

Figure 11: Bit-level concrete semantics of floating-point numbers, extending Fig. 7.

exponent e_0 to e_{10}, and a 52-bit mantissa m_0 to m_{51}. The mapping from bit values to float values is described by the function *dbl* in Fig. 11. In addition to normalized numbers (of the form $\pm 1.m_0 m_1 \cdots \times 2^x$), the standard allows denormalized (i.e., small) numbers, signed infinities $\pm\infty$, and NaNs (*Not a Numbers*, standing for error codes). This representation gives a concrete semantics to the operators *dbl-of-word* and *hi-word-of-dbl* introduced by cell synthesis. As in the case of integers, legacy abstract domains can be adapted to our new semantics by defining an abstraction for our new operators. Straightforward ones would state that $\mathit{dbl\text{-}of\text{-}word}^\sharp(e_1, e_2) = \mathit{range}(\texttt{double})$ and $\mathit{hi\text{-}word\text{-}of\text{-}dbl}^\sharp(e) = \mathit{range}(\texttt{unsigned})$, but we propose more precise ones below.

4.2 Predicate Domain on Binary Floating-Point Numbers $\mathcal{D}_p^\sharp$

The programs in Fig. 2 are idiomatic. It is difficult to envision a general domain that can reason precisely about arbitrary binary floating-point manipulations. Instead, we propose a lightweight and extensible technique based on pattern matching of selected expression fragments. However, matching each expression independently is not sufficient: it provides only a local view that cannot model computations spread across several statements precisely enough. We need to infer and propagate semantic properties to gain in precision.

To analyze Fig. 2.a, we use a domain $\mathcal{D}_p^\sharp$ of predicates of the form $V = e$, where $V \in \mathcal{V}$ and e is an expression chosen from a fixed list $\mathbb{P}$ with a parameter $W \in \mathcal{V}$. At most one predicate is associated to each V, so, an abstract element is a map from $\mathcal{V}$ to $\mathbb{P} \cup \{\, \top \,\}$, where $\top$ denotes the absence of a predicate:

$$
\begin{array}{l}
\mathcal{D}_p^\sharp \stackrel{\text{def}}{=} \mathcal{V} \to (\mathbb{P} \cup \{\, \top \,\}) \quad \text{where:} \\
\quad \mathbb{P} \quad ::= \quad W \,\hat{}\, \texttt{0x80000000} \qquad\qquad (W \in \mathcal{V}) \\
\qquad\quad\;\; | \quad \mathit{dbl\text{-}of\text{-}word}(\texttt{0x43300000}, W) \quad (W \in \mathcal{V}) \\
\gamma_p(X_p^\sharp) \stackrel{\text{def}}{=} \{\, \rho \in \mathcal{E} \mid \forall V \in \mathcal{V} : X_p^\sharp(V) = \top \vee \rho(V) \in [\![\, X_p^\sharp(V) \,]\!]\rho \,\}
\end{array}
\tag{3}
$$

The concretization is the set of environments that satisfy all the predicates, and there is no Galois connection. Figure 12 presents a few example transfer functions. Assignments and tests operate on a pair of abstractions: a predicate $X_p^\sharp$ and an interval $X_i^\sharp$. Sub-expressions from e are combined by *combine* with predicates from $X_p^\sharp$ to form idioms. The assignment then removes (in $Y_p^\sharp$) the predicates about the modified variable ($\mathit{var}(p)$ is the set of variables appearing in p) and tries to infer a new predicate. The matching algorithm is not purely syntactic: it uses a semantic interval information from $X_i^\sharp$ (e.g., to evaluate sub-expressions). Dually, successfully

$$
\begin{array}{l}
[\![V = e]\!]_p^\sharp(X_p^\sharp, X_i^\sharp) \stackrel{\text{def}}{=} \\
\quad \text{let } Y_i^\sharp = [\![V = \mathit{combine}(e, X_p^\sharp, X_i^\sharp)]\!]_i^\sharp \; X_i^\sharp \text{ in} \\
\quad \text{let } Y_p^\sharp = \lambda W : \text{if } W = V \text{ or } V \in \mathit{var}(X_p^\sharp(W)) \text{ then } \top \text{ else } X_p^\sharp(W) \text{ in} \\
\quad \text{if } e = W \texttt{ \^{} } e_1 \wedge [\![e_1]\!]_i^\sharp X_i^\sharp \in \{[2^{31}, 2^{31}], [-2^{31}, -2^{31}]\} \\
\qquad \text{then } (Y_p^\sharp[V \mapsto W \texttt{ \^{} } \texttt{0x80000000}], Y_i^\sharp) \\
\quad \text{else if } e = \mathit{dbl\text{-}of\text{-}word}(e_1, W) \wedge [\![e_1]\!]_i^\sharp X_i^\sharp = [1127219200, 1127219200] \\
\qquad \text{then } (Y_p^\sharp[V \mapsto \mathit{dbl\text{-}of\text{-}word}(\texttt{0x43300000}, W)], Y_i^\sharp) \\
\quad \text{otherwise } (Y_p^\sharp, Y_i^\sharp)
\end{array}
$$

$$[\![\mathit{assert}(e)]\!]_p^\sharp(X_p^\sharp, X_i^\sharp) \stackrel{\text{def}}{=} (X_p^\sharp, [\![\mathit{assert}(\mathit{combine}(e, X_p^\sharp, X_i^\sharp))]\!]_i^\sharp X_i^\sharp)$$

$$X_p^\sharp \cup^\sharp Y_p^\sharp \stackrel{\text{def}}{=} \lambda V : \text{if } X_p^\sharp(V) = Y_p^\sharp(V) \text{ then } X_p^\sharp(V) \text{ else } \top$$

$\mathit{combine}(e, X_p^\sharp, X_i^\sharp)$ replaces sub-expressions V_1 `-` V_2 in e with `(double)`I when:
$\exists V_1', V_2' : \forall j \in \{1, 2\} : X_p^\sharp(V_j) = \mathit{dbl\text{-}of\text{-}word}(\texttt{0x43300000}, V_j') \wedge$
$X_p^\sharp(V_1') = I \texttt{ \^{} } \texttt{0x80000000} \wedge X_i^\sharp(V_2') = [-2^{31}, -2^{31}]$

Figure 12: Abstract operator examples in the predicate domain $\mathcal{D}_p^\sharp$.

```
double frexp(double d, int* e) {
  int x = 0;
  double r = 1, dd = fabs(d);
  if (dd >= 1) { while ★ (dd > r) { x++; ♦ r *= 2; } }
  else { while (dd < r) { x--; r /= 2; } }
  *e = x;
  return d/r;
}
```

Figure 13: Floating-point decomposition into mantissa and exponent.

matched idioms refine the interval information. The join on $\mathcal{D}_p^\sharp$ removes the predicates that are not identical. As $\mathcal{D}_p^\sharp$ is flat, it has no need for a widening. Similarly to non-relational domains, the cost of each operation is sub-linear when implemented with functional maps [7, §III.H.1].

To stay concise, we only present the transfer functions sufficient to analyze Fig. 2.a. It is easy to enrich $\mathbb{P}$ with new predicates and extend the pattern matching and the propagation rules to accommodate other idioms. For instance, Fig. 2.b can be analyzed by adding the predicate $V = \mathit{hi\text{-}word\text{-}of\text{-}dbl}(W)$ and using information gathered by the bitfield domain $\mathcal{D}_b^\sharp$ during pattern matching.

4.3 Exponent Domain $\mathcal{D}_e^\sharp$

As last example, we consider the program in Fig. 13 that decomposes a float into its exponent and mantissa. Although it is possible to extract exponents using bit manipulations, as in Fig. 2.b, this function uses another method which illustrates the analysis of loops: when $|\texttt{d}| \geq 1$ it computes $\texttt{r} = 2^{\texttt{x}}$ iteratively, incrementing `x` until $\texttt{r} \geq |\texttt{d}|$. This example is, as those in Figs. 1–2, inspired from actual industrial codes and out of scope of existing abstract domains.

To provide precise bounds for the returned value, a first step is to bound `r`. We focus on

	size	with domains		w/o domains		pre-processed	
	(KLoc)	time	alarms	time	alarms	time	alarms
	154	10h44	22	10h04	22	11h38	22
	186	7h44	10	7h22	10	7h16	10
	103	54mn	**2**	44mn	451	46mn	6
(a)	493	7h34	**3**	15h27	1,833	8h40	195
	661	14h46	**2**	16h23	3,419	13h32	253
	616	22h03	**5**	26h46	5,350	20h45	300
	859	65h08	**110**	41h03	5,968	59h55	316
	2,428	48h28	**1**	44h06	3,822	44h57	674
(b)	113	25mn	30	20mn	30	17mn	30
(c)	79	3h22	7	3h09	7	3h27	7
(d)	102	46mn	64	59mn	64	59mn	64
	1,727	30h18	**2,133**	26h15	2,388	28h46	2,190

Figure 14: Analysis performance on industrial benchmarks.

the case where $|\mathtt{d}| \geq 1$. To prove that $\mathtt{r}$ and $\mathtt{d}/\mathtt{r}$ are bounded, it is necessary to infer at ★ the loop invariant $\mathtt{r} \leq 2\mathtt{d}$ and use the loop exit condition $\mathtt{d} \leq \mathtt{r}$. Several solutions exist to infer this invariant (such as using the polyhedra domain). We advocate the use of a variation $\mathcal{D}_e^\sharp$ of the, more efficient, zone domain [16]. While the original domain infers invariants of the form $V - W \in [\ell, h]$, we infer invariants of the form $V/W \in [\ell, h]$:

$$\begin{array}{rcl} \mathcal{D}_e^\sharp & \stackrel{\text{def}}{=} & (\mathcal{V} \times \mathcal{V}) \to \{\, [\ell, h] \mid \ell, h \in \mathbb{R} \cup \{\, \pm\infty \,\} \,\} \\ \gamma_e(X_e^\sharp) & \stackrel{\text{def}}{=} & \{\, \rho \in \mathcal{E} \mid \forall V, W \in \mathcal{V} : \rho(W) = 0 \vee \rho(V)/\rho(W) \in X_e^\sharp(V, W) \,\} \\ \alpha_e(X) & \stackrel{\text{def}}{=} & \lambda(V, W) : [\min \{\, \rho(V)/\rho(W) \mid \rho \in X \,\}, \max \{\, \rho(V)/\rho(W) \mid \rho \in X \,\}] \end{array}$$

The domain is constructed by applying a straightforward log transformation to [16]. A near linear cost can be achieved by using packing techniques [7, §III.H.5].

Now that $\mathtt{r}$ is bounded, in order to prove that $\mathtt{x}$ is also bounded, it is sufficient to infer a relationship between $\mathtt{x}$ and $\mathtt{r}$, i.e., $\mathtt{r} = 2^{\mathtt{x}}$ at ★, and $\mathtt{r} = 2^{\mathtt{x}-1}$ at ♦. This is possible, for instance, by using the predicate domain $\mathcal{D}_p^\sharp$ from Sec. 4.2 enriched with a new predicate parameterized by a variable W and an integer i:

$$\mathbb{P}' ::= \mathbb{P} \mid 2^{W+i} \qquad (W \in \mathcal{V},\, i \in \mathbb{Z})$$

To stay concise, we do not describe the adapted transfer functions.

5 Experimental Results

All the domains we presented have been incorporated into the Astrée static analyzer for synchronous embedded C programs [3, 7] and its extension AstréeA for multi-threaded programs [21]. Both check for arithmetic and memory errors (integer and float overflows and invalid operations, array overflows, invalid pointer dereferences). Thanks to a modular design based on a partially reduced product of communicating domains, new domains can be easily added.

Figure 14 presents the running-time and number of alarms when analyzing large (up to 2.5 MLoc) C applications from the aeronautic industry. Figure 14.a corresponds to a family

of control-command software; each one consists in a single very large reactive loop generated from a graphical specification language (similar to Lustre) and features much floating-point computations (integrations, digital filters, etc.). Figure 14.b is a software to perform hardware checks; it features many loops and is mainly integer-based. Figures 14.c–d are embedded communication software that manipulate strings and buffers; they feature much physical casts to pack, transmit, and unpack typed and untyped messages, as well as some amount of boolean and numeric control. Additionally, the applications in Fig. 14.d are multi-threaded. More details on the analyzed applications are available in [7, III.M] and [21].

In all the tests, the default domains are enabled (intervals, octagons, binary decision trees, etc., we refer the reader to [7] for an exhaustive list). The first and second columns show, respectively, the result with and without our new domains. In many cases (shown in boldface), our domains strictly improve the precision. Moreover, the improved analysis is between twice slower and twice faster. This variation can be explained as follows: adding new domains incurs a cost per abstract operation, but improving the precision may decrease or increase the number of loop iterations to reach a fixpoint. In the third column, our new domains are disabled but the code is pre-processed with ad-hoc syntactic scripts to (partially) remove their need (e.g., replacing the `cast` function in Fig. 1.b with a C cast), which is possibly unsound and incomplete (hence the remaining alarms). Comparing the first and last columns shows that being sound does not cause a significant loss of efficiency.

6 Conclusion

We presented several abstract domains to analyze C codes exploiting the binary representation of integers and floats. They are based on a concrete semantics that allows reasoning about these low-level implementation aspects in a sound way, while being compatible with legacy abstract domains. Each introduced domain focuses on a specific coding practice, so, they are not general. However, they are simple, fast to design and implement, efficient, and easy to extend. They have been included in the industrial-strength analyzer Astrée [3] to supplement existing domains and enable the precise analysis of codes exploiting the binary representation of numbers without resorting to unsound source pre-processing.

Future work includes generalizing our domains and developing additional domains specialized to code patterns we will encounter when analyzing new programs, with the hope of building a library of domains covering most analysis needs.

References

[1] GMP: The GNU multiple precision arithmetic library. `http://gmplib.org/`.

[2] How do I get rid of red OVFL outside PolySpace for Model Link TL? `http://www.mathworks.fr/support/solutions/en/data/1-5D3ONT/`.

[3] AbsInt, Angewandte Informatik. Astrée run-time error analyzer. `http://www.absint.com/astree`.

[4] ANSI Technical Committee and ISO/IEC JTC 1 Working Group. Rationale for international standard, Programming languages, C. Technical Report 897, rev. 2, ANSI, ISO/IEC, Oct. 1999.

[5] ANSI Technical Committee and ISO/IEC JTC 1 Working Group. C standard. Technical Report 9899:1999(E), ANSI, ISO & IEC, 2007.

[6] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI'01)*, pages 203–213. ACM, 2001.

[7] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385, pages 1–38. AIAA, Apr. 2010.

[8] J. Brauer and A. King. Transfer function synthesis without quantifier elimination. In *Proc. of the 20th European Symp. on Prog. (ESOP'11)*, volume 6602 of *LNCS*, pages 97–115. Springer, Mar. 2011.

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ISOP'76*, pages 106–130. Dunod, Paris, France, 1976.

[10] dSpace. TargetLink code generator. `http://www.dspaceinc.com`.

[11] P. Granger. Static analysis of arithmetic congruences. *Int. Journal of Computer Mathematics*, 30:165–199, 1989.

[12] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proc. of the Int. Joint Conf. on Theory and Practice of Soft. Development (TAPSOFT'91)*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

[13] IEEE Computer Society. Standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std. 745-1985, 1985.

[14] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the absence of runtime errors. In *Embedded Real Time Soft. and Syst. (ERTS^2 2010)*, pages 1–9, May 2010.

[15] F. Masdupuy. Semantic analysis of interval congruences. In *Proc. of the Int. Conf on Formal Methods in Prog. and Their Applications (FMPTA'93)*, volume 735 of *LNCS*, pages 142–155. Springer, 1993.

[16] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. of the 2d Symp. on Programs as Data Objects (PADO II)*, volume 2053 of *LNCS*, pages 155–172. Springer, May 2001.

[17] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. of the European Symp. on Prog. (ESOP'04)*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[18] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, Dec. 2004.

[19] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM Conf. on Lang., Compilers, and Tools for Embedded Syst. (LCTES'06)*, pages 54–63. ACM, June 2006.

[20] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[21] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Proc. of the 20th European Symp. on Prog. (ESOP'11)*, volume 6602 of *LNCS*, pages 398–418. Springer, Mar. 2011.

[22] A. Miné, X. Leroy, and P. Cuoq. ZArith: Arbitrary precision integers library for OCaml. `http://forge.ocamlcore.org/projects/zarith`.

[23] D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In *Proc. of the 7th ACM & IEEE Int. Conf. on Embedded Soft. (EMSOFT'07)*, pages 30–36. ACM, Sep. 2007.

[24] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *Proc. of the 14th European Symp. on Prog. (ESOP'05)*, volume 3444 of *LNCS*, pages 46–60. Springer, Apr. 2005.

[25] J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *Proc. of the ACM Conf. on Lang., Compilers, and Tools for Embedded Syst. (LCTES'06)*, pages 34–43. ACM, June 2006.

[26] A. Simon and A. King. Taming the wrapping of integer arithmetic. In *Proc. of the 14th Int. Symp. on Static Analysis (SAS'07)*, volume 4634 of *LNCS*, pages 121–136. Springer, Aug. 2007.