Pre-proceedings for TAPAS 2019

The 10th Workshop on Tools for Automatic Program Analysis

Affiliated to SAS 2019
The 26th Static Analysis Symposium

Part of
the 3rd World Congress
on Formal Methods

Porto, Portugal 8 October 2019

Preface

In recent years, a wide range of static analysis tools have emerged, some of which are currently in industrial use or are well beyond the advanced prototype level. Many impressive practical results have been obtained, which allow complex properties to be proven or checked in a fully or semi-automatic way, even in the context of complex software developments. In parallel, the techniques to design and implement static analysis tools have improved significantly, and much effort is being put into engineering the tools.

TAPAS 2019, *the 10th Workshop on Tools for Automatic Program Analysis* is intended to promote discussions and exchange experience between specialists in all areas of program analysis design and implementation and static analysis tool users.

These pre-proceedings enclose the contrubutions to TAPAS 2019. Besides these contributions, the program of that tenth edition will feature two invited talks and one invited keynote.

We would like to thank everyone who was involved in the organization of the workshop. We are very thankful for the members of the Programme Committee for their evaluation work, and for all the discussions on the organization of the event.

Finally, we would also like to thank the authors and the invited speakers for their contributions to the programme of TAPAS 2019.

October 2019 David Delmas

Organization

Program Committee Chair

David Delmas	Airbus and Sorbonne Université, France

Program Committee

Fausto Spoto	Università di Verona, Italy
Caterina Urban	Inria, France
Franck Vedrine	CEA LIST, France
Jules Villard	Facebook, UK
Jingling Xue	University of New South Wales, Australia
Tomofumi Yuki	Inria, France
Sarah Zennou	Airbus, France

Table of Contents

IV Organization

PrideMM: Second Order Model Checking for Memory Consistency Models

Simon Cooksey[1], Sarah Harris[1], Mark Batty[1], Radu Grigore[1], and Mikoláš Janota[2]

[1] University of Kent, Canterbury
{sjc205,seh53,mjb211,rg399}@kent.ac.uk
[2] IST/INESC-ID, University of Lisbon

Abstract. We present PrideMM, an efficient model checker for second-order logic enabled by recent breakthroughs in quantified satisfiability solvers. We argue that second-order logic sits at a sweet spot: constrained enough to enable practical solving, yet expressive enough to cover an important class of problems not amenable to (non-quantified) satisfiability solvers. To the best of our knowledge PrideMM is the first automated model checker for second-order logic formulae.

We demonstrate the usefulness of PrideMM by applying it to problems drawn from recent work on memory specifications, which define the allowed executions of concurrent programs. For traditional memory specifications, program executions can be evaluated using a satisfiability solver or using equally powerful ad hoc techniques. However, such techniques are insufficient for handling some emerging memory specifications.

We evaluate PrideMM by implementing a variety of memory specifications, including one that cannot be handled by satisfiability solvers. In this problem domain, PrideMM provides usable automation, enabling a modify-execute-evaluate pattern of development where previously manual proof was required.

1 Introduction

This paper presents PrideMM, an efficient model checker for second-order (SO) logic. PrideMM is used to automatically evaluate tests under the intricate memory specifications[3] of aggressively optimised *concurrent* languages, where no automated solution currently exists, and it is compared to existing tools over a simpler class of memory specifications.

We argue that SO logic is a sweet spot: restrictive enough to enable efficient solving, yet expressive enough to extend automation to a new class of memory specifications that seek to solve open problems in concurrent language design. PrideMM enables a modify-execute-evaluate pattern of memory-specification development, where changes are quickly implemented and automatically tested.

[3] The paper uses the term 'memory specification' instead of 'memory (consistency) model', and reserves the word 'model' for its meaning from logic.

Memory specifications define what values may be read in a concurrent system. Current evaluators rely on ad hoc algorithms [3,6,14] or satisfiability (SAT) solvers [40]. However, flaws in existing language memory specifications [5] — where one must account for executions introduced through aggressive optimisation — have led to a new class of memory specifications [22,20] that cannot be practically solved using existing ad hoc or SAT techniques.

Many memory specifications are definable in ∃SO in a natural way and one can simulate them using SAT solvers. We demonstrate this facility of PrideMM for a realistic C++ memory specification [24], reproducing previous results [40,39]. But, some memory specifications are naturally formulated in higher-order logic. For example, the Jeffrey-Riely specification (J+R) comes with a formalisation, in the proof assistant Agda [11], that clearly uses higher-order features [20]. We observed that the problem of checking whether a program execution is allowed by J+R can be reduced to the model checking problem for SO. From a program execution, one obtains an SO structure $\mathfrak{A}$ on an universe of size n, and then one asks whether $\mathfrak{A} \models \mathsf{JR}_n$, where

$$\mathsf{JR}_n \coloneqq \exists X \left(\mathsf{TC}_n(\mathsf{AeJ}_n)(\emptyset, X) \wedge \mathsf{F}(X)\right)$$

$$\mathsf{AeJ}_n(P,Q) \coloneqq \begin{cases} \mathsf{sub}^1(P,Q) \wedge \mathsf{V}(P) \wedge \mathsf{V}(Q) \wedge \\ \forall X \left(\mathsf{TC}_n(\mathsf{AJ})(P,X) \rightarrow \exists Y \left(\mathsf{TC}_n(\mathsf{AJ})(X,Y) \wedge \mathsf{J}(Y,Q)\right)\right) \end{cases}$$

We will define precisely these formulae later (§ 5.4). For now, observe that the formula JR_n is in ∃∀∃SO. In practice, this means that it is not possible to use SAT solvers, as that would involve an exponential explosion. That motivates our development of an SO model checker. It is known that SO captures the polynomial hierarchy [27, Corollary 9.9], and the canonical problem for the polynomial hierarchy is quantified satisfiability. Hence, we built our SO model checker on top of a quantified satisfiability solver (QBF solver), QFUN [17].

The contributions of our work are as follows:

1. we present a model checker for SO, built on top of QBF solvers;
2. we reproduce known simulation results for traditional memory specifications;
3. we simulate a memory specification (J+R) that is a representative of a class of memory specifications that are out of the reach of traditional simulation techniques.

2 Overview

Figure 1 shows the architecture of our memory-specification simulator. The input is a litmus test written in the LISA language, and the output is a boolean result. LISA is a programming language that was designed for studying memory specifications [1]. We use LISA for its compatibility with the state-of-the-art memory-specification checker Herd7 [3]. We transform the input program into an event structure [41]. The memory-specification generator (MSG) produces an SO formula. We have a few interchangeable MSGs (§ 5). For some memory

specifications (§ 5.1, § 5.2, § 5.3), which Herd7 can handle as well, the formula is in fact fixed and does not depend at all on the event structure. For other memory specifications (such as § 5.4), the MSG might need to look at certain characteristics of the structure (such as its size). Finally, both the second-order structure and the second-order formula are fed into a solver, giving a verdict for the litmus test.

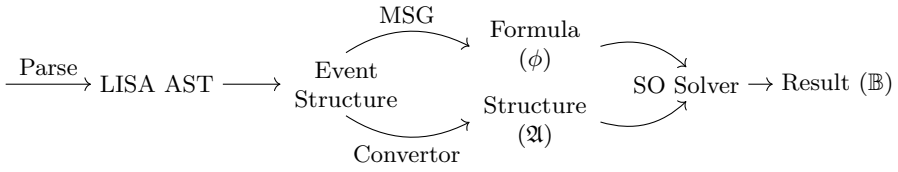


Fig. 1. From a LISA test case to a Y/N answer, given by the SO solver.

We are able to do so because of a key insight: relational second-order logic represents a sweet-spot in the design space. On the one hand, it is expressive enough such that encoding memory specifications is natural. On the other hand, it is simple enough such that it can be solved efficiently, using emerging QBF technology.

2.1 Memory Specifications

A *memory specification* describes the executions allowed by a shared-memory concurrent system; for example, under *sequential consistency* (SC) [25] memory accesses from all threads are interleaved and reads take their value from the most recent write of the same variable. Processor speculation, memory-subsystem reordering and compiler optimisations lead mainstream languages and processors to violate SC, and we say such systems exhibit *relaxed concurrency*. Relaxed concurrency is commonly described in an *axiomatic* specification (e.g. SC, ARM, Power, x86, C++ specifications [3]), where each program execution is represented as a graph with memory accesses as vertices, and edges representing program structure and dynamic memory behaviour. A set of axioms permit some execution graphs and forbid others.

Figure 2 presents a *litmus test* — a succinct pseudocode program designed to probe for a particular relaxed behaviour — together with an execution graph and an axiom. We shall discuss each in turn.

The test, called *LB+ctrl*, starts with `x` and `y` initialised to 0, then two threads concurrently read and conditionally write 1 back to their respective variables. The outcome $r_1 = 1 \wedge r_2 = 1$ (1/1) is unintuitive, and it cannot result from SC: there is no interleaving that agrees with the program order and places the writes of 1 before the reads for both `x` and `y`.

In an axiomatic specification, the outcome specified by the test corresponds to the execution graph shown in Figure 2. Initialisation is elided, but the read

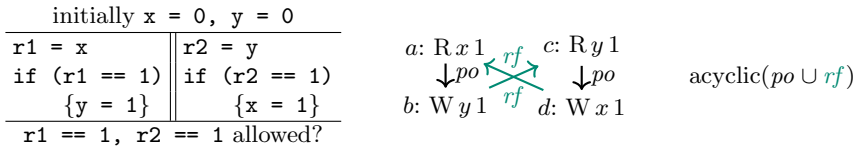


Fig. 2. LB+ctrl, an axiomatic execution of it, and an axiom that forbids it.

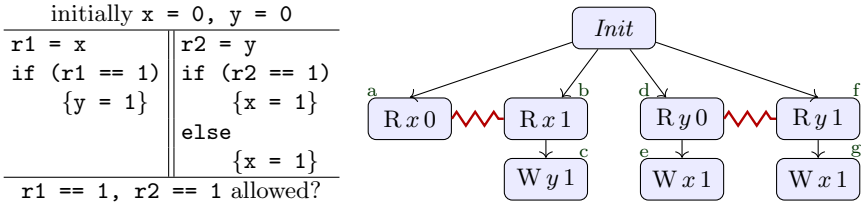


Fig. 3. LB+false-dep and the corresponding event structure.

and write of each thread is shown with *po* edges reflecting program order and *rf* edges linking writes to reads that *read from* them. The axiom of Figure 2 forbids the outcome 1/1 as the corresponding execution contains a cycle in $po \cup rf$. The SC, x86, Power and ARM specifications each include a variant of this axiom, all forbidding 1/1, whereas the C++ standard omits it [6] and allows 1/1.

MemSAT [39] and Herd7 [3] automatically solve litmus tests for axiomatic specifications using a SAT solver and ad hoc solving respectively, but not all memory specifications fit the axiomatic paradigm.

Axiomatic specifications do not fit optimised languages. Languages like C++ and Java perform dependency-removing optimisations that complicate their memory specifications. For example, the second thread of the LB+false-dep test in Figure 3 can be optimised using common subexpression elimination to `r2=y; x=1;`. On ARM and Power, this optimised code may be reordered, permitting the relaxed outcome 1/1, whereas the syntactic control dependency of the original would make 1/1 forbidden. It is common practice to use syntactic dependencies to enforce ordering on hardware, but at the language level the optimiser removes these *false* dependencies.

The memory specification of the C++ standard [15] is flawed because its axiomatic specification cannot draw a distinction between the executions leading to outcome 1/1 in LB+ctrl and LB+false-dep: to see that the dependency is false, one must consider more than one execution path, but axiomatic specifications judge single executions only [5].

Event structures capture the necessary information. A new class of specifications aims to fix this by ordering only real dependencies [22,20,31,12]. With a notable exception [22], these specifications are based on *event structures*, where all paths of control flow are represented in a single graph. Figure 3 presents the event structure for LB+false-dep. Program order is represented by arrows (→). Conflict (∿)

links events where only one can occur in an execution (the same holds for their program-order successors). For example, on the left-hand thread, the load of `x` can result in a read of value 0 (event a) or a read of value 1 (event b), but not both. Conversely, two subgraphs unrelated by program-order or conflict, e.g. $\{a, b, c\}$ and $\{d, e, f, g\}$, represent two threads in parallel execution.

It should be clear from the event structure in Figure 3 that regardless of the value read from `y` in the right-hand thread, there is a write to `x` of value 1; that is, the apparent dependency from the load of `y` is false and could be optimised away. Memory specifications built above event structures can recognise this pattern and permit relaxed execution.

The Jeffrey and Riely specification. J+R is built above event structures and correctly identifies false dependencies [20]. Conceptually, the specification is related to the Java memory specification [29]: in both, one constructs an execution stepwise, adding only memory events that can be *justified* from the previous steps. The sequence captures a causal order that prevents cycles with real dependencies. While Java is too strong, J+R allows writes that have false dependencies on a read to be justified before that read. To do this, the specification recognises confluence in the program structure: regardless of the execution path, the write will always be made. This search across execution paths involves an alternation of quantification that current ad hoc and SAT-based tools cannot efficiently simulate. However, the problem is amenable to QBF solvers.

2.2 Developing SC in SO Logic

The SC memory specification can be expressed as an axiomatic model [3] using *coherence order*, a per-variable total order of write events. An execution is allowed if there exists a reads-from relation rf and a coherence order co such that the transitive closure of $rf \cup co \cup (rf^{-1}; co) \cup po$ is acyclic. Here, po is the (fixed) program-order relation, and it is understood that co and rf satisfy certain further axioms. In our setting, we describe the sequentially consistent specification as follows. We represent rf and co by existentially-quantified SO arity-2 variables Y_{rf} and Y_{co}, respectively. For example, to say $(x, y) \in co$, we use the formula $Y_{co}(x, y)$. The program order po is represented by an interpreted arity-2 symbol $<$. Then, the SO formula that represents $rf \cup co \cup (rf^{-1}; co) \cup po$ is

$$\mathsf{R}(y, z) \;\coloneqq\; Y_{rf}(y, z) \vee Y_{co}(y, z) \vee \exists x \left(Y_{rf}(x, z) \wedge Y_{co}(x, y) \right) \vee (y < z)$$

The definition from above should be interpreted as a macro expansion rule: the left-hand side $\mathsf{R}(y, z)$ is a combinator that expands to the formula on right-hand side. To require that the transitive closure of R is acyclic we require that there exists a relation that includes R, is transitive, and irreflexive:

$$\exists Z \left(\mathsf{sub}^2(\mathsf{R}, Z) \wedge \mathsf{trans}(Z) \wedge \mathsf{irrefl}(Z) \right)$$

The combinators sub^2, trans, irrefl are defined as one would expect. For example, $\mathsf{sub}^2(P, Q)$, which says that the arity-2 relation P is included in the arity-2 relation Q, is $\forall xy\,\big(P(x, y) \to Q(x, y)\big)$. In short, the translation from the usual formulation of memory specifications into the SO logic encoding that we propose is natural and almost automatic.

To represent programs and their behaviours uniformly for all memory specifications in § 5, we use event structures. These have the ability to represent an overlay of potential executions. Some memory specifications require reasoning about several executions at the same time: this is a salient feature of the J+R memory specification.

Once we have the program and its behaviour represented as a logic structure $\mathfrak{A}$ and the memory specification represented as a logic formula ϕ, we ask whether the structure satisfies the formula, written $\mathfrak{A} \models \phi$. In other words, we have to solve a model-checking problem for second-order logic, which reduces to QBF solving because the structure $\mathfrak{A}$ is finite.

3 Preliminaries

To introduce the necessary notation, we recall some standard definitions [27]. A (finite, relational) *vocabulary* σ is a finite collection of *constant symbols* $(\mathtt{1}, \ldots, \mathtt{n})$ together with a finite collection of *relation symbols* $(\mathtt{q}, \mathtt{r}, \ldots)$. A (finite, relational) *structure* $\mathfrak{A}$ *over vocabulary* σ is a tuple $\langle A, Q, R, \ldots \rangle$ where $A = \{1, \ldots, n\}$ is a finite set called *universe* with several distinguished relations $Q, R, \ldots$ We assume a countable set of *first-order variables* (x, y, ...), and a countable set of *second-order variables* (X, Y, ...). A *variable* α is a first-order variable or a second-order variable; a *term* t is a first-order variable or a constant symbol; a *predicate* P is a second-order variable or a relation symbol. A (second-order) *formula* ϕ is defined inductively: (a) if P is a predicate and $t_1, \ldots, t_k$ are terms, then $P(t_1, \ldots, t_k)$ is a formula[4]; (b) if ϕ_1 and ϕ_2 are formulae, then $\phi_1 \circ \phi_2$ is a formula, where $\circ$ is a boolean connective; and (c) if α is a variable and ϕ is a formula, then $\exists\alpha\,\phi$ and $\forall\alpha\,\phi$ are formulae. We assume the standard satisfaction relation $\models$ between structures and formulae.

The logic defined so far is known as relational SO. If we require that all quantifiers over second-order variables are existentials, we obtain a fragment known as ∃SO. For example, the SC specification of § 2.2 is in ∃SO.

The Model Checking Problem. Given a structure $\mathfrak{A}$ and a formula ϕ, determine if $\mathfrak{A} \models \phi$. We assume that the relations of $\mathfrak{A}$ are given by explicitly listing their elements. The formula ϕ uses the syntax defined above.

Combinators. We will build formulae using the combinators defined below. This simplifies the presentation, and directly corresponds to an API for building

[4] we make the usual assumptions about arity

formulae within PrideMM.

$$\begin{aligned}
\mathsf{sub}^k(P^k, Q^k) &\coloneqq \forall \boldsymbol{x}\, \big(P^k(\boldsymbol{x}) \to Q^k(\boldsymbol{x})\big) & \mathsf{id}(x,y) &\coloneqq (x = y)\\
\mathsf{eq}^k(P^k, Q^k) &\coloneqq \forall \boldsymbol{x}\, \big(P^k(\boldsymbol{x}) \leftrightarrow Q^k(\boldsymbol{x})\big) & \mathsf{inv}(P^2)(x,y) &\coloneqq P^2(y,x)\\
\mathsf{seq}(P^2, Q^2)(x,z) &\coloneqq \exists y\, \big(P^2(x,y) \wedge Q^2(y,z)\big) & \mathsf{irrefl}(P^2) &\coloneqq \forall x\, \neg P^2(x,x)\\
\mathsf{inj}(P^2) &\coloneqq \mathsf{sub}^2\big(\mathsf{seq}(P^2, \mathsf{inv}(P^2)), \mathsf{id}\big) & \mathsf{or}(\mathsf{R},\mathsf{S})(x,y) &\coloneqq \mathsf{R}(x,y) \vee \mathsf{S}(x,y)\\
\mathsf{trans}(P^2) &\coloneqq \mathsf{sub}^2\big(\mathsf{seq}(P^2, P^2), P^2\big) & \mathsf{maybe}(\mathsf{R})(x,y) &\coloneqq \mathsf{or}(\mathsf{id}, \mathsf{R})(x,y)
\end{aligned}$$

$$\begin{aligned}
\mathsf{acyclic}(P^2) &\coloneqq \exists X^2\, \big(\mathsf{sub}^2(P^2, X^2) \wedge \mathsf{trans}(X^2) \wedge \mathsf{irrefl}(X^2)\big)\\
\mathsf{TC}_0(\mathsf{R}) &\coloneqq \mathsf{eq}^1\\
\mathsf{TC}_{n+1}(\mathsf{R})(P^1, Q^1) &\coloneqq \mathsf{eq}^1(P^1, Q^1) \vee \exists X^1\, \big(\mathsf{R}(P^1, X^1) \wedge \mathsf{TC}_n(\mathsf{R})(X^1, Q^1)\big)
\end{aligned}$$

By convention, all quantifiers that occur on the right-hand side of the definitions above are over fresh variables. Above, P^k and Q^k are arity-k predicates, x and y are first-order variables, and R and S are combinators.

Let us discuss two of the more interesting combinators: $\mathsf{acyclic}$ and TC. A relation P is acyclic if it is included in a relation that is transitive and irreflexive. We remark that the definition of $\mathsf{acyclic}$ is carefully chosen: even slight variations can have a strong influence on the runtime of solvers [18]. The combinator TC for bounded transitive closure is interesting for another reason: it is higher-order — applying an argument (R) relation in each step of its expansion. By way of example, let us illustrate its application to the subset combinator sub^1.

$$\begin{aligned}
&\mathsf{TC}_1(\mathsf{sub}^1)(P, Q)\\
&\quad= \mathsf{eq}^1(P, Q) \vee \exists X\, \big(\mathsf{sub}^1(P, X) \wedge \mathsf{TC}_0(\mathsf{sub}^1)(X, Q)\big)\\
&\quad= \begin{cases} \forall x_1\, \big(P(x_1) \leftrightarrow Q(x_1)\big) \vee \\ \quad \exists X\, \big(\forall x_2\, \big(P(x_2) \to X(x_2)\big) \wedge \mathsf{eq}^1(X, Q)\big) \end{cases}\\
&\quad= \begin{cases} \forall x_1\, \big(P(x_1) \leftrightarrow Q(x_1)\big) \vee \\ \quad \exists X\, \big(\forall x_2\, \big(P(x_2) \to X(x_2)\big) \wedge \forall x_3\, \big(X(x_3) \leftrightarrow Q(x_3)\big)\big) \end{cases}
\end{aligned}$$

In the calculation above, P, Q and X have arity 1.

4 SO Solving through QBF

From a reasoning perspective, SO model-checking is a highly non-trivial task due to quantifiers. In particular, quantifiers over relations, where the size of the search-space alone is daunting. For a universe of size n there are 2^{n^2} possible binary relations, and there are 2^{n^k} possible k-ary relations.[5]

A relation is uniquely characterised by a vector of Boolean values, each determining whether a certain tuple is in the relation or not. This insight lets us formulate a search for a relation as a SAT problem, where a fresh Boolean variable is introduced for any potential tuple in the relation. Even though the translation is exponential, it is a popular method in finite-model finding for first-order logic formulae [13,38,33].

However, in the setting of SO, a SAT solver is insufficient since the input formula may contain alternating quantifiers. We tackle this issue by translating

[5] Finding constrained finite relations is NEXP-TIME complete [26].

to quantified Boolean formulae (QBF), rather than to plain SAT. The translation is carried out in three stages.

1. each interpreted relation is in-lined as a disjunction of conjunctions over the tuples where the relation holds;
2. first-order quantifiers are expanded into Boolean connectives over the elements of the universe, i.e. $\forall x\phi$ leads to one conjunct for each element of the universe and $\exists x\phi$ leads to one disjunct for each element of the universe;
3. all atoms now are ground and each atom is replaced by a fresh Boolean variable, which is inserted under the same type of quantifier as the atom.

For illustration, consider the formula $\exists X \forall Y \forall z \left(Y(z) \rightarrow X(z)\right)$ and the universe $A = \{1, 2\}$. The formula requires a set X that is a superset of all sets. Inevitably, X has to be the whole domain. The QBF formulation is $\exists \mathsf{x}_1\mathsf{x}_2 \forall \mathsf{y}_1\mathsf{y}_2 \left((\mathsf{y}_1 \rightarrow \mathsf{x}_1) \wedge (\mathsf{y}_2 \rightarrow \mathsf{x}_2)\right)$. Intuitively, rather than talking about a set, we focus on each element separately, which is enabled by the finiteness of the universe. Using QBF enables us to arbitrarily quantify over the sets' elements.

PrideMM enables exporting the QBF formulation into the QCIR format [21], which is supported by a bevy of QBF solvers. However, since most solvers only support prenex form, PrideMM, also additionally prenexes the formula, where it attempts to heuristically minimise the number of quantifier levels.

The experimental evaluation showed that the QFUN solver [17] performs the best on the considered instances, see § 6. While the solver performs very well on the J+R litmus tests, a couple of instances were left unsolved. Encouraged by the success of QFUN, we built a dedicated solver that integrates the translation to QBF and the solving itself. The solver represents the formula in dedicated hash-consed data structures (the formulae grow in size considerably). The expansion of first-order variables is done directly on these data structures while also simplifying the formula on the go. The solver also directly supports non-prenex formulation (see [19] for non-prenex QBF solving). The solver applies several preprocessing techniques before expanding the first-order variables, such as elimination of relations that appear only in positive or only in negative positions in the formula.

5 Memory Specification Encodings

In this section, we show that many memory specifications can be expressed conveniently in second-order logic. We represent programs and their behaviours with event structures: this supports the expression of axiomatic specifications such as C++, but also the higher-order specification of J+R. For a given program, its event structure is constructed in a straightforward way: loads give rise to mutually conflicting read events and writes to write events [20]. We express the constraints over event structures with the following vocabulary, shared across all specifications.

Vocabulary. A memory specification decides if a program is allowed to have a certain behaviour. We pose this as a model checking problem, $\mathfrak{A} \models \phi$, where $\mathfrak{A}$

captures program behaviour and ϕ the memory specification. The vocabulary of $\mathfrak{A}$ consists of the following symbols:

- arity 1:`read`, `write`, `final`
- arity 2: $\leq$, `conflict`, `justifies`, `sloc`, $=$

Sets `read` and `write` classify read and write events. The symbol `final`, another set of events, identifies the executions that exhibit final register states matching the outcome specified by the litmus test.

Events x and y are in program order, written $x \leq y$, if event x arises from an earlier statement than y in the program text. We have $\texttt{conflict}(x, y)$ between events that cannot belong to the same execution; for example, a load statement gives rise to an event for each value it might read, but an execution chooses one particular value, and contains only the corresponding event. We write $\texttt{justifies}(x, y)$ when x is a read and y is a write to the same memory location of the same value. We have $\texttt{sloc}(x, y)$ when x and y access the same memory location. Identity on events, $\{\,(x, x) \mid x \in A\,\}$, is denoted by $=$.

Configurations and Executions. We distinguish two types of sets of events. A *configuration* is a set of events that contains no conflict and is downward closed with respect to $\leq$; that is, X is a configuration when $\mathsf{V}(X)$ holds, where the V combinator is defined by

$$\mathsf{V}(X) \;\coloneqq\; \begin{cases} \forall x \forall y \left(\big(X(x) \wedge X(y)\big) \rightarrow \neg \texttt{conflict}(x, y) \right) \\ \quad \wedge \forall y \left(X(y) \rightarrow \forall x \left((x \leq y) \rightarrow X(x) \right) \right) \end{cases}$$

We say that a configuration X is an *execution of interest* when every final event is either in X or in conflict with an event in X; that is, X is an execution of interest when $\mathsf{F}(X)$ holds, where the F combinator is defined by

$$\mathsf{F}(X) \;\coloneqq\; \mathsf{V}(X) \wedge \forall x \left(\begin{array}{l} \big(\texttt{final}(x) \wedge \neg X(x)\big) \rightarrow \\ \quad \exists y \left(\texttt{conflict}(x, y) \wedge \texttt{final}(y) \wedge X(y) \right) \end{array} \right)$$

Intuitively, we shall put in `final` all the maximal events (according to $\leq$) for which registers have the desired values.

Notations. In the formulae below, X will stand for a configuration, which may be the execution of interest. Variables Y_{rf}, Y_{co}, Y_{hb} and so on are used to represent the relations that are typically denoted by rf, co, hb, ... Thus, X has arity 1, while Y_{rf}, Y_{co}, Y_{hb}, ... have arity 2.

In what follows, we present four memory specifications: sequential consistency (§ 5.1), release–acquire (§ 5.2), C++ (§ 5.3), and J+R (§ 5.4). The first three can be expressed in $\exists$SO (and in first-order logic). The last one uses both universal and existential quantification over sets. For each memory specification, we shall see their encoding in second-order logic.

5.1 Sequential Consistency

The SC specification allows all interleavings of threads, and nothing else. It is described by the following SO sentence:

$$\mathsf{SC} \coloneqq \exists X\, Y_{co}\, Y_{rf}\, \big(\mathsf{F}(X) \wedge \mathsf{co}(X, Y_{co}) \wedge \mathsf{rf}(X, Y_{rf}) \wedge \mathsf{acyclic}(\mathsf{R}(Y_{co}, Y_{rf}))\big)$$

Intuitively, we say that there exists a coherence order relation Y_{co} and a reads-from relation Y_{rf} which, when combined in a certain way, result in an acyclic relation $\mathsf{R}(Y_{co}, Y_{rf})$. The formula $\mathsf{co}(X, Y_{co})$ says that Y_{co} satisfies the usual axioms of a coherence order with respect to the execution X; and the formula $\mathsf{rf}(X, Y_{rf})$ says that Y_{rf} satisfies the usual axioms of a reads-from relation with respect to the execution X. Moreover, the formula $\mathsf{F}(X)$ asks that X is an execution of interest, which results in registers having certain values.

$$\mathsf{co}(X, Y_{co}) \coloneqq \begin{cases} \mathsf{trans}(Y_{co}) \wedge \\ \forall xy \left(\begin{array}{l} \big(X(x) \wedge X(y) \wedge \mathtt{write}(x) \wedge \mathtt{write}(y) \wedge \mathtt{sloc}(x,y) \wedge (x \neq y)\big) \\ \quad \leftrightarrow \big(Y_{co}(x,y) \vee Y_{co}(y,x)\big) \end{array} \right) \end{cases}$$

$$\mathsf{rf}(X, Y_{rf}) \coloneqq \begin{cases} \mathsf{inj}(Y_{rf}) \wedge \mathsf{sub}^2(Y_{rf}, \mathtt{justifies}) \wedge \\ \quad \forall y \Big(\big(\mathtt{read}(y) \wedge X(y)\big) \rightarrow \exists x \, \big(\mathtt{write}(x) \wedge X(x) \wedge Y_{rf}(x,y)\big) \Big) \end{cases}$$

When X is a potential execution and Y_{co} is a potential coherence-order relation, the formula $\mathsf{co}(X, Y_{co})$ requires that the writes in X for the same location include some total order. Because of the later condition that $\mathsf{R}(Y_{co}, Y_{rf})$ is acyclic, Y_{co} is in fact required to be a total order per location. When X is a potential execution and Y_{rf} is a potential reads-from relation, the formula $\mathsf{rf}(X, Y_{rf})$ requires that Y_{rf} is injective, is a subset of $\mathtt{justifies}$, and relates all the reads in X to some write in X.

The auxiliary relation $\mathsf{R}(Y_{co}, Y_{rf})$ is the union of strict program-order $(<)$, reads-from (Y_{rf}), coherence-order (Y_{co}), and the from-reads relation:

$$\mathsf{R}(Y_{co}, Y_{rf})(y,z) \;\coloneqq\; (y < z) \vee Y_{co}(y,z) \vee Y_{rf}(y,z) \vee \exists x \, \big(Y_{co}(x,z) \wedge Y_{rf}(x,y)\big)$$

5.2 Release–Acquire

Release–Acquire is a simple relaxed memory specification, which is represented straightforwardly in SO logic. It is captured by the formula RA using the vocabulary established in the definition of SC:

$$\mathsf{RA} \;\coloneqq\; \exists X\, Y_{co}\, Y_{rf} \left(\begin{array}{l} \mathsf{F}(X) \wedge \mathsf{co}(X, Y_{co}) \wedge \mathsf{rf}(X, Y_{rf}) \wedge \mathsf{acyclic}(Y_{co}) \\ \quad \wedge \exists\, Y_{hb} \left(\begin{array}{l} \mathsf{sub}^2(<, Y_{hb}) \wedge \mathsf{sub}^2(Y_{rf}, Y_{hb}) \wedge \mathsf{trans}(Y_{hb}) \\ \quad \wedge \mathsf{irrefl}(Y_{hb}) \wedge \mathsf{irrefl}(\mathsf{seq}(Y_{co}, Y_{hb})) \\ \quad \wedge \mathsf{irrefl}(\mathsf{seq}(\mathsf{inv}(Y_{rf}), \mathsf{seq}(Y_{co}, Y_{hb}))) \end{array} \right) \end{array} \right)$$

The existential SO variable Y_{hb} over-approximates a relation traditionally called happens-before.

5.3 C++

To capture the C++ specification in SO logic, we follow the Herd7 specification of Lahav et al. [24]. Their work introduces necessary patches to the specification of the standard [6] but also includes fixes and adjustments from prior work [4,23]. The specification is more nuanced than the SC and RA specifications and requires additions to the vocabulary of $\mathfrak{A}$ together with a reformulation for efficiency, but the key difference is more fundamental. C++ is a *catch-fire* semantics: programs that exhibit even a single execution with a data race are allowed to do anything—satisfying every expected outcome. This difference is neatly expressed in SO logic:

$$\mathsf{CPP} \;:=\; \exists X\, Y_{co}\, Y_{rf}\, Y_{\alpha\beta}\; \left(\begin{array}{l} \mathsf{co}(X, Y_{co}) \wedge \mathsf{rf}(X, Y_{rf}) \wedge \mathsf{hb}(Y_{\alpha\beta}, Y_{rf}) \\ \quad \wedge \mathsf{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf}) \wedge (\mathsf{F}(X) \vee \mathsf{C}(Y_{\alpha\beta}, Y_{rf})) \end{array}\right)$$

The formula reuses $\mathsf{co}(X, Y_{co})$, $\mathsf{rf}(X, Y_{rf})$ and $\mathsf{F}(X)$ and includes three new combinators: $\mathsf{hb}(Y_{\alpha\beta}, Y_{rf})$, $\mathsf{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf})$ and $\mathsf{C}(Y_{\alpha\beta}, Y_{rf})$. $\mathsf{hb}(Y_{\alpha\beta}, Y_{rf})$ constrains a new over-approximation, $Y_{\alpha\beta}$, used for building a transitive relation. $\mathsf{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf})$ captures the conditions imposed on a valid C++ execution, and is the analogue of the conditions applied in SC and RA. $\mathsf{C}(Y_{\alpha\beta}, Y_{rf})$ holds if there is a race in the execution X. Note that the expected outcome is allowed if $\mathsf{F}(X)$ is satisfied or if there is a race and $\mathsf{C}(Y_{\alpha\beta}, Y_{rf})$ is true, matching the catch-fire semantics.

New vocabulary. C++ *Read-modify-write* operations load and store from memory in a single atomic step: a new `rmw` relation links the corresponding reads and writes. C++ *fence* operations introduce new events and the set `fences` identifies them. The programmer annotates each memory access and fence with a *memory order* parameter that sets the force of inter-thread synchronisation created by the access. For each choice, we add a new set: `na`, `rlx`, `acq`, `rel`, `acq-rel`, and `sc`.

Over-approximation in happens before. The validity condition, $\mathsf{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf})$, and races $\mathsf{C}(Y_{\alpha\beta}, Y_{rf})$, hinge on a relation called *happens-before*. We over-approximate transitive closures in the SO logic for efficiency, but Lahav et al. [24] define happens-before with nested closures that do not perform well. Instead we over-approximate a reformulation of happens-before that flattens the nested closures into a single one (see Appendix A).

We define a combinator for happens-before, $\mathsf{HB}(Y_{\alpha\beta}, Y_{rf})$, that is used in $\mathsf{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf})$ and $\mathsf{C}(Y_{\alpha\beta}, Y_{rf})$. It takes as argument an over-approximation of the closure internal to the reformed definition of happens-before, $Y_{\alpha\beta}$. $\mathsf{hb}(Y_{\alpha\beta}, Y_{rf})$ constrains $Y_{\alpha\beta}$, requiring it to be transitive and to include the conjuncts of the

closure, α and β below.

$$\begin{aligned}
\mathsf{HB}(Y_{\alpha\beta}, Y_{rf}) &\coloneqq \mathsf{or}(<, \mathsf{seq}(\mathsf{maybe}(<), \mathsf{sw}_{\mathsf{begin}}(Y_{rf}), Y_{\alpha\beta}, \mathsf{sw}_{\mathsf{end}}(Y_{rf}), \mathsf{maybe}(<))) \\
\alpha(Y_{rf}) &\coloneqq \mathsf{seq}(\mathsf{sw}_{\mathsf{end}}(Y_{rf}), \mathsf{maybe}(<), \mathsf{sw}_{\mathsf{begin}}(Y_{rf})) \\
\beta(Y_{rf}) &\coloneqq \mathsf{seq}(Y_{rf}, \texttt{rmw}) \\
\mathsf{hb}(Y_{\alpha\beta}, Y_{rf}) &\coloneqq \begin{cases} \mathsf{trans}(Y_{\alpha\beta}) \\ \quad \wedge \mathsf{sub}^2(\mathsf{id}, Y_{\alpha\beta}) \wedge \mathsf{sub}^2(\alpha(Y_{rf}), Y_{\alpha\beta}) \wedge \mathsf{sub}^2(\beta(Y_{rf}), Y_{\alpha\beta}) \end{cases}
\end{aligned}$$

5.4 Jeffrey–Riely

The J+R memory specification is captured by a sentence JR_n, parametrised by an integer n. Unlike the formulae we saw before, JR_n makes use of three levels of quantifiers ($\exists\forall\exists$), putting it on the third level of the polynomial hierarchy. We begin by lifting[6] `justifies` from events to sets of events P and Q:

$$\begin{aligned}
\mathsf{J}(P,Q) &\coloneqq \forall y \left(\begin{array}{l} \left(\neg P(y) \wedge Q(y) \wedge \texttt{read}(y)\right) \\ \quad \rightarrow \exists x \left(P(x) \wedge \texttt{write}(y) \wedge \texttt{justifies}(x,y)\right) \end{array} \right) \\
\mathsf{AJ}(P,Q) &\coloneqq \mathsf{J}(P,Q) \wedge \mathsf{sub}^1(P,Q) \wedge \mathsf{V}(P) \wedge \mathsf{V}(Q)
\end{aligned}$$

We read J as 'justifies', and AJ as 'always justifies'. Next, we define what Jeffrey and Riely call 'always eventually justify'

$$\mathsf{AeJ}_n(P,Q) \coloneqq \begin{cases} \mathsf{sub}^1(P,Q) \wedge \mathsf{V}(P) \wedge \mathsf{V}(Q) \wedge \\ \forall X \left(\mathsf{TC}_n(\mathsf{AJ})(P,X) \rightarrow \exists Y \left(\mathsf{TC}_n(\mathsf{AJ})(X,Y) \wedge \mathsf{J}(Y,Q) \right) \right) \end{cases}$$

The size of the formula $\mathsf{TC}_n(\mathsf{AeJ}_m)(P,Q)$ we defined above is $\Theta(mn)$. In particular, it is bounded. Finally, we let[7]

$$\mathsf{JR}_n \coloneqq \exists X \left(\mathsf{TC}_n(\mathsf{AeJ}_n)(\emptyset, X) \wedge \mathsf{F}(X) \right)$$

and ask solve the model checking problem $\mathfrak{A} \models \mathsf{JR}_n$. Since the formulae above are in MSO, it is sufficient to pick $n \coloneqq 2^{|A|}$. Since all bounded transitive closures include the subset relation, they are monotonic, and it suffices, in fact, to pick $n \coloneqq |A|$. For actual solving, we will use this observation.

6 Evaluation

We evaluate our tool in the context of Herd7 [3], which is a standard tool among memory specification researchers for building axiomatic memory specifications. No similar tool exists for higher-order event structure based memory specifications.

[6] Our definition of J is different from the original one [20]: we require that only new reads are justified, by including the conjunct $\neg P(y)$. Without this modification, our solver's results disagree with the hand-calculations reported by Jeffrey and Riely; with this modification, the results agree.

[7] The symbol $\emptyset$ denotes the empty unary relation, as expected.

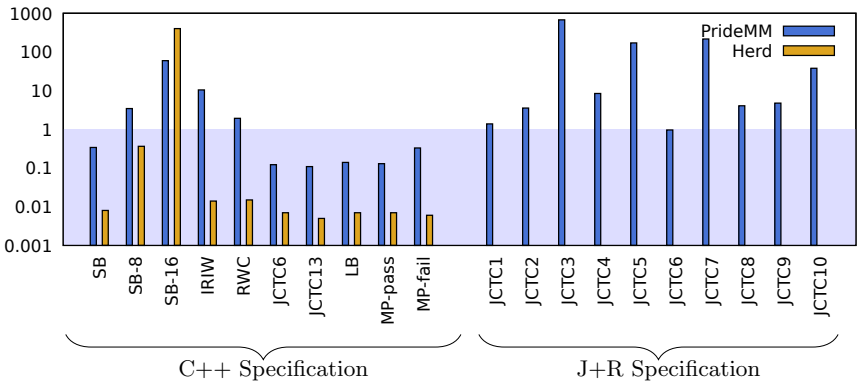


Fig. 4. Comparison of PrideMM's performance in contrast to Herd7 [3].

6.1 Comparison to existing techniques

In figure Fig. 4 we compare the performance and capabilities of PrideMM to Herd7, the de facto standard tool for building axiomatic memory specifications. Herd7 and PrideMM were both executed on a machine equipped with an Intel i5-5250u CPU and 16 GB of memory. We choose not to compare our tool to MemSAT [39], as there are more memory specifications implemented for Herd7 in the CAT language [2] than there are for MemSAT.

Performance. Notably Herd7's performance is very favourable in contrast to the performance of PrideMM, however there are some caveats. The performance of PrideMM is largely adequate, with most of the standard litmus tests taking less than 2 seconds to execute. $y \leq 1s$ is highlighted on the chart. We find that our QBF technique scales better than Herd7 with large programs. This is demonstrated in the SB-16 test, a variant of the "store buffering" litmus test with 16 threads. The large number of combinations for quantifying the existentially quantified relations which are explored naïvely by Herd7 cause it to take a long time to complete. In contrast, smarter SAT techniques handle these larger problems handily.

Expressiveness. We split the chart in figure Fig. 4 into 2 sections, the left-hand side of the chart displays a representative subset of common litmus tests showing PrideMM's strength and weaknesses. These litmus tests are evaluated under the C++ memory specification. Note that these include tests with behaviour expected to be observable and unobservable, hence there being two MP bars. The C++ memory specification is within the domain of memory specifications that Herd7 can solve, as it requires only existentially quantified relations.

The right-hand half of the chart is the first 10 Java causality test cases run under the J+R specification, which are no longer expressible in Herd7. PrideMM solves these in reasonable time, with most tests solved in less than 10 minutes.

Prob.	SAT	caqe (s)	qfun (s)	qfm (s)	Prob.	SAT	caqe (s)	qfun (s)	qfm (s)
1	N	⊥	610	**2**	10	Y	⊥	36	**10**
2	N	⊥	23	**2**	11	Y	⊥	598	**335**
3	Y	⊥	⊥	**222**	13	Y	**1**	**1**	**1**
4	Y	⊥	**2**	5	14	Y	⊥	**29**	33
5	Y	⊥	78	**51**	15	Y	⊥	512	**157**
6	N	5	4	**1**	16	N	⊥	⊥	**12**
7	Y	⊥	280	**56**	17	N	⊥	**39**	311
8	N	⊥	**2**	**2**	18	N	⊥	359	**190**
9	N	⊥	2	**1**	**#17**		#2	#15	#17

Fig. 5. Solver approaches for PrideMM on Java Causality Test Cases. ⊥ represents timeout or mem-out.

Our J+R tests replicate the results found in the original paper, but where they use laborious manual proof in the Agda proof assistant, PrideMM validates the results automatically.

6.2 QBF vs SO Solver Performance

PrideMM enables emitting the SO logic formulae and structures directly for the SO solver, or we can convert to a QBF query (see § 4). This allows us to use our SO solver as well as QBF solvers. We find that the SO solver affords us a performance advantage over the QBF solver in most of the Java causality test cases, where performance optimisations for alternating quantification are applicable.

We include the performance of the QBF solvers CAQE and QFUN, the respective winners of the CNF and non-CNF tracks at 2017's QBFEVAL competition [32]. Our QBF benchmarks were first produced in the circuit-like format QCIR [21], natively supported by QFUN. The inputs to CAQE were produced by converting to CNF through standard means, followed by a preprocessing step with Bloqqer [7].

We can also emit the structures and formulae as an Isabelle/HOL file, which can then be loaded into Nitpick [8] conveniently. We found that Nitpick cannot be run over the C++ specification or the J+R specification, timing out after 1 hr on all the litmus tests.

7 Related Work

We build on prior work from two different areas — relaxed memory specifications, and SAT/QBF solving: the LISA frontend comes from the Herd7 memory-specification simulator [3], the MSGs implement memory specifications that have been previously proposed [24,20], and the SO solver is based on a state-of-the-art QBF solver [17].

There is a large body of work on finite relational model finding in the context of memory specifications using Alloy [16]. Alloy has been used to compare memory specifications and find litmus tests which can distinguish two specifications [40], and has been used to synthesise comprehensive sets of tests for a specific memory

specification [28]. Applying SAT technology in the domain of evaluating memory specifications has been tried before, too. MemSAT [39] uses Kodkod [38], the same tool that Alloy relies on to do relational model finding. MemSynth [10] uses Ocelot [9] to embed relational logic into the Rosette [37] language. Our results are consistent with the findings of MemSAT and MemSynth: SAT appears to be a scalable and fast way to evaluate large memory specification questions. Despite this, SAT does not widen the class of specifications that can be efficiently simulated beyond ad hoc techniques.

There is work to produce a version of Alloy which can model higher-order constructions, called Alloy* [30], however this is limited in that each higher order set requires a new signature in the universe to represent it. Exponential expansion of the sets quantified in the J+R specification leaves model finding for J+R executions intractable in Alloy* too.

While Nitpick [8] can model higher order constructions, we found it could not generate counter examples in a reasonable length of time of the specifications we built. There is work to build a successor to Nitpick called Nunchaku [34], however, at present Nunchaku does not support higher order quantification. Once Nunchaku is more complete we intend to output to Nunchaku and evaluate its performance in comparison to our SO solver.

There is a bevy of work on finite model finding in various domains. SAT is a popular method for finite model finding in first-order logic formulae [13,33]. There are constraint satisfaction-based model finders, e.g. the SEM model finder [42], relying on dedicated symmetry and propagation. Reynolds et al. propose solutions for finite model finding in the context of SMT [35,36] (CVC4 is in fact used as backend to Nunchaku).

8 Conclusion

This paper presents PrideMM, a case study of the application of new solving techniques to a problem domain with active research. PrideMM allows memory specification researchers to build a new class of memory specifications with richer quantification, and still automatically evaluate these specifications over programs. In this sense we provide a Herd7-style modify-execute-evaluate pattern of development for higher-order memory specifications that were previously unsuitable for mechanised model finding.

References

1. Alglave, J., Cousot, P.: Syntax and analytic semantics of LISA. https://arxiv.org/abs/1608.06583 (2016)
2. Alglave, J., Cousot, P., Maranget, L.: Syntax and analytic semantics of the weak consistency model specification language CAT. https://arxiv.org/abs/1608.07531 (2016)
3. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014). https://doi.org/10.1145/2627752, http://doi.acm.org/10.1145/2627752
4. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and opencl. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 634–648 (2016). https://doi.org/10.1145/2837614.2837637, http://doi.acm.org/10.1145/2837614.2837637
5. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 283–307 (2015). https://doi.org/10.1007/978-3-662-46669-8_12, https://doi.org/10.1007/978-3-662-46669-8_12
6. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 55–66 (2011). https://doi.org/10.1145/1926385.1926394, http://doi.acm.org/10.1145/1926385.1926394
7. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: The 23rd International Conference on Automated Deduction CADE (2011)
8. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 131–146. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/https://doi.org/10.1007/978-3-642-14052-5_11
9. Bornholt, J., Torlak, E.: Ocelot: A solver-aided relational logic DSL (2017), https://ocelot.memsynth.org/
10. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 467–481 (2017). https://doi.org/10.1145/3062341.3062353, http://doi.acm.org/10.1145/3062341.3062353
11. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda - A functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 73–78. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_6, https://doi.org/10.1007/978-3-642-03359-9_6
12. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures. PACMPL **3**(POPL), 70:1–70:28 (2019), https://dl.acm.org/citation.cfm?id=3290383

13. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications (2003)
14. Gray, K.E., Kerneis, G., Mulligan, D.P., Pulte, C., Sarkar, S., Sewell, P.: An integrated concurrency and core-isa architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In: Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015. pp. 635–646 (2015). https://doi.org/10.1145/2830772.2830775, http://doi.acm.org/10.1145/2830772.2830775
15. ISO/IEC: Programming languages – C++. Draft N3092 (March 2010), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf
16. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (Apr 2002). https://doi.org/10.1145/505145.505149, http://doi.acm.org/10.1145/505145.505149
17. Janota, M.: Towards generalization in QBF solving via machine learning. In: AAAI Conference on Artificial Intelligence (2018)
18. Janota, M., Grigore, R., Manquinho, V.: On the quest for an acyclic graph. In: RCRA (2017)
19. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. Artificial Intelligence **234**, 1–25 (2016). https://doi.org/http://dx.doi.org/10.1016/j.artint.2016.01.004
20. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 759–767. LICS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2933575.2934536, http://doi.acm.org/10.1145/2933575.2934536
21. Jordan, C., Klieber, W., Seidl, M.: Non-CNF QBF solving with QCIR. In: AAAI Workshop: Beyond NP. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)
22. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 175–189 (2017), http://dl.acm.org/citation.cfm?id=3009850
23. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 649–662 (2016). https://doi.org/10.1145/2837614.2837643, http://doi.acm.org/10.1145/2837614.2837643
24. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 618–632 (2017). https://doi.org/10.1145/3062341.3062352, http://doi.acm.org/10.1145/3062341.3062352
25. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439, https://doi.org/10.1109/TC.1979.1675439
26. Lewis, H.R.: Complexity results for classes of quantificational formulas. Journal of Computer and System Sciences **21**(3), 317–353 (1980). https://doi.org/https://doi.org/10.1016/0022-0000(80)90027-6, http://www.sciencedirect.com/science/article/pii/0022000080900276

27. Libkin, L.: Elements of Finite Model Theory. Springer (2004)
28. Lustig, D., Wright, A., Papakonstantinou, A., Giroux, O.: Automated synthesis of comprehensive memory model litmus test suites. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 661–675. ASPLOS '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3037697.3037723, http://doi.acm.org/10.1145/3037697.3037723
29. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 378–391 (2005). https://doi.org/10.1145/1040305.1040336
30. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. In: ICSE (2015)
31. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 622–633 (2016). https://doi.org/10.1145/2837614.2837616
32. QBF Eval 2017, http://www.qbflib.org/event_page.php?year=2017
33. Reger, G., Suda, M., Voronkov, A.: Finding finite models in multi-sorted first-order logic. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT. Lecture Notes in Computer Science, vol. 9710, pp. 323–341. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_20
34. Reynolds, A., Blanchette, J.C., Cruanes, S., Tinelli, C.: Model finding for recursive functions in SMT. In: Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings. pp. 133–151 (2016). https://doi.org/10.1007/978-3-319-40229-1_10
35. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite model finding in SMT. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. pp. 640–655 (2013). https://doi.org/10.1007/978-3-642-39799-8_42, https://doi.org/10.1007/978-3-642-39799-8_42
36. Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7898, pp. 377–391. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_26, https://doi.org/10.1007/978-3-642-38574-2_26
37. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 530–541. PLDI '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594340, http://doi.acm.org/10.1145/2594291.2594340
38. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4424, pp. 632–647. Springer (2007). https://doi.org/10.1007/978-3-540-71209-1_49

39. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: Checking axiomatic specifications of memory models. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 341–350. PLDI '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1806596.1806635
40. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 190–204 (2017), http://dl.acm.org/citation.cfm?id=3009838
41. Winskel, G.: Event structures, pp. 325–392. Springer Berlin Heidelberg, Berlin, Heidelberg (1987). https://doi.org/10.1007/3-540-17906-2_31
42. Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI. pp. 298–303. Morgan Kaufmann (1995), http://ijcai.org/Proceedings/95-1/Papers/039.pdf

Appendix A Reformulation of happens before

Lahav et al. [24] define happens before, $\texttt{hb}$, in terms of *sequenced before* $\texttt{sb}$, the C++ name for program order, and *synchronises with*, $\texttt{sw}$, inter-thread synchronisation. Their $\texttt{rf}$ and $\texttt{rmw}$ relations match Y_{rf} and $\texttt{rmw}$ in our vocabulary. Fixed sequences of memory events initiate and conclude synchronisation, and these are captured by $\texttt{sw}_{begin}$ and $\texttt{sw}_{end}$. In the definition below, semicolon represents forward relation composition.

$$\texttt{sw} \coloneqq \texttt{sw}_{begin}; (\texttt{rf}; \texttt{rmw})^*; \texttt{sw}_{end}$$
$$\texttt{hb} \coloneqq (\texttt{sb} \cup \texttt{sw})^+$$

For efficiency we over-approximate transitive closures in the SO logic, but the nesting over-approximation that follows from the structure of $\texttt{hb}$ does not perform well. Instead we over-approximate a reformulation of $\texttt{hb}$.

$$\texttt{hb'} \coloneqq \texttt{sb} \cup (\texttt{sb}^?; \texttt{sw}_{begin}; ((\texttt{sw}_{end}; \texttt{sb}^?; \texttt{sw}_{begin}) \cup (\texttt{rf}; \texttt{rmw}))^*; \texttt{sw}_{end}; \texttt{sb}^?)$$

By unpacking the definition of $\texttt{sw}$, the reformulation flattens the nested closures into a single one. The closure combines fragments of happens before where at the start and end of the fragment, a synchronisation edge has been initiated but not concluded. Within the closure, the synchronisation edge can be concluded and a new one opened, or some number of read-modify-writes can be chained together with $\texttt{rf}$.

We explain the definition of $\texttt{hb'}$ by considering the number of $\texttt{sw}$ edges that consitute a particular $\texttt{hb}$ edge. If a $\texttt{hb}$ edge contains no $\texttt{sw}$ edge, then because $\texttt{sb}$ is transitive, the $\texttt{hb}$ edge must be a single $\texttt{sb}$ edge. Otherwise, the $\texttt{hb}$ edge is made up of a sequence of one or more $\texttt{sw}$ edges with $\texttt{sb}$ edges before, between and after some of the $\texttt{sw}$ edges. The first $\texttt{sw}$ edge is itself a sequence of edges starting with $\texttt{sw}_{begin}$. This is followed by any number of $\texttt{rf};\texttt{rmw}$ edges. At the end of the $\texttt{sw}$ edge there are two possibilities: this edge was the final $\texttt{sw}$ edge, or there is another in the sequence to be initiated next. The first conjunct of the closure, $\texttt{sw}_{end}; \texttt{sb}^?; \texttt{sw}_{begin}$ captures the closing and opening of $\texttt{sw}$ edges, the second captures the chaining of read-modify-writes. The end of the definition closes the final $\texttt{sw}$ edge with $\texttt{sw}_{end}$.

`fkcc`: the Farkas Calculator

Christophe Alias

CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon
Christophe.Alias@ens-lyon.fr
http://foobar.ens-lyon.fr/fkcc

Abstract. In this paper, we present FKCC, a scripting tool to prototype quickly program analysis and transformation exploiting the affine form of Farkas lemma. Our language is general enough to prototype in a few lines sophisticated termination and scheduling algorithms. The tool is freely avalailable and may be tried online via a web interface. We believe that FKCC is the missing chain to accelerate the development of program analysis and transformations exploiting the affine form of Farkas lemma.

Keywords: Farkas lemma · Scripting tool · Termination · Scheduling.

1 Introduction

Several program analysis and transformation requires to handle conjunction of affine constraints C and C' with a universal quantification as $\forall x \models C : C'$. For instance, this appears in loop scheduling [6, 7], loop tiling [2], program termination [1] and generation of invariants [3]. Farkas lemma – affine form – provides a way to get rid of that universal quantification, at the price of introducing quadratic terms. In the context of program termination and loop scheduling, it is even possible to use Farkas lemma to turn universally quantified quadratic constraints into *existantially quantified affine constraints*. This requires tricky algebraic manipulations, not easy to applied by hand, neither to implement.

In this paper, we propose a scripting tool, FKCC, which makes possible to manipulate easily Farkas lemma to benefit from those nice properties. More specifically, we made the following contributions:

- A general formulation for the resolution of equations $S(\boldsymbol{x}) = 0$ where S is summation of affine forms including Farkas terms. So far, this resolution was applied for specific instances of Farkas summation. Though that generalization is easy to find, it provides the basis of the FKCC scripting language.
- A scripting language to apply and exploit Farkas lemma; among polyhedra, affine functions and affine forms.
- Our tool, FKCC, implementing these principles, available at http://foobar.ens-lyon.fr/fkcc. FKCC may be download and tried online *via* a web interface. FKCC comes with many examples, making possible to learn the tool quickly.

This paper is structured as follows. Section 2 presents the affine form of Farkas, our resolution theorem, and explains how it applies to compute scheduling functions. Then, Section 3 define the syntax outlines informally the semantics

of the FKCC language. Section 4 presents two complete use-cases of FKCC. Finally, Section 5 concludes this paper and draws future research perspectives.

2 Farkas Lemma in Program Analysis and Compilation

This section presents the theoretical background of this paper. we first introduce the affine form of Farkas lemma. Then, we present our theorem to solve equations $S(\boldsymbol{x}) = 0$ where S is s summation including Farkas terms. This formalization will then be exploited to design the FKCC language.

Lemma 1 (Farkas Lemma, affine form). *Consider a convex polyhedron* $\mathcal{P} = \{\boldsymbol{x},\ A\boldsymbol{x} + \boldsymbol{b} \geq 0\} \subseteq \mathbb{R}^n$ *and an affine form* $\phi : \mathbb{R}^n \to \mathbb{R}$ *such that* $\phi(x) \geq 0 \quad \forall x \in \mathcal{P}$.
Then: $\exists \boldsymbol{\lambda} \geq \boldsymbol{0}, \lambda_0 \geq 0$ *such that:*

$$\phi(\boldsymbol{x}) = {}^t\boldsymbol{\lambda}(A\boldsymbol{x} + \boldsymbol{b}) + \lambda_0 \quad \forall \boldsymbol{x}$$

Hence, Farkas lemma makes possible to remove the quantification $\forall \boldsymbol{x} \in \mathcal{P}$ by encoding directly the positivity over $\mathcal{P}$ into the definition of ϕ, thanks to the Farkas multipliers $\boldsymbol{\lambda}$ and λ_0. In the remainder, *Farkas terms* will be denoted by: $\mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(\boldsymbol{x}) = {}^t\boldsymbol{\lambda}(A\boldsymbol{x} + \boldsymbol{b}) + \lambda_0$. We now propose a theorem to solve equations $S(\boldsymbol{x}) = 0$ where S involves Farkas terms. The result is expressed as a conjunction of affine constraints, which is suited for integer linear programming:

Theorem 1. *Consider a summation* $S(\boldsymbol{x}) = \boldsymbol{u} \cdot \boldsymbol{x} + v + \sum_i \mathfrak{F}(\lambda_{i0}, \boldsymbol{\lambda_i}, A_i, \boldsymbol{b}_i)(\boldsymbol{x})$ *of affine forms, including Farkas terms. Then:*

$$\forall \boldsymbol{x} :\ S(\boldsymbol{x}) = 0 \quad \textit{iff} \quad \begin{cases} \boldsymbol{u} + \sum_i {}^tA_i\boldsymbol{\lambda}_i = \boldsymbol{0} \wedge \\ v + \sum_i (\boldsymbol{\lambda}_i \cdot \boldsymbol{b}_i + \lambda_{0i}) = 0 \end{cases}$$

Proof. We have:

$$\begin{aligned} S(\boldsymbol{x}) &= {}^t\boldsymbol{x}\left(\sum_i {}^tA_i\boldsymbol{\lambda}_i\right) + \sum_i (\boldsymbol{\lambda}_i \cdot \boldsymbol{b}_i + \lambda_{0i}) + \boldsymbol{u} \cdot \boldsymbol{x} + v \\ &= {}^t\boldsymbol{x}\left(\boldsymbol{u} + \sum_i {}^tA_i\boldsymbol{\lambda}_i\right) + v + \sum_i (\boldsymbol{\lambda}_i \cdot \boldsymbol{b}_i + \lambda_{0i}) \end{aligned}$$

$S(\boldsymbol{x}) = \boldsymbol{\tau} \cdot \boldsymbol{x} + \tau_0 = 0$ for any $\boldsymbol{x}$ iff $\boldsymbol{\tau} = \boldsymbol{0}$ and $\tau_0 = 0$. Hence the result. □

Application to scheduling Consider the polynomial product kernel depicted in Figure 3.(a). Farkas lemma and Theorem 1 may be applied to compute a *schedule*, this is a way to reorganize the computation of the program to fulfill various criteria (overall latency, locality, parallelism, etc). On this example, a schedule may be expressed as an *affine form* $\theta : (i, j) \mapsto t$ assigning a *timestamp* $t \in \mathbb{Z}$ to each iteration (i, j). This way, a schedule *prescribes* an execution order $\prec_\theta := \{((i, j), (i', j')) \mid \theta(i, j) < \theta(i', j')\}$. Figure 3.(b) illustrates the order

prescribed by the schedule $\theta(i,j) = i$: a sequence of vertical wave fronts, whose iterations are executed in parallel.

A schedule must be positive everywhere on the set of *iteration vectors* $\mathcal{D}_N = \{(i,j) \mid A\ {}^t(i,j,N) + \boldsymbol{b}\}$ (referred to as *iteration domain*). In general, the iterations domains are parametrized (typically by the array size N) and the schedule may depends on N. Hence we have to consider vectors (i,j,N) instead of (i,j):

$$\theta(i,j,N) \geq 0 \quad \forall (i,j) \in \mathcal{D}_N \tag{1}$$

Applying Farkas lemma, this translates to:

$$\exists \lambda_0 \geq 0, \boldsymbol{\lambda} \geq 0 \quad \text{such that} \quad \theta(i,j,N) = \mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(i,j,N) \tag{2}$$

Moreover, a schedule must *satisfy the data dependences* $(i,j) \rightarrow (i',j')$. $\rightarrow$ is generally expressed as a Presburger relation [8], in turned *abstracted* as a rational convex polyhedron Δ_N containing the correct vectors (i,j,i',j') and sometimes false positives. Here again, $\Delta_N = \{(i,j,i',j') \mid C\ {}^t(i,j,i',j',N) + \boldsymbol{d} \geq 0\}$ is parametrized by structure parameter N. This way, the correctness condition translates to:

$$\theta(i',j',N) > \theta(i,j,N) \quad \forall (i,j,i',j') \in \Delta_N \tag{3}$$

Note that $\theta(i',j',N) > \theta(i,j,N)$ is equivalently written as the positivity of an affine form over a convex polyhedron: $\theta(i',j',N) - \theta(i,j,N) - 1 \geq 0$. Applying Farkas lemma:

$$\exists \mu_0 \geq 0, \boldsymbol{\mu} \geq 0 \text{ such that } \theta(i',j',N) - \theta(i,j,N) - 1 = \mathfrak{F}(\mu_0, \boldsymbol{\mu}, C, \boldsymbol{d})(i,j,i',j',N)$$

Substituting θ using Equation (2), this translates to $S(i,j,i',j',N) = 0$, where $S(i,j,i',j',N)$ is defined as the summation:

$$\mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(i',j',N) - \mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(i,j,N) - \mathfrak{F}(\mu_0, \boldsymbol{\mu}, C, \boldsymbol{d})(i,j,i',j',N) - 1$$

Since $-\mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b}) = \mathfrak{F}(-\lambda_0, -\boldsymbol{\lambda}, A, -\boldsymbol{b})$, we may apply theorem 1 to obtain a system of affine constraints with $\lambda_0, \boldsymbol{\lambda}, \mu_0, \boldsymbol{\mu}$. Linear programming may then be applied to find out the desired schedule [2, 7]. The same principle might be applied in *termination analysis* to derive a ranking function [1], this will be developed in Section 4.

3 Language

This section specifies the input language of FKCC and outlines informally its semantics. Figure 1 depicts the input syntax of FKCC. Keywords and syntax sugar are written with `verbatim` letters, identifiers with *italic* letter and syntactic categories with roman letters. Among identifiers, p is a parameter, v is a variable (typically a loop counter) and id is an FKCC identifier.

program ::= (**parameters** **=** { *p*, ..., *p* }**;**)? instruction; ...; instruction;

instruction ::= object | *id* **:=** object | **lexmin** polyhedron | **lexmax** polyhedron | **set** *id*

object ::= polyhedron | affine_form | affine_function

polyhedron ::=
  [*p*, ..., *p* **]** **->** { **[** *v*, ..., *v* **]** **:** inequation **and** ... **and** inequation }
| polyhedron ***** ... ***** polyhedron
| **solve** affine_form **=** **0**
| **define** affine_form **with** *v*
| **keep** *v*, ..., *v* **in** polyhedron
| **find** *id*, ..., *id* **s.t.** affine_form **=** **0**

affine_form ::= leaf_affine_form | leaf_affine_form [**+-**] ... [**+-**] leaf_affine_form

leaf_affine_form ::=
  { **[** *v*, ..., *v* **]** **->** expression }
| **positive_on** polyhedron
| leaf_affine_form **.** affine_function
| *int*
| *int* ***** leaf_affine_form

affine_function ::= { **[** *v*, ..., *v* **]** **->** **[** expression, ..., expression **]** }

Fig. 1. **fkcc** syntax

Program, instructions, polyhedra An FKCC program consists of a sequence of instructions. There is no other control structure than the sequence. An instruction may assign an FKCC object (polyhedron, affine form or affine function) to an FKCC identifier, or may be an FKCC object alone. In the latter, the FKCC object is streamed out to the standard output. Also, we often need to compute the lexicographic optimum of a polyhedron, typically to pick an optimal schedule. FKCC uses *parameteric integer linear programming* [5] *via* the Piplib library. The result is a discussion on the parameter value:

```
parameters := {N};
lexmin [N] -> {[i,j]: 0 <= i and i <= N and 0 <= j and j <= N};
```

would give:

```
if(N >= 0)
  {
    [] -> {[0,0]}
  }
else
  {
    (no solution)
  }
;
```

Note that structure parameters *must* be declared with the `parameters` construct. When no parameters are involved, the `parameters` construct may be omitted. The ensure the compatibility with ISCC syntax, the parameters of a polyhedron *may* be declared on preceding brackets `[N] -> ...`. This is purely optional: FKCC actually does not analyze this part. The instruction `set` *id* emits `id :=` to the standard output. This makes possible to generate ISCC scripts for further analysis. Finally, the set intersection of two polyhedra `P` and `Q` is obtained with `P*Q`.

Affine forms An affine form may be defined as a *Farkas term*:

```
iterations := [] -> {[i,j,N]: 0 <= i and i <= N and 0 <= j and j <= N};
theta := positive_on iterations;
```

If `iterations` is $\{\boldsymbol{x} \mid A\boldsymbol{x} + \boldsymbol{b} \geq 0\}$, then `theta` is defined as $\mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})$ where λ_0 and $\boldsymbol{\lambda}$ are fresh positive variables. In that case, the polyhedron is *never* parametrized: the *parameters must be handled as variables*. In particular, do not name variables with identifiers declared as parameters with `parameters :=`, as they would be treated as parameters whatever the context. Affine forms might be summed, scaled and composed with *affine functions*, typically to adjust the input dimension:

```
to_target := {[i,j,i',j',N]->[i,j,N]};
to_source := {[i,j,i',j',N]->[i',j',N]};
sum := theta.to_target - 2*theta.to_source + 1 + {[i,j,i',j',N] -> 2*i-i'};
```

In a summation of affine forms, affine forms must have the same input dimension. Also, a constant (`1`) is automatically interpreted as an affine form (`[i,j,i',j',N] -> 1`). Affine forms may also be stated explicitely (`{[i,j,i',j',N] -> 2*i-i'}`). The terms of the summation are simply separated with `+` and `-`, no parenthesis are allowed.

Resolution The main feature of FKCC is the resolution of equations $S(\boldsymbol{x}) = 0$ where S is a summation of affine forms including Farkas terms. This is obtained with the instruction `solve`:

```
solve sum = 0;
```

The result is a polyhedron with Farkas multipliers (obtained after applying theorem 1):

```
[] -> {[lambda_0,lambda_1,lambda_2,lambda_3,lambda_4] :
       (2+lambda_0)+(-1*lambda_1) >= 0 and (-2+(-1*lambda_0))+lambda_1 >= 0 and
       lambda_2+(-1*lambda_3) >= 0 and (-1*lambda_2)+lambda_3 >= 0 and
       (-1*lambda_1)+(-1*lambda_3) >= 0 and lambda_1+lambda_3 >= 0 and
       (-1+(-2*lambda_0))+(2*lambda_1) >= 0 and (1+(2*lambda_0))+(-2*lambda_1) >= 0 and
       (-2*lambda_2)+(2*lambda_3) >= 0 and (2*lambda_2)+(-2*lambda_3) >= 0 and
       1+(-1*lambda_4) >= 0 and -1+lambda_4 >= 0 and lambda_4 >= 0 and
       lambda_0 >= 0 and lambda_1 >= 0 and lambda_2 >= 0 and lambda_3 >= 0 and
       lambda_4 >= 0 and lambda_0 >= 0 and lambda_1 >= 0 and lambda_2 >= 0 and lambda_3 >= 0};
```

At this point, we need to recover the coefficients of our affine form `theta` in terms of $\boldsymbol{\lambda}$ (`lambda_0,lambda_1,lambda_2,lambda_3`) and λ_0 (`lambda_4`). Observe that $\texttt{theta}(\boldsymbol{x}) = \mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(\boldsymbol{x}) = {}^t\boldsymbol{\lambda} A \boldsymbol{x} + \boldsymbol{\lambda} \cdot \boldsymbol{b} + \lambda_0$. If the coefficients of `theta` are written: $\texttt{theta}(\boldsymbol{x}) = \boldsymbol{\tau} \cdot \boldsymbol{x} + \tau_0$, we simply have: $\boldsymbol{\tau} = {}^t\boldsymbol{\lambda} A$ and $\tau_0 = \boldsymbol{\lambda} \cdot \boldsymbol{b} + \lambda_0$. This is obtained with `define`:

```
define theta with tau;
```

The result is a conjunction of definition equalities, gathered in a polyhedron:

```
[] -> {[lambda_0,lambda_1,lambda_2,lambda_3,lambda_4,tau_0,tau_1,tau_2,tau_3] :
       ((-1*lambda_0)+lambda_1)+tau_0 >= 0 and (lambda_0+(-1*lambda_1))+(-1*tau_0) >= 0 and
       ((-1*lambda_2)+lambda_3)+tau_1 >= 0 and (lambda_2+(-1*lambda_3))+(-1*tau_1) >= 0 and
       ((-1*lambda_1)+(-1*lambda_3))+tau_2 >= 0 and (lambda_1+lambda_3)+(-1*tau_2) >= 0 and
       (-1*lambda_4)+tau_3 >= 0 and lambda_4+(-1*tau_3) >= 0};
```

The first coefficients `tau_k` define $\boldsymbol{\tau}$, the last one defines the constant τ_0. On our example, `theta(i,j,N) = tau_0*i + tau_1*j + tau_2*N + tau_3`. Now we may gather the results and eliminate the λ to keep only $\boldsymbol{\tau}$ and τ_0:

```
keep tau_0,tau_1,tau_2,tau_3 in ((solve sum = 0)*(define theta with tau));
```

The result is a polyhedron with the solutions. Here, there are no solutions: the result is an empty polyhedron. All these steps may be applied once with the `find` command:

```
find theta s.t. sum = 0;
```

The coefficients are automatically named `theta_0`, `theta_1`, etc with the same convention as `define`. We point out that `define` *choose fresh names* for coeffients (e.g. `tau_4`, `tau_5` on the second time with ``tau'') whereas `find` *always choose the same names*. Hence `find` would be prefered when deriving separately constraints on the same coefficients of `theta`. `find` may filter the coefficients for several affine forms expressed as Farkas terms in a summation:

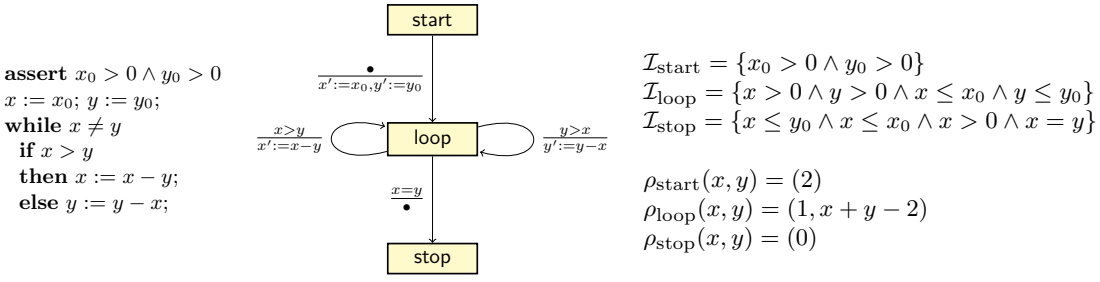
```
find theta_S,theta_T s.t.
  theta_T.to_target - theta_S.to_source - 1
     - (positive_on dependences_from_S_to_T) = 0;
```

This is typically used to compute schedules for programs with multiple assignments (here S and T with dependence from iterations of S to iterations of T). Finally, note that `keep tau_0,tau_1,tau_2,tau_3 in P;` projects `P` on variables `tau_0,tau_1,tau_2,tau_3`: the result is a polyhedron with integral points of coordinates `(tau_0,tau_1,tau_2,tau_3)`. This way, the order in which `tau_0,tau_1,tau_2,tau_3` are specified to `keep` impacts directly a further lexicographic optimization.

4 Examples

This section show how FKCC might be used to specify in a few lines termination analysis and loop scheduling.

4.1 Termination analysis



(a) Kernel (b) Affine automaton (c) Invariants and ranking

Fig. 2. Termination example

Consider the example depicted on Figure 2. The program computes the gcd of two integers x_0 and y_0 (a). It is translated to an affine automaton (b) (also called integer interpreted automaton), in turn analysed to check the termination (c): does the program terminates for *any* input (x_0, y_0) satisfying the precondition $x_0 > 0 \wedge y_0 > 0$?

This problem is – as most topics in static analysis – undecidable in general. However, we may conclude when it is possible to derive statically an abstraction precise enough of the program execution. In [1], we provide a termination algorithm based on the computation of a *ranking*. A ranking is an application $\rho_{\text{label}} : \mathbb{Z}^n \to (\mathcal{R}, \prec)$ which maps each reachable state of the automaton

$\langle label, \boldsymbol{x} \rangle$ to a *rank* belonging to well-founded set. On our example a reachable state could be $\langle loop, (x:3, y:3, x_0:3, y_0:6) \rangle$ after firing the initial transition and the right transition.

The ranking is decreasing on the transitions: for any transition $\langle label, \boldsymbol{x} \rangle \to \langle label', \boldsymbol{x'} \rangle$, we have: $\rho_{\text{label'}}(\boldsymbol{x'}) \prec \rho_{\text{label}}(\boldsymbol{x})$. Since ranks belong to a well founded set, there are – by definition – no infinite decreasing chain of ranks. Hence infinite chains of transitions from an initial state never happen.

On [1], we provide a general method for computing a ranking of an affine automata. Our ranking is *affine per label*: $\rho_{\text{label}}(\boldsymbol{x}) = A_{label}\boldsymbol{x} + b_{label} \in \mathbb{N}^p$. Figure 2.(c) depicts the ranking found on the example. Ranks ordered with the lexicographic ordering $\ll$, the well-founded set is $(\mathbb{N}^p, \ll)$. This means that, by decreasing order, `start` comes first (2), then all the iterations of `loop` (1), and finally `stop` (0). The transitions involved to compute those constants are the transitions from `start` to `loop` and the transitions from `loop` to `stop`. Then, transitions from `loop` to `loop` (left, denoted τ_1 and right, denoted τ_2) are used to computed the second dimension of ρ_{loop}. *In the remainder, we will focus on the computation of the second dimension of* ρ_{loop} *(*$x + y - 2$*) from transitions* τ_1 *and* τ_2. We will write $\rho_{\text{loop}}(\boldsymbol{x})$ for $\rho_{\text{loop}}(\boldsymbol{x})[1]$ to simplify the presentation.

Positivity on reachable states The ranking must be positive on reachable states of `loop`. The set of $\boldsymbol{x}$ such that $\langle loop, \boldsymbol{x} \rangle$ is reachable from an initial state is called the *accessibility set* of `loop`. In general, we cannot compute it – this is the undecidable part of the analysis. Rather, we compute an over-approximation thanks to linear relation analysis [4, 9]. This set is called an *invariant* and will be denoted by $\mathcal{I}_{\text{loop}}$. Figure 2.(c) depicts the invariants on the program. All the challenge is to make the invariant close enough to the accessibility set so a ranking can be computed. In FKCC, the assertion $\boldsymbol{x} \models \mathcal{I}_{\text{loop}} \Rightarrow \rho_{\text{loop}}(\boldsymbol{x}) \geq 0$ translates to:

```
I_loop := [] -> {[x,y,x0,y0]: x>0 and y>0 and x <= x0 and y <= y0};
rank := positive_on I_loop;
```

Decreasing on transitions Now it remains to find a ranking decreasing on transitions τ_1 and τ_2. We first consider τ_1. The assertation $\boldsymbol{x} \models \mathcal{I}_{\text{loop}} \wedge x > y \Rightarrow \rho_{\text{loop}}(x - y, x, x_0, y_0) < \rho_{\text{loop}}(x, y, x_0, y_0)$ translates to:

```
tau1 := [] -> {[x,y,x0,y0]: x>y};
s1 := find rank s.t. rank - (rank . {[x,y,x0,y0]->[x-y,y,x0,y0]}) - 1
                         - positive_on (tau1*I_loop) = 0;
```

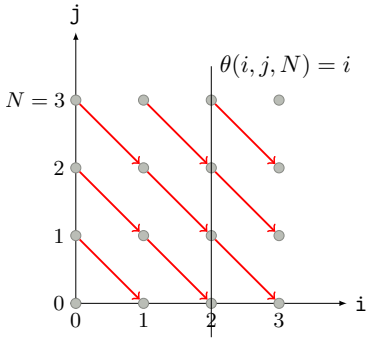
Similarly we compute a solution set `s2` from τ_2 and $\mathcal{I}_{\text{loop}}$. Finally, the ranking is found with the instruction `lexmin (s1*s2);`, which outputs the result:

```
[] -> {[1,1,0,0,-2]};
```

This corresponds to the dimension $x + y - 2$.

for $i := 0$ **to** N
  for $j := 0$ **to** N
    c[i+j] := c[i+j] + a[i]*b[j];

(a) Product of polynomials



(b) Iterations and schedule

Fig. 3. Scheduling example

4.2 Scheduling

Figure 3 depicts an example of program (a) computing the product of two polynomials specified by their array of coefficients `a` and `b`, and the iteration domain with the data dependence across iterations (b) and an example schedule θ prescribing a parallel execution by vertical waves, as discussed in Section 2.

Positivity Similarly to the ranking, the positivity condition (1) translates to:

```
iterations := [] -> { [i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N};
dependence := [] -> { [i,j,i',j',N]: 0 <= i and i <= N  and 0 <= j and
                        j <= N and 0 <= i' and i' <= N  and 0 <= j' and
                        j' <= N and i+j = i'+j' and i<i'};

# theta(i,j,N) >= 0 for any iteration (i,j,N)
theta := positive_on iterations;
```

Correctness We enhance the correctness condition (2) by making possible to *select* the dependence to satisfy. For each dependence class d, we use a 0-1 variable ϵ_d. Here we have a single dependence class from S to S, so have only one 0-1 variable ϵ:

$$\theta(i',j',N) \geq \theta(i,j,N) + \epsilon \quad \forall (i,j,i',j') \in \Delta_N$$

On the ranking example, we would have four classes ($i = start \to loop, \tau_1, \tau_2, e = loop \to stop$). This makes possible to choose which dependence class is satified ($\epsilon_d = 1$) or just respected ($\epsilon_d = 0$). This is the way multidimensional schedules are built [7]: on the termination example we would have $\epsilon_i = \epsilon_e = 1, \epsilon_{\tau_1} = \epsilon_{\tau_2} = 0$ for the first dimension, then $\epsilon_{\tau_1} = \epsilon_{\tau_2} = 1$ for the second dimension. Here it is kind of artificial, since we have a single dependence. However, the presentation generalizes easily to several dependence classes. This translates as:

```
parameters := {inv_eps,eps};
```

```
to_target := {[i,j,i',j',N]->[i',j',N]};
to_source := {[i,j,i',j',N]->[i,j,N]};

# s -> t ==> theta(s) <= theta(t) + eps, 0 <= eps <= 1
theta_correct := solve (theta . to_target) - (theta . to_source)
                       + {[i,j,i',j',N] -> -1*eps}
                       - (positive_on dependence) = 0;
theta_def := define theta with theta;
eps_correct := [] -> {[i]: 0 <= eps and eps <= 1 and inv_eps = 1-eps};
```

Here is the trick: parameters are forbidden to define Farkas terms; however parameters are perfectly allowed in summation. In that case, **the resolution interprets parameters as constants**. Hence the trick to set ϵ as a parameter and to put it in the summation by declaring an explicit affine form {`[i,j,i',j',N] -> -1*eps`}. We then keep the definition of theta coefficients in terms fo Farkas multipliers (`theta_def`) and the domain of ϵ (`eps_correct`).

Optimality We seek a schedule θ with a minimal latency $\ell(\theta)$ (number of steps). When θ is an affine form, $\ell(\theta)$ may be bound by an affine form $L(N)$ of the structure parameters [6]: $\ell(\theta) \leq L(N)$. This means that:

$$\forall (i,j) \in \mathcal{D}_N : \quad \theta(i,j,N) \leq L(N)$$

Which is, again, completely Farkas compliant. It remains to express $L(N)$, which have to be positive provided $\mathcal{D}_N$ is not empty i.e. $N \geq 0$. This translates to:

```
# L(N) >= 0 on the parameter domain
latency := positive_on ([] -> {[N]: N >= 0});

# theta(i,j,N) <= L(N)
theta_bounded := solve (latency . {[i,j,N] -> [N]}) - theta
                        - (positive_on iterations) = 0;
bound_def := define latency with latency;
```

Finally, it remains to gather the constraints (positivity, correctness, optimality) to obtain the result:

```
lexmin (keep inv_eps,latency_0,latency_1,theta_0,theta_1,theta_2,theta_3,eps
        in theta_correct*theta_def*eps_correct*theta_bounded*bound_def);
```

By priority order, we want to (i) maximize the depedence satisfied (minimize `inv_eps`), then (ii) to minimize the latency (L(N) = latency_0*N + latency_1). This amounts to find the lexicographic minimum with variables ordered as `(inv_eps,latency_0,latency_1)`. Note that `eps` and `inv_eps` are parameters. Adding them to the variable list of `keep` has the effect to turn them to counters `eps_counter` and `inv_eps_counter`. We obtain the following result, pretty-printed using the `-pretty` option:

```
theta_0 = 0
theta_1 = -1
theta_2 = 1
```

```
theta_3 = 0
latency_0 = 1
latency_1 = 0
eps_counter = 1
inv_eps_counter = 0
```

Hence $\theta(i, j, N) = N - j$, $L(N) = N$ and the dependence was satisfied (`eps_counter = 1`).

5 Conclusion

In this paper, we have presented FKCC, a scripting tool to prototype program analysis and transformations using the affine form of Farkas lemma. The script language of FKCC is powerful enough to write in a few lines tricky scheduling algorithms and termination analysis. The object representation (polyhedron, affine functions) is compatible with ISCC[10], a widespread polyhedral tool featuring manipulation of affine relations. FKCC provides features to generate ISCC code, and conversely, the output of ISCC might be injected in FKCC. This will allow to take profit of both worlds.

We believe that scripting tools are mandatory to avaluate rapidly research ideas. So far, Farkas lemma-based research was locked by two facts: (i) applying by hand Farkas Lemma is almost impossible and (ii) implementing an analysis with Farkas lemma is traditionnally tricky, time consuming and highly bug prone. With FKCC, computer scientists are now freed from these constraints.

References

1. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: International Static Analysis Symposium (SAS'10) (2010)
2. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 101–113 (2008). https://doi.org/10.1145/1375581.1375595
3. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using nonlinear constraint solving. In: Springer-Verlag (ed.) CAV. pp. 420–432. No. 2725 in LNCS (2003)
4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th ACM Symposium on Principles of Programming Languages (POPL'78). pp. 84–96. Tucson (Jan 1978)
5. Feautrier, P.: Parametric integer programming. RAIRO Recherche Opérationnelle **22**(3), 243–268 (1988)
6. Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. International Journal of Parallel Programming **21**(5), 313–348 (Oct 1992). https://doi.org/10.1007/BF01407835

7. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II: Multi-dimensional time. International Journal of Parallel Programming **21**(6), 389–420 (Dec 1992)
8. Feautrier, P., Lengauer, C.: Polyhedron model. In: Encyclopedia of Parallel Computing, pp. 1581–1592 (2011)
9. Gonnord, L.: Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires. Ph.D. thesis, Université Joseph Fourier - Grenoble (2007)
10. Verdoolaege, S.: Counting affine calculator and applications. In: IMPACT (2011)

Experiments in Context-Sensitive Incremental and Modular Static Analysis in CiaoPP
(Extended Abstract)

Isabel Garcia-Contreras[1,2], Jose F. Morales[1], and Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute
{isabel.garcia, josef.morales, manuel.hermenegildo}@imdea.org
[2] Universidad Politécnica de Madrid

Keywords: Program Analysis · Abstract Interpretation · Fixpoint Algorithms · Incremental Analysis · Modular Analysis · Logic Programming

Introduction and motivation. Abstract interpretation is a widely used technique for automatically detecting errors and proving program properties related to correctness, security, cost, etc. Performing such analysis during software development helps in early bug detection, but, given the size and complex structure of real-life programs, triggering a complete reanalysis for each set of changes is often too costly. However, development iterations normally involve small modifications in practice, which are often isolated within a small number of files or components. This can be taken advantage of to reduce the cost of re-analysis by reusing previous information. In particular, in CiaoPP [4, 2], incrementality-based cost reductions have been achieved to date at two levels: on one hand, ***modular* context-sensitive analysis** has been used to obtain global information on the whole program by iterating over local analyses of its components (modules). While this technique has been primarily aimed at reducing memory footprint, it can also achieve some incrementality. On the other hand, **fine grain context-sensitive *incremental* analysis** [3] identifies, invalidates, and recomputes only those parts of the analysis results that are affected by program changes. This analysis has been used to achieve very high levels of incrementality, with finer granularity (e.g., at program line level), but it does not take advantage of the module structure.

Objectives. We have recently been working on combining these two techniques within CiaoPP in order to achieve incrementality both at the intra- and inter-modular levels. Extending the *context-sensitive*, fine-grained, incremental analysis techniques to the modular setting requires dealing with the fact that the analysis of a module depends on the analysis of other modules in complex ways, through several paths to different versions (summaries) of the procedures. In order to bridge this gap we have developed a framework that analyzes separately the modules of a modular program, using context-sensitive fixpoint analysis while achieving both inter-modular (coarse-grain) and intra-modular (fine-grain) incrementality. Our objective is to give an overview (and demo) of the approach and, specially, report on the results (i.e., incremental gains) obtained so far.

Algorithm. The essence of the algorithm is that the concrete (possibly infinite) program execution trees are abstracted as graphs (essentially, regular trees), with the analysis information split by procedure (predicate), and partitioned by module, while tracking the dependencies between predicates and modules. We solve the problems related to the propagation of the fine-grain change information across module

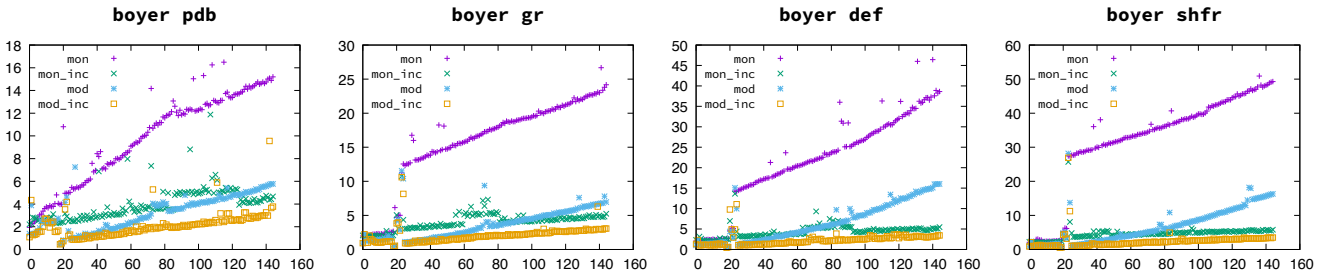


Fig. 1. Incremental analysis with different settings and domains

boundaries. We also work out the actions needed to recompute the analysis incrementally after multiple additions and deletions across modules in the program. We show that the analysis result is always correct and it is the best (most accurate) over-approximation of the actual behavior of the program. The full description of the algorithm is given in [1].

Experiments. The results of our experiments are also detailed in [1] and can be reproduced in `https://github.com/ciao-lang/ciaopp_tests`, but we include here some results for one characteristic program: a stylized version of the `Boyer-Moore` theorem prover, written modularly in Prolog. The experiment chosen consists in adding all the predicates in all the modules in the program one clause at a time, (re)running the analysis after each addition, as a stress test of the algorithm. The experiment is run with different domains: reachability (`pdb`), groundess (`gr`), dependency tracking via propositional clauses (`def`), and sharing and freeness, (pointer sharing and uninitialized pointers, `shfr`). The results (in terms of CiaoPP analysis time) are given in Fig. 1. The analyzer settings shown are the traditional monolithic, non-incremental analysis (`mon`), the (monolithic) incremental algorithm (`mon_inc`), the modular (coarse-grain incremental) algorithm (`mod`) and the current, fine-grain modular algorithm (`mod_inc`). It can be seen that the modular incremental approach is faster in most cases, and all incremental approaches are significantly faster than the traditional, non-incremental approach. In spite of the different costs of the various abstract domains (e.g., analyzing with `def` takes at most $40ms$ while analyzing with `pdb` takes at most $17ms$), the trend is clear: incremental analysis performs better than running the analysis from scratch. In general, the implementation and evaluation show that the proposed modular algorithm achieves competitive and, in some cases, improved, performance when compared to existing non-modular, fine-grain incremental analysis techniques. Furthermore, thanks to the more detailed propagation of inter-modular analysis information, our new algorithm outperforms the traditional modular analysis even when analyzing from scratch.

References

1. Garcia-Contreras, I., Morales, J.F., Hermenegildo, M.V.: An Approach to Incremental and Modular Context-sensitive Analysis. Tech. Rep. 1804.01839 (v4), arXiv (July 2019)
2. Hermenegildo, M., Puebla, G., Bueno, F., Lopez-Garcia, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Comp. Progr. **58**(1–2) (2005)
3. Hermenegildo, M.V., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM TOPLAS **22**(2), 187–223 (March 2000)
4. Hermenegildo, M., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. TPLP **12**(1–2), 219–252 (2012)

Boost the Impact of Continuous Formal Verification in Industry

Felipe R. Monteiro[1], Mikhail R. Gadelha[2], and Lucas C. Cordeiro[3]

[1]Federal University of Amazonas, Brazil, felipemonteiro@ufam.edu.br
[2]SIDIA Instituto de Ciência e Tecnologia, Brazil, mikhail.gadelha@sidia.com
[3]University of Manchester, UK, lucas.cordeiro@manchester.ac.uk

Abstract. Software model checking has experienced significant progress in the last two decades, however, one of its major bottlenecks for practical applications remains its scalability and adaptability. Here, we describe an approach to integrate software model checking techniques into the DevOps culture by exploiting practices such as continuous integration and regression tests. In particular, our proposed approach looks at the modifications to the software system since its last verification, and submits them to a continuous formal verification process, guided by a set of regression test cases. Our vision is to focus on the developer in order to integrate formal verification techniques into the developer workflow by using their main software development methodologies and tools.

Keywords: Formal Software Verification, Model Checking, DevOps.

1 Motivation

Currently, the formal verification community faces a pressing problem to ensure security and reliability of large codebases, which have a significant impact in millions of users [1]. Even minor defects can lead to huge impacts for companies and costumers [2]; for instance, in September 2018, attackers exploited three Facebook vulnerabilities and stole access tokens from as many as 50 million users, in order to take over their accounts [3]. In this particular context, software verification plays an important role in ensuring the overall product reliability. Even though formal techniques have been dramatically evolved over the past 15 years, the main challenges in the formal methods community remain **scalability and adoptability** [4]. *So how can we scale formal verification techniques for real-world software systems? How can we increase adoption of formal verification techniques by software engineers in industry?*

In order to tackle both aforementioned questions, **our vision** is to integrate formal verification techniques into the workflow of the main software development methodologies and tools. Our work is inspired by recent insights described by Sadowski et al. [2] who describe a set of lessons from building static analysis tools at Google. We believe that formal methods can be effective in improving software quality assurance of a large number of organisations around the globe. In particular, our approach aims to provide a solution that applies formal verification in a way that is both *(i)* low-effort *e.g.*, fits into existing processes, and *(ii)* scalable to the large software systems used in industry. Here our focus is on

software model checking techniques combined with DevOps culture, particularly, continuous integration (CI). On one hand, we have software model checking [4], which has been successfully applied to discover subtle errors but, for larger applications, often suffers from the state-space explosion problem [5]. On the other hand, we have continuous integration, which has been widely adopted by the software development community, but relies on a test suite that typically does not cover significant parts of the state-space [6].

We propose a continuous formal verification (CFV) approach, which aims to automatically detect design errors and integration problems as quickly as possible. First, we concentrate the verification effort to code changes rather than the entire system, thus, we only re-verify the code changes that could potentially break the properties of a system; this verification process should run fast (*e.g.*, in less than 5 minutes) in order to provide quick (and useful) feedback for developers. Second, we select the regression tests related to each code change (*e.g.*, an updated function), generalize these tests, and formally verify the code changes using software model checking. Lastly, we gather all the information from this analysis and report it back to the analytic and development team, who will carry out this process continuously; this step is crucial according to Sadowski et al. since careful developer workflow integration is key for any static analysis tool adoption by engineers [2].

Our **main contributions** are twofold. Firstly, we propose a feasible integration of software model checking into DevOps practices, thus making formal verification techniques accessible to software engineers. Here, our approach will focus on the developer and their feedback; the goal is to increase the adoption of our approach in real-world software projects by integrating our verification tool into the developer workflow. Secondly, we propose to reduce the impact of state-space explosion in development practices using existing regression tests in the verification process, which will provide quick and useful feedback for developers so that they can easily locate and fix bugs.

2 Continuous Formal Verification

The essence of this approach relies on the principle of compositional analysis [1]. Practically, we are inspired by CI practice, a well-known concept in Extreme Programming, proposed by Martin Fowler [6]. CI is particularly relevant when coupled with tools to automatically build and test a project's code base. Since the builds are generated after every incoming code changes (*i.e.*, commits and pull requests), problems can be detected much earlier. We can take advantage of such modularity to apply formal techniques in a continuous environment by model checking a software component only after it is changed, *i.e.*, we place our approach at diff-time. We use the same information (*i.e.*, development history and regression test cases), but in a way to substantially reduce verification complexity and increase coverage in a pull-based development model (*e.g.*, GitHub[1]).

The development cycle initiates with the developer submitting changes to the code base through a software configuration management (SCM) system. For each system build, we thus use the information from the SCM system to identify

[1] More info at https://help.github.com/articles/about-pull-requests/

the components that have actually been modified and focus on these. Importantly, we focus on C projects and each function is considered as a component. Equivalence checking [7] is then performed to identify which changes have an actual impact on the code base. At this point, the regression test suite (containing unit and functional tests) is of paramount importance, since we select the regression tests correspondent to the non-equivalent changed components. To increase coverage, these regression tests are passed to a generalization process. Finally, we check the use of non-equivalent components through the generalized regression tests and collect the reports (*e.g.*, counterexamples) and send it to the analytics team. In order to adopt such an approach, a project must comply with two basic guidelines: *(i)* the development process must be based on a continuous integration environment and *(ii)* it must include a regression test suite. We are currently building tools that can completely automate our CFV process.

The following sections describe the two steps of the process, highlighting key challenges: identifying relevant code changes, and the model checking of generalized test cases. As an illustrative example, we use the a GitHub project called `vec`[2], an ANSI-C type-safe dynamic array implementation. The repository contains 22 test cases, which *intend* to cover all possible execution paths related to the functionalities of the type-safe dynamic array. We focus on the function `vec_insert_`, shown in Fig. 1, to exemplify our proposed approach.

```
1  #define vec_unpack_(v) \
2    (char**)&(v)->data, &(v)->length, &(v)->capacity, sizeof(*(v)->data)
3
4  #define vec_insert(v, idx, val) \
5    ( vec_insert_(vec_unpack_(v), idx) ? -1 : \
6      ((v)->data[idx] = (val), 0), (v)->length++, 0 )
7
8  int vec_insert_(char **data, int *length, int *capacity, int memsz,
9                  int idx) {
10   int err = vec_expand_(data, length, capacity, memsz);
11   if (err) return err;
12   memmove(*data + (idx + 1) * memsz,
13           *data + idx * memsz,
14           (*length - idx) * memsz);
15   return 0;
16 }
```

Fig. 1: Implementation of the `vec_insert` function that adds the `val` value in the `idx` index of the `vec` structure. We omit the function `vec_expand_` for simplicity: it reallocates the vector if it needs to be expanded.

2.1 Checking for Relevant Code Changes

We begin from the principle that if a modified version of a component is computationally equivalent to its older version, then it is not necessary to prove that all properties that hold for the old version still hold for the modified one. Thus, we use equivalence checking to check whether the modified components need to

[2] Available at `https://github.com/rxi/vec`

be re-verified. Naturally, proving the equivalence of two functions is in general undecidable [7], and the effort we spend in trying to do so might be wasted. However, such an approach can potentially reduce the immediate verification effort, since proving the equivalence of two function versions can be less expensive than re-verifying the function [7]. In addition, by proving that two versions of a function are computationally equivalent, we eliminate the effort to re-verify any other function that depends on it (unless that function has been changed as well). Therefore, this approach limits the propagation of changes through the system and, consequently, reduces the effort to overall system verification.

The equivalence check will happen in two steps: a (1) fast and imprecise abstract syntax tree (AST) structural equivalence check [8], and a (2) slow and precise formal check e.g. bounded model checking (BMC). In the AST structural equivalence check, easy cases will be caught without the need to formally verify it, e.g., a function is renamed and the call sites are updated, or comments are added to a function body. If the AST is structurally not equivalent, we then encode the old and the new functions, and check if they are equivalent for the same inputs. A time limit is set for the formal check since it is more useful to spend time running the regression tests than checking their equivalence; if the time limit is exhausted, we assume they are not equivalent and start the tests.

In our illustrative project `vec` we find commits that would benefit from our approach. In commit $40d5cc17$[3], the developer changes that name of a macro `vec_absindex` used in an early version of the function shown in Fig. 1, and in commit $7d8588bc$[4], the developer removes the support for negative indexes when accessing arrays. In the former, the ASTs would found to be equivalent, neither triggering the next formal check nor starting the tests, while in the latter, the formulas would be found to be not equivalent by the formal check, triggering the regression tests.

Open Challenges. There are many techniques that could be applied to perform equivalence checking such as SYMDIFF [9] and CORK [10] tools or through directed incremental symbolic execution (DiSE) [11]; in future, we will evaluate their performance in this CFV setting. We will also exploit this module by generating test cases from code changes [12].

2.2 Model Checking Generalized Tests

It is of paramount importance a software project follows two key best-practice principles: *(i)* keep the project as modular as possible and create short functions that focus on one particular objective *(ii)* provide at least one regression/unit test for every function. Such an approach is key to a successful compositional analysis of the software project, where the combination of the analysis result of its parts represents the analysis result of the whole.

After pruning the unmodified components, we only focus on the existing regression test cases related to the modified ones, in order to reduce the state space to be explored by the model checker. However, we do not generate new concrete values for the test cases with the purpose of maximizing the code coverage. Instead, we combine existing test cases with non-deterministic input values to maximize the coverage of this verification. The use of regression tests also help

[3] https://github.com/rxi/vec/commit/40d5cc17ea41923c66286078bae82cc09c6458f7
[4] https://github.com/rxi/vec/commit/7d8588bc96c4c7aa68beb38f15704bd6135c0a5e

to reduce the state space by breaking the global model (containing the entire program) into local models (containing only the functions under verification) and generate on demand the reachable states to be visited by the model checker, starting with the state described by the test case. This reduces the number of paths and variables to be considered during model checking.

In our illustrative project `vec`, by measuring the number of linearly independent paths in all functions, i.e., the project's cyclomatic complexity [13], we clearly see the benefit of focusing on the regression tests. In the case of `vec`, the entire system has a cyclomatic complexity of 24; in contrast, its regression tests have an average cyclomatic complexity of 1.

Through BMC, we can check for all possible paths in the implementation shown in Fig. 1, by non-deterministically assigning a value for each function parameter (*i.e.*, `pos`, and `val`) assuming a valid initialized structure (*i.e.*, `v`). Rather than modifying the program, we modify the regression tests and replace the concrete input values by non-deterministic choices. Here, we replace the series of function invocations with a non-deterministic one (see lines 5–7 of Fig. 2b). We can try to get full coverage in this particular module because we already pruned the state space by only selecting the modified parts of the system.

```
1  test_section("vec_insert");
2  vec_int_t v;
3  vec_init(&v);
4  int i;
5  for (i = 0; i < 1000; i++)
6    vec_insert(&v, 0, i);
7  test_assert(v.data[0] == 999);
8  test_assert(
9    v.data[v.length - 1] == 0);
10 vec_insert(&v, 10, 123);
11 test_assert(v.data[10] == 123);
12 test_assert(v.length == 1001);
13 vec_insert(&v, v.length - 2, 678);
14 test_assert(v.data[999] == 678);
15 test_assert(
16   vec_insert(&v, 10, 123) == 0);
17 vec_insert(&v, v.length, 789);
18 test_assert(
19   v.data[v.length - 1] == 789);
20 vec_deinit(&v);
```

(a) Original test.

```
1  test_section("vec_insert");
2  vec_int_t v;
3  vec_init(&v);
4  int val = nondet_int();
5  size_t pos = nondet_size_t();
6  vec_insert(&v, pos, val);
7  test_assert(v.data[pos] == val);
8  vec_deinit(&v);
```

(b) Generalized version.

Fig. 2: Generalization of the regression test for the function shown in Fig. 1.

Open Challenges. Our main difficulty here is how to deal with false negatives as the non-deterministic choice of values for program variables may force the exploration of paths that are infeasible in the original program. So, we need to find a balance between coverage and soundness. We also need to increase automation as much as possible. One may combine techniques to automatically generate tests based on counterexamples [14] or source code [15]. We will also increase the power of this analysis by using conditional verifiers [16] or applying different model checking approaches (*i.e.*, explicit-state).

3 Related Work

Fitzgerald and Stol [17] present a holistic overview of the activities related to continuous software engineering, which includes continuous testing and verification. Although they do not propose a new approach, they highlight the importance of continuous (and automatic) testing and verification in the context of DevOps. Interestingly, Beyer and Lemberger [18] perform a comparison between software testers and software model checkers, which shows that model checkers are mature enough to be used in practice (they even outperform testing tools), and the combination of both techniques could lead to even better results. Indeed, there are many reports of successful attempts that use formal techniques in large software systems.

For instance, Klein *et al.* [19] show how to scale formal proofs based on software architecture to real systems at low cost; Godefroid, Levin, and Molnar [20] describe the remarkable impact of SAGE tool, which performs dynamic symbolic execution to hunt for security issues in Microsoft applications; Cordeiro, Fischer, and Marques-Silva [21] as well as Yin and Knight [22] propose approaches to conduct formal verification of large software systems. Furthermore, there are two important studies that tackle the combination of formal techniques with continuous integration, which led to promising results and reflect the need and scientific challenges in the industry to follow this road. First, Chudnov *et al.* [23] describe how Amazon Web Services (AWS) prove the correctness of their Transport Layer Security (TLS) protocol implementation, and how they use CI tools to keep proving the software properties during its lifetime. Similarly, O'Hearn [1] presents Infer, a static analyzer used at Facebook following a continuous reasoning approach. Neither Chudnov *et al.* nor O'Hearn try to handle model checking in a continuous process; the latter states this as an open challenge for the community.

These cases highlight the impact of formal techniques in real software systems; however, they do not present guidelines to generalize these approaches to a wide range of software projects, which could lead to a significant adoption of formal techniques by practitioners. Thus, there is still an open-call for approaches that could potentially popularize formal techniques in software engineering practices.

4 Conclusions and Future Work

Model checking of entire systems is usually not feasible for many industrial applications due to the state-space explosion problem, however, one of the scalability challenges can be solved through leveraging changes to the system. Thus, we propose CFV, an approach with the potential to detect software vulnerabilities by combining dynamic and static verification to reduce the state space. This potential propels us to further research this topic: we are currently developing an automated software tool to tackle the key challenges of equivalence checking and test case generalization, so it can be applied to large open-source projects. We are also working in close collaboration with software developers at Samsung with the goal of integrating our automated reasoning tool into their workflow, thus increasing adoption of formal methods in industry.

References

1. O'Hearn, P.W.: Continuous Reasoning: Scaling the Impact of Formal Methods. In: LICS. (2018) 13–25
2. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google. Commun. ACM **61**(4) (2018) 58–66
3. Rosen, G.: Security Update Facebook, Inc. (2018) [Online; accessed September-2018].
4. Clarke, E.M., Henzinger, T.A., Veith, H. In: Introduction to Model Checking. Springer International Publishing, Cham (2018) 1–26
5. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An Industrial-Strength C Model Checker. In: ASE. (2018) 888–891
6. Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B.: The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Study. In: ASE. (2017) 60–71
7. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Softw. Test., Verif. Reliab. **23**(3) (2013) 241–258
8. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proceedings of the 23rd International Conference on Computer Aided Verification. CAV'11, Berlin, Heidelberg, Springer-Verlag (2011) 669–685
9. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs. In: CAV. (2012) 712–717
10. Lopes, N.P., Monteiro, J.: Automatic Equivalence Checking of Programs with Uninterpreted Functions and Integer Arithmetic. STTT **18**(4) (2016) 359–374
11. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed Incremental Symbolic Execution. In: PLDI '11. (2011) 504–515
12. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In: SAS. (2011) 112–128
13. Bang, L., Aydin, A., Bultan, T.: Automatically computing path complexity of programs. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015, New York, NY, USA, ACM (2015) 61–72
14. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from Witnesses. In Dubois, C., Wolff, B., eds.: TAP. (2018) 3–23
15. Christakis, M., Emmisberger, P., Godefroid, P., Müller, P.: A General Framework for Dynamic Stub Injection. In: ICSE. (2017) 586–596
16. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based Construction of Conditional Verifiers. In: ICSE. (2018) 1182–1193
17. Fitzgerald, B., Stol, K.J.: Continuous Software Engineering: A Roadmap and Agenda. Journal of Systems and Software **123** (2017) 176–189
18. Beyer, D., Lemberger, T.: Software Verification: Testing vs. Model Checking. In: HVC. (2017) 99–114
19. Klein, G., Andronick, J., Fernandez, M., Kuz, I., Murray, T., Heiser, G.: Formally Verified Software in the Real World. Commun. ACM **61**(10) (2018) 68–77
20. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox Fuzzing for Security Testing. Queue **10**(1) (2012) 20:20–20:27
21. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: Continuous Verification of Large Embedded Software Using SMT-Based Bounded Model Checking. In: ECBS. (2010) 160–169
22. Yin, X., Knight, J.: Formal Verification of Large Software Systems. In: NFM. (2010) 192–201
23. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous Formal Verification of Amazon s2n. In: CAV. (2018) 430–446

Handling Heap Data Structures in Backward Symbolic Execution

Robert Husák, Jan Kofroň, and Filip Zavoral

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
{husak, zavoral}@ksi.mff.cuni.cz, jan.kofron@d3s.mff.cuni.cz

Abstract. Backward symbolic execution (BSE), also known as weakest precondition computation, is a useful technique to determine validity of assertions in program code by transforming its semantics into boolean conditions for an SMT solver. Regrettably, the literature does not cover various challenges which arise during its implementation, especially when we want to reason about heap objects using the theory of arrays and to use the SMT solver efficiently. In this paper, we present our achievements in this area. Our contribution is threefold. First, we summarize the two most popular state-of-the-art approaches used for BSE, denoting them as *disjunct propagation* and *conjunct combination*. Second, we present a novel method for modelling heap operations in BSE using the theory of arrays, optimized for incremental checking during the analysis and handling the input heap. Third, we compare both approaches with our heap handling implementation on a set of program examples, presenting their strengths and weaknesses. The evaluation shows that conjunct combination is the most efficient variant, exceeding the straightforward implementation of disjunct propagation in an order of magnitude.

Keywords: backward symbolic execution, weakest precondition, heap data structures, input heap, theory of arrays

1 Introduction

Symbolic execution is an established technique to explore semantics of programs, create tests with high code coverage and discover bugs [2]. To achieve that, it systematically explores the state space of the program reachable from the entry point, transforming the possible execution paths into boolean constraints. These constraints are usually passed to an SMT solver to determine the reachability of the corresponding paths. To reason about objects on the heap, several of the practically-usable tools [6, 18] use the theory of arrays [10], which can be handled by the most of the state-of-the-art SMT solvers [8].

If we are not interested in the exploration of the whole program and we want to inspect only one particular problematic place instead, we can use the *backward* variant of symbolic execution, sometimes referred to also as the *weakest precondition* analysis [7, 9]. As its name suggests, backward symbolic execution starts at the assertion of our interest and traverses the execution direction backwards. If it manages to reach the entry point and find an assignment satisfying the path constraints, it can provide us with a valuable test case. Otherwise, if no under-approximation is used and the assertion violation is proved to be unreachable, it is validated.

As we can see, each run of backward symbolic execution can be very expensive in terms of resources. However, the information it provides is potentially very detailed and useful for detecting the causes of errors. Therefore, it is important to use it in an appropriate context. There is a plethora of techniques which use some kind of abstraction, enabling them to efficiently analyse large programs for the cost of introducing false positives [17, 15, 11]. Backward symbolic execution can be then used only at the places where these techniques found potential errors in order to examine them further. For example, the authors of Snugglebug were able to verify 29 of 38 feasible *null* dereference exceptions found by FindBugs [11] in a Java codebase of 750 kLOC [7]. Another usage of backward symbolic execution is to run it in an interactive fashion, enabling programmers to gather as much information about a specific program error as possible [12].

Although backward symbolic execution can be indeed very useful, it is not as popular in the literature as the forward variant. Therefore, many important design considerations and potential complications have to be rediscovered during each implementation. For example, when calling an SMT solver multiple times, it is often efficient for the subsequently analysed conjunctions to share a common prefix. As we illustrate in Section 2, that complicates the way to handle the constraints, because we then should not alter the existing ones, only add new ones. We tackle this problem and other issues in our contributions:

1. We summarize the existing algorithms commonly used for backward symbolic execution in Section 3.
2. We present a novel way to transform heap operations into boolean constraints in Section 4. These transformations fit into the mentioned algorithms and utilize performance enhancements of the state-of-the-art SMT solvers.
3. We compare the performance of all the presented approaches on a set of code examples in Section 5.

In Section 6, we compare our approach to the most important papers and tools related to our work, while Section 7 concludes.

2 Problem

All the issues related to implementing a backward symbolic execution tool stem from its very nature. Forward symbolic execution starts with a fixed set of symbolic input variables and all the gradually constructed constraints can be essentially build from them. With backward symbolic execution, the situation is different, as the set of input variables constantly changes according to the variables encountered along the way. Every time a variable is read, it is added to this set; every time it is assigned to, it is removed from it.

We will illustrate the approaches on a simple method `ScalarExample` in Listing 1.1. The forward variant starts with a symbolic variable a assigned to `a`, at lines 2 and 3 it then assigns 1 to `b` and 2 to `c`. When it comes to the assertion `a != b` at line 4, it interprets it using the known values and negates the expression to discover any error inputs, resulting in a simple condition $a = 1$. The backward variant starts directly at the

Listing 1.1: Sample C# code to demonstrate symbolic execution

```
1  void ScalarExample(int a) {
2      int b = 1;
3      int c = 2;
4      Debug.Assert(a != b);
5  }
6
7  void HeapExample1(Node a) {
8      a.next = new Node();
9      Node b = a.next;
10     Debug.Assert(a != b);
11 }
12
13 void HeapExample2(Node a) {
14     Node b = a.next;
15     Node c = new Node();
16     Debug.Assert(b != c);
17 }
```

assertion, creating a condition $a = b$, where a and b are the input variables corresponding to their symbolic variables a and b, respectively. As the condition is not dependent on c in any way, it can safely skip its assignment at line 3. Next, the assignment of 1 into b at line 2 must effectively remove it from the set of input variables and replace it by 1 in the condition, resulting again in $a = 1$. Although this simple example does not demonstrate any significant differences, things become more complicated when heap operations and various efficiency optimizations are involved:

Constantly changing input heap: The rule with the constantly changing set of input variables applies to the input heap as well. Moreover, its analysis gets more complicated, because the objects from the input heap might get intertwined with the ones created during the analysed program run. Consider the assertion a != b on the line 10 in Listing 1.1. At first, the objects in the input heap might be possibly referenced by both a and b. The field read at line 9 causes b to be loaded from the current input heap. However, we cannot assume that the loaded reference is from the input heap as well, because it can always be assigned an explicitly allocated object, as at line 8. Therefore, we need to provide a way to correctly distinguish between the input heap and the explicitly allocated objects and to enable their various interactions.

Incremental solving: There are many usage scenarios of SMT solvers where we need to call them successively on similar formulas. E.g., in the case of symbolic execution, we might want to explore two independent code branches sharing the same prefix. Therefore, a modern SMT solver can be usually used incrementally, with the possibility to cache certain knowledge between subsequent calls. To add and remove assertions, they offer two useful mechanisms: an *assertion stack* and *assumptions*. The former enables us to use a stack-based system of scopes containing the particular assertions, with the

ability to destroy all the data of the topmost scope while retaining the remaining ones. The latter works by adding every assertion a in the form of $l \implies a$, where l is a literal specified later during each call of the solver. Because we want to utilize these features to optimize our SMT calls, it is important that we construct the formulas in a proper way, possibly combining the strengths of both techniques.

3 Backward Symbolic Execution

3.1 Notation

Let us clarify the terminology and semantics of various formulas and symbols used in this paper. If we speak about a *function* or *mapping* $g : A \to B$, it is understood as a partial function, hence defined on the subset of its domain A. If $g(a)$ for $a \in A$ is not defined, we denote it as $g(a) = \mathit{undef}$. The function $g[a \to b]$ is defined to be the same as g, except for it maps a to b. This notation can be generalized for a set: $g[\{a_1, ..., a_n\} \to b] = g[a_1 \to b]...[a_n \to b]$.

As to the formalism used for SMT queries, many-sorted first-order logic is used. Because the meaning of *sort* in logic corresponds to the meaning of *type* in computer science, we will use these two names interchangeably, according to the context. The *signature* $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P})$ comprises a set of *sorts* $\mathcal{S}$, a set of *function symbols* $\mathcal{F}$ and a set of *predicate symbols* $\mathcal{P}$. *Symbolic variables* Σ_v, *terms* Σ_t, *atoms* Σ_a, and *formulas* Σ_f are derived from the signature, using the standard recursive way. A formula $\varphi[a/b]$ is constructed by replacing all the occurrences of a in φ by b. The function $\textsc{Fresh}_{\Sigma_v}$ retrieves a symbolic variable not yet used in any context. In general, for any domain A, $\textsc{Fresh}_A$ retrieves a variable $a \in A$, which is not yet used in the analysis.

Let C be a set of classes contained in the analysed program. For each class $c \in C$, there is a corresponding set F_c containing all its fields. All the fields in the program are contained in $F = \bigcup_{c \in C} F_c$. To enable working with reference symbolic variables, we introduce a set of reference sorts $R = \{\sigma_c \mid c \in C\} \subset \mathcal{S}$. As an instrument to reason about types of fields and variables, we use function $t : V \cup F \to \mathcal{S}$. $F_{t(v)}$ as an abbreviation for F_c where $\sigma_c = t(v)$. Reference fields F_R and value fields F_V are defined as follows:

$$F_R = \{f \in F \mid t(f) \in R\} \qquad F_V = F \setminus F_R$$

Analogicaly, reference and value variables:

$$V_R = \{v \in \Sigma_v \mid t(v) \in R\} \qquad V_V = \Sigma_v \setminus V_R$$

Note that the sorts R representing reference variables are used only to ease formal description of heap operations. They are effectively replaced by functions on arrays and integers, as we describe in Section 4. All the reference variables are expected to point to objects on the heap, there is no notion of low-level pointers and of accessing stack variables by references.

To reason about a certain program, we expect it to be given as a control flow graph (CFG). Each node n contains at most one operation n.op and each edge $e = (n_1, \psi_e, n_2)$ is marked with a condition ψ_e. The possible operations follow: scalar assignment of term

$v_t \leftarrow_s t$, reference assignment $v_t \leftarrow_r v_r$, reference comparison assignment[1] $v_t \leftarrow (v_r^1 = v_r^2)$, new object creation $v_t \leftarrow_r \texttt{new}\ T$, field read $v_t \leftarrow v_r.f$ and field write $v_r.f \leftarrow v_v$. Note that the last two operations can occur both for reference and variable fields, in some cases we denote it by $\leftarrow_r$ and $\leftarrow_s$, respectively. Assertions are modelled as edges to special nodes.

To keep the scope limited, this paper does not directly address handling loops, interprocedural analysis or recursion. In order to evaluate our approach on programs of smaller size, we use a simple preprocessor for CFGs, which unwinds the loops for a given number of iterations. To handle interprocedural calls, we plan to extend it to handle inlining of the procedures up to a certain level of recursive calls. We are aware that this approach is underapproximate and does not scale well on larger programs. In order to mitigate this issue, we will inspire from the existing tools which were able to efficiently extend backward symbolic execution into an interprocedural analysis. Snugglebug uses directed call graph construction and tabulation, enabling it to explore the call graph lazily and reuse certain summaries obtained for each procedure [7]. Alter combines backward and forward symbolic execution to combine method summaries, utilizing interpolant computation to learn from infeasible paths [16].

3.2 Algorithm

Whereas in forward symbolic execution we usually want to reasonably spread our analysis among the state space to achieve high code coverage [2], backward symbolic execution often works by gathering summaries towards the entry point [7, 1]. At least in the intraprocedural case it is a natural approach, as we are interested in finding a feasible path between the entry node and the target node.

```
BSE(cfg, n_trg)
 1: var states: node → state
 2: states[n_trg] ← state representing true
 3: for all node n in cfg sorted by reverse dependency on n_trg do
 4:     var deps ← {(ψ_e, n_dep, states[n_dep]) | edge (n, ψ_e, n_dep) in cfg}
 5:     states[n] ← Merge(n, deps)
 6:     if DoSolve() ∧ Solve(GetCondition(states, n)) = UNSAT then
 7:         states[n] ← state representing false
 8: return states
```

Fig. 1: Backward symbolic execution algorithm

An overall algorithm structure is shown in Fig. 1. Given a target node n_{trg}, it traverses *cfg* backwards towards the entry node and gathers useful information along the way. The information is stored in the *states* associative array. In the beginning, because we expect *cfg* to be acyclic, we can sort its nodes according to their topological order in

[1] We did not put reference comparison directly in the edge conditions so that we can describe its processing later in the unified manner with the other heap operations, see Section 4.

the reversed *cfg*, skipping those not reachable from n_{trg}. This way, when processing a node, we are sure that the dependent nodes were already processed. For each node, we gather the states of the directly adjacent nodes and their corresponding edge conditions into *deps*. Merge is a core function responsible for inferring the state of a given node according to its dependencies. DoSolve is a heuristic returning *true* for the entry node and possibly also during the exploration so that certain infeasible parts get pruned. GetCondition is used to gather the condition corresponding to a given state, returns false if n_{trg} is unreachable from that node. Eventually the algorithm retrieves all the computed states. The caller can then extract interesting pieces of information from it, such as a possible input driving the execution towards n_{trg}.

state: formula in DNF
MergeDisj(n, *deps*)
1: var *merged* ← disjunction of $\{\psi_e \wedge d \mid d \text{ disjunct in } \varphi, (\psi_e, n_{dep}, \varphi) \in deps\}$
2: **return** ProcessOperationDisj(Simplify(*merged*), n.op)
ProcessOperationDisj(*state*, $v_t \leftarrow_s t$)
1: **return** *state*[v_t / t]
GetConditionDisj(*states*, n)
1: **return** *states*[n]

Fig. 2: Backward symbolic execution implementation using disjunct propagation

In the literature, we have identified two main possible implementations of this algorithm. The first, listed in Fig. 2, is based on formulas in DNF and their propagation in the form of disjuncts [7]. MergeDisj merges the disjunctions in all the dependent nodes and enhances them by their corresponding edge conditions, simplifying the resulting formula by Simplify and passing it to ProcessOperationDisj. Simplify applies various techniques of reducing a disjunction size while maintaining its semantics. ProcessOperationDisj handles an assignment $v_t \leftarrow_s t$ by replacing the target variable v_t by the term t representing its value, GetConditionDisj simply returns the disjunction for the given node. Heap operations and the implementation of GetCondition for heaps is described in Section 4.

In Fig. 3, the other implementation is listed [1]. Instead of propagating a set of disjuncts to the entry node, it associates each node n with a condition ψ_n describing its semantics and control flow. As seen in GetConditionConj, to reason about the whole path, we can pass a conjunction of these conditions to an SMT solver, which enables an efficient incremental usage. Since we can reason about mutable variables, our state contains also a map *vers* containing a version number for each encountered program variable. Unlike the previous case, we need to store certain information about a symbolic heap in each state; the details will be provided in Section 4.

MergeConj works as follows. Each node n is associated with a propositional variable c_n to express that the control flow reached it. The condition ψ_n is an implication with c_n on the left side. The right side consists of two parts: a join condition ψ_{join} and an operation condition ψ_{op}. The purpose of ψ_{join} is to model the branching of the control

state: (node condition ψ_n, *vers*: $V_V \to \mathbb{N}$, *heap*)
MergeConj(n, *deps*)

1: var *mergedVers* ← merge *vers* in *deps* to get the highest of each entry
2: (var *mergedHeap*, var *heapJoinConds*) ← MergeHeaps(*heaps* in *deps*)
3: var ψ_{join} ← disjunction of
{versioned $\psi_e \wedge c_{n_{dep}} \wedge$ JoinVers(*vers*, *mergedVers*) $\wedge$ *heapJoinCond* for *heap*
| $(\psi_e, n_{dep}, \psi_{n_{dep}}, vers, heap) \in deps$}
4: (var ψ_{op}, var *finalVers*, var *finalHeap*) ← ProcessOperationConj(*mergedVers*, *mergedHeap*, *n.op*)
5: **return** $(c_n \implies \psi_{join} \wedge \psi_{op}$, *mergedVers*, *mergedHeap*)

ProcessOperationConj(*vers*, *heap*, $v_t \leftarrow_s t$)

1: **if** $vers[v_t] = undef$ **then**
2: **return** (*true*, *vers*, *heap*)
3: **else**
4: var $oldVer \leftarrow vers[v_t]$
5: var $newVers \leftarrow vers[v_t \to oldVer + 1, \{\text{unknown variables in } t\} \to 0]$
6: **return** ($v_t^{oldVer} = t$ versioned by *newVers*, *newVers*, *heap*)

GetConditionConj(*states*, n)

1: **return** $c_n \wedge$ conjunction of $\psi_{n'}$ where n' is reachable from n

Fig. 3: Backward symbolic execution implementation using conjunct combination

flow by creating a disjunction on the edge conditions where each disjunct redirects the flow to the corresponding $c_{n_{dep}}$ and possibly synchronizes the variable versions of the dependent nodes using JoinVers. An operation condition, created by ProcessOperationConj, handles an assignment by making the given variable under its current version equal to the given term and associating the variable with a new version. Notice that if v_t has not been encountered so far, we can safely ignore the operation. Heap operation handling is described in Section 4, including the merging of heaps.

As we can see, each implementation is connected with certain advantages and disadvantages. The disjunct propagation approach is based on maintaining sets of disjuncts and simplifying them, while the operations are handled as term substitutions. As a result, the final condition can be potentially much simpler than in the other case, because it does not contain any helper variables representing various versions and Simplify can help to get rid of various repetitive patterns. On the other hand, if the simplification is not successful enough, the size of the resulting formula can be exponential with respect to the number of calls to Merge. Furthermore, it cannot fully utilize incremental SMT solvers, as they work by adding immutable conjuncts to an assertion stack. The conjunction combination case is able to use them efficiently and the generated condition size is usually linear with respect to the number of the analysed nodes, which is redeemed by the presence of helper variables.

Although in this work, the implementations are handled as two separate techniques, we plan to pursue a way to efficiently combine them, using the best features of both. Creating simple procedure summaries might be crucial for developing an efficient in-

terprocedural algorithm, whereas utilizing an incremental SMT solver might help with exploring large program state.

4 Modelling Heap Using Array Theory

4.1 Main Idea

The array theory enables SMT solvers to reason about heap memory in forward symbolic execution and concolic execution [6, 18]. Its axioms, in addition to those of theory of uninterpreted functions, follow [4]:

$$\forall a, i, j\ (i = j \Rightarrow read(write(a, i, v), j) = v)$$

$$\forall a, i, j\ (i \neq j \Rightarrow read(write(a, i, v), j) = read(a, j))$$

$$\forall a, b\ ((\forall i(a[i] = b[i]) \Leftrightarrow a = b)$$

As we can see, array theory generalises the operations of the array data structure, with the only difference being the immutability of the array variables. In the forward variant of symbolic execution, a common approach is to associate an array with each defined field and represent all the references by integers [18]. Reading a value from an instance can be then naturally modelled by using the *read* operation on the corresponding reference and array. Writing a value is similarly performed by using the *write* operation to produce a new version of the particular array. To ensure that different allocations of new objects do not reference the same object, we can use an internal counter and increment it every time an allocation is performed (allocation site counting) [3]. To denote *null* references, 0 is used.

All these principles can be directly adopted for backward symbolic execution as well [7]. However, to our knowledge there is a serious problem not sufficiently tackled in the literature. If we do not analyse a program from its very start, we expect that there are existing objects on the heap, prior to the entry point, where the analysis begins from, being called the input heap. Therefore, each reference can point either to an object located in the input heap, to *null*, or to an object allocated explicitly during the analysis. The problem is that if we do not constrain the references from the input heap to be distinct from the explicitly allocated objects, the SMT solver might produce a model where the references from those two distinct groups are equal.

Consider the method `HeapExample1` in Listing 1.1. Apart from the instance created at line 8, there is also an instance passed as the parameter `a`. Because this instance was created before the method call, we must assert that it is distinct from the former. Otherwise, an SMT solver might create an invalid model where $a = b$, so the input heap contains a reference to the explicitly created instance before it even exists.

Furthermore, all the references from `a` in the beginning of the method must point either to *null* or to other input heap instances. In method `HeapExample2`, we can see the reason. If we do not constrain the reference loaded from `a.next` in any way, the SMT solver can create a model where $b = c$.

A natural approach used in our solution is to restrict all the input heap objects to be represented as negative integers. In the case of forward symbolic execution, we can

remember the first version of the variable representing each field and then constrain it whenever we access it from any reference. In `HeapExample1`, we start with an input reference $a \leq 0$ and an array variable $next^0$ representing the field `next` in the beginning. At line 8 we assert $next^1 = write(next^0, a, 1)$, making $next^1$ the current version of the field. Nevertheless, when we access the field at line 9, we can retrospectively add a constraint $read(next^0, a) \leq 0$, making `a.next` from the input heap either to be *null* or to reference another object from the input heap. As we only add constraints and never alter the existing ones, this approach is naturally efficient for incremental solving.

When trying to using this approach in backward symbolic execution, we encounter a major problem. Because the view of the input heap continues to change as the analysis proceeds backwards, we cannot use any single version of the array variable representing the given field. For example, if we decide to set the input heap constraint at line 9 as $read(next^0, a) \leq 0$, we prevent `a.next` to be assigned any explicitly created instance, which exactly happens at line 8.

As we explain below, we tackle this problem by creating a helper "input" array variable for each field and firmly asserting its equality with the current field variable version only when explicitly checking the condition. A similar solution is created also for the reference variables, as they face the same issue.

4.2 Operation Definitions

The implementation of heap operation handling for the disjunct propagation algorithm from Fig. 2 is shown in Fig. 4. To mark symbolic variables corresponding to reference variables and fields, we use the s superscript. For a reference variable v, v^s represents a symbolic integer variable; for a field f, f^s represents a symbolic array variable indexed by integers. The value sort of f^s is $t(f)$ if $f \in F_V$, integer otherwise. The semantics of a reference variable v is as follows. If $v^s = 0$, v is *null*; therefore, $null^s = 0$. If $v^s > 0$, v references an object explicitly created during the analysed part of the program. Otherwise, if $v^s < 0$, v references an object in the input heap, i.e., it is created in the not yet analysed code.

We can see that assignments, comparisons and new object creations are implemented as simple replacements of the corresponding target variables in the existing formula. $\textsc{Fresh}_{\mathbb{N}_+}$ ensures that each created object is represented by a distinct number. A field write replaces all the occurrences of the given field array variable f^s by an expression that writes the given value v_v to f^s on the index given by the instance v_r. Because v_v can be either a reference variable or a scalar value (term), we use a helper function $\textsc{Symb}$ which optionally adds the s superscript if $v_v \in V_R$. Because the operation would not have been executed if v_r was *null*, we also add the condition $v_r^s \neq 0$.

When reading a value from a field, we distinguish between the scalar case $\leftarrow_s$ and the reference case $\leftarrow_r$. In the scalar case, we just replace the read variable by the formula representing array read and assert that v_r is not *null*. In the reference case, we also need to handle the aforementioned problems with input heap. Therefore, for each field f, we create also a helper symbolic array variable f^{in}, which is never rewritten during any operation. By adding $read(f^{in}, v_r^s) \leq 0$ we ensure that any read from the input heap using v_r will always either be *null* or reference an input heap object. These variables are then used in $\textsc{GetConditionDisjHeap}$, where we associate all the constraints gathered for

ProcessOperationDisjHeap(*state*, *op*)

switch *op* **do**
  case $v_t \leftarrow_r v_v$
    return $state[v_t^s \,/\, v_r^s]$
  case $v_t \leftarrow (v_1 = v_2)$
    return $state[v_t \,/\, (v_1^s = v_2^s)]$
  case $v_t \leftarrow_r$ `new` T
    return $state[v_t^s \,/\, \text{Fresh}_{\mathbb{N}^+}()]$
  case $v_r.f \leftarrow v_v$
    return $state[f^s \,/\, write(f^s, v_r^s, \text{Symb}(v_v))] \wedge v_r^s \neq 0$
  case $v_t \leftarrow_s v_r.f$
    return $state[v_t \,/\, read(f^s, v_r^s)] \wedge v_r^s \neq 0$
  case $v_t \leftarrow_r v_r.f$
    return $state[v_t^s \,/\, read(f^s, v_r^s)] \wedge v_r^s \neq 0 \wedge read(f^{in}, v_r^s) \leq 0$

GetConditionDisjHeap(*states*, *n*)

var *inputRefs* ← gather reference symbolic variables in *states*[*n*]
return $states[n] \wedge \bigwedge_{f \in F_R} f^s = f^{in} \wedge \bigwedge_{v \in inputRefs} v \leq 0$

Fig. 4: Heap operation modelling in the disjunct propagation approach from Fig. 2

them with their corresponding fields. We also identify all the input heap references and constrain them to be ≤ 0 as well.

As we can see from the algorithms in Fig. 2 and Fig. 4, the disjunct propagation approach is straightforward to implement and the condition transformations directly correspond to the operations. However, its efficiency heavily depends on the implementation of formula handling, especially their substitution and simplification. The best results are supposed to be obtained by a custom implementation which reflects all the requirements of the particular project [7]. It is also possible to reuse existing solutions, for example the efficient algorithms for terms in Z3 using its API [8].

Nevertheless, even with the best implementation possible, the conditions in certain programs can grow beyond a reasonable complexity, where every term substitution or simplification consumes too many resources. Therefore, we will now focus on the implementation of heap operations in Fig. 5 for the conjunct combination based algorithm shown in Fig. 3. Although the semantics regarding fields as array variables and references as integer variables remains the same, there are several differences, making the operations more complex. Because each condition is associated with the semantics of a single node and we cannot manipulate conditions for the already processed nodes, we are not allowed to use term substitution. Instead, we utilize a version-based mechanism similar to the implementation of assignment in ProcessOperationConj, where the version of the given variable is incremented and its equality with the particular term is added to the condition.

As a result, each node is also associated with a symbolic heap (η, α). The environment η contains all the current input heap reference variables and maps each of them either to 0 or to an integer symbolic variable. In the beginning of the analysis, η con-

tains only the mapping from *null* to 0. The field version map α associates each field $f \in F$ with a non-negative integer representing the current version of its array symbolic variable. If $\alpha[f] = i$, the variable is denoted f^i. Initially, all fields have the version 0.

heap: (environment $\eta : V_R \to \{0\} \cup \Sigma_v$, field versions $\alpha : F \to \mathbb{N}^0\}$
ProcessOperationConjHeap($vers$, (η, α), op)

1: var $\varphi \leftarrow true, vers' \leftarrow vers, \eta' \leftarrow \eta, \alpha' \leftarrow \alpha$
2: **switch** op **do**
3: **case** $v_t \leftarrow_r v_v$
4: **if** $\eta[v_t] \neq undef$ **then**
5: **if** $\eta[v_v] = undef$ **then**
6: $\eta' \leftarrow \eta[v_v \to \eta[v_t], v_t \to undef]$
7: **else**
8: $\eta' \leftarrow \eta[v_t \to undef]$
9: $\varphi \leftarrow \eta[v_t] = \eta[v_v]$
10: **case** $v_t \leftarrow (v_1 = v_2)$
11: $\eta' \leftarrow \text{Init}(\eta, v_1, v_2)$
12: $vers' \leftarrow vers[v_t \to vers[v_t] + 1]$
13: $\varphi \leftarrow v_t^{vers[v_t]} = (\eta'[v_1] = \eta'[v_2])$
14: **case** $v_t \leftarrow_r$ `new` T
15: **if** $\eta[v_t] \neq undef$ **then**
16: $\eta' \leftarrow \eta[v_t \to undef]$
17: $\varphi \leftarrow \eta[v_t] = \text{Fresh}_{\mathbb{N}^+}()$
18: **case** $v_r.f \leftarrow v_v$
19: $\eta' \leftarrow \text{Init}(\eta, v_r, v_v)$
20: $\alpha' \leftarrow \alpha[f \to \alpha[f] + 1]$
21: $\varphi \leftarrow (f^{\alpha[f]} = write(f^{\alpha'[f]}, \eta'[v_r], \text{Symb}(\eta', v_v))) \wedge (\eta'[v_r] \neq 0)$
22: **case** $v_t \leftarrow_s v_r.f$
23: $\eta' \leftarrow \text{Init}(\eta, v_r)$
24: $vers' \leftarrow vers[v_t \to vers[v_t] + 1]$
25: $\varphi \leftarrow (v_t^{vers[v_t]} = read(f^{\alpha[f]}, \eta'[v_r]) \wedge \eta'[v_r] \neq 0)$
26: **case** $v_t \leftarrow_r v_r.f$
27: $\eta' \leftarrow \text{Init}(\eta, v_r)$
28: $\varphi \leftarrow (\eta'[v_r] \neq 0)$
29: **if** $\eta[v_t] \neq undef$ **then**
30: **if** $v_t = v_r$ **then**
31: $\eta' \leftarrow \eta'[v_r \to \text{Fresh}_{\Sigma_v}()]$
32: **else**
33: $\eta' \leftarrow \eta'[v_t \to undef]$
34: $\varphi \leftarrow \varphi \wedge (\eta[v_t] = read(f^{\alpha[f]}, \eta'[v_r]) \wedge read(f^{in}, \eta'[v_r]) \leq 0)$
35: **return** $(\varphi, vers', (\eta', \alpha'))$

GetConditionConjHeap($states$, n)

1: var $inputRefs \leftarrow$ gather symbolic variables in η of the heap in $states[n]$
2: **return** GetConditionConj($states$, n) $\wedge \bigwedge_{f \in F_R} f^{\alpha[f]} = f^{in} \wedge \bigwedge_{v \in inputRefs} v \leq 0$

Fig. 5: Heap operation modelling in the conjunct combination approach from Fig. 3

Let us proceed to the semantics of ProcessOperationConjHeap. The reference assignment $v_t \leftarrow_r v_v$ distinguishes three cases. If we have not yet encountered v_t, it is not contained in η and we are not interested in any value assigned to it. Otherwise, if we do not know v_v, we associate it with variable[2] $\eta[v_t]$. If both v_t and v_v are known, we must assert the equality of their symbolic variables. Eventually, in any case, we must remove v_t from η, because by being assigned to it was effectively removed from the set of input heap references. When comparing two references v_1 and v_2, we use helper function Init, which associates them in η with fresh symbolic integer variables, if they are not already present there. Then, the scalar assignment of boolean term $\eta[v_t] = \eta[v_v]$ to v_t is performed, updating the version of v_t in *vers* accordingly. A new object creation is again modelled only if we have encountered the target reference variable v_t before. Its symbolic integer variable $\eta[v_t]$ is asserted to be equal with a fresh positive number and v_t is removed from η. A field write $v_r.f \leftarrow v_v$ needs to manipulate α by incrementing the version of f and using its two distinct versions to express the write. Note that due to the backward approach of our analysis, the version being written to is the current one.

Again, a field read operation is the most complicated one to model. In both scalar and reference cases, we use Init to ensure that there is a symbolic integer variable corresponding to v_r, constrain it not to be equal to *null* by $\eta'[v_r] \neq 0$ and use *read* to model the read of the field from the heap. In the scalar case $\leftarrow_s$, we must also handle the assignment into v_t by increasing its version in *vers*. In the reference case $\leftarrow_r$, when we are interested in the reference stored in v_t, we also use the helper f^{in} array variable enabling us to constrain the input heap later in GetConditionConjHeap. Note that we also explicitly handle the situation when $v_t = v_r$ in order not to accidentally remove v_r from the environment. MergeHeaps uses the same version map merging as MergeConj utilizing JoinVers. To merge environments with two or more distinct values corresponding to one reference variable, it is suitable to randomly pick one of them and constrain all the others to point to it. In the algorithm, we must avoid introducing unintentional aliases in the resulting environment.

4.3 Example

To demonstrate the operations on a real-life example, let us examine the assertion in Fig. 6, which corresponds to inspecting the reachability of the node n_{13} in the CFG. Notice that the heap operations from the code were decomposed into the atomic ones, producing helper variables such as `tv`, `tn` or `rnv`.

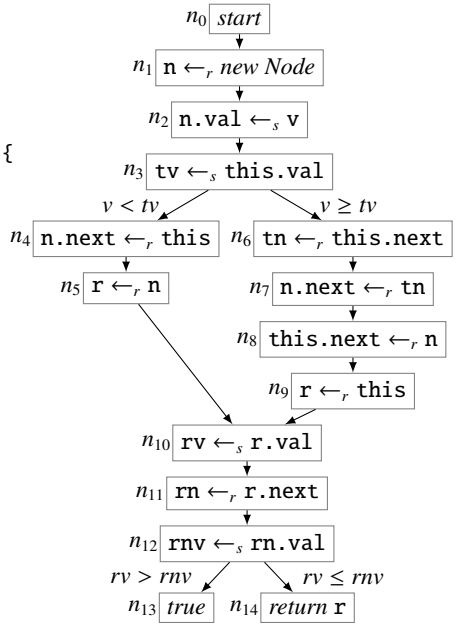
The solution using the disjunct propagation approach is depicted in Table 1. Each row captures the current state of the condition computed for it, starting from n_{13} and going backwards to n_0. The table is divided into four blocks according to the shape of the CFG. To simplify the notation, we do not use the *s* superscripts to denote symbolic variables, as all the variables in the condition are symbolic. Instead, they are differentiated by their font, as the program variables from the CFG use a monospaced one.

Since the reachability from n_{13} to n_{13} is trivial, the condition starts with *true*. Next, to reach it from n_{12}, the condition $rv > rnv$ is added and the field read is performed, replacing *rnv* with $read(val, rn)$ and ensuring that *rn* is not *null*. The next read into

[2] We expect that *null* cannot be on the left side of the assignment.

```
1  class Node {
2    public int val;
3    public Node next;
4
5    public Node AddSmaller(int v) {
6      Node n = new Node();
7      n.val = v;
8      Node r;
9      if (v < this.val) {
10       n.next = this;
11       r = n;
12     } else {
13       n.next = this.next;
14       this.next = n;
15       r = this;
16     }
17     Assert(r.val <= r.next.val);
18     return r;
19   }
20 }
```

(a) C# code



(b) CFG

Fig. 6: Sample C# code with heap objects and the corresponding CFG

rn is a reference one; therefore, $read(next^{in}, r) \leq 0$ is added. The helper variable *rv* is replaced by its semantics in n_{10}. Notice that if we called GetConditionDisjHeap at this point, the condition $next = next^{in} \wedge r \leq 0$ would be temporarily added, ensuring that the input heap consisting of *r* is separated from the objects potentially created during the analysis.

In n_9, the last node of the `else` branch, the assignment `r` $\leftarrow_r$ `this` causes the replacement of *r* by *this*. After the field write in n_8, $read(write(next, this, n), this)$ is simplified to *n*. Notice that now *next* is not a part of the formula and *this* and *n* are already constrained not to be *null*, so the operations in n_7 and n_6 do not have any effects. The semantics of the positive `if` branch is similar, as it replaces *r* by *n* and then reduces both occurrences of $read(next, n)$ to *this*.

Node n_3 merges the disjuncts from nodes n_6 and n_4, adds their respective conditions and performs the replacement of *tv* by $read(val, this)$. By the assignment `n.val` $\leftarrow_s$ `v` in n_2, we reduce $read(val, n)$ to *v*. The creation of new object in n_1 replaces *n* by 1 in both disjuncts, simplifying away the conditions $n \neq 0$. Finally, the condition for n_0 enhanced with input heap handling is passed to the SMT solver, proving the assertion by returning *UNSAT*.

Table 1: The verification of the assertion in Fig. 6 using disjunct propagation

n_{13}	$true$
n_{12}	$rv > read(val, rn) \wedge rn \neq 0$
n_{11}	$rv > read(val, read(next, r)) \wedge read(next, r) \neq 0$ $\wedge\ read(next^{in}, r) \leq 0 \wedge r \neq 0$
n_{10}	$read(val, r) > read(val, read(next, r)) \wedge read(next, r) \neq 0$ $\wedge\ read(next^{in}, r) \leq 0 \wedge r \neq 0$
n_9	$read(val, this) > read(val, read(next, this)) \wedge read(next, this) \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_8, n_7, n_6	$read(val, this) > read(val, n) \wedge n \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_5	$read(val, n) > read(val, read(next, n)) \wedge read(next, n) \neq 0$ $\wedge\ read(next^{in}, n) \leq 0 \wedge n \neq 0$
n_4	$read(val, n) > read(val, this) \wedge this \neq 0$ $\wedge\ read(next^{in}, n) \leq 0 \wedge n \neq 0$
n_3	$(v \geq read(val, this) \wedge read(val, this) > read(val, n) \wedge n \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0)$ $\vee\ (v < read(val, this) \wedge read(val, n) > read(val, this) \wedge this \neq 0$ $\wedge\ read(next^{in}, n) \leq 0 \wedge n \neq 0)$
n_2	$(v \geq read(write(val, n, v), this) \wedge read(write(val, n, v), this) > v \wedge n \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0)$ $\vee\ (v < read(write(val, n, v), this) \wedge v > read(write(val, n, v), this) \wedge this \neq 0$ $\wedge\ read(next^{in}, n) \leq 0 \wedge n \neq 0)$
n_1, n_0	$(v \geq read(write(val, 1, v), this) \wedge read(write(val, 1, v), this) > v$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0)$ $\vee\ (v < read(write(val, 1, v), this) \wedge v > read(write(val, 1, v), this) \wedge this \neq 0$ $\wedge\ read(next^{in}, 1) \leq 0)$

Table 2 shows how the conjunct combination variant works. As its name suggests, the assertions created for all the relevant nodes are combined using conjunction. In order to determine the reachability from n_1, we must combine all the conditions in the table. Notice that for each node n_i, there exist an environment η_i, a field version map α_i and a helper c_i to express that the control flow reached it.

The semantics of the operations is the same as in the former case, but the construction is different. In general, η_i and α_i keep track of the symbolic variables which represent the current versions of references and fields, respectively. As we can see in n_8, n_7, n_4 and n_2, every field read causes the corresponding α_i to create another version of its corresponding array symbolic variable. Whenever we read an unknown reference, we create a symbolic integer variable for it, such as in the case of η_{12}. As soon as that reference is being assigned to, we forget it, e.g. in η_{11}.

Let us have a look on the assignments in n_9 and n_5. The former one causes all the usages of `this` in the `else` branch to be represented by r, whereas the latter one does the same in the positive `if` branch for `n`. Their versions are properly united after being merged in n_3.

Table 2: The verification of the assertion in Fig. 6 using conjunct combination

n_{13}	$c_{13} \implies \mathit{true}$	$\eta_{13} = \{(\mathit{null}, 0)\}$ $\alpha_{13} = \{(\texttt{next}, 0), (\texttt{val}, 0)\}$
n_{12}	$c_{12} \implies c_{13} \wedge rv > rnv \wedge rnv = \mathit{read}(\mathit{val}^0, rn) \wedge rn \neq 0$	$\eta_{12} = \eta_{13}[\texttt{rn} \to rn]$
n_{11}	$c_{11} \implies$ $c_{12} \wedge rn = \mathit{read}(\mathit{next}^0, r) \wedge r \neq 0 \wedge \mathit{read}(\mathit{next}^{in}, r) \leq 0$	$\eta_{11} = \eta_{12}[\texttt{rn} \to \mathit{undef}]$
n_{10}	$c_{10} \implies c_{11} \wedge rv = \mathit{read}(\mathit{val}^0, r) \wedge r \neq 0$	$\eta_{10} = \eta_{11}[\texttt{r} \to r]$
n_9	$c_9 \implies c_{10}$	$\eta_9 = \eta_{10}[\texttt{r} \to \mathit{undef}, \texttt{this} \to r]$
n_8	$c_8 \implies c_9 \wedge \mathit{next}^0 = \mathit{write}(\mathit{next}^1, r, n) \wedge r \neq 0$	$\eta_8 = \eta_9[\texttt{n} \to n]$ $\alpha_8 = \alpha_{13}[\texttt{next} \to 1]$
n_7	$c_7 \implies c_8 \wedge \mathit{next}^1 = \mathit{write}(\mathit{next}^2, n, tn) \wedge n \neq 0$	$\eta_7 = \eta_8[\texttt{tn} \to tn]$ $\alpha_7 = \alpha_8[\texttt{next} \to 2]$
n_6	$c_6 \implies$ $c_7 \wedge tn = \mathit{read}(\mathit{next}^2, r) \wedge r \neq 0 \wedge \mathit{read}(\mathit{next}^{in}, r) \leq 0$	$\eta_6 = \eta_7[\texttt{tn} \to \mathit{undef}]$
n_5	$c_5 \implies c_{10}$	$\eta_5 = \eta_{10}[\texttt{r} \to \mathit{undef}, \texttt{n} \to r]$
n_4	$c_4 \implies c_5 \wedge \mathit{next}^0 = \mathit{write}(\mathit{next}^1, r, \mathit{this}) \wedge r \neq 0$	$\eta_4 = \eta_5[\texttt{this} \to \mathit{this}]$ $\alpha_4 = \alpha_{13}[\texttt{next} \to 1]$
n_3	$c_3 \implies$ $((c_4 \wedge v < tv \wedge \mathit{next}^1 = \mathit{next}^2 \wedge r = n)$ $\vee (c_6 \wedge v \geq tv \wedge r = \mathit{this}))$ $\wedge\ tv = \mathit{read}(\mathit{val}^0, \mathit{this}) \wedge \mathit{this} \neq 0$	$\eta_3 = \{(\mathit{null}, 0), (\texttt{this}, \mathit{this}), (\texttt{n}, n)\}$ $\alpha_3 = \{(\texttt{next}, 2), (\texttt{val}, 0)\}$
n_2	$c_2 \implies c_3 \wedge \mathit{val}^0 = \mathit{write}(\mathit{val}^1, n, v) \wedge n \neq 0$	$\alpha_2 = \alpha_3[\texttt{val} \to 1]$
n_1	$c_1 \implies c_2 \wedge n = 1$	$\eta_1 = \eta_3[\texttt{n} \to \mathit{undef}]$
n_0	$c_0 \implies c_1$	

We can see that in our simple example, the formula resulting from disjunct propagation is much shorter than the one from conjunct combination. However, in case of

larger programs with more branches, the number of disjuncts can grow in an exponential manner if we do not simplify them efficiently.

5 Evaluation

We implemented the techniques into a development version of AskTheCode, an open-source tool for backward symbolic execution of C# code, which uses Z3 as the SMT solver. In order to compare the efficiency of the aforementioned approaches, we prepared a simple program which can be parametrized so that its complexity and validity of the assertions can vary. *Degree counting*(a, b) is an algorithm receiving a linked list as the input. Each of its nodes contains an additional reference to another node and the algorithm calculates for each node its in-degree: the number of nodes referencing it. The assertion fails if it encounters a node whose in-degree is greater than its zero-based index in the list and also greater than a given number a. The second parameter b specifies the number of loop unwindings, i.e., the number of nodes inspected from the start of the list. As a result, the assertion is refutable if and only if $a + 2 \leq b$. Increasing b produces a larger CFG with also potentially more complicated conditions, but the counterexample might be easier to find due to a larger number of paths corresponding to it.

The execution time of analysis of each input variant is shown in Table 3[3]. Notice that there are multiple approaches both to disjunct propagation *Disj* and to conjunct combination *Conj*. Because we considered creating a custom implementation of term simplification and efficient representation too complex, we decided to use the well-optimized terms available in the API of Z3. $Disj_{Set}$ uses a set of Z3 terms to represent the disjuncts in each state. Their uniqueness is ensured by the hash consing implemented in Z3. The simplification is performed for each term separately. On the other hand, $Disj_{Z3}$ represents each state using a Z3 term; merging is performed by creating a disjunction of all the terms in the dependent nodes. $Disj_{Comb}$ is a combination of the two approaches. A state is represented as a Z3 term set, but the merging is performed by creating a disjunction term and putting it as a single item of the set. In $Conj_{Never}$, DoSolve always returns *false*, so no intermediate calls of the SMT solver are performed. An opposite extreme is $Conj_{Always}$, where DoSolve always returns *true*. In $Conj_{Loops}$, *true* is returned only for entry nodes of loops. The underlying solver is used incrementally, which enables it to reuse the information gained during the previous checks.

The results show that for our problem, conjunct combination was more efficient than disjunct propagation. The best times were obtained for $Conj_{Never}$, where the SMT solver was called only once at the very end of the analysis. However, in case of more complicated examples where an early check may prevent the analysis from inspecting large regions of code, the incremental usage of the SMT solver might be useful. The results of $Conj_{Always}$ show that it is unnecessary and inefficient to call it on every operation, as it causes an overhead of more than 250% on average. Instead, when we carefully select the nodes where to perform these additional checks like we did in $Conj_{Loops}$, the overhead is less than 25% on average.

[3] We conducted the experiments on a desktop with an Intel Core i7 CPU and 6GB RAM.

Table 3: Performance evaluation, the times are in milliseconds

Test Case	$Disj_{Set}$	$Disj_{Z3}$	$Disj_{Comb}$	$Conj_{Never}$	$Conj_{Always}$	$Conj_{Loops}$
Degree counting (0, 3)	298	775	668	18	55	20
Degree counting (1, 3)	302	773	688	21	60	26
Degree counting (2, 3)	284	752	675	15	59	20
Degree counting (1, 4)	2062	1225	791	31	119	46
Degree counting (2, 4)	2075	1152	822	43	121	53
Degree counting (3, 4)	1949	874	754	25	115	32
Degree counting (2, 5)	13334	1856	1287	91	242	102
Degree counting (3, 5)	13381	1947	1360	85	232	99
Degree counting (4, 5)	13226	1764	1125	40	246	50
Degree counting (3, 6)	81282	4728	4052	200	469	219
Degree counting (4, 6)	80853	4566	4214	161	427	178
Degree counting (5, 6)	80915	3116	2364	62	390	78

We believe that implementing a custom well-optimized simplifier will lead a substantial performance improvement of disjunct propagation, as achieved in the case of Snugglebug [7]. However, writing such a simplifier might be a challenging feat, whereas the utilization of incremental solving can efficiently move the problem to a well-optimized SMT solver.

6 Related Work

The disjunct propagation approach originates from Snugglebug [1], a tool using weakest preconditions to assess the validity of assertions in Java code. Snugglebug uses the algorithm for intraprocedural analysis, utilizing a custom-made simplifier over the propagated disjuncts. For interprocedural analysis, various other methods are used, such as directed call graph construction or tabulation. The SMT solver is utilized only at the entry point, as many infeasible paths are rejected using the simplifier. The conjunct combination approach is used in UFO [1] as the under-approximation subroutine. UFO, however, does not handle heap objects.

Microsoft Pex [18] is a tool generating unit tests for .NET programs using dynamic symbolic execution. It executes the program with concrete inputs and observes its behaviour, using the Z3 SMT solver to generate new inputs steering the execution to uncovered parts of the code. It also uses the array theory to model heap operations, but the way it works with the input heap is different from our pure symbolic approach.

KLEE [6] is a symbolic virtual machine utilizing the LLVM [13] infrastructure, used mainly for C and C++ projects. It uses array theory not only to reason about heap operations, but also about pointers, low-level memory accesses, etc. This differs from our approach, because we target only higher level languages with reference semantics, without the usage of pointers. Furthermore, KLEE does not support running symbolic execution backwards.

Symbolic execution tools JBSE [5] and Java StarFinder (JSF) [14] both employ lazy initialization to reason about heap objects, which lazily enumerates all the possible

shapes of the heap. They differ by the languages used for specification of the heap objects' invariants. Whereas JBSE uses custom-made HEX, JSF utilizes separation logic. Although we use a different approach for the core of the heap operations, taking heap invariants into account might help us to prune infeasible paths and save resources.

7 Conclusion

In this paper, we focused on the task of demand-driven program analysis by studying methods of efficiently implement backward symbolic execution. We identified two main approaches used for the core algorithm, namely disjunct propagation and conjunction combination. The former one has the benefit of easier implementation and creating potentially simpler conditions passed to an SMT solver, while the latter one is more predictable in terms of the resulting condition size and can better utilize incremental SMT solvers. To handle heap operations in both approaches, we use the theory of arrays, paying attention to properly handle the notion of an input heap throughout the analysis. The evaluation on our code examples shows that the effort put into the implementation of the conjunct combination approach is reasonable, because its results exceeded the straightforward implementation of disjunct propagation in an order of magnitude.

Due to the narrow focus of this work, the application of our technique is currently limited mainly by the inability to soundly handle loops, interprocedural calls and recursion. Our future work will mainly focus on removing these limitations by exploring the possibilities of computing and reusing procedure summaries, possibly learning from infeasible paths using interpolants. We will build on our knowledge of how disjunct propagation and conjunct combination perform in different circumstances, combining them to reach a valuable synergy.

Acknowledgements

This work was supported by the project PROGRESS Q48, the Czech Science Foundation project 17-12465S the grant SVV-2017-260451.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: TACAS (2012). https://doi.org/10.1007/978-3-642-28756-5_12
2. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 50 (2018)
3. Bjørner, N.: Engineering theories with z3. In: Yang, H. (ed.) Programming Languages and Systems. pp. 4–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
4. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer-Verlag, Berlin, Heidelberg (2007)
5. Braione, P., Denaro, G., Pezzè, M.: Jbse: A symbolic executor for java programs with complex heap inputs. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 1018–1022. ACM (2016)

6. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855756
7. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: A powerful approach to weakest preconditions. SIGPLAN Not. **44**(6), 363–374 (Jun 2009), http://doi.acm.org/10.1145/1543135.1542517
8. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
9. Dinges, P., Agha, G.: Targeted test input generation using symbolic-concrete backward execution. In: 29th IEEE/ACM International Conference on Automated Software Engineering (ASE). ACM, Västerås, Sweden (September 15-19 2014)
10. Goel, A., Krstić, S., Fuchs, A.: Deciding array formulas with frugal axiom instantiation. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning. pp. 12–17. SMT '08/BPR '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1512464.1512468
11. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Notices **39**, 92–106 (12 2004). https://doi.org/10.1145/1028664.1028717
12. Husák, R., Kofroň, J., Zavoral, F.: Askthecode: Interactive call graph exploration for error fixing and prevention. Electronic Communications of the EASST (in press)
13. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977673
14. Pham, L.H., Le, Q.L., Phan, Q.S., Sun, J., Qin, S.: Testing heap-based programs with java starfinder. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 268–269. ACM (2018)
15. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (May 2002). https://doi.org/10.1145/514188.514190, http://doi.acm.org/10.1145/514188.514190
16. Sinha, N., Singhania, N., Chandra, S., Sridharan, M.: Alternate and learn: Finding witnesses without looking all over. In: Proceedings of the 24th International Conference on Computer Aided Verification. pp. 599–615. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_42, http://dx.doi.org/10.1007/978-3-642-31424-7_42
17. Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Aliasing in object-oriented programming. chap. Alias Analysis for Object-oriented Programs, pp. 196–232. Springer-Verlag, Berlin, Heidelberg (2013), http://dl.acm.org/citation.cfm?id=2554511.2554523
18. Tillmann, N., De Halleux, J.: Pex: White box test generation for .net. In: Proceedings of the 2Nd International Conference on Tests and Proofs. pp. 134–153. TAP'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792786.1792798

AuthCheck: Program-state Analysis for Access-control Vulnerabilities

Goran Piskachev[1], Tobias Petrasch[2], Johannes Späth[1], and Eric Bodden[1,3]

[1] Fraunhofer IEM, Germany
{goran.piskachev,johannes.spaeth}@iem.fraunhofer.de
[2] BCG Platinion, Germany
petrasch.tobias@bcgplatinion.com
[3] Paderborn University, Germany
eric.bodden@upb.de

Abstract. According to security rankings such as the SANS Top 25 and the OWASP Top 10, access-control vulnerabilities are still highly relevant. Even though developers use web frameworks such as Spring and Struts, which handle the entire access-control mechanism, their implementation can still be vulnerable because of misuses, errors, or inconsistent implementation from the design specification. We propose AuthCheck, a static analysis that tracks the program's state using a finite state machine to report illegal states caused by vulnerable implementation. We implemented AuthCheck for the Spring framework and identified four types of mistakes that developers can make when using Spring Security. With AuthCheck, we analyzed an existing open-source Spring application with inserted vulnerable code and detected detected the vulnerabilities.

Keywords: static analysis, access control, authentication, authorization, web systems, security

1 Introduction

With increasing popularity and amount of processed data, web applications are attractive targets for attackers. The access-control vulnerabilities are still ones of the most relevant as rankings show. For instance, five of the SANS Top 25[4] most dangerous vulerabilities are related to access-control. On the OWASP Top 10[5] ranking, on place two is *broken authentication* vulnerability and on place five is *broken authorization* vulnerability.

Nowadays, web frameworks are heavily used by software developers [19]. Modern frameworks, such as Spring [6] and Struts. [7] provide mechanism for access-control making developers' implementation effort smaller. At runtime, the actual

[4] https://cwe.mitre.org/top25/
[5] https://www.owasp.org/index.php/Top_10-2017_Top_10
[6] https://spring.io/
[7] https://struts.apache.org/

access-control checks of such mechanism are performed within frameworks' code, hereby software developers do not need to write customized access-control code and implementation bugs are avoided.

Instead of writing access-control code manually, frameworks allow software developer to specify the access rules via framework specific APIs. Spring, for instance, provides a fluent interface with specification language SpEL [3] combined with Java annotations to allow the specification of access rules.

However, implementing the access-control rules using the frameworks APIs according to a design specification, created by the software architect, remains a challenging task. The access-control is often a combination of annotations of methods, a specifications in a configuration class, and a set of permission groups for the resources of the system (i.e., URI). The resulting access control of the implementation easily diverges from the design specification and the application may accidentally grant an unauthorized user access to confidential data.

In this paper, we propose a typestate-inspired analysis for detecting three access-control vulnerabilities:

- *CWE-306* missing authentication [8] - The system does not perform an identity check on a request to a resource which by design should be accessed only be identified requests.
- *CWE-862* missing authorization [9] - The system does not perform a check whether an authenticated request has the correct rights to access a resource.
- *CWE-863* incorrect authorization [7] - The system performs an authorization check on the resources, but this check is wrong.

Our static analysis uses *finite state machines* (FSMs) of each vulnerability to track the authorization state of the program. The state changes are triggered by method calls that authorize the user or access a critical resource along the control flow paths.

The main contributions of this paper are:

- AuthCheck: a program-state analysis for access-control vulnerabilities,
- an implementation of AuthCheck for the Spring Security framework,
- a running example and four typical errors in Spring Security, and
- a case study demonstrating the applicability of the implementation.

The following section introduces our running example within the Java Spring framework. In Section 3, we provide background information and definitions for the AuthCheck approach, which is then introduced in Section 4. Implementation details are discussed in Section 5. A case study and limitations are discussed in Section 6.

2 Running Example

As running example consider a minimal web-application that helps a user to organize her tasks. An anonymous user browsing the web application must only see the web applications version number. A user that is authenticated can view

tasks assigned to herself. An administrator (group *ADMIN*) can create new tasks for a particular user.

Table 1 details the design specification of the web-application's REST-API [10]. The specification maps the URI of an incoming request to the actual API method which shall be invoked to process the incoming request. Table 1 additionally details the permissions required for each request. A software architect specifies these requirements and hands them to a software developer.

Table 1: Specification resources and access rules in the running example

HTTP	URI	Resource	Description	Access rule
GET	/version	version()	Returns application's version.	No rule
GET	/profile	profile()	Returns user profile.	Authenticated
GET	/task	retrieveAll()	Returns list of all tasks.	*USER* or *ADMIN*
CREATE	/task	create()	Creates new task.	*ADMIN*

Spring-based Implementation The software developer uses the Spring framework[1] to implement the software as specified. Spring provides a security component [2] that ships with a mechanism for access control of resources. Spring handles requests from users via chain of filters (chain of responsibility design pattern [11]). The requests are matched and processed based on their URIs.

```
1  public class WebSecurityConfig extends
       WebSecurityConfigurerAdapter {
2  @Override
3  protected void configure(HttpSecurity http) throws
       Exception {
4  http.csrf().disable().sessionManagement()
5  .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
6  .and().authorizeRequests()
7  .antMatchers(HttpMethod.GET, "/version").permitAll()
8  .antMatchers(HttpMethod.GET,
       "/task").access("hasAnyRole('USER', 'ADMIN')")
9  .antMatchers(HttpMethod.CREATE, "/task").hasRole("USER")
10 .antMatchers(HttpMethod.GET,
       "/profile").authenticated().and().httpBasic();
11 }}
```

Listing 1.1: Resource and access-control configuration of the running example implemented with Spring Security

Listing 1.1 shows the implementation of Table 1 using Spring Security. By the use of a fluent interface the developer can implement the chain of filters that

is applied upon each incoming request at runtime. Each filter is created through the method `antMatcher(..)` defined by the HTTP method and the URI of the resources which that filter can process. The `permitAll()` method allows any request to access the resource. The `authenticated(..)` method creates a filter that restricts the incoming request to the one where the useer is authenticated. The `hasRole(..)` method allows access to the resource by any request that has the role of the specified group. The `access(...)` method evaluates the specified argument which has to be defined in the *Spring Expression Language (SpEL)* [3], and when evaluated to true, allows the corresponding request to access the resource.

The implementation has inconsistency with the specification. The software developer erroneously allowed basic users (*USER*) to create new tasks as opposed to restricting the action to *ADMIN*s only. AuthCheck detects the deviation from the specification automatically.

3 Background and Definitions

3.1 Typestate Analysis and Program-state Analysis

Typestate analysis [20] is a data-flow analysis that can detect invalid states of objects from the code being analyzed. The analysis uses specification of all possible states of the object, typically expressed as *final state machine (FSM)*. For example, using the *FSM* of the type *java.io.FileWriter*, in a given program, the analysis can report if any object of type *java.io.FileWriter* is not closed at the end of the program. Another example is CogniCrypt [13], a typestate analysis for detecting API misuses of cryptographic libraries.

To detect access-control vulnerabilities, such as *CWE-306* [8], *CWE-862* [9], and *CWE-863* [7], we designed a program-state [5] [12] analysis. Similar to the typestate analysis, the program-state analysis uses *FSM* to track the states, not of single objects, but the state of the program. Figure 1 shows the *FSM* that models the program states when detecting *CWE-306*. Based on our running example (Section 2), the *acm()* (authentication-critical method) is replaced by one of the resources, e.g. *profile()*. The legal states are *NA* (not authenticated) and *A* (authenticated). The *init()* transition models the entry point of the analysis, which in this case is the arrival of a request from a user. If the request is for the resource *profile()*, the application has to make sure that the call to the method *authenticate()* from Spring was successfully called before. This is modeled by the transition with label *authenticate()*. If this transition was fired, the state of the *FSM* will be changed from *NA* to *A*. In case, the implementation of the application does not contain a call to the method *authenticate()*, when the resource *profile()* is requested, the *FSM* will go to the state *CWE-306*, which models an illegal state and this can be reported.

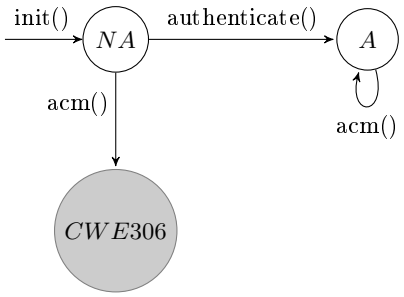


Fig. 1: *FSM* for missing authentication *CWE-306*

3.2 Definitions

Before we introduce the AuthCheck approach (Section 4), we define the term *web application*. In the following, we introduce the required terms. A user is a client program, e.g. web browser, that can send requests to the server.

Definition 1. *Authorization group*
Authorization group g is boolean characteristic of a user u with a unique name and access rights. A user can belong to more authorization groups. The set of all authorization groups is G:

$$G = \{g_i \mid g_i \text{ is an authorization group}, 1 \leq i \leq m,\ m \in \mathbb{N}\}$$

The function $userGroups$: $U \rightarrow Pow(G)$ maps each user $u \in U$ to the authorization groups. $Pow(G)$ is the power set of G. We define the help function $hasRole$: $U \times G \rightarrow \mathbb{B}$, that expresses whether a user u belongs to an authorization group g: $hasRole(u, g) := g \in userGroups(u)$

Each user that is authenticated in the system belongs to the special authorization group ANONYMOUS.

Authorization formula is a boolean formula a, formed by the function *hasRole*, *true*, *false*, and the operators $\vee, \wedge, \neg$.

Definition 2. *Resource*
An authentication and authorization critical resource is a 4-tuple $r = (m, p, s, a)$, where m is HTTP method, p is URI*, s is a method signature, and a is an authorization formula that defines the access rule of the resource. Access to the resource is given when a is evaluated to true for a request of a user u. Users identify each resource with the* URI *p and the HTTP method m. The coresponding method in the system is identified by the signature s.*

Definition 3. *Web application*
A web application W, is 2-tuple $W = (R, G)$, where R is a set of resources and G is a set of authorization groups.

Example The web application from Section 2 has the authorization groups *ADMIN*, for administrators and *USER*, for basic users. By default, it also has the *ANONYMOUS* group. Thus, $G = \{ANONYMOUS, ADMIN, USER\}$. The set of resources has 4 elements (Table 1). The first resource is defined as $r_1 = (GET, /version, String\ version(), a_1)$, where $a_1(u) = true$.

We consider a user u with $userGroups(u) = \{ANONYMOUS, USER\}$. If this user requests the resource r_1, the access will be allowed because $a_1(u) = true$. However, a request to the resource r_4 will be denied because $a_4(u) = hasRole(u, ANONYMOUS) \wedge hasRole(u, ADMIN) = false$.

4 Approach

We present AuthCheck, a program-state analyis for detecting three access-control vulnerabilities, *CWE-306*, *CWE-862*, and *CWE-863*. The analysis uses a call graph of the program (deatailed in Subsection 5.2) and an *access-control specification model (ACSM)*, like the one in Table 1. *ACSM* is defined as a web application $S = W_S$, where $W_S = (R_S, G_S)$ (Definition 3). *ACSM* can be created manually by software architects or automated from requirements and design specifications. Either way, we assume that the following information is available: resource API, URI, and access rule, that is aware of the autorization groups in the system.

AuthCheck checks whether the call graph confirms the *ACSM* by checking each path from the call graph. To extract all paths, the depth first search *DFS* algorithm is used. AuthCheck uses *FSM* for each vulnerability, e.g. Figure 1. Algorithm 2 shows the tracking of each path with the *FSM*. The *FSM* starts in the initial state (e.g. *NA* in Figure 1) and for each node of the path a new state of the *FSM* is calculated (line 4 in Algorithm 2). If an error state is reached (e.g. *CWE-306* in Figure 1), a new vulnerability will be reported.

For each path, the function *DetectVuln* is called which is defined by Algorithm 2. *DetectVuln* uses the *FSM* to analyse the path.

Algorithm 1 Check the call graph against vulnerabilities

```
1: function CheckCallGraph(CallGraph, FSM)
2:     Paths ← DFS(CallGraph)
3:     Vul ← ∅
4:     for each p ∈ Paths do
5:         Vul ← Vul ∪ DetectVuln(p, FSM)
6:     return Vul
```

The complexity of Algorithm 1 is $\mathcal{O}(|V| + |E| + |P| \cdot T(\text{DetectVuln}))$, where V is the number of nodes, E is the number of edges, and P is the number of paths in the call graph. In *DetectVuln*, every node of the path is analyzed, resulting in $\mathcal{O}(|P|)$. The worst case path is the one with all nodes from the call

Algorithm 2 Checking each path against vulnerabilities

```
1: function DetectVuln(Path, FSM)
2:     v ← FSM→init()
3:     for each n ∈ Path do
4:         v ← FSM→nextState(n)
5:         if v ∈ FSM.ERROR_STATES then
6:             return new Vulnerability(v)
7:     if v ∈ FSM.NOT_ERROR_STATES then
8:         return new Vulnerability(v)
9:     return ∅
```

graph $|V|$. Additionally, the number of paths in the worst case is $|E|$. Thus, the total complexity of Algorithm 1 is

$$\mathcal{O}(|V| + |E| + |P| \cdot |V|) = \mathcal{O}(|V| + |E| + |V| \cdot |E|) = \mathcal{O}(|V| \cdot |E|)$$

In the following, we discuss the three the *FSM* used by AuthCheck.

Missing Authentication A program is vulnerable to *CWE-306* when an authentication-critical method (*acm()*) can be accessed by user that has not been authenticated before. AuthCheck models this vulnerability as shown in Figure 1. Authentication critical methods are all resources that in the *ACSM* have an access rule that requires authentication. The error state in Figure 1 is reached when an authentication-critical method is processed next in a given path and the current state of the *FSM* is *NA* (not authorized). In this case, the program-state analysis will create a vulnerability (Algorithm 2, line 8).

Missing Authorization and Incorrect Authorization *CWE-862* occurs in a given program when a non-authorized user u can request an authorization-critical method (*azcm()*). If the user is authorized but the belonging group g does not confirm the access rule for that authorization-critical method as specified in the *ACSM* ($hasRole(u, g) = false$), then incorrect authorization occurs (*CWE-863*). Figure 2 shows the *FSM* that AuthCheck uses to model *CWE-862* and *CWE-863*. The transitions with the label *azcm()* without an argument denote calls to an authorization-critical method when the user is not authorized. When there is an argument g, the user has been authorized and the belonging group is being checked. This happens in state *A2*. When the user's group evaluates to true the self transition of state *A2* is fired, otherwise the transition to state *CWE-863* is fired. AuthCheck performs a group hierarchy check.

Strategies for detecting critical methods The transitions *acm()* in Figure 1 and *azcm()* in Figure 2 denote an authentication-critical and authorization-critical method. These methods correspond to the resources defined in the *ACSM*. In the following, we discuss AuthCheck's strategies for detecting these methods in the call graph.

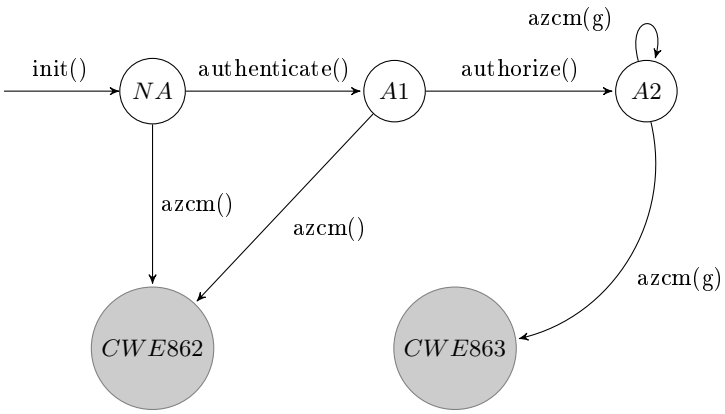


Fig. 2: *FSM* for missing authorization *CWE-862* and incorrect authorization *CWE-863*

Algorithm 3 Identifying methods as authentication-critical

```
1: function isMethodAuthenticationCritical(R, s')
2:     for each r ∈ R do
3:         if r_s = s' then
4:             return true
5:     return false
```

In the case of *CWE-306*, the authentication-critical methods are detected by iterating the set of all resources R from the *ACSM* for each method M that is currently processed in the path. The complexity for this strategy is $\mathcal{O}(|M| \cdot |R|)$.

Algorithm 4 shows the AuthCheck strategy to identify the authentication-critical methods in the call graph for *CWE-862*. When checking the *CWE-862*, each method M currently processed in the path is classified as authorization-critical if the method is contained in the *ACSM* as a resource and the access rule is not tautological. The evaluation of the authorization formula depends on the number of relevant authorization groups used in the authorization formula. For the calculation, all possible combinations $\forall\ g \in Pow(G')$ of relevant authorization groups G' must be evaluated. For a worst-case authorization formula with $|G|$ authorization groups, the resulting complexity is $\mathcal{O}(|M| \cdot |R| \cdot 2^{|G|})$.

Algorithm 5 shows the AuthCheck strategy to identify the authorization-critical methods in the call graph. For each resource in the *ACSM*, it checks whether its signature matches the signature of the method M currently processed in the call graph. In addition, the authorization formulas are checked. The runtime depends on the number of relevant authorization groups. For the calculation, all possible combinations $\forall\ g \in Pow(G')$ of relevant authorization groups G' must be evaluated. For a worst-case authorization formula with $|G|$ authorization groups, the resulting complexity is $\mathcal{O}(|M| \cdot |R| \cdot 2 \cdot 2^{|G|})$.

Algorithm 4 Identifying methods as authorization-critical

```
1: function isMethodAuthorizationCritical(R, s')
2:     for each r ∈ R do
3:         if r_s = s' and r_a is not tautological then
4:             return true
5:     return false
```

Algorithm 5 Identifying methods as authorization-critical and group-belonging

```
1: function isMethodAuthorizationCritical(R, s', a')
2:     for each r ∈ R do
3:         if r_s = s' and eval(r_a) = eval(a') then
4:             return true
5:     return false
```

5 Spring Security AuthCheck

We implemented the AuthCheck concept from Section 4, as a Java application that checks the implementation of a given Java Spring Security application and a given *ACSM*. We used the Soot framework [14] for the analysis. In the following, we discuss the architecture of our implementation, the insights of the call graph construction, and the four typical developer's mistakes with Spring Security that AuthCheck can detect. Our implementation is available on Github [18].

5.1 Architecture

The AuthCheck tool follows a pipeline architecture, since it consists of several sequential phases that work on shared artifacts. Our AuthCheck implementation consists of 3 phases:

1. *Call graph construction*: parses the code, the Spring Security configuration, and annotations, and constructs the call graph,
2. *Call graph update*: patches missing edges into the call graph based on Spring Security configuration,
3. *CWE analysis*: analyzes the call graph against *CWE-306*, *CWE-862*, and *CWE-863* based on Section 4.

The design an extendable architecure of the tool. Figure 3 shows the meta-model of the main components of the tool. The root class is the *Analysis* that contains all components. The *Phase* can process objects of type *Artifact*. In our implementation, the call graph instance, *FSMs*, and *ACSM* are defined as artifacts. The final results of the analysis are stored in a *Result* object which can be presented via *Presenter* object. Our tool has one presenter, that generates HTML pages (see Section 6 and Figure 4). In this architecture new phases can be added easily. Furthermore, new types of vulnerabilities can be created as *FSM* and added as artifacts in the analysis.

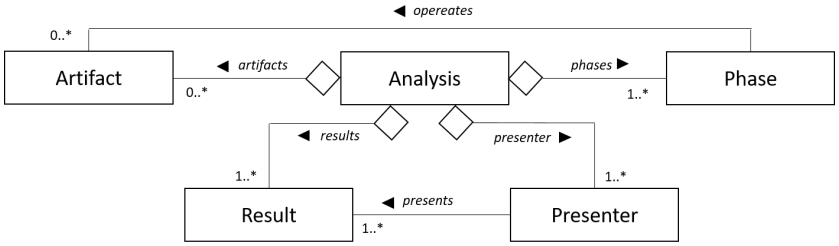


Fig. 3: UML class diagram of AuthCheck implementation for Spring Security

5.2 Call Graph Construction

Phase 1 constructs the call graph using the class hierarchy algorithm and extracts the Spring Security configuration needed in phase 2 to complete the missing edges in the call graph due to reflection. The extracted information is prepared according to Definition 3. Each critical method is annotated with its URI and HTTP method. This is transferred together with the signature of the method into a resource according to the Definition 2.

The Spring Security configuration is extracted from the program using an intraprocedural analysis. A special case is the method *access(a)*, which can take as an input a *SpEL* formula. For this, we use the Spring mechanism to evaluate the string values containg the *SpEL* formula.

An authorization formula is assigned to a resource when the defined filter matches the method and the URI. If multiple authorization formulas are applied to a resource, they are associated with a logical AND ($\wedge$).

The extracted information is stored as web application (Definition 3). Then, in phase 2, the missing edges are added to the call graph according to Algorithm 6. The algorithm gets the extracted web application W_J and generated call graph *CallGraph*. For each resource, it is checked whether the Spring Framework performs an authorization check, authentication check, or no access check. Accordingly, an edge is created to the critical method from the *authorize()*, *authenticate()*, or *init()* methods.

Algorithm 6 Adding missing edges in the call graph

```
1: function createMissingEdges(W_J = (R_J, G_J), CallGraph)
2:     for each r ∈ R_J do
3:         if isMethodAuthorizationCritical(R_J, r_sig) then
4:             CallGraph→addEdgeFromAuthorize(r_sig)
5:         else if isMethodAuthenticationCritical(R_J, r_sig) then
6:             CallGraph→addEdgeFromAuthenticate(r_sig)
7:         else
8:             CallGraph→addEdgeFromInit(r_sig)
```

5.3 Developers' mistakes

As demonstrated in Listing 1.1, the access-control rules in Spring Security are specified with the SpEL fluent interface. With this approach, we foresee two factors that can lead to inconsistencies of the implementation and the intended design. First, the developer should be familiar with the domain specific language SpEL in order to specify the *antMatchers* correctly, i.e. in the correct order. Second, the string values of some of the arguments are not parsed and automatically checked. Based on that, we identified 4 mistakes that developers can make when using Java Spring Security.

Missing or wrong authentication rule: The developer forgets to include the authentication filter *authenticated()* for the URI of a particular resource in the configuration or uses the filter *permitAll()* to incorrectly allow access to all users. However, in the specification model, the resource requires valid authentication. If no filter is specified, this is equivalent to the filter *permitAll()*. As a result, any user without authentication is able to request this resource. The error causes the security vulnerability missing authentication *CWE-306*.

Missing authorization rule: The developer forgets to include one of the authorization filters *hasRole(role)* or *access(rule)* for the URI of a particular resource. However, according to the *ACSM*, the resource requires a valid authorization. The filter *authenticated()* leads to the same error because it only checks the authentication of the user. Depending on the filter used, either all users or only authenticated users are able to request this resource. The error causes the security vulnerability of missing authorization *CWE-862*.

Incorrect authorization rule: The developer incorporates an authorization filter *hasRole(role)* or *access(rule)* for the URI of a certain resource, but a wrong authorization formula is used. As a result, a user without the required access rights is able to request this resource. The error causes the security problem of incorrect authorization *CWE-863*.

Method call with higher access rights: The developer creates a correct configuration for the resource, but in a deeper layer of the application, a call to a method is created that requires higher access rights and therefore should not be called by the user. The error causes the security problem of incorrect authorization *CWE-863*.

We implemented an extended version of the running example from Section 2 that includes the four mistakes and serves as a test scenario for our implementation. It is available under [18]. The tool generates a HTML page with all vulnerabilities detected. Figure 4 shows a detected *CWE-306* in our running example, including the path and description for solving the issue.

6 Case Study

We used the open-source project *FredBet*[8] to perform a case study that demonstrates the applicability of our analysis.

[8] https://github.com/fred4jupiter/fredbet

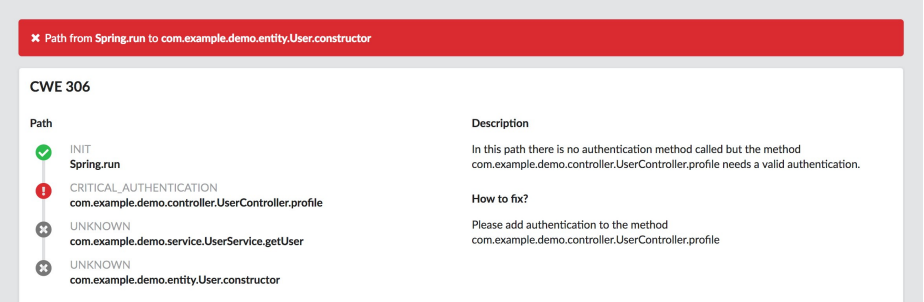


Fig. 4: Screenshot from AuthCheck generated output with *CWE-306*

6.1 FredBet

The web application *FredBet* is a football betting system developed with Java Spring Boot and Spring Security. FredBet offers the possibility to initiate an online football bet with several users. In addition to the betting, the web application offers statistics about the matches, rankings, a profile management, and many other functions. The application is actively developed since 2015 and as of July 2019, it's repository has more than 1300 commits.

Since we have access only to the implementation and no design specification is available from which we can infer an *ACSM*, we decided to create the *ACSM* based on the implementation and insert the four types of mistakes discussed in Subsection 5.3. We focused on the *AdminController* from FredBet and made code modifications. AuthCheck analyzed the *AdminController* and detected the inserted vulnerabilities.

6.2 Limitations

When applying AuthCheck to FredBet, we realized that the specification scope in Spring Security is much broader than the available documentation. This means that there are many ways to specify the same configuration information when one develops an application. For example, a developer can specify an URI for a given class containing critical methods. This URI is then concatenated to the URIs of the critical methods it contains. Then, the annotations can have different formats or even some can be skipped, like the HTTP method, which in such case, a default value *GET* will be considered by the framework. The configuration of the *antMathers* (see Listing 1.1) can have different parameters. Such broad scope of specification options, is currently not supported by the AuthCheck parser. Even though, this is a technical disadvantage, in order to prepare AuthCheck for more complex web applications, the parser needs to be extended.

7 Related Work

Security vulnerabilities caused by the misuses of access-control mechanisms have been investigated by Dalton et al. [6]. The approach examines access-control problems by analyzing the flow of user credentials within the web application. In contrast to AuthCheck, their approach is dynamic and can not be used for early detection of the vulnerabilities.

Sun et al. [21] introduced a static analyis approach for the detection of access-control vulnerabilities. They assume that the source code contains implicit documentation of intended accesses. From this, sitemaps for different authorization groups are generated and checked whether forced browsing can happen. Another static analysis specific for access-control of XML documents was introduced by Murata et al. [16]. They use XPath representation for the access-control rules and XQuery for specifying the requests. The analysis checks all paths defined by the query against the XPath rules. Naumovich et al. [17] proposed a static analysis for Java EE applications where the resources are security fields from the Java Beans objects.

In the area of model checking, few approaches address the access-control protocols [15] [22]. In these approaches, the focus is to validate the message communication of the defined protocols. Similarly, Alexander et al. applied model checking to verify the authentication mechanism in the comminication of a set of interacting virtual machines [4].

8 Conclusion and Future Work

Even though sophisticated Java web frameworks, such as Spring, provide secure mechanism for access control of resources, for many developers using the APIs and the configuration specifications correctly, can be challenging. Thus, these misuses may cause access-control vulnerabilities in the code. In this paper, we presented AuthCheck, a static analysis, that tracks the program-state to detect the vulnerabilities *CWE-306*, *CWE-862*, and *CWE-863*. Based on *finite state machine* specification of each vulnerability, AuthCheck checks each path. We implemented the approach on top of the Soot framework and applied it to one open-source project on which we detected four types of errors that were previously inserted in the existing application.

We plan to evaluate the precision of AuthCheck in cooperation with industry to overcome the problem of the open-source projects of not having a design specification on which we can check the implementation against. Additionally, in future the choice of the call graph algorithm should be evaluated.

References

1. Spring framework, java spring. `https://spring.io/projects`, online; accessed 9 March 2019

2. Spring framework, java spring security. https://spring.io/guides/topicals/spring-security-architecture, online; accessed 9 March 2019
3. Spring framework, spring expression language. https://docs.spring.io/spring/docs/5.0.5.RELEASE/spring-framework-reference/core.html, online; accessed 12 March 2019
4. Alexander, P., Pike, L., Loscocco, P., Coker, G.: Model checking distributed mandatory access control policies. ACM Trans. Inf. Syst. Secur. **18**(2), 6:1–6:25 (Jul 2015)
5. Ball, T., Rajamani, S.K.: The slam project: Debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN POPL. pp. 1–3. POPL '02, ACM, New York, NY, USA (2002)
6. Dalton, M., Kozyrakis, C., Zeldovich, N.: Nemesis: Preventing authentication and access control vulnerabilities in web applications. In: Proceedings of USENIX. pp. 267–282. SSYM'09, USENIX Association, Berkeley, CA, USA (2009)
7. Enumeration, C.C.W.: Incorrect authorization. https://cwe.mitre.org/data/definitions/863.html, accessed 12 March 2019
8. Enumeration, C.C.W.: Missing authentication for critical function. https://cwe.mitre.org/data/definitions/306.html, accessed 12 March 2019
9. Enumeration, C.C.W.: Missing authorization. https://cwe.mitre.org/data/definitions/862.html, accessed 12 March 2019
10. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis (2000), university of California, Irvine
11. Gamma, E., Vlissides, J., Johnson, R., Helm, R.: Design Patterns CD: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1998)
12. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. In: Proceedings of the 10th International Conference on Model Checking Software. pp. 235–239. SPIN'03, Springer-Verlag, Berlin, Heidelberg (2003)
13. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In: ECOOP. pp. 10:1–10:27 (2018)
14. Lam, P., Bodden, E., Lhotak, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011) (Oktober 2011)
15. Marrero, W., Clarke, E., Jha, S.: A model checker for authentication protocols. In: Rutgers University (1997)
16. Murata, M., Tozawa, A., Kudo, M., Hada, S.: Xml access control using static analysis. ACM Trans. Inf. Syst. Secur. **9**(3), 292–324 (Aug 2006)
17. Naumovich, G., Centonze, P.: Static analysis of role-based access control in j2ee applications. SIGSOFT Softw. Eng. Notes **29**(5), 1–10 (Sep 2004)
18. Petrasch, T., Piskachev, G., Spaeth, J., Bodden, E.: Authcheck spring implementation. https://github.com/secure-software-engineering/authcheck/, online
19. del Pilar Salas-ZÃąrate, M., Alor-HernÃąndez, G., Valencia-GarcÃıa, R., RodrÃıguez-Mazahua, L., RodrÃıguez-GonzÃąlez, A., Cuadrado, J.L.L.: Analyzing best practices on web development frameworks: The lift approach. Science of Computer Programming **102** (2015)
20. Strom, R.E.: Mechanisms for compile-time enforcement of security. In: Proceedings of the 10th ACM SIGPLAN POPL. pp. 276–284. ACM, New York, NY, USA (1983)
21. Sun, F., Xu, L., Su, Z.: Static detection of access control vulnerabilities in web applications. In: Proceedings of USENIX. USENIX Association, Berkeley, CA, USA (2011)

22. Xu, Y., Xie, X.: Modeling and analysis of authentication protocols using colored petri nets. In: Proceedings of the 3rd ASID. ASID'09, IEEE Press, Piscataway, NJ, USA (2009)

Leveraging Highly Automated Theorem Proving for Certification

Deni Raco, Bernhard Rumpe, and Sebastian Stüber

Software Engineering, RWTH Aachen University, Germany www.se-rwth.de

Abstract. This work demonstrates how highly automated theorem proving can be leveraged for sparing testing costs during certification of safety-critical software such as in avionics. A verification framework for distributed interactive systems is presented. Components are modeled as stream processing functions. The functional methodology is modular with respect to serial, parallel and feedback composition. To specify and formally verify properties of distributed systems, a stream-based verification infrastructure is encoded in the theorem prover Isabelle. Composition operators for components are provided, thus allowing to scale proofs from individual components to complex networks. The underlying mathematical theory FOCUS stands out among competitors by the fact that refinement is fully compositional. The associativity and commutativity of the provided general composition operator enables a compositional verification. In this paper the stream theory encoded in Isabelle is demonstrated. This represents a small part of our encoding of the methodology focusing only on channel histories specifications, yet sufficient to demonstrate dealing successfully with underspecification, supporting automatic refinement checking during design time, time-sensitive specification, as well as verifying safety and liveness properties. The theory is evaluated in a case study where the occurring refinement steps from system requirements, to high level requirements, to low level requirements, until an implementation is reached, are demonstrated to be proven correct at the push of a button.

Keywords: Formal Verification · Distributed Systems · Certification

1 Industrial Background

In a software development process, such as the one depicted in Fig.1 for avionics, checking the correctness of the refinement steps from system requirements, to high level requirements, to low level requirements, until the source code, can get very costly to execute by reviewing and testing[2]. Formal methods can provide help for this. Well-known classes of formal methods are[40]:

- Theorem Proving:
  - Most powerful, most expressive formal methods tools.
  - Require expertise and continuous interaction to successfully use.

- Model Checking:
  - Less expressive than theorem provers.
  - Mostly automated, but still require expertise to use successfully.
- Abstract Intepretation and Static Analysis [29, 7, 38, 22, 42]:
  - Least expressive, targeted to very specific artifacts.
  - Require some expertise to discharge false positives.

Theorem proving [5] can assure correctness, a confidence level in general not reachable by testing. Furthermore, it can result in sparing costs on testing (partially replaces it, partially complements it), which is known as the most costly part of the development of safety-critical systems [10, 2].

Using theorem proving, one can show e.g. that the high-level requirements are consistent (i.e. do not contradict each other) by proving that there is at least one implementation satisfying the high-level requirements. One can show that the system architecture and the high-level requirements of the components comply with the system requirements by proving that the system requirements are satisfied by the design instantiated with any components that satisfy the high-level requirements [5].

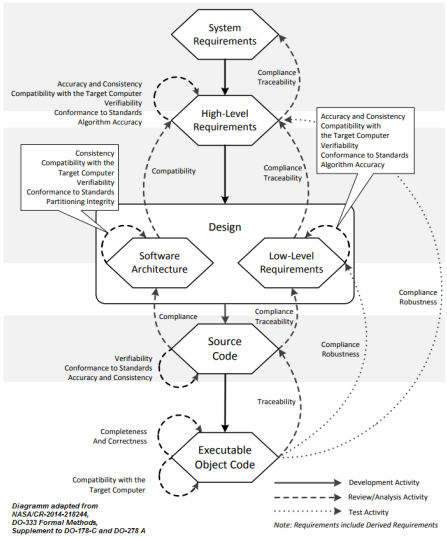


Fig. 1. Software Development Process as in DO-178C [5]

The usual challenges when using theorem proving are:

- Sound semantics of the underlying formal method?
- Degree of automation acceptable?
  - Significant expertise and user training necessary?
- Is the theorem proving tool qualified itself?

A set of 5 criteria as defined by Airbus for the use of formal methods is [41]:

- Soundness
- Cost Savings
- Analysis of unaltered programs
- Usability by normal software engineers on normal machines
- Ability to be integrated into the DO-178B conforming process

To deal with this requirements, the underlying methodology of this paper offers:

- Sound semantics from a well-estabilished stream-theory [19, 13, 34, 4, 8, 9, 3, 33]
  - Time-sensitivity, stateful modeling, nondeterminism and refinement supported
- User-friendly, yet sufficiently expressive modelling language [11]
  - Practicable for industrial application without need for theorem-proving skills
- Can be integrated into the conformance process of functional safety standards
  - Tool qualification of Isabelle [26] practicable due its axiomatic, conservative nature
- Scalability through compositional verification (allowed by commutativity and associativity of the general composition operator [19, 3])
  - Saves costs on testing (can partly replace it, and partly complement it)

The unique selling point of the stream-based formalism FOCUS [4] is:

- Refinement is fully compositional. Refinement of a subcomponent implies refinement of the overall composition.
- Properties of the old system hold directly for the new refined one aswell, thus sparing tests and integration costs after each refinement step.
- Verifying refinement of underspecification accompanies the design phase

The focus in this paper is in showing how the developed methodology handles typical development challenges in a software life cycle. The framework is thus demonstrated to deal successfully with important aspects of development of distributed systems such as underspecification, refinement and time-sensitive specifications, as well as safety and liveness properties. For this, refinement of specifications concerning streams allowed in a communication bus of the Flight Guidance System are presented. For the purpose of this paper, this is sufficient to demonstrate how theorem proving can be leveraged to spare costs in testing, achieved by enriching an encoded infrastructure sufficiently with theorems to ensure the proof for a refinement of a specification is found at the push of the button. For scaling the methodology over networks of components and a compositional verification of properties of the overall system at the push of a button, see [19] and https://www.youtube.com/watch?v=krl4Q7MAAlo for a short video demo.

In particular, the contributions of this paper consist of:

- The encoding of streams with domain-theoretical concepts in the theorem prover Isabelle
- an infrastructure for time-sensitive specifications
- a sufficient collection of functions and theorems over these structures to ensure that refinement proofs of the case study are performed at the push of a button without further user interaction
- refinement of requirements
- liveness property specification
- a case study consisting in checking refining requirements evaluate the encoded abstract theories

This paper does not go in depth in demonstrating details of our formalization of components as stream processing functions (see the previous work for a demonstration [19]) due to the limited space. This paper rather focuses in depth on one specific part of our contribution, namely in this case modeling channel histories of a distributed system, underspecification, and full automated refinement checking. This is a simple, yet sufficiently expressive example to demonstrate a run through all the development process phases when refining requirements such as those occurring in the guidance certification standards.

The rest of the paper consists in the following: The next section presents a short overview of the underlying theory. Subsequently, a formalization of a communication bus eg. in the Flight Guidance System is demonstrated, as well as the automatic refinement checking process from system requirements, to high-level requirements, low-level requirements, until the code.

2 Modular Hierarchical Methodology Providing Correctness by Design

Distributed systems in particular have proven to be more error-prone than sequential software [4, 27]. Their correct development has been a challenge in the past decades. To reduce ambiguities, formal methods have been proposed. Formal methods are known to be challenging in becoming practical in the industry due to their high costs [18]. Methodologies like CSP [15, 14], FOCUS [4], CCS [24], Petri Nets [31], or the π-calculus [25] are used to detect potential sources of errors earlier [12, 23, 1]. They support the correctness of the typical steps of development, which are usually structural decomposition and refinement. In order to spare integration costs, these two need to be compatible.

For the stepwise development of systems, the mathematical methodology FOCUS [4] is used in this work. The communication between components of a distributed system is modeled here by streams of messages flowing in unidirectional communication channels. The advantage that FOCUS has compared to the others is that refinement is indeed fully compositional. This means that one can decompose a system into components, refine those separately, and have the composed system be correct by construction. An example is shown in Figure 2.

To specify and formally verify properties of distributed systems, a part (the stream encoding) of a verification tool chain (Figure 3) is demonstrated in this

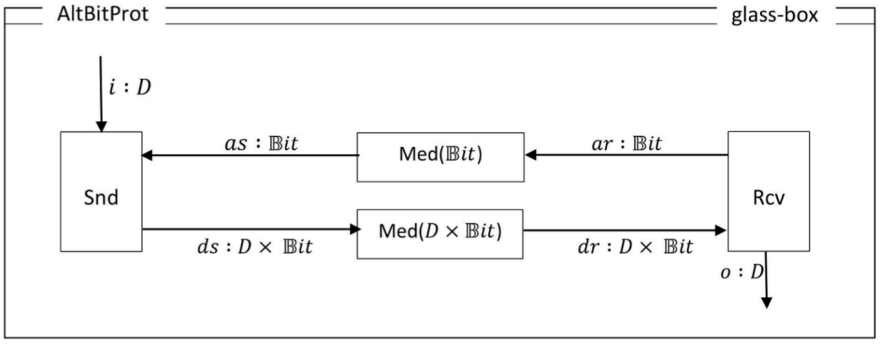


Fig. 2. Example of a Distributed System: The Alternating Bit Protocol [4]. For example the sender has an input as with messages from the alphabet $Bit := \{0, 1\}$.

paper. A developer of a distributed system is provided with an architecture description language (ADL) [11],[19] to specify in a user-friendly way the components and their interaction. This language also allows a developer to specify a desired property of the system. By the push of a button, the created system model is then transformed into an equivalent specification in the theorem prover Isabelle [26]. The property is transformed in a theorem and exterior solvers are called to prove or disprove the property. A more detailed overview is given in [19].

The ADL MontiArc [11] was created with MontiCore [16]. This language is used to describe component-and-connector architectures. The component behavior is described here by an automaton with input/output [34]. The user can specify the component interfaces, their behavior, their interaction, as well as the desired property of the system in the language OCL [32, 6, 35], which is embedded in the ADL. The ADL components are then translated into equivalent automata encoded in the theorem prover Isabelle. These automata are transformed within Isabelle into (sets of) stream processing functions, which constitute the semantics of the automata [34]. The desired property of the system written from the user in the ADL is translated into a theorem in Isabelle. Finally, general theorems over stream processing functions written in Isabelle support highly-automated property verification.

There are a couple of reasons for enforcing the specification of components by means of an ADL featuring automata, rather then giving the user just an encoding of the stream data type and set of theorems over streams in Isabelle and total freedom of specifying components over (tuples of) streams [8, 9].

First, the ADL is for a user more comfortable than writing recursive (specified typically as least fixed points [37]) stream processing functions in Isabelle or their composition [3].

Second, the automata with input and output of this methodology are designed to describe the *realizable* components (and only these) [34]. Realizability is reflected by the following properties called *monotonicity* and *continuity*.

Since streams model history, not every function is a model of a real life interactive component. After emitting a message, a component cannot take it back. Thus an extension of the input can only lead to the extension of the output. This property is called *monotonicity*. This property is also needed to guarantee the existence of least fixed points [34] to give meaning to (streams flowing in) feedback loops [33].

Also, a component cannot react to infinity, thus one cannot look at the infinite input stream to produce an output. This property is known as *continuity* [34]. This property is also needed not only to guarantee the existence, but also to calculate the least fixed point by approximation from the finite prefixes of a stream.

Allowing a user to specify any kind of function from streams to streams would also mean to expect the user to write a proof that the function is *realizable*. Since this would be not practical, a better way would be to provide the user a language (the ADL with automata of this paper) by which only realizable functions can be specified. The proof that the (sets of) stream processing functions, which are obtained from the semantics of (non-deterministic) automata, are realizable [34], is one of the important general theorems, which is encoded in the theorem prover Isabelle.

Third, the automata of this methodology are general enough to represent every realizable stream processing function [34].

Proper composition operators such as serial, parallel, feedback, and a general operator combining all these, are encoded to support a modular modeling. The composition of realizable components by these operators results in a realizable composed component [34].

Furthermore, a proper specification methodology should be able to describe underspecification. This may arise due to lack of information during the life cycle of a development process or due to non-deterministic behavior during run time. In this methodology this is reflected by non-deterministic automata, which have as semantics a set of stream processing functions.

Behavioral refinement is an important development step performed by decreasing underspecification through elimination of alternatives and thus making a specification more precise.

Refinement of the behavior of an automata is reflected by set-inclusion of the (sets of) stream processing functions representing their semantics [34]. Thus, the properties of the refined system are derived in this methodology per construction from the proven properties of the previous one, thus sparing testing and integration costs on the new system.

Refinement being fully compositional guarantees that after decomposing the system, refining the components separately, and then composing back, the system does not gain new behaviors.

A specification should also able to handle timing information. An airbag controller for instance is highly time-dependent. The specification framework presented in this work is able to represent timing. One way to model time is to extend the alphabet of messages of a stream with a dummy element $\sim$ (read: eps, see next section for a formal description in Isabelle). By assuming a discrete global clock, each time unit in a stream consists either in a message, or an $\sim$ used to model the absence of messages. Using timed automata with input and output, one can represent in this methodology the desired timed stream processing functions. The realizability properties in the timed case (such as e.g. it is invalid to look in the future of the input when producing the actual output message) are equivalently represented by the concepts of *weak causality* [34], as well as the stronger version by adding delay and thus dealing with feedback loops called *strong causality* [34].

In this methodology, a code generator from the ADL MontiArc to Isabelle generates specifications of the components, their composition, and helper theorems to support verification. A library of theorems encoded in the theorem prover Isabelle leads to high automation of the proofs.

Some of the mathematical fundaments of the concepts above have been formalized in theorem provers, such as HOLCF in [17]. The works [36, 8, 9] formalized streams in the theorem prover Isabelle with domain-theoretical concepts, and constitute a foundation of this paper. Furthermore tools such as AutoFOCUS [39] use stream-based semantics to model distributed systems. Also, the Ptolemy Project [20, 21] tackles the challenge to formalize component networks. In the Isabelle-formalization in this paper a high-level API is provided to hide fixed-point theoretical concepts from the developer.

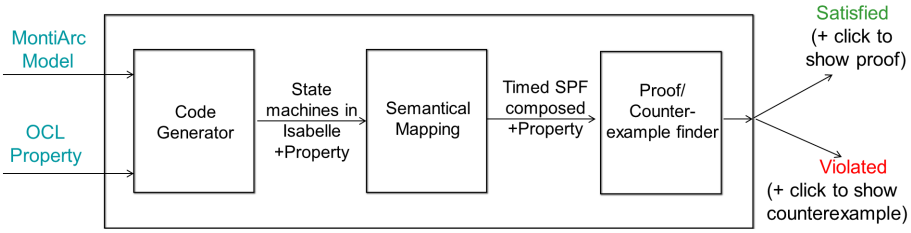


Fig. 3. Tool Chain

3 Verification of Distributed Software Development Steps

3.1 Algebra of Stream Processing Functions

FOCUS [4] and its main construct *stream* consitute the mathematical underpinning of the methodology. An (untimed) *stream* is a (potentially infinite) sequence

of messages over a set M. M^ω is the set of all streams and constitutes the union of finite streams M^* and infinite ones M^∞. Similar to lists, a constructor " : " with signature $M \Rightarrow M^\omega \Rightarrow M^\omega$ is used to create streams by appending an element to an existing stream.

To specify time-sensitive behavior, a variant of timed streams (so-called *time-synchronous streams*) is used. The message set is extended by an element $\sim$ (read *eps*). As mentioned, it is interpreted as "no messages arrived in that timeslot". A discrete global clock is assumed. An element of the stream is then a message or an $\sim$ (which has the length of one time frame as well). This way one can react to the absence of messages.

The concatenation of two streams is denoted by $\frown$. A prefix ordering $\sqsubseteq$ is defined on the set of streams to approximate infinite streams:

$$\forall x, y \in M^\omega.\ x \sqsubseteq y \Leftrightarrow \exists s \in M^\omega.\ \ x \frown s = y$$

M^ω forms a complete partial order [34].

3.2 Abstract Theories in Isabelle

An important goal of the paper is to demonstrate the verification of requirements refinement. The theorem prover Isabelle [26] is a generic system for implementing formalisms in higher-order logic. Informally HOL can be described with the equation

$$\text{HOL} = \text{Functional Programming} + \text{Logic}.$$

An Isabelle-Theory consists of data type definitions, functions and proofs [19, 28]:

```
theory ExampleTheory
imports Main
begin
(* definitions and lemmas *)
end
```

Please note that the proofs of the abstract lemmas below are omitted for space reasons, whereas the proof of the case study lemmas is fully demonstrated (and consists in applying the abstract lemmas of the theory). The logic of computable functions (HOLCF) has been encoded by Regensburger [30] and Huffman [17]. Based on this, the data type of natural numbers extended with the infinite element is encoded using the `domain` command [17]:

```
domain lnat = lnsuc (lazy lnpred::lnat)
```

lnsuc is a constructor and its inverse is lnpred. The domain command makes sure the type `lnat` is an element of the classes zero and ord by generating a bottom element and an order relation.

```
instantiation lnat :: "{ord, zero}"
begin
  definition lnzero_def: "(0::lnat) ≡ ⊥"
  definition lnless_def: "(m::lnat) < n ≡ m ⊑ n ∧ m ≠ n"
  definition lnle_def:   "(m::lnat) ≤ n ≡ m ⊑ n"
instance ..
end
```

A finite natural number `n` is represented by `Fin n`, where lntake is a function which retrieves a specified number of elements of the recursively constructed data type lnat.

```
definition Fin::"nat ⇒ lnat" where
"Fin k ≡ lntake k·∞"
```

The dot after the letter k above is an abbreviation used when a continuous function is applied [17].

Then, the infinite element is encoded:

∞ is the maximum of all `lnat`s

```
definition Inf'::"lnat" ("∞") where
"Inf' ≡ fix·lnsuc"
```

`Fix` denotes the least fixed point operator calculated as Kleene-Approximation. A couple of theorems for increasing automation follows:

Bottom element of lnat is 0:

```
lemma bot_is_0: "(⊥::lnat) = 0"
```

0 is not equal ∞:

```
lemma Inf'_neq_0[simp]: "0 ≠ ∞"
```

∞ is a fixed point of `lnsuc`:

```
lemma fold_inf[simp]: "lnsuc·∞ = ∞"
```

The brackets with content [simp] ensure that this rewriting rule is integrated in the core of Isabelle and the lemma does not need to be explicitly called by name during a proof. The preparations are done to be ready to encode the stream data type. The implementation is similar to that of lazy lists in Haskell, and apart from little technical details, looks as following, where 'a is a type parameter abstracting from the sort of messages:

```
domain 'a stream = lsconc "'a" (lazy "'a stream")
```

`domain` [17] creates a new (potentially recursive) data type enhanced with an order (in this case the prefix order) and a bottom element (here the empty stream). `lsconc` is the name of the constructor that appends an element to the rest of the stream.

The data type can be extended into time-synchronous streams by using the constructor Msg with arity 1 and eps for "no data" (abbreviated as $\sim$).

```
datatype 'a tsyn = Msg 'a | ∼
```

For instance the following time-synchronous stream over natural numbers $<[Msg\ 3, \sim, Msg\ 5, \sim, \sim, ...]>$ is interpreted as: message 3 arrives in the first time slot, no message arrives in the second time slot, message 5 arrives in the third time slot, and then no message arrives anymore. Depending on the application to be modeled, the granularity (duration of a time slot) can be interpreted at will (a time slot can correspond to 1 millisecond, or perhaps to 1 minute).

Now a couple of functions, abbreviations, as well as lemmas over streams follow.

The empty stream is denoted as ϵ.

```
abbreviation sbot :: "'a stream"  ("ϵ")
where "sbot ≡ ⊥"
```

`sup'` is used to construct a stream by a single element.

```
definition sup':: "'a ⇒ 'a stream" ("↑_" [1000] 999) where
"sup' a ≡ updis a && ϵ"
```

The `updis` command above lifts an arbitrary type to a discrete pointed partial order [17].

`sdom` retrieves the set of all values in a stream and `snth` is used to get the n-th element of a stream.

```
definition sdom:: "'a stream → 'a set" where
"sdom ≡ Λ s. {snth n s | n. Fin n < #s}"
```

`slen` retrieves the length of a stream. It is defined as the number of its elements or ∞ for inifinite streams.

```
definition slen:: "'a stream → lnat" where
"slen ≡ fix·(Λ h. strictify·(Λ s. lnsuc·(h·(srt·s))))"
```

The command `strictify` above turns a function into a strict one [17].

UNIV denotes the set of all elements in a data type.

The function `slookahd` applies a function to the head of stream. If the stream is empty, $\perp$ (the polymorphic parameter 'a of streams has also an order defined in it and has also a least element) is returned. This function is especially useful for defining own stream-processing functions.

```
definition slookahd:: "'a stream → ('a ⇒ 'b) →
    ('b::pcpo)" where
"slookahd ≡ Λ s f. if s = ϵ then ⊥ else f (shd s)"
```

Hereby `shd` returns the head of a stream.

`sfilter` removes all elements from the stream which are not included in the given set.

```
definition sfilter:: "'a set ⇒'a stream →'a stream" where
"sfilter M ≡ fix·(Λ h s. slookahd·s·(λ a.
```

```
    (if (a ∈ M) then ↑a • (h·(srt·s)) else h·(srt·s))))"
```

Type classes, such as those in Haskell, are supported: `class 'a::countable` means that the type 'a is restricted to belong in the class of countable types, where there exists an injective mapping from the type 'a into the natural numbers.

```
class countable =
  assumes ex_inj: "∃to_nat :: 'a ⇒ nat. inj to_nat"
```

`sinftimes` concatenate a stream infinitely often to itself (an abbreviation in form of a suffix is introduced in brackets).

```
definition sinftimes  :: "'a stream ⇒ 'a stream" ("_∞")
    where
"sinftimes ≡ fix·(Λ h. (λs.
  if s = ϵ then ϵ else (s • (h s))))"
```

Finally, a collection of lemmas over streams to improve high automation:
Only the empty stream has length zero:

```
lemma only_empty_has_length_0 : "#s ≠ 0⟹s ≠ ϵ"
```

Filtering with a superset of the stream's domain does not change the stream:

```
lemma sfilter_sdoml3:
  "sdom·s ⊆ X⟶sfilter X·s = s"
```

A couple of connections between sfilter and sdom (notice the infix abbreviation of the `sfilter` function):

```
lemma sfilter_bot_dom: "(A ⊖ s) = ⊥⟹sdom·s ⊆ UNIV - A"

lemma sdom_sfilter1: assumes "x∈sdom·(A⊖s)"
shows "x∈A"

lemma sfilterEq2sdom_h: "sfilter A·s = s⟶sdom·s ⊆ A"

lemma sfilterEq2sdom_h: "sfilter A·s = s⟶sdom·s ⊆ A"
```

If the head of a stream is in M, then `sfilter` will not drop the head of the stream:

```
lemma sfilter_in[simp]:
"a ∈ M⟹sfilter M·(↑a • s) = ↑a • sfilter M·s"
```

If the stream is not empty, then the following holds for the length:

```
lemma srt_decrements_length : "s ≠ ϵ⟹#s =
    lnsuc·(#(srt·s))"
```

If x isn't empty then concatenating head and rest leaves the stream unchanged:

```
lemma surj_scons: "x≠ϵ⟹↑(shd x) • (srt·x) = x"
```

If filtering everything except z from the stream x doesn't produce the empty stream, then z must be an element of the domain of x:

```
lemma sfilter2dom:
  "sfilter {z}·x ≠ ϵ⟹z ∈ sdom·x"
```

Mapping a stream to head and rest is injective:

```
lemma inject_scons: "↑a • s1 = ↑b • s2⟹a = b ∧ s1 = s2"
```

If the head of a stream is in M, then `sfilter` will keep the head of the stream:

```
lemma sfilter_in[simp]:
"a ∈ M⟹sfilter M·(↑a • s) = ↑a • sfilter M·s"
```

After filtering by filter `T`, the head of the result is in `T`:

```
lemma sfilter_ne_resup: "sfilter T·s ≠ ϵ⟹shd (sfilter
   T·s) ∈ T"
```

A relevant connection between `sfilter` and `sinftimes`:

```
lemma sfilter_sinftimes_in[simp]:
"sfilter {a}·(sinftimes (↑a)) = sinftimes (↑a)"
```

Repeating a stream infinitely often is equivalent to repeating it once and then again infinitely often:

```
lemma sinftimes_unfold: "sinftimes s = s • sinftimes s"
```

Prepending a singleton stream increases the length by 1:

```
lemma slen_scons[simp]: "#(↑a•as) = lnsuc·(#as)"
```

For nonempty `s`, `sinftimes s` is infinite:

```
lemma slen_sinftimes: "s ≠ ϵ⟹#(sinftimes s) = ∞"
```

Finally, infinitely cycling the empty stream produces the empty stream again:

```
lemma strict_icycle[simp]: "sinftimes ϵ = ϵ"
```

The neccessary structures were introduced, so that the case study can now be specified and verified.

3.3 Case Study in Isabelle: Automatic Checking of Requirements Refinement

A bus in the Flight Guidance System is modeled and a specification indicates which streams are allowed to flow in it. The system requirement in this case is a set of streams fulfilling a fairness property, which is required to guarantee that a communication protocol using this bus works correctly (the protocol is not further specified, but such fairness requirements are not unusual; notice for example that the Alternating Bit Protocol [4] is only correct under the assumption of a fair medium).

By the following system requirement (SysReq), the only stream histories of messages allowed to flow in the bus are only those such that, after waiting an infinite amount of time, the number of actual proper messages occurring in the bus is infinite (thus the "no-data" symbol eps can occur only a finite amount of time).

```
definition SysReq :: "(nat tsyn stream) set" where
"SysReq ≡ {s. #((UNIV-{∼})⊖s) = ∞}"
```

To guarantee the SysReq, the developed high level requirements can look as follows: The length of the stream should be infinite and every element of the stream is an actual message, starting from the second one. The data type can be abstracted to any countable one (typical for high level requirements; fixed only after reaching low level requirements), in the sense that, if a property holds for any arbitrary countable type, then it also holds on natural numbers of (as wished in SysReq). The decision, whether the first element is a proper message, or an eps, is postponed for a point in time where the overall architecture of the system is known. If this bus is eg. to be embedded in connection with a feedback loop, then the first element being an "no-data" can act as a delay of 1 unit and thus make sure that the semantics of the stream flowing in the feedback loop is uniquely defined. If not, then this "delay" shall not be needed.

```
definition HLR :: "(('a::countable) tsyn stream) set"
    where
"HLR ≡ {s. #s=∞ ∧ sdom·(srt·s) ⊆ UNIV-{∼}}"
```

The refinement checking is formulated as a subset relation between HLR and SysReq and the encoding of sufficient abstract theorems ensures that the proof is found at the push of a button without the need of user interaction:

```
lemma "HLR ⊆ SysReq"
  by (smt HLR_def SysReq_def Collect_mono mem_Collect_eq
      Inf'_neq_0 fold_inf lnat.sel_rews(2)
       sfilter_sdoml3 sfilter_bot_dom sfilter_in
           sfilter_nin slen_empty_eq srt_decrements_length
       surj_scons sfilter2dom sdom_def)
```

Next, low level requirements shall be specified, namely the underspecification is reduced eg. after determining that there is no feedback loops in the architec-

ture, and thus no necessity for introducing a delay. So the first message is fixed, leading to LLR (yet a deterministic implementation is not reached yet):

```
definition LLR :: "(nat tsyn stream) set" where
"LLR = {s. #s=∞ ∧ sdom·s ⊆ UNIV-{∼}}"
```

The correctness of the refinement is checked again fully automatically:

```
lemma "LLR ⊆ HLR"
  by (smt HLR_def LLR_def DiffD2 sdom_sfilter1
      sfilter_sdoml3 sfilter_srtdwl3 singletonI surj_scons
  Inf'_neq_0 inject_scons sfilterEq2sdom_h sfilter_in
      sfilter_ne_resup slen_empty_eq Collect_mono)
```

An interesting question concerning requirements is usually whether they are consistent, i.e. is there at least an implementation fulfilling these?

Finally, an implementation is chosen, namely the infinite stream consisting of only the number one:

```
definition Code :: "nat tsyn stream" where
"Code = ↑(Msg 1)∞"
```

The consistency of LLR is shown fully automatically by proving that the Code-implementation above is an element of the LLR set.

```
lemma "Code ∈ LLR"
by (smt LLR_def Code_def Diff_UNIV Diff_empty
    Diff_eq_empty_iff bot_is_0 insert_iff tsyn.distinct(1)
    mem_Collect_eq only_empty_has_length_0 sfilterEq2sdom
        sfilter_sinftimes_in sinftimes_unfold
    slen_scons slen_sinftimes strict_icycle
        subset_Diff_insert subset_singleton_iff
        lnat.con_rews)
```

In conclusion, theorem proving can be leveraged to spare costs for certification activities by enriching the corresponding encodings with a large number of general abstract theorems. For scaling up to networks of components, a code generator can help by generating helpful theorems about the model and their proof. The code generator mapping a modeling language into a formal language generally needs to be qualified. On the other hand, the qualification of the generated proofs does not pose a threat though, since the proofs will finally be checked by the theorem prover Isabelle and the qualification of Isabelle is not hard due to its axiomatic and conservative nature.

References

1. Akroun, L., Salaün, G.: Automated verification of automata communicating via fifo and bag buffers. Formal Methods in System Design **52**(3), 260–276 (2018)
2. Brahmi, A., Delmas, D., Essoussi, M.H., Randimbivololona, F., Atki, A., Marie, T.: Formalise to automate: deployment of a safe and cost-efficient process for

avionics software. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018). Toulouse, France (Jan 2018), https://hal.archives-ouvertes.fr/hal-01708332
3. Broy, M., Rumpe, B.: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Informatik Spektrum **30**(1), 3–18 (2007)
4. Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Verlag Heidelberg (2001)
5. Cofer, D.D., Miller, S.P.: Do-333 certification case studies. In: NASA Formal Methods (2014)
6. Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., Wills, A.C.: The Amsterdam Manifesto on OCL. In: Object Modeling with the OCL, pp. 115–149, LNCS 2263. Springer Verlag, Berlin (2002)
7. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: International Workshop on Formal Methods for Industrial Critical Systems. pp. 53–69. Springer (2009)
8. Gajanovic, B., Rumpe, B.: Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen. Informatik-Bericht 2006-03, Technische Universität Braunschweig, Carl-Friedrich-Gauss-Fakultät für Mathematik und Informatik (2006)
9. Gajanovic, B., Rumpe, B.: Alice: An advanced logic for interactive component engineering. In: 4th International Verification Workshop (Verify'07). Bremen (2007)
10. Gigante, G., Pascarella, D.: Formal methods in avionic software certification: The do-178c perspective. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies. pp. 205–215. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
11. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural modeling of interactive distributed and cyber-physical systems, Technical report / Department of Computer Science, RWTH Aachen, vol. 2012,3. RWTH and Technische Informationsbibliothek u. Universitätsbibliothek and Niedersächische Staats- und Universitätsbibliothek, Aachen and Hannover and Göttingen (2012)
12. Hall, A.: Seven myths of formal methods. IEEE Software **7**(5), 11–19 (Sep 1990). https://doi.org/10.1109/52.57887
13. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of" semantics"? Computer **37**(10), 64–72 (2004)
14. Heim, R., Nazari, P.M.S., Ringert, J.O., Rumpe, B., Wortmann, A.: Modeling robot and world interfaces for reusable tasks. In: Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on. pp. 1793–1798. IEEE (2015)
15. Hoare, C.A.R.: Communicating sequential processes. In: The origin of concurrent programming, pp. 413–443. Springer (1978)
16. Hölldobler, K., Rumpe, B.: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32, Shaker Verlag (December 2017)
17. Huffman, B.C.: HOLCF '11: A definitional domain theory for verifying functional programs. Portland State University, [Portland, Or.] (2012)
18. Kasauli, R., Knauss, E., Kanagwa, B., Nilsson, A., Calikli, G.: Safety-critical systems and agile development: A mapping study. pp. 470–477 (08 2018)
19. Kriebel, S., Raco, D., Rumpe, B., Stüber, S.: Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In: Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19).

CEUR Workshop Proceedings, vol. 2308, pp. 87–94. CEUR-WS.org (February 2019)
20. Lee, E.A.: Computing needs time. Communications of the ACM **52**(5), 70–79 (May 2009)
21. Lee, E.A.: Fundamental limits of cyber-physical systems modeling. ACM Transactions on Cyber-Physical Systems **1**(1) (11 2016), http://chess.eecs.berkeley.edu/pubs/1183.html
22. Li, Y., Tan, T., Xue, J.: Effective soundness-guided reflection analysis. In: Blazy, S., Jensen, T. (eds.) Static Analysis. pp. 162–180. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
23. Maoz, S., Pomerantz, N., Ringert, J.O., Shalom, R.: Why is my component and connector views specification unsatisfiable? pp. 134–144 (2017)
24. Milner, R.: Communication and concurrency, vol. 84. Prentice hall New York etc. (1989)
25. Milner, R.: Communicating and mobile systems: the pi calculus. Cambridge university press (1999)
26. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A proof assistant for Higher-Order Logic, Lecture notes in artificial intelligence, vol. 2283. Springer, Berlin [etc.] (2002)
27. Olveczky, P.C.: Designing Reliable Distributed Systems. Springer-Verlag London (2017)
28. Paulson, T.N.L.C., Wenzel, M.: A proof assistant for higher-order logic (2013)
29. Payet, E., Spoto, F.: Checking Array Bounds by Abstract Interpretation and Symbolic Expressions, pp. 706–722 (06 2018)
30. Regensburger, F.: HOLCF: Eine konservative Erweiterung von HOL um LCF. na (1994)
31. Reisig, W.: Petri nets: an introduction, vol. 4. Springer Science & Business Media (2012)
32. Richters, M., Gogolla, M.: On formalizing the uml object constraint language ocl. In: International Conference on Conceptual Modeling. pp. 449–464. Springer (1998)
33. Ringert, J.O., Rumpe, B.: A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. International Journal of Software and Informatics **5**(1-2), 29–53 (July 2011)
34. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Doktorarbeit, Technische Universität München (1996)
35. Rumpe, B.: Modellierung mit UML, vol. 2nd Edition. Springer (2011)
36. Spichkova, M.: Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle. VDM Verlag Dr. Müller Aktiengesellschaft & Co. KG, Saarbrücken (2008)
37. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific journal of Mathematics **5**(2), 285–309 (1955)
38. Urban, C., Ueltschi, S., Müller, P.: Abstract interpretation of ctl properties. In: Podelski, A. (ed.) Static Analysis. pp. 402–422. Springer International Publishing, Cham (2018)
39. Voss, S., Zverlov, S.: Design space exploration in autofocus3 - an overview. In: Mařík, V., Lastra, J.M., Skobelev, P. (eds.) IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems. Springer (2014)
40. Wagner, L.: Formal methods for certification: Why and how? Safe & Secure Systems and Software Symposium (S5) URL: http://www.mys5.org/Proceedings/2016/Day_3/2016-S5-Day3_1505_Wagner.pdf (2016), accessed on 19.07.2019

41. Wiels, V.: Formal methods in aerospace: Constraints, assets and challenges. URL: https://richmodels.epfl.ch/_media/madrid13-slides-virginie-wiels.pdf, accessed on 19.07.2019
42. Yuki, T., Feautrier, P., Rajopadhye, S., Saraswat, V.: Array dataflow analysis for polyhedral x10 programs. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 23–34. PPoPP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2442516.2442520, http://doi.acm.org/10.1145/2442516.2442520