



Automatic Data Dependence Analysis by Deductive Verification

Asmae Heydari Tabar¹ · Richard Bubel² · Reiner Hähnle²

Received: 22 September 2025 / Accepted: 21 September 2026
© The Author(s) 2026

Abstract

Parallelization of programs is of essence in the high-performance computing (HPC) area. But parallelization is only correct if certain data dependences can be reliably excluded, for example, read-after-write dependences, where a read access occurs as the subsequent access on a data element after a write access. Data dependence analyses should not merely be correct, but as precise as possible, so as not to miss any opportunity for parallelization. Various dynamic and static analyses were proposed in the HPC community, but, until recently, none based on program logic and deductive verification, even though the latter approach has important advantages, including soundness, precision, and modularity. Previously a proof-of-concept was given that fully automatic dependence analysis based on deductive verification is possible, but it suffered from limitations that rendered it impractical: the restriction to one-dimensional arrays and flat loops and, importantly, it does not differentiate between inter- and intra-iteration data dependences, which often results in a loss of precision. In this paper we lift all three limitations and show that a deductive approach is a competitive alternative to the kinds of dependence analyses used hitherto in HPC. Our axiomatization in a deduction system constitutes the first formalization of the semantics of inter-iteration dependence.

Keywords Data dependence analysis · Deductive verification · Invariant generation · Parallel computation · Priming

1 Introduction

The context of this work is parallelization of programs in high-performance computing (HPC). Typical HPC programs manipulate very large arrays and matrices. To make use

✉ Asmae Heydari Tabar
asmae.heydari-tabar@kit.edu

✉ Richard Bubel
richard.bubel@tu-darmstadt.de

✉ Reiner Hähnle
reiner.haehnle@tu-darmstadt.de

¹ Fakultät für Informatik, Karlsruher Institut für Technologie, Am Fasanengarten 5, 76131 Karlsruhe, Baden-Württemberg, Germany

² Fachbereich Informatik, Technische Universität Darmstadt, Hochschulstrasse 10, 64289 Darmstadt, Hesse, Germany

of supercomputers, it is of essence that such computations can be performed in parallel. To this end, the HPC community developed parallelization patterns [33, 35, 36] that permit parallelization of programs via directives expressed in annotation languages such as OpenMP (www.openmp.org). However, these parallelization patterns are only correct (i.e., behavior preserving) if certain data dependences¹ can be guaranteed to be absent [33, 38]. Such data dependences express potential conflicts between subsequent accesses of the same data element. For example, if a is a two-dimensional array and i, j are index variables, then $\text{noRaW}(a[i][j])$ expresses the property that no read access on $a[i][j]$ took place as the subsequent access after any write access.

Data dependence analyses should not merely be correct, but as *precise* as possible, so as not to miss any opportunity for parallelization. The HPC community developed a variety of dependence analyses that can be classified into dynamic [39, 40, 44, 48, 66, 71] and static [14, 28, 43, 46, 56, 57]. Recently, also hybrid approaches were suggested [47]. Dynamic dependence analysis is very time-consuming, because a whole instrumented program must be executed many times under realistic load. Moreover, no guarantees with respect to either soundness or precision can be made. Static dependence analysis, on the other hand, can guarantee soundness, but tends to be imprecise, because it rests on syntactic criteria and is not value-sensitive [4].

It is a natural question, therefore, *to investigate whether dependence analysis based on a precise deductive approach is able to improve precision, while retaining correctness and full automation*. To do this, and to provide an affirmative answer, is the objective of the present paper. It may seem surprising that deductive reasoning had not been tried before to analyze data dependence, however, formal verification as an analysis technique is virtually unknown in the HPC community and, vice versa, data dependence as an interesting property is unknown in the verification community.

Deductive verification [29] as a basis for data dependence analysis has a number of potential advantages: (i) being based on a program logic for the programs under analysis, it is provenly sound; (ii) deductive verification has full path coverage and it is value-sensitive, so full precision is achievable in principle; (iii) on the tool side, one benefits from the substantial progress in deductive verification technology and SMT solving; (iv) deductive verification is by necessity procedure-modular, so there is no need to analyze a whole program, but one can focus on the—typically compact—computation-intensive parts.

The crucial question is: Does dependence analysis based on deductive verification scale to typical HPC programs? In particular, can it be made fully automatic? This is far from obvious: While deductive verification is very powerful and it was possible to prove correctness of highly complex code (for example, [19, 62] and many other case studies in [29]), it typically involves to supply invariants and auxiliary specifications manually by formal verification experts. Obviously, this is a show stopper for usage in the HPC realm. As will be detailed below, we rely on two crucial ingredients to achieve automation: a domain-specific language with limited expressiveness in which dependence relations are stated, plus a tailored abstract interpretation [17]. Together, these permit to express dependence relations schematically and to derive them automatically by predicate abstraction, thus eliminating the need to supply annotations manually.

In a previous paper [68] we provide a *proof-of-concept* that fully automatic dependence analysis based on deductive verification of imperative programs with loops over arrays is principally possible. It has, however, serious limitations that prevent the approach from being usable in *practice*: (i) It works only on one-dimensional arrays, whereas typical HPC

¹ Not to be confused with much simpler *data flow* properties that are commonly analyzed by static checking.

applications involve *multi-dimensional* matrices; (ii) it is limited to simple loops, whereas HPC programs typically contain *nested* loops; (iii) most importantly, it can analyze merely the dependences within a single loop iteration and loses precision when analyzing dependences across subsequent loop iterations. The reason for the latter is that loop iterations executed in parallel cannot interfere with each other, therefore, treating a loop as if it were executed sequentially introduces spurious dependences.

In the present paper, we lift all of these restrictions and show experimentally that fully automatic dependence analysis based on deductive verification of *realistic* HPC programs is not only possible, but competitive with state-of-art tools in the HPC community. In consequence, our work establishes deductive verification as a new, viable alternative in the portfolio of dependence analysis tools.

Of the three contributions mentioned above, taking into account inter-iteration dependence is by far the most challenging. It requires a substantial extension of the program logic in [68]. The resulting axiomatization is, to our best knowledge, the first formal semantics of inter-iteration dependence. Operationally, the system uses a combination of symbolic execution (to analyze values and paths) and predicate abstraction (to compute loop invariants). This combination is the key reason why deductive verification can be used in a fully automatic mode in the dependence analysis setting. However, it necessitates shifting between a weakest precondition and a strongest postcondition representation of intermediate symbolic states after symbolic execution of each loop body. The challenge was to discover a suitable language that permits inter-iteration shifting and still renders itself to automation.

The paper is organized as follows: In Sect. 2 we review all technical concepts needed as a background for the remaining paper. Section 3 is the central contribution: it defines a program logic, a formal semantics, and a calculus for reasoning about inter-iteration dependences. In Sect. 4 we explain how the approach can be lifted to multi-dimensional arrays and nested loops. Section 5 reports on our experiments with a standard benchmark suite used in the HPC community. We discuss related work in Sect. 6 and conclude in Sect. 7.

Sections 3–5 have not been published before, while Sect. 2 has been substantially rewritten and Sect. 6 was considerably extended compared to the account in [68].

2 Background

We introduce concepts and notions that form the starting point of this paper. The account is partially based on [15, 68] and uses additional material from [1, 69].

Our approach has been implemented for nearly full sequential Java. For ease of presentation, we explain it along a simpler and smaller language called Java_{HPC} which includes only the primitive types **boolean** and **int** as well as a one- and two-dimensional array type of element type **int**. Arrays are reference types in Java and not scalar. All programs have the usual Java semantics.

2.1 Syntax

As a program logic to specify and verify dependences in Java_{HPC} programs, we use the dynamic logic JavaDL_{Dep} of [15, 68]. JavaDL_{Dep} extends typed first-order logic with a binary program modality $[\cdot]$. The formula $[s]\phi$ (read “box $s\phi$ ”) expresses: *If* the execution of statement s terminates, *then* formula ϕ holds in the reached final state. JavaDL_{Dep} subsumes

$$\begin{aligned}
\langle term \rangle &::= \dots \mid \{ \langle update \rangle \}^* \langle term \rangle & \langle fml \rangle &::= \dots \mid [\langle stmtnt \rangle]^* \langle fml \rangle \mid \{ \langle update \rangle \}^* \langle fml \rangle \\
\langle update \rangle &::= x \text{ '}' \text{ '=' } \langle term \rangle \quad (x \in PV) & & \text{(elementary update)} \\
&\mid \backslash R \text{ '}' \langle term \rangle \text{ '}' \mid \backslash W \text{ '}' \langle term \rangle \text{ '}' & & \text{(read/write access update)} \\
&\mid \backslash R' \text{ '}' \langle term \rangle \text{ '}' \mid \backslash W' \text{ '}' \langle term \rangle \text{ '}' & & \text{(renamed read/write access update)} \\
&\mid \langle update \rangle \text{ '}' \parallel \text{ '}' \langle update \rangle & & \text{(parallel update)} \\
&\mid \langle update \rangle \text{ '}' ; \text{ '}' \langle update \rangle & & \text{(sequential update)}
\end{aligned}$$

Fig. 1 Grammar of JavaDL_{Dep}

the Hoare calculus: The formula $pre \rightarrow [s]post$ is equivalent to the Hoare triple $\{pre\}s\{post\}$ (assuming pre and $post$ are first-order formulas).

Definition 1 (*Signature* Σ) Let a *type hierarchy* $\mathcal{T} = (\text{Sort}, \leq)$ be a partially ordered set that consists of a set of type names Sort and a partial order $\leq: \text{Sort} \times \text{Sort}$.

A $\mathcal{T}$ -typed JavaDL_{Dep} signature $\Sigma_{\mathcal{T}} = (\text{PSym}, \text{FSym}, \text{VSym})$ consists of the non-empty sets PSym of predicate symbols, FSym of function symbols, and VSym of first-order variables.

JavaDL_{Dep} has *rigid* as well as *non-rigid* predicate (PSym_r and PSym_{nr}) and function (FSym_r and FSym_{nr}) symbols. *Rigid* symbols have the same meaning in all (program) states, while the interpretation of *non-rigid* symbols depends on the state. The set of *program variables* $PV \subseteq \text{FSym}_{nr}$ consists of all non-rigid constants. Typical rigid symbols are theories, literals, and values that retain the same meaning throughout a program execution.

Updates are a separate syntactic category to represent state changes syntactically: An *elementary update* $l := r$ assigns the value of term r to program variable l . An update u can be composed with another update u' into a *sequential update* $u; u'$ or into a *parallel update* $u \parallel u'$. Parallel updates are applied simultaneously in the same state, i.e., they do not influence each other. If a program variable occurs more than once on the left-hand side of a parallel update, the syntactically last update wins. In contrast to assignment statements, updates are composed of first-order logic terms, where the right-hand side may contain logical variables that can be quantified over.

The grammar of terms, formulas, and updates is given in Fig. 1. JavaDL_{Dep} terms and formulas are inductively defined as usual (abbreviated with "..."). Java statements are represented by *stmtnt*. Additionally, an update u can be applied to a term $\{u\}t$ or to a formula $\{u\}\phi$, similar to explicit substitutions, with the intended meaning that term t (formula ϕ) is evaluated in the state reached after performing the state changes described by u . Read/write access updates are explained below.

Example 1 In state s , where i and j are 42 and 0, respectively, the formula below behind the update is evaluated in the state s' that differs from s only in that the values of i and j are swapped:

$$i \doteq 42 \wedge j \doteq 0 \rightarrow \{i := j \parallel j := i\}(i \doteq 0 \wedge j \doteq 42)$$

Data dependences are properties about the access order of heap memory locations, here pairs of an array a and an index i . Hence, we need to syntactically represent sets of memory locations. (Memory) location sets are represented as terms of type `LocSet`, providing constructors to build location sets. Here, we use mostly the constructors $\underline{a[l..h]}$ to denote the set of locations $\{(a, i) \mid l \leq i \leq h\}$ and $\underline{a[l]}$ to denote the singleton location set $\{(a, l)\}$. We employ the convention that syntactic expressions are underlined, while semantic values are not. Local variables are not modeled as memory locations as they do not live on the heap. In addition to constructors, there are rigid function symbols for standard set operations *union*, *intersect*, *setMinus*, etc., with their usual semantics.

To express dependence properties, the program logic JavaDL_{Dep} provides non-rigid, unary predicate symbols, called dependence predicate symbols, $\text{PSym}_{Dep} = \{\text{noR}, \text{noW}, \text{noRaW}, \text{noWaR}, \text{noWaW}\} \subseteq \text{PSym}_{nr}$. These predicate symbols take a set of memory locations ls as argument and specify that there are no read $\text{noR}(ls)$ or no write $\text{noW}(ls)$ accesses on any of the memory locations in ls ; similarly, no-read-after-write $\text{noRaW}(ls)$, no-write-after-read $\text{noWaR}(ls)$, and no-write-after-write $\text{noWaW}(ls)$ data dependence predicates are defined.

Example 2 Let a, i denote program variables of type $\text{int}[]$ and int , respectively. The formula $\text{noRaW}(a[0..i])$ states that no array element between $a[0]$ and $a[i]$ (inclusive) was read if it had been written to before with no other access in between.

Elementary updates express state changes on program variables. Dedicated updates are used to capture memory accesses during symbolic execution and the order between them. These *memory access updates* $\backslash R(ls)$ and $\backslash W(ls)$ (see Fig. 1) express that there are read and write accesses, respectively, on all locations in ls .

Example 3 (*Memory Access Update*) The formula

$$\{\backslash W(a[0..i]); \backslash R(a[j..a.length-1])\} \text{noRaW}(a[0..i])$$

states that there is no read-after-write access on any element of a between (and including) indices 0 and i in any execution state with previous write accesses on $a[0..i]$, followed by read accesses on $a[j..a.length-1]$. Obviously, this can possibly hold only when $i < j$.

We conclude this section with a brief discussion and a justification of our decision to use data dependence predicates and memory access updates. At first sight, an approach that tracks memory accesses by recording them as an *explicit* sequence of accesses seems less heavy and one could simply reuse existing first-order theories over finite sequences (or lists). However, the drawback of such an approach is that expressing data dependence properties, such as no-read-after-write, results in complex first-order formulas with nested quantifiers. This severely impacts the prospects for automation. In our approach, we use abstract interpretation instead of first-order inference to synthesize data dependence properties. This is the key to scalable automation, but it necessitates to strictly encapsulate the theory of memory access histories, with the data dependence predicate and memory access updates being the only interface.

2.2 Semantics

The semantics of JavaDL_{Dep} is defined relative to a *Kripke structure* $\mathcal{K}$, i.e. a labeled transition system whose states are first-order interpretations, plus access histories. Its transition relation encodes the semantics of a programming language. Since first-order interpretations of non-rigid symbols, such as program variables, determine the execution state of a program, we use the terms *state* and *interpretation* interchangeably.

However, the standard notion of state in a Kripke semantics is insufficient to characterize read and write access updates, because it records only state *changes*, but not how they were *arrived at*. For this reason, we add a possible history of read and write accesses to each execution state, so that a state becomes a *pair* (σ, Acc) of an execution state σ relative to an access history Acc .

Definition 2 (*Domain, JavaDL_{Dep} State*) A domain $\mathcal{D}$ is a non-empty set. For any type T we denote with $\mathcal{D}_T \subseteq \mathcal{D}$ the domain elements in T . A JavaDL_{Dep} state s is a pair (σ, Acc) with

- a first-order interpretation σ of function and predicate symbols and

- a finite sequence $Acc = \langle acc_1 \rangle \circ \dots \circ \langle acc_n \rangle$ of memory read and write accesses with $acc_i ::= Read(ls) \mid Write(ls)$, where $ls \in \mathcal{D}_{LocSet}$. The operator $\circ$ is the usual concatenation of sequences.

The following definition provides a useful tool to express state changes in a concise manner.

Definition 3 (*Modification of State and Access Function*) We write $s[x \leftarrow d]$ for state s' coinciding with s , except for the value of program variable x of type T , which is evaluated to $d \in \mathcal{D}_T$.

Definition 4 (*Kripke Structure*) A JavaDL_{Dep} transition system or *Kripke structure* $\mathcal{K} = (\mathcal{S}, \llbracket \cdot \rrbracket)$ consists of: (i) an infinite set of states $\mathcal{S}$, where all $s \in \mathcal{S}$ coincide on their domain and the interpretation of rigid functions and predicates; (ii) a state transition relation $\llbracket \cdot \rrbracket : \text{Prg} \times \mathcal{S} \times \mathcal{S}$ that associates program prg with all pairs of states (s, t) such that if prg is executed in s then it terminates in t in accordance to Java's program semantics.

As JavaDL_{Dep} is deterministic, there is at most one tuple $(s, t) \in \llbracket \text{prg} \rrbracket$ for each program prg and state $s \in \mathcal{S}$.

Definition 5 (*Variable Assignment*) A *variable assignment* $\beta : \text{VSym} \rightarrow \mathcal{D}$ maps each first-order variable v of type T to an element of its domain $\mathcal{D}_T$.

The domain for location sets $\mathcal{D}_{LocSet} = 2^{Location}$, is the power set of *locations*. We use locations to abstract away from memory addresses. Intuitively, a location (a, i) of an array element $a[i]$ is the memory address where its value is stored:

Definition 6 (*Location*) A memory *location* for array element $a[i]$ is a pair (a, i) , where $a \in \mathcal{D}$ denotes an array and $i \in \mathbb{Z}$.

Observe that memory locations are a semantic notion and not part of the syntax. On the syntax side, we use *location terms* which are interpreted over memory locations.

The semantics of JavaDL_{Dep} is given in terms of a *state transition valuation* $\llbracket p \rrbracket_{\mathcal{K}, s, \beta}$ (for programs) and a *valuation function* $val_{\mathcal{K}, s, \beta}$ for updates, terms and formulas. Its definition is inductive over the term and formula structure. The definition continues the base case to arbitrary terms and formulas from interpretations of function, predicate symbols and programs given by the Kripke structure, its states and the variable assignment. Figure 2 shows only the cases needed in addition to the standard definitions in classical first-order logic.

A program p is evaluated with respect to a Kripke structure $\mathcal{K}$, an execution state s , and a variable assignment β , to a set of tuples of states that is either empty (if p does not terminate) or is a singleton set $\{(s, s')\}$, where s' denotes the reached final state (if p terminates). It is a singleton set, because JavaDL_{Dep} is deterministic.

State transition valuation $\llbracket p \rrbracket_{\mathcal{K}, s, \beta}$ uses an auxiliary function $acc_{\mathcal{K}, s, \beta} : \text{Term} \rightarrow \text{MemoryAccess}^*$ to extract all read accesses needed to evaluate a given term or program expression to keep track of read accesses:

$$\begin{aligned} acc_{\mathcal{K}, s, \beta}(c) &= \emptyset \text{ (} c \text{ rigid constant or a local program variable)} \\ acc_{\mathcal{K}, s, \beta}(a[i]) &= \langle Read(\llbracket a \rrbracket, i) \rangle \\ acc_{\mathcal{K}, s, \beta}(f(t_1, \dots, t_n)) &= acc_{\mathcal{K}, s, \beta}(t_1) \circ \dots \circ acc_{\mathcal{K}, s, \beta}(t_n) \text{ (} f \text{ is a rigid function symbol)} \end{aligned}$$

Dependence predicates are non-rigid, i.e., their truth value not only depends on their argument, but also on the value of Acc in the state $s = (\sigma, Acc)$, where they are evaluated.

Programs

$$\begin{aligned}
\llbracket stmt_1; stmt_2 \rrbracket_{\mathcal{K}, s, \beta} &:= \{(s, s'') \mid (s, s') \in \llbracket stmt_1 \rrbracket_{\mathcal{K}, s, \beta} \text{ and } \\
&\quad (s', s'') \in \llbracket stmt_2 \rrbracket_{\mathcal{K}, s', \beta}\} \\
\llbracket x = t \rrbracket_{\mathcal{K}, s, \beta} &:= \{(s, s') \mid s' = (\sigma', Acc') \text{ with } s = (\sigma, Acc), \text{ where } \\
&\quad \sigma' = \sigma[x \leftarrow val_{\mathcal{K}, s, \beta}(t)] \text{ and } \\
&\quad Acc' = Acc \circ acc_{\mathcal{K}, s, \beta}(t)\} \text{ (} x \text{ is a local variable)} \\
\llbracket a[i] = t \rrbracket_{\mathcal{K}, s, \beta} &:= \{(s, s') \mid s' = (\sigma', Acc') \text{ with } s = (\sigma, Acc), \text{ where } \\
&\quad \sigma' = \sigma[a[i] \leftarrow val_{\mathcal{K}, s, \beta}(t)] \text{ and } \\
&\quad Acc' = Acc \circ acc_{\mathcal{K}, s, \beta}(t) \circ acc_{\mathcal{K}, s, \beta}(i) \circ \langle Write(a[i]) \rangle\} \\
\llbracket \text{while } (b) \{ stmt \} \rrbracket_{\mathcal{K}, s, \beta} &:= \{(s, t) \mid \text{there is a sequence } s = s_0 s'_0 \cdots s_n s'_n = t \\
&\quad \text{with } \mathcal{K}, s_i, \beta \models b, i < n \\
&\quad \text{and } \mathcal{K}, t, \beta \not\models b \text{ and } (s'_i, s_{i+1}) \in \llbracket stmt \rrbracket_{\mathcal{K}, s_i, \beta}, i < n \\
&\quad \text{with } s'_i = (\sigma_i, Acc_i \circ acc_{\mathcal{K}, s_i, \beta}(b)) \}
\end{aligned}$$

Updates

$$\begin{aligned}
val_{\mathcal{K}, s, \beta}(x := t) &= s' \text{ with } s = (\sigma, Acc), s' = (\sigma', Acc) \text{ and } \sigma' = \sigma[x \leftarrow val_{\mathcal{K}, s, \beta}(t)] \\
val_{\mathcal{K}, s, \beta}(\backslash R(ls)) &= s' \text{ with } s = (\sigma, Acc) \text{ and } s' = (\sigma, Acc \circ \langle Read(val_{\mathcal{K}, s, \beta}(ls)) \rangle) \\
val_{\mathcal{K}, s, \beta}(\backslash W(ls)) &= s' \text{ with } s = (\sigma, Acc) \text{ and } s' = (\sigma, Acc \circ \langle Write(val_{\mathcal{K}, s, \beta}(ls)) \rangle)
\end{aligned}$$

Terms

$$val_{\mathcal{K}, s, \beta}(\{u\}t) = val_{\mathcal{K}, s', \beta}(t) \text{ with } val_{\mathcal{K}, s, \beta}(u) = s'$$

Dependence Predicates

$$\begin{aligned}
s(\text{noR}) &= \{(ls) \mid s = (\sigma, Acc), Acc = \langle M_1(ls_1) \rangle \circ \cdots \circ \langle M_n(ls_n) \rangle, M_i \in \{Read, Write\} \\
&\quad \text{and for all } j = 1 \dots n \text{ for which } M_j = Read \text{ it holds that } ls \cap ls_j = \emptyset\}
\end{aligned}$$

Fig. 2 Excerpt of JavaDL_{Dep} Semantics

Figure 2 also shows the semantics of the no-read predicate `noR`. It states that the formula `noR(ls)` is evaluated to true in a state $s = (\sigma, Acc)$, if the sequence of memory accesses Acc does not contain any read access $Read(ls')$ on a set of locations ls' that contains a location in $val_{\mathcal{K}, s, \beta}(ls)$.

Updates are used to represent state changes occurring during symbolic execution. State updates $x := t$ describe value changes to program locations, meaning they modify the first component σ of a state. In contrast, memory access updates $\backslash R'(ls)$, $\backslash W'(ls)$ record which location has been read or written. They modify the second state component Acc .

2.3 Calculus

To reason about the validity of a formula ϕ we use a *sequent calculus* [27]. The rules for the classical typed first-order fragment of JavaDL_{Dep} is standard, see, for example [1, Chapter 2]. The underlying data structure of the calculus is a sequent

$$\overbrace{\phi_1, \dots, \phi_n}^{\text{antecedent}} \Longrightarrow \overbrace{\psi_1, \dots, \psi_m}^{\text{succedent}} \quad (n, m \geq 0)$$

whose antecedent and succedent are sets of JavaDL_{Dep} formulas. This sequent has the same meaning as the formula

$$\left(\bigwedge_{i=1}^n \phi_i \right) \rightarrow \left(\bigvee_{j=1}^m \psi_j \right).$$

A sequent calculus proof forms a finite tree whose root is labeled with the sequent to be proven valid. It then applies schematic rules

$$\text{rule name} \frac{seq_1 \cdots seq_k}{seq} \quad \begin{array}{l} \leftarrow \text{premises} \\ \leftarrow \text{conclusion} \end{array}$$

to the leaves of the tree by matching the sequent in the conclusion to a leaf sequent and appending the instantiated sequents of the premise as children to the former leaf.

Rules with no premise are called axioms. They close the branch on whose leaf they are applied. A proof is closed if all its branches are closed. Soundness of the calculus ensures that a closed proof implies validity of the proof's root sequent.

The JavaDL_{Dep} calculus reduces sequents that contain formulas with modalities to pure first-order sequents. This is achieved by rules that execute the programs within the modalities symbolically. We explain some representative rule schemata:

If the next statement to be symbolically executed is a simple assignment to a local variable, then the rule schema

$$\text{assignmentLocalVariable} \frac{\Gamma \Longrightarrow \{u\}\{v := t\}[r]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[v = t; r]\phi, \Delta}$$

simply moves the assignment into an update. Schema variables v , t , u , r , and ϕ match a local program variable, a side-effect free expression (without memory accesses), an update, a statement, and a formula, respectively. Γ , Δ match sets of formulas.

A more complex rule is required for assignments to local variables that read from an array, because the read access needs to be recorded:

$$\text{assignVarWithArrayRead} \frac{\Gamma \Longrightarrow \{u\}\{v := a[i] \parallel \backslash R(a[i])\}[r]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[v = a[i]; r]\phi, \Delta}$$

As an example of a rule schema that causes a proof branch, we use symbolic execution of a conditional statement that splits the proof into a branch, where the guard of the if-statement is true and one, where it evaluates to false:

$$\text{conditional} \frac{\Gamma, \{u\}b \Longrightarrow [s_1; r]\phi, \Delta \quad \Gamma, \{u\}\neg b \Longrightarrow [s_2; r]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\text{if } (b) \{s_1\} \text{ else } \{s_2\} r]\phi, \Delta}$$

The following two rule schemata permit reasoning about loops. We start with a rule schema that unwinds a loop once:

$$\text{unwindLoop} \frac{\Gamma \Longrightarrow \{u\}[\text{if } (b) \{s; \text{while } (b) \{s\} r\}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\text{while } (b) \{s\} r;]\phi, \Delta}$$

Its disadvantage is that it is not possible to complete a proof when the loop has no fixed bound on the number of iterations. In program verification *loop invariants* are used to describe the behavior of loops in a finite manner. Loop invariants are properties that are not affected by any loop iteration, i.e. a loop invariant is true after executing the loop body in any iteration provided that it held at the beginning. This is sufficient to approximate the effect of executing a loop for *any* number of iterations. To establish a conjectured loop invariant *LoopInv*, one

must “forget” any knowledge about the computation state embodied in u , Γ , Δ when the loop is encountered, because that might have changed during a previous loop iteration. This forgetting can be simply achieved by deleting the context u , Γ , Δ in the rule premise that proves *LoopInv* to be invariant. This is inefficient, because one has to add any aspect of the initial context that is needed after loop execution to the loop invariant which in turn becomes unwieldy.

Instead, we use a more efficient approach to manage the loop context. Instead of forgetting the complete context, we selectively “forget” only the knowledge about the values of those program variables and locations that might potentially influence the behavior of a loop iteration, i.e. exactly those that can potentially be modified during a loop iteration. To achieve this, we use an update (see Sect. 2.1) that assigns each of the potentially changed program variables and locations a fixed but unknown value using a fresh Skolem constant. For instance, if a loop only modifies the loop counter i , we construct the update $\{i := c\}$, where c is a fresh constant. We refer to updates constructed in such a manner as *anonymizing updates* to emphasize their intended usage. We provide a more extensive example and refer to [1, Chapter 3] for further details.

Example 4 (*Anonymization*) Given the following program (assume that initially $i \doteq 0$):

```
while ( $i < a.length$ ) {  
   $a[i+1] = a[i]$ ;  
   $i++$ ; }
```

At the beginning of the k -th loop iteration, the values of all entries of a from 1 up to and including $k - 1$ may have been modified as well as the value of i . We capture this as a state transition that “forgets” exactly this information, achieved by the following *anonymizing update* $\{v\}$:

$$\{v\} = \{i := k \parallel \text{heap} := \text{anon}(\text{heap}, \underline{a[1..k]}, \text{heap}') \parallel \backslash \text{anonAccess}(id)\}$$

It consists of three parts: (i) the first update anonymizes the local program variable i by assigning it the value of a fresh constant k (about which we know nothing); (ii) the second update anonymizes the part of the heap that is possibly modified by the loop. This is achieved by using the *anon*-function which takes the heap to be anonymized, the set of locations whose values should be set to unknown values and a fresh heap' with unknown values. The term $\text{anon}(\text{heap}, \underline{a[1..k]}, \text{heap}')$ evaluates then to a heap that coincides with heap on the values of all locations, except those contained in $\underline{a[1..k]}$. The values of the latter are taken from the fresh heap heap' about which nothing is known; (iii) the third part anonymizes the *Acc* part of the state and is described below in Sect. 4.3.

The loop invariant rule thus splits a proof into three branches, proving that the loop invariant *LoopInv* 1. holds initially, 2. it is preserved by the loop body, and, 3. together with the negated loop guard, it is sufficiently strong to prove that after executing the remaining program r the postcondition ϕ holds. The three premises of the *loop_inv* rule below realize these branches, respectively. The second premise executes the loop body for an *arbitrary* iteration, and the third premise executes the remaining program after the loop. Hence, the second and third premise both contain the anonymizing update $\{v\}$, but they retain the context u , Γ , Δ .

$$\text{loop_inv} \frac{\begin{array}{c} \Gamma \Longrightarrow \{u\} \text{LoopInv}, \Delta \\ \Gamma, \{u\}\{v\}(b \wedge \text{LoopInv}) \Longrightarrow \{u\}\{v\}[s] \text{LoopInv}, \Delta \\ \Gamma, \{u\}\{v\}(\neg b \wedge \text{LoopInv}) \Longrightarrow \{u\}\{v\}[x]\phi, \Delta \end{array}}{\Gamma \Longrightarrow \{u\}[\text{while } (b) \{s\} r;]\phi, \Delta}$$

For ease of presentation, we do not use the sequent rule format for simple rewrite rules, instead we use the notation $\phi \rightsquigarrow^{\text{cond}} \psi$ to express that if *cond* can be proven then formula ϕ can be replaced by ψ . The formula *cond* may be omitted if it is the formula *true*.

2.4 Loop Invariant Generation

The technique in [68] generates loop invariants for simple loops to analyze data dependences. They propose a version of loop invariant generation by predicate abstraction [37], adapted to deductive verification and dependence analysis.

Loop invariant generation roughly works as follows: Let a set of abstract predicates $\mathcal{A}$ contain formulas that express dependence properties known to hold in the state before executing the loop. Now the loop is unwound once. Afterwards we check which of the formulas in $\mathcal{A}$ hold in the state reached. The provable ones are kept in $\mathcal{A}$, the others are replaced with weakened versions (determined using an abstraction lattice) that can be proven. Then the process restarts with unwinding the loop once more, until the set $\mathcal{A}$ is stable. Using a finite lattice and appropriate weakening operations, this happens after finitely many rounds.

2.4.1 “Priming” and the Shift Update Rule

Our forward symbolic execution calculus, including unwinding, accumulates state updates *in front of* modalities. However, for technical reasons, predicate abstraction is easier to implement, if state changes are expressed as *equations* corresponding to the strongest postcondition, not as weakest preconditions that result from update application.

In more detail, the power of our approach is derived from a fine-grained combination of symbolic execution and predicate abstraction, i.e. before and after each loop iteration in a program. In the KeY verifier, the symbolic states arising during symbolic execution are constantly simplified, irrelevant parts of the state are eliminated, unreachable states are deleted. This is crucial to combat state explosion and to realize efficient symbolic execution.

To perform predicate abstraction, the information embodied in the accumulated updates must be abstracted. In the presence of relations between two or more variables, this become technically complex and hard to manage. The situation is exacerbated, because we perform abstraction as part of generating a loop invariant for dependence properties. To illustrate the problem, consider a typical update resulting from symbolically executing a loop iteration:

$$\{v_1 := t_1 \parallel \dots \parallel v_n := t_n \parallel \backslash Ev(ls_1); \dots \backslash Ev(ls_k)\},$$

where $\backslash Ev$ is either $\backslash R$ or $\backslash W$. To infer data dependence, an abstraction must be found that maintains the kind *and* the order of the memory access updates. This might be achieved with a suitable regular (or context-free) language to describe access patterns within updates. Still, it is necessary to infer which data dependences are absent (or present), requiring reasoning and predicates that express the absence (presence) of these data dependences. Such a design becomes highly technical and hard to understand.

Instead, we opt for a *standard*, well-understood abstraction technique, not based on updates, but on predicates to keep track of data dependences. An additional advantage is

that symbolic execution remains efficient during fixpoint generation, because at the start of each iteration, the initial update preceding the subsequent loop iteration is concise and only accumulates the state changes for a single iteration. This combines the advantages of efficient forward symbolic execution with the flexibility of predicate abstraction.

The price we have to pay for this, is that we must be able to “shift” an update-based representation of symbolic states into a predicate-based one. Fortunately, dynamic logic is expressive enough to do this without leaving the logical framework: Indeed, abstraction, loop execution, and deductive verification all are performed *within* the logical framework introduced so far. The single new concept needed is a suitable first-order theory of access predicates. Before we discuss the actual shifting rule, we illustrate the idea behind it with an example:

Example 5 Unwinding the loop in the following sequent once

$$\Gamma, i \doteq 0 \Longrightarrow \langle \text{while } (i < 42) \{ i++; \} \rangle \phi, \Delta$$

$$\Gamma, i < 42, i \doteq 0 \Longrightarrow \underbrace{\{i := i + 1\}}_u \langle \text{while } (i < 42) \{ i++; \} \rangle \phi, \Delta. \quad (1)$$

Translating (“shifting”) update u into a set of equations results in the sequent:

$$\Gamma', i' \doteq 0, i' < 42, i \doteq i' + 1 \Longrightarrow \langle \text{while } (i < 42) \{ i++; \} \rangle \phi, \Delta' \quad (2)$$

Intuitively, *shifting* changes the state in which a sequent is evaluated. Sequent (2) is no longer evaluated in the initial state of sequent (1), but in the shifted state under update u . Consequently, all occurrences of i (including those in Γ, Δ) are renamed to primed versions. For this reason, the state shifting technique is also called “priming”. The equation $i \doteq i + 1$ now connects the old and new value of i , capturing the information that used to be embodied in update u . Sequent (2) contains no update anymore, only equations that are suitable as input for predicate abstraction.

The shifting rule for state properties (without dependences) was proposed in [69]:

$$\text{shiftUpdate} \frac{\{u'\}\Gamma, \text{upd}(u, u') \Longrightarrow \phi, \{u'\}\Delta}{\Gamma \Longrightarrow \{u\}\phi, \Delta} \quad (3)$$

As illustrated above, the idea behind the rule is, instead of transforming ϕ to its weakest precondition under update u , to transform the other parts of the sequent into their strongest postcondition. This is done by *advancing* the symbolic state of the sequent except ϕ through *priming*. Formula $\text{upd}(u, u')$ establishes the relation between the original update u and its renamed (“primed”) version u' . The formal definition is given below [69].

Definition 7 The formula $\text{upd}(u, u')$ establishes the relation between the original update u and its renamed update u' in the shiftUpdate rule. It is defined as

$$\text{upd}(u, u') = \bigwedge_{i \in \{1, \dots, n\}} a_i \doteq \{u'\}\{u\}a_i, \text{ where}$$

- $\{a_1, \dots, a_n\} = \text{updated}(u)$ with $\text{updated} : \text{Upd}_\Sigma \rightarrow 2^{PV}$
- $\text{updated}(u) = \begin{cases} \{a\} & u = a := t \\ \text{updated}(u_1) \cup \text{updated}(u_2) & u = u_1 || u_2 \\ \text{updated}(u_2) & u = \{u_1\}u_2 \end{cases}$
- $u' = a_1 := a'_1 || \dots || a_n := a'_n$

- $a'_i \in \text{FSym}_r$ fresh constants for all $i \in \{1, \dots, n\}$

Update u' substitutes each updated program variable a_i with a fresh constant a'_i representing the old (pre-update) value of a_i . Formula upd links the old instances with the updated ones. Weiß [69] showed that $\{u'\}\Gamma \wedge upd$ is the strongest post condition of Γ under u .

Example 6 In preparation for applying the `shiftUpdate` rule to the following sequent:

$$\Gamma, i \doteq 42 \Longrightarrow \underbrace{\{i := 0\}}_u \phi, \Delta$$

formula $upd(u, u')$ is:

$$i \doteq \underbrace{\{i := i'\}}_{u'} \underbrace{\{i := 0\}}_u i$$

where $i' \in \text{FSym}_r$ is a fresh constant. Observe that $\{i := i'\}\{i := 0\}i$ simplifies to 0.

Example 7 Applying the `shiftUpdate` rule on the following sequent:

$$\Gamma, i \doteq 42 \Longrightarrow \underbrace{\{i := 0\}}_u \phi, \Delta$$

results in the sequent:

$$\underbrace{\{i := i'\}}_{u'} \Gamma, \underbrace{\{i := i'\}}_{u'} i \doteq 42, \underbrace{i \doteq \{i := i'\}\{i := 0\}i}_{upd(u, u')} \Longrightarrow \phi, \underbrace{\{i := i'\}}_{u'} \Delta.$$

After update simplification we have:

$$\underbrace{\{i := i'\}}_{u'} \Gamma, i' \doteq 42, i \doteq 0 \Longrightarrow \phi, \underbrace{\{i := i'\}}_{u'} \Delta$$

where the old value of i is saved in i' , and its new value is 0.

2.4.2 Shifting Memory Access Updates

To shift *memory access updates*, in the paper [68] the data dependence logic was extended with predicates and updates that record the effect of the memory access updates as formulas (instead of updates) to compute the strongest postcondition of formulas.

In the previous section, shifting was achieved with new symbols by priming program variables. The equality predicate then suffices to relate the value of primed and unprimed program variables. This would be sufficient for our data dependence logic if we had adopted the approach of modeling the sequence of memory accesses explicitly in first-order logic. As discussed at the end of Sect. 2.1, this is not viable for a high degree of automation.

To make shifting work for memory access updates we introduce a small set of additional predicates (one for each data dependence predicate and data dependence update plus two general read/write memory access predicates) to be able to express the strongest postcondition of data dependence predicates.

We sketch this solution for read memory access updates. To shift $\backslash R$ -updates three ingredients are required: (i) non-rigid predicates $\text{rPred} : \text{LocSet} \times \text{int}$ to record read accesses as a formula; (ii) history dependence predicates $\text{noRHist} : \text{LocSet} \times \text{int}$ to express that no read happened until a given number of preceding memory accesses; and (iii) primed read memory access updates $\backslash R'(\text{LocSet})$ to express the strongest postcondition. The exact

$$\underbrace{\langle acc_1(ls_1) \rangle \circ \dots \circ \langle acc_{n-(i+1)}(ls_{n-(i+1)}) \rangle}_{\text{noRHist}(ls, i+1)} \circ \overbrace{\langle acc_{n-i}(ls_{n-i}) \rangle}^{\text{rPred}(ls, i)} \circ \langle acc_{n-(i-1)}(ls_{n-(i-1)}) \rangle \circ \dots \circ \langle acc_n(ls_n) \rangle$$

Fig. 3 Access subsequences referred to by $\text{rPred}(ls, i)$ and $\text{noRHist}(ls, i+1)$

usage of these predicates and updates, and their interplay, will become clear below when the shifting rule for memory access is introduced.

Intuitively, the formula $\text{rPred}(ls, i)$ holds in a state $s = (\sigma, acc_1 \dots acc_{n-i} \dots acc_n)$ if and only if $acc_{n-i} = \text{Read}(val_{\mathcal{K}, s, \beta}(ls))$. In other words, the formula $\text{rPred}(ls, i)$ expresses that the i -th most recent memory access is a read access on exactly the locations in $val_{\mathcal{K}, s, \beta}(ls)$.

Definition 8 (*History data dependence predicate*) The syntax of the history data dependence predicates is

$$\langle fml \rangle ::= \text{noRHist} \langle ' \langle term \rangle, \langle term \rangle ' \rangle \mid \text{noWHist} \langle ' \langle term \rangle, \langle term \rangle ' \rangle \mid \dots$$

where the first argument is a location set and the second an integer.

The semantics is as follows:

$$\begin{aligned} s(\text{noRHist}) &= \{(ls, i) \mid s = (\sigma, Acc), 0 \leq i < n, \\ &\quad Acc = \langle M_1(ls_1) \rangle \circ \dots \circ \langle M_{n-i}(ls_{n-i}) \rangle \circ \dots \circ \langle M_n(ls_n) \rangle, \\ &\quad \text{for all } 0 < j \leq n - i \text{ with } M_j = \text{Read it holds that } ls \cap ls_j = \emptyset\} \end{aligned}$$

(analogously for noWHist) and

$$\begin{aligned} s(\text{noRaWHist}) &= \{(ls, i) \mid s = (\sigma, Acc), 0 \leq i < n, \\ &\quad Acc = \langle M_1(ls_1) \rangle \circ \dots \circ \langle M_{n-i}(ls_{n-i}) \rangle \circ \dots \circ \langle M_n(ls_n) \rangle, \\ &\quad \text{and for all } 0 < j < k \leq n - i \text{ with } M_j = \text{Write and } M_k = \text{Read} \\ &\quad \text{it holds that } ls \cap ls_j \cap ls_k = \emptyset\} \end{aligned}$$

(analogously for noWaRHist , noWaWHist).

Formula $\text{noRHist}(ls, i+1)$ holds in state s if there has been no read access on any location in ls up to and including $acc_{n-(i+1)}$. Figure 3 visualizes the subsequences of the memory accesses that determine the validity of the formulas.

Primed updates $\backslash \mathcal{R}'(ls)$, $\backslash \mathcal{W}'(ls)$ are used to transform the formula on which they are applied to their strongest postcondition. Hence, the update $\backslash \mathcal{R}'(ls); \backslash \mathcal{R}(ls)$ is equivalent to the empty update. Formally, we define the primed versions as follows:

Definition 9 (*Renamed memory access update predicate*) The syntax of a renamed memory access update is

$$\langle update \rangle ::= \dots \mid \backslash \mathcal{R}' \langle ' \langle term \rangle ' \rangle \mid \backslash \mathcal{W}' \langle ' \langle term \rangle ' \rangle$$

where the term given as argument denotes a location set.

Their semantics is defined as follows:

$$val_{\mathcal{K}, s, \beta}(\backslash \mathcal{R}'(loc)) = s' \text{ where}$$

$$\begin{array}{c}
\text{noRImpliesNoRHist} \qquad \text{noR}(loc) \rightsquigarrow \text{noRHist}(loc, lb) \\
\\
\text{noRHistOnBothSides} \\
\frac{\Gamma, \text{noRHist}(loc_1, lb_1), lb_1 \leq lb_2 \implies \text{noRHist}(loc_2 \setminus loc_1, lb_2), \Delta \quad \Gamma, \text{noRHist}(loc_1, lb_1), lb_1 > lb_2 \implies \text{noRHist}(loc_2, lb_2), \Delta}{\Gamma, \text{noRHist}(loc_1, lb_1) \implies \text{noRHist}(loc_2, lb_2), \Delta}
\end{array}$$

Fig. 4 Excerpt of subsumption relations between history data dependence predicates

$$s' = \begin{cases} (\sigma, acc_seq), & \text{if } s = (\sigma, acc_seq \circ \langle Read(val_{\mathcal{K},s,\beta}(loc)) \rangle) \\ sChoice(s, val_{\mathcal{K},s,\beta}(loc)), & \text{otherwise} \end{cases}$$

$val_{\mathcal{K},s,\beta}(\backslash W'(loc)) = s'$ where

$$s' = \begin{cases} (\sigma, acc_seq), & \text{if } s = (\sigma, acc_seq \circ \langle Write(val_{\mathcal{K},s,\beta}(loc)) \rangle) \\ sChoice(s, val_{\mathcal{K},s,\beta}(loc)), & \text{otherwise} \end{cases}$$

Function $sChoice(s, val_{\mathcal{K},s,\beta}(loc))$ takes a state s and location set $val_{\mathcal{K},s,\beta}(loc)$ and returns an arbitrary state. This way, we ensure that the evaluation function is total and well-defined.

Example 8 The formula $\{\backslash R'(\underline{a}[0])\} \text{noR}(\underline{a}[0])$ holds in a state $s' = (\sigma', acc_1 \cdots acc_{n-1} acc_n)$ if in the predecessor state $s = (\sigma, acc_1 \cdots acc_{n-1})$ the formula $\text{noR}(\underline{a}[0])$ did hold. In other words, if up to and including state s the location $\underline{a}[0]$ has never been read.

We can now define the shift rule for read memory access updates (analogously for write memory updates). Observe the structural similarity to the shift rule for updates (3). The renamed memory access predicates were purposely defined to make this possible.

$$\text{shiftRead} \frac{\{\backslash R'(ls)\} \Gamma, \text{rPred}(\{\backslash R'(ls)\} ls, 0) \implies \phi, \{\backslash R'(ls)\} \Delta}{\Gamma \implies \{\backslash R(ls)\} \phi, \Delta}$$

Predicate $\text{rPred}(\{\backslash R'(ls)\} ls, 0)$ expresses that the first memory access that happened (labeled by integer 0) is a read memory access on the location set ls . The formal definition is given in the Appendix A.

Example 9 Applying the shiftRead rule on the sequent $\text{noR}(\underline{a}[0]) \implies \{\backslash R(\underline{a}[0])\} \phi$ results in the sequent $\{\backslash R'(\underline{a}[0])\} \text{noR}(\underline{a}[0]), \text{rPred}(\{\backslash R'(\underline{a}[0])\} \underline{a}[0], 0) \implies \phi$.

In the formula $\{\backslash R'(\underline{a}[0])\} \text{noR}(\underline{a}[0])$ in the example's antecedent, one must apply a read memory access update to a read-related dependence predicate. This necessitates to express that no read access happened during the last relevant states (here only one state). It is exactly for this purpose that we introduced the history dependence predicates. Figure 4 shows some subsumption relations that illustrate how data dependence predicates are rendered into history data dependence predicates:

- Rule schema **noRImpliesNoRHist** states that if predicate $\text{noR}(loc)$ holds after any number of memory accesses, it holds after any specific number of lb memory accesses.
- In the rule **noRHistOnBothSides**, having $lb_1 \leq lb_2$ and consequently $n - lb_1 \geq n - lb_2$, infers that $\text{noRHist}(loc_1, lb_1)$ takes into account a larger segment of memory accesses, therefore, it has more information than $\text{noRHist}(loc_2, lb_2)$ regarding the read memory accesses on $loc_1 \cap loc_2$.

renamedReadAppOnNoR

$$\{\backslash R'(rLoc)\}noR(loc) \rightsquigarrow noRHist(\{\backslash R'(rLoc)\}loc, 1)$$

renamedReadAppOnReadPred

$$\{\backslash R'(rLoc)\}rPred(loc, lb) \rightsquigarrow rPred(\{\backslash R'(rLoc)\}loc, \{\backslash R'(rLoc)\}lb + 1)$$

Fig. 5 Some rules for renamed memory access update application

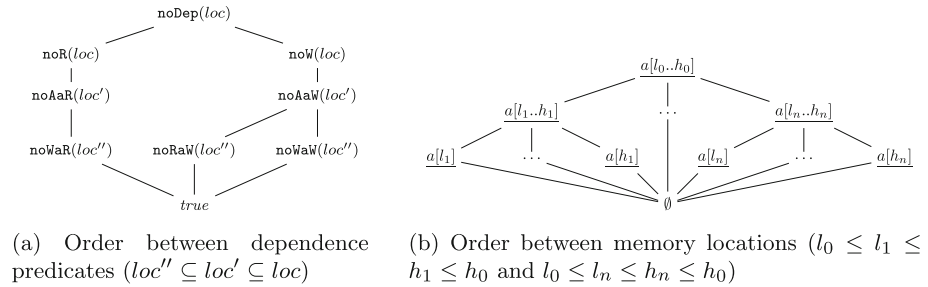


Fig. 6 Lattices used for predicate abstraction of data dependence properties

Other subsumption relations are analogous. Intuitively, applying a renamed memory access update *ages* memory access and history data dependence predicates. Rules *renamedReadAppOnNoR* and *renamedReadAppOnReadPred* in Fig. 5 are examples of rules reflecting this effect. Rules for application of renamed read memory access update on the rest of memory access and history data dependence predicates are analogous and therefore omitted. For a full account of all rules, see [67].

Example 10 The formula $\{\backslash R'(\underline{a}[0])\}noR(\underline{a}[0])$ is rewritten to $noRHist(\underline{a}[0], 1)$.

2.4.3 Predicate Abstraction

Weakening of dependence predicates during predicate abstraction is based on a lattice induced by predicates $PSym_{Dep}$, considering the partial order of read and write accesses they exclude. For example, $noR(loc)$ implies $noRaW(loc')$ and $noWaR(loc')$, assuming that $loc' \subseteq loc$.

Figure 6a shows excerpts of the dependence predicate lattice and Fig. 6b of the complementary lattice for memory locations, where the partial order is the subset relation for location sets.²

In predicate abstraction, the predicates are equipped with an abstraction relation based on a partial order. The product lattice of the data dependence predicate lattice and the location set lattice LS provides that abstraction relation.

Definition 10 (Abstract Domain) The abstract domain D_{Dep} consists of atomic formulas constructed from predicates $PSym_{Dep}$ and LS , i.e. $D_{Dep} = \{p(ls) \mid p \in PSym_{Dep} \text{ and } ls \in LS\}$.

Definition 11 (Abstract Language) The abstract language $L_{Dep} = \langle D_{Dep}, \sqsubseteq_{Dep} \rangle$ is the abstract domain D_{Dep} equipped with the partial order $\sqsubseteq_{Dep}$ that is induced by logical consequence.

² The predicates $noAaR$, $noAaW$ (no access after read, write, respectively) are present for technical reasons to complete the partial order into a lattice and can be ignored in what follows.

Definition 12 (*Data Dependence Loop Invariant*) A data dependence loop invariant LI_{Dep} has the form $\bigwedge_i p_i$, where $p_i \in D_{Dep}$.

It is well known that inference of general, sufficiently strong functional loop invariants is a difficult and largely unsolved problem. However, here the focus is on data dependence properties over arrays. In consequence, it suffices to generate functional invariants with arithmetic constraints over array bounds and loop counters, which is often feasible [37, 70].

The quality of the generated loop invariants depends on the level of abstraction embodied in the initial set of predicates provided for predicate abstraction. The data dependence loop invariant algorithm initially receives $\{noDep(a[0..a.length - 1])\}$ as input for a loop over array a , i.e. no dependences are present which is an obvious upper bound. The algorithm then unwinds the loop, shifts the generated updates using `shiftUpdate`, `shiftRead`, and `shiftWrite`. Based on the resulting sequent, it refines the dependences with help of the abstraction relations in Fig. 6. This process is repeated until the set of predicates reaches a fixed point. The conjunction of predicates left in the predicate set is the data dependence loop invariant. It is guaranteed to be correct, as each predicate is proved to hold. On the other hand, it is not necessarily inductive, as it might fail to imply the postcondition. In the worst case, one obtains *true*, i.e. no dependence can be excluded.

Example 11 Consider a program which shifts the content of array a to the left:

```
while (i < a.length - 1) { a[i] = a[i+1]; i++; }
```

Our approach generates the following loop invariant which happens to be the most precise one that is correct:

$$\begin{aligned} & noR(a[0]) \wedge noW(a[i..a.length-1]) \\ & \wedge noRaW(a[0..a.length-1]) \wedge noWaW(a[0..a.length-1]) \\ & \wedge 0 \leq i \leq a.length \end{aligned}$$

So far, we established the foundations of our approach and explained how intra-data dependences can be automatically generated. The central problem to be addressed in subsequent sections is to analyze *inter-data* dependences that exist between *different* loop iterations. This involves to solve several technical challenges (see Sects. 3 and 4.3):

1. find a suitable representation for sequences of memory access updates *grouped* according to different loop iterations;
2. define shift update (or “priming”) rules for *grouped* access updates to switch between an update-based and a predicate-based representation of the symbolic state;
3. extend anonymizing updates (Sect. 2.3) to *grouped* access updates.

We stress that these are not mere technical issues. The machinery is needed to formally define the *semantics of inter-iteration dependences*. Therefore, the considerable technical effort spent on it, is not only justifiable, but it is necessary.

3 Loop Invariant Generation for Inter-iteration Data Dependences

The granularity of tracking data dependences of $JavaDL_{Dep}$ as proposed in [15, 68] and described in Sect. 2 is unsuitable for many loop parallelization patterns, because it tracks *all* data dependences. For example, a simple loop parallelization that assigns each loop iteration its own process requires that there are no data dependences between *two different* loop iterations, but is oblivious to data dependences *within* the same iteration. The approach

sketched in Sect. 2 is not able to distinguish between intra- and inter-iteration dependences and prevents taking advantage of such a parallelization opportunity.

Therefore we extend JavaDL_{Dep} by adding support for *inter-iteration* data dependence predicates that permit dependences within the same loop iteration, but not those stretching between different loop iterations. We also provide an algorithm to generate corresponding loop invariants.

3.1 Syntax and Semantics

We extend the logic JavaDL_{Dep} to $\widehat{\text{JavaDL}}_{Dep}$ for reasoning about inter-iteration dependences. To specify and to reason about inter-iteration dependences $\widehat{\text{JavaDL}}_{Dep}$ defines three new non-rigid predicates: $\widehat{\text{noRaW}}$, $\widehat{\text{noWaR}}$ taking four arguments of type LocSet , and $\widehat{\text{noWaW}}$ with three arguments of type LocSet . Their semantics is similar to the dependence predicates introduced in Sect. 2 without $\widehat{\cdot}$, except that they are only sensitive to data dependences that stretch over *different* loop iterations. The extra arguments express history or future contexts and will be explained later.

```

1 sum = 0; i = 0;
2 while (i < a.length) {
3   a[i] = b[i] + c[i];
4   sum += a[i];
5   i=i+1;
6 }

```

Listing 1 Program `arraySum`: component-wise Array Addition

Example 12 Consider the program `arraySum` in Listing 1, which adds the content of arrays `b` and `c` component-wise and stores the result in array `a`. In addition, it computes the sum over all elements stored in `a`. Formula $\widehat{\text{noRaW}}(a[0..a.length - 1])$ does not hold after execution of `arraySum`, because array element $a[i]$ is read in line 4 and written in line 3 in the subsequent loop iteration. In contrast, $\widehat{\text{noRaW}}(a[0..a.length - 1], \emptyset, \emptyset, \emptyset)$ expresses that there are no read-after-write dependences *across* loop iterations and it holds after execution of the program.

As different loop iterations need to be distinguished, the memory access sequence of $\widehat{\text{JavaDL}}_{Dep}$ states must provide more structure than those of JavaDL_{Dep} . Instead of a sequence of memory accesses $\widehat{\text{JavaDL}}_{Dep}$ uses a *sequence of sequences* of memory accesses $\widehat{\text{Acc}}$. Each element in $\widehat{\text{Acc}}$ (i.e. a sequence of memory accesses) consists of the memory accesses of a single loop iteration.

Definition 13 (*Domain, $\widehat{\text{JavaDL}}_{Dep}$ State, Kripke Structure*) Given a non-empty domain $\mathcal{D}$. A $\widehat{\text{JavaDL}}_{Dep}$ state $s = (\sigma, \widehat{\text{Acc}})$ is a pair of an interpretation σ as in JavaDL_{Dep} (Def. 10) and

a finite sequence $\widehat{\text{Acc}} = \text{racc}_1 \circ \cdots \circ \text{racc}_n$ of sequences of memory read and write accesses $\text{racc}_i ::= \langle \text{acc}_{i,1} \rangle \circ \cdots \circ \langle \text{acc}_{i,m} \rangle$, where $\text{acc}_{i,j} ::= \text{Read}(ls) \mid \text{Write}(ls)$, $ls \in \mathcal{D}_{\text{LocSet}}$, and $\circ$ is the concatenation between elements of $\widehat{\text{Acc}}$ (the big circles are separators to save brackets and improve readability).

$$\begin{aligned}
\llbracket x = t \rrbracket_{\mathcal{K}, s, \beta} &:= \{(s, s') \mid s' = (\sigma', \widehat{Acc}') \text{ with } s = (\sigma, \widehat{Acc}), \sigma' = \sigma[x \leftarrow val_{\mathcal{K}, s, \beta}(t)] \text{ and} \\
&\quad \widehat{Acc}' = \widehat{Acc} \circ \langle acc_{\mathcal{K}, s, \beta}(t) \rangle = \widehat{acc}_1 \circ \dots \circ \widehat{acc}_n \circ \langle Read(t) \rangle \\
&\quad = racc_1 \circ \dots \circ racc_{n-1} \circ (acc_{n,1} \circ \dots \circ acc_{n,m} \circ \langle Read(t) \rangle)\} \\
&\quad (x \text{ is a local variable}) \\
\llbracket a[k] = t \rrbracket_{\mathcal{K}, s, \beta} &:= \{(s, s') \mid s' = (\sigma', \widehat{Acc}') \text{ with } s = (\sigma, \widehat{Acc}), \sigma' = \sigma[a[k] \leftarrow val_{\mathcal{K}, s, \beta}(t)] \text{ and} \\
&\quad \widehat{Acc}' = \widehat{Acc} \circ acc_{\mathcal{K}, s, \beta}(t) \circ acc_{\mathcal{K}, s, \beta}(k) \circ \langle Write(a[k]) \rangle\} \\
\llbracket \text{while}(b)\{stmnt\} \rrbracket_{\mathcal{K}, s, \beta} &:= \{(s, t) \mid \text{there is a sequence } s = s_0 \ s'_0 \ \dots \ s_n \ s'_n = t \text{ with} \\
&\quad \mathcal{K}, s_i, \beta \models b, \ i < n \text{ and } \mathcal{K}, t, \beta \not\models b \text{ and} \\
&\quad (s'_i, s_{i+1}) \in \llbracket stmnt \rrbracket_{\mathcal{K}, s'_i, \beta} \text{ with } s'_i = (\sigma_i, \widehat{Acc}_i \circ \langle acc_{\mathcal{K}, s'_i, \beta}(b) \rangle)\} \\
&\quad (\widehat{Acc} = \widehat{acc}_1 \circ \dots \circ \widehat{acc}_n \text{ and } \widehat{acc}_i = acc_{i,1} \circ \dots \circ acc_{i,m_i}, \ i \in \{1 \dots n\})
\end{aligned}$$

Fig. 7 Program semantics of JavaDL_{Dep}

A JavaDL_{Dep} Kripke structure $\mathcal{K} = (\mathcal{S}, \llbracket \cdot \rrbracket)$ consists of an infinite set of JavaDL_{Dep} states and a state transition valuation $\llbracket \cdot \rrbracket_{\mathcal{K}, s, \beta}$, whose definition for selected program elements is shown in Fig. 7.

Executing an assignment $x = t$ from side-effect free expression t to a local variable x in a state $s = (\sigma, \widehat{Acc})$ terminates in a state s' which is almost identical to state s except for the value of variable x and the possible extension of read memory accesses resulting from evaluating t . As an assignment inside a loop body does neither start nor end in the current loop iteration, the reads are appended to the sequence of memory accesses of the *current* iteration (i.e. the last element of $\widehat{Acc}$). Similarly for $\llbracket a[i] = t \rrbracket_{\mathcal{K}, s, \beta}$. In contrast, a loop *ends* the current iteration and *starts* a new one with each evaluation of its guard. This is reflected in the semantics by appending a *new* sequence to $\widehat{Acc}_i$ at the start of each loop iteration i . A minor observation is that on loop exit the final evaluation of the loop guard (evaluating to false) starts a *new* iteration and thus ends the current one.

It remains to define the semantics for the inter-iteration dependence predicates, which will enable us to specify and reason about inter-iteration data dependences. To streamline the definition of their semantics, we first introduce two projection functions that aggregate the locations of all read/write accesses for a given sequence of memory accesses:

Definition 14 (Projection) For $\widehat{acc} = acc_1 \circ \dots \circ acc_n$ the projection function for read accesses $\downarrow_R$ defined as

$$\downarrow_R \widehat{acc} := \{ls \mid ls = \bigcup_{i=1}^n ls_i \text{ where } acc_i = \langle Read(ls_i) \rangle\}$$

collects all memory locations read in loop iteration $\widehat{acc}$. The projection function for write accesses $\downarrow_W$ is defined analogously.

Example 13 Let $\widehat{acc} := \langle Read(\{(a, 0)\}) \rangle \circ \langle Write(\{(a, 1)\}) \rangle \circ \langle Read(\{(a, 2)\}) \rangle$ be a sequence of memory accesses. Then $\downarrow_R \widehat{acc} = \{(a, 0), (a, 2)\}$ and $\downarrow_W \widehat{acc} = \{(a, 1)\}$.

Definition 15 (Inter-iteration Data Dependence Predicates: Semantics) Let $ls, rLs, wLs, futRLS$ and $futWLS$ denote sets of locations in $\mathcal{D}^{\text{LocSet}}$. The *semantics* of predicate symbols $\widehat{\text{noRaW}}$ (similar for predicate $\widehat{\text{noWaR}}$, omitted) and $\widehat{\text{noWaW}}$ is defined as follows:

$$\begin{aligned}
s(\widehat{\text{noRaW}}) &= \{(ls, rLs, wLs, futRLS) \mid s = (\sigma, \widehat{acc}_1 \circ \dots \circ \widehat{acc}_n), \\
&\quad rLs, wLs, futRLS \in \mathcal{D}^{\text{LocSet}} \text{ such that } ls \cap (wLs \cup \downarrow_W \widehat{acc}_n) \cap futRLS = \emptyset \text{ and} \\
&\quad \text{if } n > 1 : (ls, \emptyset, \emptyset, futRLS \cup rLs \cup \downarrow_R \widehat{acc}_n) \in s'(\widehat{\text{noRaW}}) \text{ with}
\end{aligned}$$

$$s' = (\sigma, \widehat{acc}_1 \circ \cdots \circ \widehat{acc}_{n-1}))$$

$$\begin{aligned} s(\widehat{noRaW}) &= \{(ls, wLs, futWLS) \mid s = (\sigma, \widehat{acc}_1 \circ \cdots \circ \widehat{acc}_n), wLs = \downarrow_W \widehat{acc}_n, \\ &\quad wLs \text{ and } futWLS \in \mathcal{D}^{LocSet} \text{ such that } ls \cap wLs \cap futWLS = \emptyset \text{ and} \\ &\quad \text{if } n > 1 : (ls, \downarrow_W \widehat{acc}_n, futWLS \cup wLs) \in s'(\widehat{noRaW}) \text{ with} \\ &\quad s' = (\sigma, \widehat{acc}_1 \circ \cdots \circ \widehat{acc}_{n-1}))\} \end{aligned}$$

The second and third argument of $\widehat{noRaW}$ are the read and write accesses encountered so far, the fourth argument accumulates the read accesses in the already processed iterations in the recursive step.

Example 14 Consider the states

$$\begin{aligned} s_1 &= (\sigma_1, \langle Read(\{(a, 0)\}) \rangle \circ \langle Write(\{(a, 0)\}) \rangle \circ \langle \rangle) \text{ and} \\ s_2 &= (\sigma_2, \langle Write(\{(a, 0)\}) \rangle \circ \langle Read(\{(a, 0)\}) \rangle \circ \langle \rangle) \end{aligned}$$

both of which just completed an iteration.

To determine that there is no read-after-write on array element $a[0]$, i.e. formula $\widehat{noRaW}(a[0], \emptyset, \emptyset, \emptyset)$ is satisfied in s_i , we establish $(\{(a, 0)\}, \emptyset, \emptyset, \emptyset) \in s_i$.³ As the final memory access sequence is empty, the conditions before the recursive step in the semantics are obviously satisfied. For the first recursive step, we check

$$(\{(a, 0)\}, \emptyset, \emptyset, \emptyset) \in s'_1(\widehat{noRaW})$$

with $s'_1 = (\sigma_1, \langle Read(\{(a, 0)\}) \rangle \circ \langle Write(\{(a, 0)\}) \rangle)$. This is straightforward and it remains to check

$$(\{(a, 0)\}, \emptyset, \emptyset, \emptyset) \in s''_1(\widehat{noRaW})$$

with $s''_1 = (\sigma_1, \langle Read(\{(a, 0)\}) \rangle)$. This holds as $\{(a, 0)\} \cap (\emptyset \cup \emptyset) \cap \emptyset = \emptyset$.

For state s_2 we check instead in the final recursive step that

$$(\{(a, 0)\}, \emptyset, \emptyset, \{(a, 0)\}) \in s''_2(\widehat{noRaW})$$

with $s''_2 = (\sigma_2, \langle Write(\{(a, 0)\}) \rangle)$ which does not hold, because $\{(a, 0)\} \cap (\emptyset \cup \{(a, 0)\}) \cap \{(a, 0)\} = \{(a, 0)\} \neq \emptyset$.

In the updates we need to represent the $\circ$ symbol syntactically that indicates the start of a new iteration (i.e., the end of the previous iteration) to group memory access updates correctly. This is done with a “marker” update $\backslash next$, between which the memory access updates for a single iteration are grouped.

Definition 16 (*Marker Update Semantics*) The semantics of marker update $\backslash next$ is defined as

$$val_{\mathcal{K}, s, \beta}(\backslash next) = \{(s, s') \mid s' = (\sigma, \widehat{Acc} \circ \langle \varepsilon \rangle) \text{ with } s = (\sigma, \widehat{Acc})\}$$

and thus starts a new iteration by appending a new empty sequence of memory accesses to $\widehat{Acc}$.

³ In the rewrite rules below we also need other arguments than $\emptyset$.

Inter-Iteration Dependence Predicates

$$\begin{aligned}
s(\widehat{\text{noRaWHist}}) &= \{(ls, label) \mid s = (\sigma, \widehat{Acc}), \widehat{Acc} = \widehat{acc}_1 \circ \dots \circ \widehat{acc}_n, \\
&\quad 0 \leq label < n, \text{ and for all } 0 < i < j \leq n - label \downarrow_R \widehat{acc}_j \cap \downarrow_W \widehat{acc}_i \cap ls = \emptyset\} \\
&\quad (\widehat{\text{noWaRHist}}, \widehat{\text{noWaWHist}} \text{ are defined analogously}) \\
s(\widehat{\text{noRaTHist}}) &= \{(ls, label) \mid s = (\sigma, \widehat{Acc}), \widehat{Acc} = \widehat{acc}_1 \circ \dots \circ \widehat{acc}_n, \\
&\quad 0 \leq label < n, \text{ and for all } 0 < i \leq n - label \downarrow_R \widehat{acc}_i \cap ls = \emptyset\} \\
s(\widehat{\text{rPred}}) &= \{(ls, label) \mid s = (\sigma, \widehat{Acc}), \widehat{Acc} = \widehat{acc}_1 \circ \dots \circ \widehat{acc}_n, \\
&\quad 0 \leq label < n, ls = \downarrow_R \widehat{acc}_{n-label}\}
\end{aligned}$$

Semantics of Update $\backslash\text{next}'$

$$val_{\mathcal{K}, s, \beta}(\backslash\text{next}') = s' \text{ with } s' = (\sigma, \widehat{Acc} \circ \langle \varepsilon \rangle) \text{ and } s = (\sigma, \widehat{Acc})$$

Fig. 8 Semantics for selected inter-iteration history data dependence predicates and update $\backslash\text{next}'$

During loop invariant generation (not appearing in the finally generated loop invariants), we need predicates to express up to which iteration a given data dependence property holds. Hence, we need to define inter-iteration versions of the history predicates (see Sect. 2.4 and [68]) as well as the corresponding read and write predicates. These predicates are $\widehat{\text{noRaWHist}}$, $\widehat{\text{noWaRHist}}$, $\widehat{\text{noWaWHist}}$, $\widehat{\text{noRaTHist}}$, $\widehat{\text{noWaTHist}}$, as well as $\widehat{\text{rPred}}$ and $\widehat{\text{wPred}}$. All have the type signature $\text{LocSet} \times \text{int}$. For example, $\widehat{\text{noRaWHist}}(ls, i)$ expresses that no read-after-write access occurs on the locations in ls up to the i -th preceding iteration. Their semantics is summarized in Fig. 8 together with that of the $\backslash\text{next}'$ update, which is needed to shift $\circ$ (cf. Sect. 2.3).

3.2 Sequent Calculus

The loop invariant generation algorithm for JavaDL_{Dep} can now be reused for $\text{JavaDL}_{Dep}^{\widehat{}}$. It is merely necessary to (i) replace the unwindLoop rule by a version that also marks the start of a new iteration, (ii) a shifting rule for the new kind of updates, as well as (iii) axiomatizations and simplification rules for the new data inter-dependence predicates. The modified unwindLoop rule

$$\text{markedUnwindLoop} \frac{\Gamma \Longrightarrow \{u\} \{ \backslash\text{next} \} [\text{if } (b) \{s; \text{while } (b) \{s\} r; \} \phi, \Delta]}{\Gamma \Longrightarrow \{u\} [\text{while } (b) \{s\} r; \} \phi, \Delta}$$

marks the beginning of the next loop iteration using a $\backslash\text{next}$ -update.

We need to be able to shift the $\backslash\text{next}$ -update, hence, we replace the previous shift rule with

$$\begin{array}{c}
\text{shiftNextUpdate} \\
\frac{\overbrace{\{su' \parallel \dots; \backslash W'(wls_1); \dots; \backslash R'(rIs_1); \backslash\text{next}'\} \Gamma,}^u \\
\quad \text{rPred}(\bigcup_{i=1..n} rIs_i, 0), \text{wPred}(\bigcup_{j=1..m} wls_j, 0) \Longrightarrow [\text{while } (b) \{s\} r; \} \phi, \{u\} \Delta]}{\Gamma \Longrightarrow \{su' \parallel \backslash\text{next}; \backslash R(rIs_1); \dots; \backslash W(wls_1); \dots\} [\text{while } (b) \{s\} r; \} \phi, \Delta}
\end{array}$$

$$\begin{aligned}
& \text{readAccessOnInterIterationNoRaW} \\
& \quad \overbrace{\{\backslash\text{next}, \backslash\text{R}(\text{loc})\}}^a \widehat{\text{noRaW}}(ls, rLs, wLs, futRLS) \rightsquigarrow \\
& \quad \{\backslash\text{next}\} \widehat{\text{noRaW}}(\{a\}ls, (\{a\}rLs) \cup \text{loc}, \{a\}wLs, \{a\}futRLS) \\
& \text{writeAccessOnInterIterationNoRaW} \\
& \quad \overbrace{\{\backslash\text{next}, \backslash\text{W}(\text{loc})\}}^a \widehat{\text{noRaW}}(ls, rLs, wLs, futRLS) \rightsquigarrow \\
& \quad \{\backslash\text{next}\} \widehat{\text{noRaW}}(\{a\}ls, \{a\}rLs, (\{a\}wLs) \cup \text{loc}, \{a\}futRLS) \\
& \text{checkNoRaWForIteration} \\
& \quad \{u, \backslash\text{next}\} \widehat{\text{noRaW}}(ls, rLs, wLs, futRLS) \rightsquigarrow (\{u, \backslash\text{next}\}(ls \cap wLs \cap futRLS = \emptyset)) \wedge \\
& \quad \{u\} \widehat{\text{noRaW}}(\{\backslash\text{next}\}ls, \emptyset, \emptyset, \{\backslash\text{next}\}futRLS \cup \{\backslash\text{next}\}rLs) \\
& \text{checkNoRaWForLastIteration} \\
& \quad \{\backslash\text{next}\} \widehat{\text{noRaW}}(ls, rLs, wLs, futRLS) \rightsquigarrow \{\backslash\text{next}\}(ls \cap wLs \cap futRLS = \emptyset)
\end{aligned}$$

Fig. 9 Selected rules for inter-dependence updates and predicates

Here, su is a state update not containing memory access updates, and $\backslash\text{R}(rLs_1), \dots, \backslash\text{W}(wLs_1), \dots$ is a sequence of n read and m write updates ($n, m \in \mathbb{N}_0$). The semantics of the while-statement implies that rule `shiftNextUpdate` can only be applied once a loop iteration is complete, but the next one has not yet started. Shifting the preceding updates is equivalent to replacing all formulas in the sequent by their strongest postcondition. This is achieved by applying an update consisting of the corresponding renaming updates on each of the sequent formulas. Predicates $\text{rPred}(\bigcup_{i=1..n} rLs_i)$ and $\text{wPred}(\bigcup_{j=1..m} wLs_j)$ preserve the information about the read and write memory accesses in the current iteration of the post-state.

As a result of introducing new predicates ($\widehat{\text{noRaW}}$, $\widehat{\text{noWaR}}$, and $\widehat{\text{noWaW}}$) and updates ($\backslash\text{next}$ and $\backslash\text{next}'$), we need to define sequent calculus rules for application of updates on these predicates. The most important ones are shown in Fig. 9. Intuitively, the top two rules accumulate the read and write accesses of the current iteration, while rule `checkNoRaWForIteration` performs the emptiness check before performing the recursive step in case there are further iterations.

Example 15 We prove the following sequent, the loop body s is the one of program `arraySum` in Listing 1.

$$\frac{\widehat{\text{noRaW}}(a[0..a.\text{length}-1], \emptyset, \emptyset, \emptyset) \rightsquigarrow}{\{i := 0 \mid \text{sum} := 0\} [\text{while } (i < a.\text{length}) \{s\}] \widehat{\text{noRaW}}(a[0..a.\text{length}-1], \emptyset, \emptyset, \emptyset)}$$

First unwind the loop by applying rule `markedUnwindLoop`, continue on the branch where the loop guard is true and symbolically execute the loop iteration until we are left with the sequent

$$\begin{aligned}
& \widehat{\text{noRaW}}(a[0..a.\text{length}-1], \emptyset, \emptyset, \emptyset) \Longrightarrow \\
& \quad \{u \mid \backslash\text{next}, \backslash\text{R}(\underline{b[0]}), \backslash\text{R}(\underline{c[0]}), \backslash\text{W}(\underline{a[0]}), \backslash\text{R}(\underline{a[0]})\} \\
& \quad [\text{while } (i < a.\text{length}) \{s\}] \widehat{\text{noRaW}}(a[0..a.\text{length}-1], \emptyset, \emptyset, \emptyset)
\end{aligned}$$

where update u contains the state updates that are not of interest here. We shift the updates and obtain the following sequent (u' denotes the renaming update corresponding to u):

$$\begin{aligned}
& \{u' \mid \backslash\text{next}', \backslash\text{R}'(\underline{b[0]}), \backslash\text{R}'(\underline{c[0]}), \backslash\text{W}'(\underline{a[0]}), \backslash\text{R}'(\underline{a[0]})\} \widehat{\text{noRaW}}(a[0..a.\text{length}-1], \emptyset, \emptyset, \emptyset), \\
& \text{rPred}(\underline{a[0]}), \dots \Longrightarrow [\text{while } (i < a.\text{length}) \{s\}] \widehat{\text{noRaW}}(a[0..a.\text{length}-1], \emptyset, \emptyset, \emptyset)
\end{aligned}$$

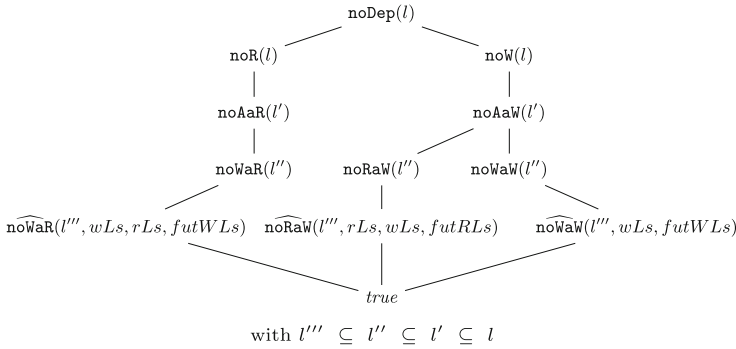


Fig. 10 Partial View on Predicate Abstraction Lattice for Inter-iteration Data Dependence Predicates

After applying the update simplification rules we obtain a first-order formula over inter-iteration predicates that can be used in the loop invariant generation algorithm explained in the subsequent section.

The final set of rules is needed to reason about subsumption relations between data dependence predicates. Subsumption relations are shown in Fig. 10. For example, if there is a no-read-after-write data dependence on memory location set l'' , there cannot be such a dependence on any memory locations l''' with $l''' \subseteq l''$. Neither inside a loop iteration nor between different iterations like $\widehat{\text{noRaW}}$. Rewrite rules such as the ones listed below, formalise this lattice in our calculus:

$$\begin{array}{l}
 \text{noRSubsumption} \quad \widehat{\text{noR}}(ls) \rightsquigarrow \widehat{\text{noRaHist}}(ls, lb) \\
 \text{interNoRaWSubsumption} \quad \widehat{\text{noRaW}}(ls) \rightsquigarrow \widehat{\text{noRaWHist}}(ls, lb) \\
 \text{interNoRaWLocSetSubsumption} \\
 \frac{\Gamma, \widehat{\text{noRaWHist}}(loc, label), label \leq lb \implies \widehat{\text{noRaWHist}}(ls \setminus loc, lb), \Delta \quad \Gamma, \widehat{\text{noRaWHist}}(loc, label), label > lb \implies \widehat{\text{noRaWHist}}(ls, lb), \Delta}{\Gamma, \widehat{\text{noRaWHist}}(loc, label) \implies \widehat{\text{noRaWHist}}(ls, lb), \Delta}
 \end{array}$$

Shifting utilizes the predicates $\widehat{\text{rPred}}$ and $\widehat{\text{wPred}}$ to capture knowledge about the post-state. This information is used by our calculus with rules like:

$$\begin{array}{l}
 \text{interReadPredANDNoRaHistSameLabel} \\
 \frac{\Gamma, \widehat{\text{rPred}}(loc, lb), \widehat{\text{noRaHist}}(ls, lb), loc \cap ls \doteq \emptyset \implies \phi, \Delta}{\Gamma, \widehat{\text{rPred}}(loc, lb), \widehat{\text{noRaHist}}(ls, lb) \implies \phi, \Delta}
 \end{array}$$

$$\begin{array}{l}
 \text{interReadPredANDNoRaHistRightSameLabel} \\
 \frac{\Gamma, \widehat{\text{rPred}}(loc, lb) \implies loc \cap ls \doteq \emptyset \wedge \widehat{\text{noRaHist}}(ls, lb + 1), \Delta}{\Gamma, \widehat{\text{rPred}}(loc, lb) \implies \widehat{\text{noRaHist}}(ls, lb), \Delta}
 \end{array}$$

Rule `interReadPredANDNoRaHistSameLabel` derives from the assumptions that there was no read on location set loc and in the same iteration locations in loc have been read, the fact that loc and ls do not share any memory locations. Rule `interReadPredANDNoRaHistRightSameLabel` simplifies the proof obligation that

there is no read on ls at the same iteration that loc is read, by splitting it into two conjuncts: The first conjunct expresses that loc and ls do not share a common memory location and the second conjunct then only requires to prove that there was no read up to (and including) the preceding iteration $lb + 1$.

4 Nested Loop Invariant Generation

The data dependence loop invariant generation technique developed so far supports simple loops. In this section, we extend the data dependence program logic to generate loop invariants for nested loops. To simplify the presentation, we base the description on intra-iteration dependences, as discussed in Sect. 2.4. For inter-iteration dependences, see Appendix B.

In our application domain HPC, nested loops are executed over multi-dimensional arrays. We show how to support programs that contain doubly nested loops over two-dimensional arrays. The generalization to more deeply nested loops on N -dimensional ($N > 2$) arrays is a straightforward extension and omitted from the presentation.

We outline the steps required to generate data dependence loop invariants: The first step is to provide an initial set of abstract predicates. Following the technique described in Sect. 2.4, we initialize the data dependence predicate set with the upper bound of the abstraction lattice in Fig. 6a.

In the second step, the nested loop for which the loop invariant is generated is unwound and symbolically executed by applying rule `unwindLoop` (see Sect. 2.3). Symbolic execution of the outer loop eventually reaches the inner loop. To symbolically execute the inner loop (and continue with the statements after the loop), unwinding the inner loop is not a viable option.

Instead, we restart the loop invariant generation process for the inner loop to compute a loop invariant L_{Dep}^{inner} that describes its data dependences (and other properties) as precisely as possible. Afterwards, we use the computed loop invariant L_{Dep}^{inner} to approximate the effect of the inner loop by making use of the third premise of rule `loop_inv` (see Sect. 2.3) for the inner loop.

This eliminates the inner loop, and symbolic execution of the outer loop body continues with the remaining statements. After this outline, we describe now the technical details.

4.1 Multi-dimensional Arrays

To support multi-dimensional arrays one needs to propagate, to maintain, and to simplify multi-dimensional symbolic location sets. In the one-dimensional case, location sets for an array a take the form of finite unions of contiguous segments $[i..j]$ for some $0 \leq i \leq j \leq a.length-1$. These must be generalized to finite unions of N -dimensional rectangles:

$$[i_1..j_1] \times \cdots \times [i_N..j_N] \text{ where } 0 \leq i_k \leq j_k \leq a[0].length-1 \text{ for all } 1 \leq k \leq N$$

The challenge is to achieve a sufficient level of automation for solving such constraints. To this end, we implemented support for automated constraint solving over multi-dimensional location sets.

4.2 Generation Algorithm

Computation of the outer and inner loop invariants are closely interconnected. Every time we reach the inner loop during the computation of the outer loop invariant, we generate a new loop invariant for the inner loop.

This approach takes into account the changes that occur when we unwind the outer loop as a precondition for generating the inner loop invariant. It provides significant precision, especially for loops that are not perfectly nested. (Commonly, loops with no statements between the outer and inner loop are referred to as *perfectly nested*.) However, it is computationally expensive.

A copy of the current set of predicates that describe the outer loop is used as the initial set of predicates for generating the inner loop invariant. We explain Algorithm 1 along the example in Listing 2.

```

function invariantGenerator
  input : Sequent  $seq : pre \implies [loop] post$ , Data Dependence Predicate set  $D_{init}$ 
  output: Loop Invariant
   $D_{old}, D_{ref} \leftarrow D_{init}, \emptyset$ ;
  while  $D_{old} \neq D_{ref}$  do
     $seq \leftarrow$  Apply rule unwindLoop on  $seq$ ;
     $seq \leftarrow$  Symbolic execution on  $seq$  and merging branches below  $seq$ ;
    /*  $seq$  has the form  $pre' \implies \{u\}[innerLoop; rest; loop]post$ , where  $rest$ 
       is the code after the inner loop and  $u$  is the result of
       symbolically executing the statements between the outer and
       the inner loop */
    if ( $innerLoop; rest$  not empty) then
      /* inner loop is present */
       $seq_{inner} \leftarrow pre' \implies \{u\}[innerLoop] true$ ;
       $L_{inner} \leftarrow invariantGenerator(seq_{inner}, D_{ref})$ ;
       $seq \leftarrow pre' \wedge \{u'\}L_{inner} \implies \{u'\}[rest; loop]post$ ;
      /*  $seq \leftarrow$  Premise use case after applying rule loop_inv on  $seq$ 
         with  $L_{inner}$  */
       $seq \leftarrow$  Symbolic execution on  $seq$  and merging branches below  $seq$ ;
    end
     $seq \leftarrow$  Apply shift update rules on  $seq$ ;
     $D_{ref}, D_{old} \leftarrow refine(seq, D_{old}), D_{ref}$ ;
     $seq \leftarrow (\bigwedge D_{ref} \implies [loop] post)$ ;
  end
  return  $\bigwedge D_{ref}$ ;

```

Algorithm 1: Data Dependence Loop Invariant Generation Algorithm for Nested Loops

4.2.1 Computation of the Inner Loop Invariant

To compute the inner loop invariant the data dependence loop invariant generation algorithm is run on the inner loop for each iteration of the outer loop. We do not use the postcondition during the loop invariant generation process, so we simply take the formula *true* as postcondition.

```

1  i = 0; j = 0;
2  while (i < N) {
3    j = 0;

```

```

4   while (j < M) {
5       a[i][j] = a[i][j] + 1;
6       j++;
7   }
8   i++;
9 }

```

Listing 2 Increment Elements of a Two-dimensional Array

Example 16 Listing 2 shows an imperfect nesting of loops with the assumption that the array object is not null and no access happens outside the array bounds. After the first unwind of the outer loop (line 2) and symbolically executing its body, we reach the inner loop (line 4). The sequent used as input for the algorithm to generate the inner loop invariant is:

$$\begin{aligned}
 & a.length \geq N - 1, a[0].length \geq M - 1, a \neq \text{null}, \\
 & i \doteq 0, j \doteq 0, \text{noDep}(\underline{a[0..a.length-1][0..a[0].length-1]}), i < N \implies \\
 & \quad [\text{while } (j < M) \{ a[i][j] = a[i][j] + 1; j++; \}] \text{true}.
 \end{aligned}$$

The generated loop invariant for the inner loop is:

$$\begin{aligned}
 & \text{noR}(\underline{a[i..a.length-1][j..a[0].length-1]}) \wedge \\
 & \text{noW}(\underline{a[i..a.length-1][j..a[0].length-1]}) \wedge \\
 & \text{noRaW}(\underline{a[0..a.length-1][0..a[0].length-1]}) \wedge \\
 & \text{noWaW}(\underline{a[0..a.length-1][0..a[0].length-1]}) \wedge \\
 & i \doteq 0 \wedge i < N \wedge 0 \leq j \leq M \wedge \\
 & a \neq \text{null} \wedge a.length \geq N - 1 \wedge a[0].length \geq M - 1.
 \end{aligned}$$

4.2.2 Using the Inner Loop Invariant

After computation of the inner loop invariant L_{inner} we return to the generation of the outer loop invariant where symbolic execution stopped at the beginning of the inner loop. Using L_{inner} , we apply the loop invariant rule `loop_inv` (see Sect. 2.3) on the inner loop. The rule splits the proof in three branches. The first two premises ensure correctness of the loop invariant, which we can safely ignore, because our approach guarantees the generation of correct loop invariants.⁴ We continue on the branch opened by the third premise, called *use case*, where symbolic execution commences with the first statement after the inner loop.

Example 17 Continuing Example 16, the use case branch of rule `loop_inv` is

$$\begin{aligned}
 & \{u\}\{v\}(\text{noR}(\underline{a[i..a.length-1][j..a[0].length-1]}) \\
 & \quad \wedge \text{noW}(\underline{a[i..a.length-1][j..a[0].length-1]}) \\
 & \quad \wedge \text{noRaW}(\underline{a[0..a.length-1][0..a[0].length-1]}) \\
 & \quad \wedge \text{noWaW}(\underline{a[0..a.length-1][0..a[0].length-1]}) \\
 & \quad \wedge i \doteq 0 \wedge i < N \wedge a \neq \text{null} \wedge a.length \geq N - 1 \wedge a[0].length \geq M - 1 \\
 & \quad \wedge j \doteq M) \implies \{u\}\{v\}[i++; \text{while}(i < N) \dots] \text{post}
 \end{aligned}$$

⁴ Even otherwise, they could be ignored, because they are checked once the outer loop invariant is available.

where update u are the current symbolic values resulting from executing the loop in an arbitrary iteration and v is a suitable anonymizing update.

The conjunction of $0 \leq j \leq M$ from the inner loop invariant, and the negation $j \geq M$ of the inner loop guard gives $j \doteq M$.

As mentioned in Sect. 2.3, applying the loop invariant rule requires to use an anonymizing update, denoted by v , to “forget” the part of the pre-state that might have been changed by the loop. However, in the data dependence program logic, simply anonymizing the values of memory locations is insufficient because the loop may also perform memory accesses. Therefore, the anonymizing update v must also anonymize these accesses. Anonymizing memory accesses differs from anonymizing the memory state because accesses are not recorded as the value of a variable but rather as memory access updates. The concept is to introduce an anonymizing memory access update that represents an unknown sequence of arbitrary read or write memory accesses (including the possibility of no access). In the following section we provide the theoretical foundations of anonymizing memory access updates.

4.3 Anonymization of Memory Access Updates

For defining the semantics of anonymized memory access updates, we need a helper function:

Definition 17 (*Sequence of anonymized memory accesses updates*) The function $anonAcc : \mathbb{Z} \rightarrow \text{MemoryAccess}^*$ gives a sequence of (possibly empty) unknown memory accesses for the specified length argument.

We can now define syntax and semantics of anonymized memory access updates.

Definition 18 (*Anonymized memory access update*) The syntax of an *anonymized memory access update* is defined by extension of the update grammar rule

$$\langle \text{update} \rangle ::= \dots \mid \backslash \text{anonAccess}^*(' \langle \text{term} \rangle ')'$$

where the argument term is of type int .

Its *semantics* is defined as

$$\text{val}_{\mathcal{K},s,\beta}(\backslash \text{anonAccess}(id)) = s' \text{ where } s' = (\sigma, Acc') \text{ with } s = (\sigma, Acc) \text{ and } \\ Acc' = Acc \circ \text{anonAcc}(\text{val}_{\mathcal{K},s,\beta}(id))$$

To use anonymizing memory updates in the calculus, we need to be able to find out the length of an anonymized sequence of memory access updates.

Definition 19 (*Length of anonymized sequence of memory accesses updates*) The length of a sequence of anonymized memory access updates is given by $\text{anonLength} : \text{int} \rightarrow \text{int}$. Let id denote a term of type int , then

$$\text{val}_{\mathcal{K},s,\beta}(\text{anonLength}(id)) = n \text{ where } \text{anonAcc}(\text{val}_{\mathcal{K},s,\beta}(id)) = acc_1 \circ \dots \circ acc_n$$

Example 18 For the sequent in Example 17 the anonymizing update v is specified as

$$v := v' \parallel \backslash \text{anonAccess}(id)$$

where v' takes care of anonymizing the program variables and array content, id is a fresh Skolem constant of type int .

$$\begin{array}{l}
\text{shiftAnonAcc} \frac{\{\backslash\text{anonAccess}'(id)\}\Gamma \Longrightarrow \phi, \{\backslash\text{anonAccess}'(id)\}\Delta}{\Gamma \Longrightarrow \{\backslash\text{anonAccess}(id)\}\phi, \Delta} \\
\\
\text{anonAccApp} \\
\quad \{\backslash\text{anonAccess}(id)\}\text{noRHist}(loc, lb) \rightsquigarrow \\
\quad \text{noRHist}(\{\backslash\text{anonAccess}(id)\}loc, \{\backslash\text{anonAccess}(id)\}i - \text{anonLength}(id)) \\
\\
\text{renamedAnonAccApp} \\
\quad \{\backslash\text{anonAccess}'(id)\}\text{noRHist}(loc, lb) \rightsquigarrow \\
\quad \text{noRHist}(\{\backslash\text{anonAccess}'(id)\}loc, \{\backslash\text{anonAccess}'(id)\}i + \text{anonLength}(id))
\end{array}$$

Fig. 11 A selection of calculus rules for (renamed) anonymized memory access update

As before, to enable shifting (“priming”), we need an update to compute the strongest postcondition in presence of an unknown sequence of memory accesses.

Definition 20 (*Renamed anonymized memory access update*) The *syntax* of a *renamed anonymized memory access update* is:

$\langle \text{update} \rangle ::= \dots \backslash\text{anonAccess}'(' \langle \text{term} \rangle ')$

where the argument term is of type `int` and the *semantics* is as follows:

$$\begin{array}{l}
\text{val}_{\mathcal{K},s,\beta}(\backslash\text{anonAccess}'(id)) = s' \text{ where } s' = (\sigma, \text{Acc}') \text{ with } s = (\sigma, \text{acc}_1 \circ \dots \circ \text{acc}_n) \text{ and} \\
\text{Acc}' = \begin{cases} \text{acc}_1 \circ \dots \circ \text{acc}_{n-\text{val}_{\mathcal{K},s,\beta}(\text{anonLength}(id))} & \text{if } \text{val}_{\mathcal{K},s,\beta}(\text{anonLength}(id)) < n \\ \varepsilon & \text{otherwise} \end{cases}
\end{array}$$

A selection of the sequent calculus rules designed for the new update kinds $\backslash\text{anonAccess}(id)$ and $\backslash\text{anonAccess}'(id)$ is shown in Fig. 11. We explain them briefly.

- Rule `shiftAnonAcc`, like previous shifting rules, achieves shifting for anonymized memory access updates, i.e. it replaces all formulas in the sequent by their strongest postcondition. In contrast to the previous shifting rules, it does not add information on the memory locations accessed in the anonymized update, because these are unknown.
- Rule `anonAccApp` applies the anonymized memory access update $\backslash\text{anonAccess}(id)$ to a formula $\text{noRHist}(loc, lb)$, which expresses that there was no read on any location in ls , except possibly for the preceding i accesses. Applying the update modifies the number of preceding accesses to the shorter sequence of accesses in the pre-state. This means instead of going back i steps, we only need to go back $lb - \text{anonLength}(id)$ steps.
- The converse rule `renamedAnonAccApp` for a renaming update $\backslash\text{anonAccess}'(id)$, instead of decreasing the number of steps going back, increases them by the number of the memory accesses represented by the anonymized update.

To handle inter-iteration dependences (Sect. 3) it is necessary to extend the theory of anonymization of memory access updates to anonymized *sequences* of memory access updates. While being relatively straightforward, the details are rather technical and can be found in Appendix B.

Example 19 Our approach generates the following inter-iteration data dependence invariant for Listing 2 under the assumption that the array object is not null and no access happens

outside of the array bounds. The invariant guarantees that the nested loop can be completely parallelized in a sound manner.

$$\begin{aligned}
 & \text{noR}(\underline{a[i..a.length-1][0..a[0].length-1]}) \\
 & \wedge \text{noW}(\underline{a[i..a.length-1][0..a[0].length-1]}) \\
 & \wedge \widehat{\text{noRaW}}(\underline{a[0..a.length-1][0..a[0].length-1]}) \\
 & \wedge \widehat{\text{noWaR}}(\underline{a[0..a.length-1][0..a[0].length-1]}) \\
 & \wedge \widehat{\text{noWaW}}(\underline{a[0..a.length-1][0..a[0].length-1]}) \\
 & \wedge 0 \leq i \leq N \wedge a \neq \text{null} \wedge a.length \geq N - 1 \wedge a[0].length \geq M - 1
 \end{aligned}$$

Theorem 4.1 (Soundness and Completeness of $\text{JavaDL}_{Dep}^{\widehat{}}$ Calculus) *The calculus rules of $\text{JavaDL}_{Dep}^{\widehat{}}$ program logic are sound and complete.*

Proof See [67]. □

Corollary 4.2 (Soundness of $\text{JavaDL}_{Dep}^{\widehat{}}$ Calculus) *Since all the rules in $\text{JavaDL}_{Dep}^{\widehat{}}$ are sound, its calculus is sound.*

5 Experiments

We implemented our data dependence analysis approach on top of the deductive verification tool KeY [1, 12]. Notable additions include (i) an extension of the core logic JavaDL to JavaDL_{Dep} so as to support data dependences, in particular, memory access updates and the data dependence predicates; (ii) extending the theory of location sets by matrices and improving the existing theory by providing additional lemmas and simplification rules; (iii) improved automation by enabling the proof search strategy to select more specific lemmas when reasoning about location sets.

Like the majority of HPC applications, our evaluation focuses on loops over arrays and matrices. This section shows the result of our analysis for single loops and nested loops in Sects. 5.1 and 5.2, respectively.

Test cases are inspired by well-known HPC test suites such as Polybench [51] and NAS [5]. We focus on the computation-intensive parts of test cases from these test suites. The result shows that our approach is automatic and highly precise.

The work reported in this paper bumps our approach up from *Technology Readiness Level*⁵ (TRL) 3 to TRL 4–5 compared to our earlier [68]. We do not claim that our tool is ready to be fully integrated into a parallelization tool chain. After all, existing analysis tools in HPC have decades of partially commercial development under their hood, while our approach is relatively new. Because of the manual effort to prepare programs from test suites, the number of test cases is somewhat limited (yet representative, see Sect. 5.3).

We carried out our experiments on a laptop with a Core i7-8750H CPU @ 2.2 GHz, 6 Cores (12 logical processors, HT). The number of processors does not affect the performance of KeY as it is not parallelized. 2 GB of memory is allocated to KeY. For each side proof, the timeout is set to 5 seconds. The number of conducted proofs is an indication of the number of data dependence or comparison predicates in the intermediate steps. However, some predicates are retried for different reasons, e.g., to ensure that a fixed point is reached.

⁵ https://www.nasa.gov/wp-content/uploads/2017/12/458490main_trl_definitions.pdf.

The maximum number of steps per such side proof is 5000, unless stated otherwise. Proof statistics include generating abstract predicates and performing symbolic execution.

5.1 Single Loops

Tables 1 and 2 show the experimental results for the general data dependence loop invariant generation approach (based on [68]) and the inter-iteration data dependence loop invariant, respectively. The precondition used in all tests is:

$$\text{noDep}(\text{a}[0..a.length-1]) \wedge a \neq \text{null} \wedge a.length > N.$$

Predicate $\text{noDep}(\text{a}[0..a.length-1])$ states that the data dependence history before encountering the loop to be analyzed is not considered (i.e. no dependences exist) so that we focus on parallelizing the current loop in isolation. The remaining precondition ensures that the array is not the **null** reference and that the length of array ($a.length > N$) is sufficiently large, and hence, no out of bounds array accesses can occur. The post condition is set to true and expresses reachability.

The average time taken for generating the loop invariant over three runs is displayed in the third column. Several factors influence the time required to generate the loop invariant. For example, the number of iterations the loop invariant generation algorithm needs to reach the fixpoint is a significant factor. The fourth column (No. of Iterations) illustrates this factor. Additionally, the presence of conditionals and complex array access patterns also impact the time needed for data dependence loop invariant generation.

The generated loop invariants differ considerably in most cases when only inter-iteration dependence is taken into account. This is discussed in detail below. In each test case the time to generate the inter-iteration invariant is lower than for the data dependence loop invariant. This can be explained with the fact that less dependences are created in the former case. The number of proofs for symbolic execution and generating abstract predicates is shown in the fifth column (No. of Proofs).

The generated data dependence loop invariant is correct if it does not under-approximate data dependences and precise if it does not over-approximate. All of the generated data dependence loop invariants are correct. However, our approach can lose precision when dealing with conditionals (test cases `conditional` and `conditionalDiffAccesses1/2/-3`). The reason is that when the symbolic execution engine reaches the if-then-else, it splits the proof into two branches. To represent both proof branches within a single loop invariant we join them using the symbolic state merging framework [63]. Therefore, the imprecise data dependence loop invariant still catches all the data dependences. The imprecision is about the fact that it does not state whether the data dependence is absent in the if-branch or the else-branch. It is worth noting that the inter-iteration data dependence loop invariant for `conditional` is precise because the conditional statements in these programs only affect the intra-iteration data dependences.

Our approach loses precision when dealing with non-affine array accesses, as illustrated in Listing 16a, due to the design of sub-array generation as part of the weakening algorithm in [68]. On the other hand, unlike other state-of-the-art approaches, we have the capability to analyze such programs at all.

We discuss some test cases in detail. The program in Listing 12a increases the value of each array element by one. There is only one type of data dependence present in this loop: an intra-iteration *WaR*. Figure 12b shows the generated loop invariant. It is fully precise and correctly captures the intra-iteration *WaR* data dependence. Since it is an intra-iteration data

Table 1 Data dependence loop invariant generation for single loops

Test case	Precise	Time (s)	# Iterations	# Proofs
Listing 12a	✓	21.1	3	145
arrayIncreaseWithFunc	✓	21.1	3	145
Listing 13a	✓	23.4	3	132
Listing 14a	✓	74.7	4	161
Listing 15a	✓	32.1	3	136
conditional	×	20.9	3	118
conditionalWithDiffAccesses1	×	130.5	4	215
conditionalWithDiffAccesses2	×	26.8	3	130
conditionalWithDiffAccesses3	×	36.6	4	143
shiftArrayWithAbruptTermination	✓	28.2	3	154
stencil	✓	43.2	3	143
Listing 16a	×	57.7	3	164

Table 2 Inter-iteration data dependence loop invariant generation for single loops

Test case	Precise	Time (s)	# Iterations	# Proofs
Listing 12a	✓	16.7	3	158
arrayIncreaseWithFunc	✓	17.8	3	158
Listing 13a	✓	21.8	3	164
Listing 14a	✓	20.5	4	164
Listing 15a	✓	21.7	3	144
conditional	✓	14.0	3	112
conditionalWithDiffAccesses1	×	44.7	3	154
conditionalWithDiffAccesses2	×	23.7	3	144
conditionalWithDiffAccesses3	×	24.2	3	182
shiftArrayWithAbruptTermination	✓	16.7	3	154
stencil	✓	21.33	3	144
Listing 16a	×	20.6	4	164

dependence, its absence is recorded in the *inter*-iteration loop invariant. Figure 12c shows the fully precise inter-iteration loop invariant.

Similar to Listing 12a, Listing 13a contains a *WaR* data dependence on all elements of array *a*. Moreover, there is a read operation in line 5 that follows a write operation in line 4, which together constitute a *RaW* data dependence. Since both data dependences in Listing 13a are intra-iteration data dependences, they are both ignored by the inter-iteration loop invariant in Fig. 13c.

In Listing 14a, there is *WaR* data dependence on all elements of array *a* except for *a*[0], since it is never read. The generated loop invariant is depicted in Fig. 14b. As this data dependence is an inter-iteration data dependence, it is also present in the inter-iteration loop invariant in Fig. 14c.

Similar to Listing 14a, the program in Listing 15a contains a *WaR* data dependence. Moreover, there is a read in line 5 after the write in line 4, which together constitute a *RaW*

```

1 i = 0;
2 while (i <= N) {
3   a[i] = a[i] + 1;
4   i = i + 1;
5 }

```

(a) Array increase

$$\begin{array}{ll}
& \text{noR}(\underline{a[i \dots a.length-1]}) \\
& \wedge \text{noW}(\underline{a[i \dots a.length-1]}) \\
& \wedge \widehat{\text{noRaW}}(\underline{a[0 \dots a.length-1]}, \text{empty}, \text{empty}, \text{empty}) \\
& \wedge \widehat{\text{noWaR}}(\underline{a[0 \dots a.length-1]}, \text{empty}, \text{empty}, \text{empty}) \\
& \wedge \widehat{\text{noWaW}}(\underline{a[0 \dots a.length-1]}, \text{empty}, \text{empty}) \\
& \wedge a \neq \text{null} \wedge a.length > N \\
& \wedge i \geq 0 \wedge i \leq N+1
\end{array}
\quad
\begin{array}{l}
\text{(b) Loop invariant}
\end{array}$$

(b) Loop invariant

(c) Inter-iteration loop invariant

Fig. 12 “Array increase” example and its loop invariants

```

1 i = 0;
2 sum = 0;
3 while (i <= N) {
4   a[i] = a[i] + 1;
5   sum = sum + a[i];
6   i = i + 1;
7 }
8

```

(a) Array increase and sum

$$\begin{array}{ll}
& \text{noR}(\underline{a[i \dots a.length-1]}) \\
& \wedge \text{noW}(\underline{a[i \dots a.length-1]}) \\
& \wedge \widehat{\text{noRaW}}(\underline{a[0 \dots a.length-1]}, \text{empty}, \text{empty}, \text{empty}) \\
& \wedge \widehat{\text{noWaR}}(\underline{a[0 \dots a.length-1]}, \text{empty}, \text{empty}, \text{empty}) \\
& \wedge \widehat{\text{noWaW}}(\underline{a[0 \dots a.length-1]}, \text{empty}, \text{empty}) \\
& \wedge a \neq \text{null} \wedge a.length > N \\
& \wedge i \geq 0 \wedge i \leq N+1
\end{array}
\quad
\begin{array}{l}
\text{(b) Loop invariant}
\end{array}$$

(b) Loop invariant

(c) Inter-iteration loop invariant

Fig. 13 “Array increase and sum” example and its loop invariants

data dependence. Since the *RaW* data dependence is an intra-iteration one, it is disregarded by the inter-iteration loop invariant in Fig. 15c. Although the loop invariant precisely captures the data dependences, it fails to include predicate $\text{noR}(\underline{a[0]})$ due to the heuristic used for generating sub-arrays. Because of that we cannot claim that our approach generates a fully precise data dependence loop invariant for this test case. On the other hand, this loss

```

1 i = 0;
2 while (i <= N-1) {
3   a[i] = a[i+1];
4   i=i+1;
5 }

```

(a) Array shift

$$\begin{array}{ll}
\text{noR}(\underline{a[0]} \cup \underline{a[i+1..a.length-2]}) & \text{noR}(\underline{a[0]} \cup \underline{a[i+1..a.length-2]}) \\
\wedge \text{noW}(\underline{a[i..a.length-2]}) & \wedge \text{noW}(\underline{a[i..a.length-2]}) \\
\wedge \text{noRaW}(\underline{a[0..a.length-2]}) & \wedge \widehat{\text{noRaW}}(\underline{a[0..a.length-2]}, \text{empty}, \text{empty}, \text{empty}) \\
\wedge \text{noWaW}(\underline{a[0..a.length-2]}) & \wedge \widehat{\text{noWaW}}(\underline{a[0..a.length-2]}, \text{empty}, \text{empty}) \\
\wedge a \neq \text{null} \wedge a.length > N & \wedge a \neq \text{null} \wedge a.length > N \\
\wedge i \geq 0 \wedge i \leq N & \wedge i \geq 0 \wedge i \leq N
\end{array}$$

(b) Loop invariant

(c) Inter-iteration loop invariant

Fig. 14 “Array shift” example and its loop invariants

```

1 i = 0;
2 sum = 0;
3 while (i <= N-1) {
4   a[i] = a[i+1];
5   sum = sum + a[i];
6   i=i+1;
7 }

```

(a) Array shift and sum computation

$$\begin{array}{ll}
& \text{noR}(\underline{a[i+1..a.length-2]}) \\
\text{noR}(\underline{a[i+1..a.length-2]}) & \wedge \text{noW}(\underline{a[i..a.length-2]}) \\
\wedge \text{noW}(\underline{a[i..a.length-2]}) & \wedge \widehat{\text{noRaW}}(\underline{a[0..a.length-2]}, \text{empty}, \text{empty}, \text{empty}) \\
\wedge \text{noWaW}(\underline{a[0..a.length-2]}) & \wedge \widehat{\text{noWaW}}(\underline{a[0..a.length-2]}, \text{empty}, \text{empty}) \\
\wedge a \neq \text{null} \wedge a.length > N & \wedge a \neq \text{null} \wedge a.length > N \\
\wedge i \geq 0 \wedge i \leq N & \wedge i \geq 0 \wedge i \leq N
\end{array}$$

(b) Loop invariant

(c) Inter-iteration loop invariant

Fig. 15 Array shift and sum Listing, and its invariants

of precision does not harm the data dependence analysis goal for the example, because there are no data dependences on $\underline{a[0]}$.

Listing 16a has a non-linear array index. As the KeY deductive verifier supports non-linear operations on integers, we can, in principle, handle such array accesses. On the other hand, we cannot generate fully precise results in such cases. The reason is that our sub-array generation (part of the weakening algorithm) is designed for linear array access. It would be too expensive to generate non-linear sub-arrays.

```

i = 0;
while (i <= a.length - 1) {
  a[i] = a[i*i];
  i = i + 1;
}

```

(a) Non-linear array access

	$\text{noR}(\underline{a[i..a.length-1]})$
$\text{noR}(\underline{a[i..a.length-1]})$	$\wedge \text{noW}(\underline{a[i..a.length-1]})$
$\wedge \text{noW}(\underline{a[i..a.length-1]})$	$\wedge \widehat{\text{noRaW}}(\underline{a[0..a.length-1]}, \text{empty}, \text{empty}, \text{empty})$
$\wedge \text{noRaW}(\underline{a[0..a.length-1]})$	$\wedge \widehat{\text{noWaR}}(\underline{a[0..a.length-1]}, \text{empty}, \text{empty}, \text{empty})$
$\wedge \text{noWaW}(\underline{a[0..a.length-1]})$	$\wedge \widehat{\text{noWaW}}(\underline{a[0..a.length-1]}, \text{empty}, \text{empty})$
$\wedge a \neq \text{null} \wedge a.length > N$	$\wedge a \neq \text{null} \wedge a.length > N$
$\wedge i \geq 0 \wedge i \leq a.length$	$\wedge i \geq 0 \wedge i \leq a.length$

(b) Loop invariant

(c) Inter-iteration loop invariant

Fig. 16 “Non-linear array access” example and its loop invariants

Table 3 Data dependence loop invariant generation results for nested loops

Test case	Precise	Time (min)	No. of Iterations	No. of proofs	Proof steps
readMatrix	✓	14.7	3	2046	5000
conditionalRead	×	14.8	3	2046	5000
Listing 17a	✓	37.9	3	2389	10000
Listing 18a	✓	54.4	3	2452	10000
Listing 19a	✓	73.1	3	2769	10000
multipleArrays	✓	0.65	4	333	5000

5.2 Nested Loops

Similar to Sect. 5.1, in the following test cases, except for test case *Multiple Arrays*, the precondition is:

$$\text{noDep}(\underline{a[0..a.length-1]}) \wedge a \neq \text{null} \wedge a.length > N \wedge a[0].length > M.$$

Our approach requires matrices to have a rectangular shape, where the length of all rows is equal to $a[0].length$.

We present test cases with an imperfect nesting of loops to demonstrate that our approach can handle them, unlike tools that exclusively handle perfectly nested loops [14, 47].

Table 3 contains the outcome of our experiments with nested loops. The last column shows the maximum number of steps each proof is allowed to take. All the generated loop invariants are correct. Except for the loop invariant of *conditionalRead*, all the results are precise. However, our approach loses precision when generating the loop invariant for *conditionalRead* due to the presence of a conditional statement.

Unlike the results for single loops in Table 1, where our approach on average takes around a minute to generate results, it takes several minutes to generate data dependence loop invariants

while ($i \leq N - 1$) {	$\text{noR}(\underline{a[i..a.length-2] [0..a[0].length-2]})$
$j = 0;$	$\wedge \text{noW}(\underline{a[i..a.length-2] [0..a[0].length-2]})$
while ($j \leq M - 1$) {	$\wedge \text{noRaW}(\underline{a[0..a.length-2] [0..a[0].length-2]})$
$a[i][j] = a[i][j] + 1;$	$\wedge \text{noWaW}(\underline{a[0..a.length-2] [0..a[0].length-2]})$
$j = j+1;$	$\wedge a \neq \text{null} \wedge a.length > N \wedge a[0].length > M$
}	$\wedge i \geq 0 \wedge i \leq N \wedge j \geq 0 \wedge j \leq M$
$i = i+1;$	
}	
(a) Matrix value increase	(b) Outer loop invariant

Fig. 17 Nested loop with intra-iteration *WaR*

while ($i \leq N - 1$) {	$\text{noR}(\underline{a[i..a.length-2] [0..a[0].length-2]})$
$j = 0;$	$\wedge \text{noW}(\underline{a[i..a.length-2] [0..a[0].length-2]})$
while ($j \leq M - 2$) {	$\wedge \text{noRaW}(\underline{a[0..a.length-2] [0..a[0].length-2]})$
$a[i][j] = a[i][j + 1];$	$\wedge \text{noWaW}(\underline{a[0..a.length-2] [0..a[0].length-2]})$
$j = j+1;$	$\wedge a \neq \text{null} \wedge a.length > N \wedge a[0].length > M$
}	$\wedge i \geq 0 \wedge i \leq N \wedge j \geq 0 \wedge j \leq M-1$
$i = i+1;$	
}	
(a) Shifting row elements to left	(b) Outer loop invariant

Fig. 18 Nested loop with inter-iteration data dependence in inner loop

for nested loops over matrices (time is given in *minutes*). This is due to the presence of matrices, rather than nested loops. For example, the test case `multipleArrays` needs less than a minute to analyze a nested loop over *one-dimensional* arrays. With more effort spent on improving reasoning about multi-dimensional location sets, it would be possible to reduce the runtime, however, the important aspect is that full automation is achievable. In an offline analysis is here, the runtime is somewhat secondary.

Listing 17a shows an imperfectly nested loop with intra-iteration *WaR* data dependence in the inner loop. The outer loop invariant in Fig. 17b captures this data dependence and reveals that there is no *RaW* or *WaW* data dependence on matrix *a*. Furthermore, it demonstrates that at each iteration *i*, the array elements of row *i* and subsequent rows are not read or written.

In Listing 18a, there is an inter-iteration *WaR* data dependence in the inner loop. The generated loop invariant for the outer loop is shown in Fig. 18b and it correctly detects and describes this data dependence. The loop preserves predicates about the matrix not being null and its dimensions being lower bounds. These predicates appear in the loop invariant.

Additionally, the analysis generates precise predicates showing the lower and upper bounds of the loop indices.

In Listing 19a there exists an inter-iteration *WaR* data dependence within the outer loop. This dependency is correctly captured by the precise loop invariant generated for the outer loop shown in Fig. 19b.

5.3 Threats to Validity

We reflect on the design of our experiments and in particular the threats to the validity of the evaluation.

while (i <= N - 2) {	<u>noR(a[i+1..a.length-2] [0..a[0].length-2])</u>
j = 0;	
while (j <= M - 1) {	<u>noW(a[i..a.length-2] [0..a[0].length-2])</u>
a[i][j] = a[i+1][j];	<u>noRaW(a[0..a.length-2] [0..a[0].length-2])</u>
j = j+1;	<u>noWaW(a[0..a.length-2] [0..a[0].length-2])</u>
}	
i = i+1;	<u>a ≠ null ∧ a.length > N ∧ a[0].length > M</u>
}	<u>i ≥ 0 ∧ i ≤ N-1 ∧ j ≥ 0 ∧ j ≤ M</u>
(a) Shifting matrix rows up	(b) Outer loop invariant

Fig. 19 Nested loop with inter-iteration data dependence in outer loop

A primary *internal threat* that could impede generalization of the experimental results is that the programming language we analyse is Java, while in our application domain *High-Performance Computing* (HPC) most programs are written in C. The reason why we can still transfer our results is situated in the nature of HPC programs whose critical components (for parallelization) are loops. Pointer arithmetic typically is restricted to the indexed access of array elements. This usage scenario is benign and aligns directly with array accesses in Java. The examples used in our experiments are direct and straightforward translations from C.

Another issue to be aware of is that data dependence analysis is integrated into parallelization tools. Unfortunately, all state-of-the-art tools we looked at, do not provide statistics on their data dependence analysis results in isolation. This makes it difficult to compare our approach directly with HPC parallelization tools. In consequence, in future work we intend to integrate our analysis into a state-of-art parallelization tool, see Sect. 7.

Among the *external threats* is the relatively small number of experiments done in our evaluation. However, more important than *quantity* of the test cases is their *quality*. Many of the programs in HPC test suites [5, 51] constitute minor variations of each other. We carefully analyzed their main features and extracted those that pose a challenge for dependence analysis.

Loops with conditional statements are difficult for dynamic data dependence analysis and often cause under-approximation, while our approach ensures correctness. Over-approximation in our approach occurs due to the weakening heuristics used during abstraction and the design of the abstract domains (contiguous areas). As Example 15a shows, it is far from obvious, whether full precision pays off with respect to the final goal of parallelization. On the other hand, our approach is fully precise in *collecting* dependences during symbolic execution and, therefore, could be refined, if so desired.

We include experiments containing nested loops with row- as well with as column-wise transposition of values. Many of our test cases give substantially different results once only inter-iteration dependence is taken into account. Our test cases include non-linear array access patterns, demonstrating that our approach does not reject programs with non-linear array indices but can deal with them to a certain degree (and otherwise soundly over-approximates).

Our approach works on code sequences or loops that were isolated from a surrounding program that also has code for data setup, user interaction, visualization, etc. This is sufficient for the application scenario of code parallelization and it drastically reduces the overall time and memory usage relative to what would be required to analyze the entire code of an application program.

In our experiments we assume well-behaving programs in the sense that no runtime exceptions are triggered, for example by accessing an array out-of-bounds or a null reference. In

general, this might introduce dependencies undetected by our experimental setup. However, in the context of our application domain this is a reasonable assumption, because otherwise programs would abort at runtime (they are often C programs leading to a segmentation fault). A possible spurious data dependence is of no interest for the parallelization effort, because it will disappear when fixing the underlying issue, or else, if the fix introduced that data dependence, then it would be detected in a subsequent run of our tool. On the other hand, our approach would work perfectly well in presence of runtime exceptions, because we could simply use the state merging technique (as we do already for conditionals) to include the proof branches that contain exceptional behavior.

6 Related Work

Related work can be mainly categorized into two groups: (i) data dependence profilers (Section 6.1), and (ii) loop invariant generation (Section 6.2). Section 6.3 covers other work that investigates related problems.

6.1 Data Dependence Profiling

Detecting data dependences is an NP-hard problem [52], therefore it can only be approximated. In the literature this approximation is done statically, dynamically, or in a hybrid fashion.

Static Approaches. Static data dependence analysis approaches [2, 3, 7, 23, 45, 53–55, 65] soundly find all data dependences that can occur based on compile time information. For efficiency reasons, the analysis is over-approximated. For this reason, these approaches are prone to false positives as some data dependences might be resolved only at runtime. In addition to loss of parallelization opportunities due to over-approximation of data dependences, these techniques are limited to loops with affine linear bounds and linear array indices [7, 14, 23, 43, 45].

In contrast, our approach is based on a fully precise and faithful program logic using symbolic execution interleaved with first-order reasoning to prove the absence of data dependences. We are not limited to affine nested loops, and the loop index does not need to change linearly. Our logic is able to represent non-linear integer constraints and the calculus is able to prove some non-linear problems automatically. We have the possibility to export sub-goals to SMT solvers [9]. Because of this, we can profit from the quickly evolving capabilities regarding non-linear arithmetic in the automated reasoning area.

Dynamic Approaches.

Dynamic parallelization tools [26, 39, 40, 66, 71] use dynamic data dependence analysis techniques that capture data dependences at runtime. These analyses are susceptible to false negatives, because they rely on runtime information which may be incomplete: It is impossible to know whether the executed runs cover all dependences that may occur. Therefore, these approaches might suggest unsound parallelization based on the erroneous absence of data dependences in traversed paths. In addition, these techniques that are based on profiling have a large runtime overhead (a factor of 100 or more [50] relative to a normal run). Increasing the diversity of inputs, enlarges the set of data dependences identified by dynamic analysis tools, but also increases the overhead correspondingly.

Although our approach is static, it retains advantages of the dynamic approaches, because it uses fully precise symbolic execution. By distinguishing different symbolic execution paths we can safely use all path conditions from proven symbolic execution paths in conditional parallelization even for a failed proof attempt.

Hybrid Approaches.

Combining static and dynamic data dependence analysis approaches results in hybrid approaches with higher precision and potentially reduced runtime. Sampaio et al. [60] apply static dependence analysis and use the result for parallelization. It then generates tests to validate at runtime whether it can be taken. Rus et al. [58] use static analysis. They formulate conditions and insert them into the source code. These conditions evaluate at runtime whether a loop can be parallelized or not.

DiscoPoP [47, 50] runs dynamic analysis only for code sections where the data dependence detection requires runtime information. The framework has a lower profiling overhead than the other approaches named here. It uses the combination of the following techniques to profile data dependences in a hybrid fashion: (i) Static extraction of data dependences from loops. Subsequently, loops are excluded and profiling data dependences is only performed outside loops. In the end, static and dynamic dependences are merged. This technique is, however, limited to polyhedral loops. (ii) Static identification of data dependences among scalar variables, not including passed-by-reference parameters and pointers. Then, identify memory instructions that create the dependences and exclude them from parallelization. This technique includes profiling data dependences in polyhedral loops.⁶

Compared to the hybrid tools, our approach is not limited to specific shapes of loops or functions to produce sound results. In addition, we support multi-dimensional arrays and pointers that are left out of the analysis by the hybrid tools mentioned above.

6.2 Loop Invariant Generation

Various loop invariant generation techniques are general or generic in the sense that they do not cover data dependences, while our approach is domain specific. On the other hand, these techniques are not general enough because they come with many limitations, e.g., only covering polynomials, only allowing integers, not covering arrays, etc., while our approach supports a larger grammar [25].

Loop Invariant Generation with Predicate Abstraction

There is a vast research corpus on loop invariant generation using a variety of techniques like abstract interpretation (including shape analysis and predicate abstraction) [6, 17, 32], templates [16, 21, 22, 49], recurrence equations [31, 41], learning [64], etc. Our approach is an extension of [70] to loop invariant generation for dependence predicates. All mentioned approaches focus on *functional* properties of a loop. To the best of our knowledge, none of them has been adapted to generate loop invariants that capture (the absence of) data dependences.

The approaches used in the paper [24] and in the Boogie verifier [8] like our approach abstract a loop with predicates containing free variables. However, the first-order calculus that quantifies over these variables is separate from the calculus that infers an invariant in a proof. In consequence, there is no overarching soundness argument which relies on correct translation of proof objects between both formalisms. In contrast, our approach uses *the same* logical calculus for generating the loop invariant as for inferring loop invariants, dispensing

⁶ Loops with affine bounds and array accesses.

with the need to translate back and forth, and soundness is based on the same semantics. In addition, our approach permits fine-grained interleaving of constraint solving, predicate abstraction, and symbolic execution which is important for performance.

Loop Invariant Generation for Nested Loops

Loop invariant generation for nested loops with conditionals is rarely addressed in the literature. The method in [34] generates loop invariants only when program variables are integers and assignments to program variables are polynomial expressions. In contrast, our approach allows arrays in addition to integer variables, and assignments can include function and procedure calls on program variables.

6.3 Other Work with a Relation to Data Dependences

Information Flow

There is some resemblance between data dependences and secure information flow [59]. Indeed, both problems can be approached by deductive verification (our approach and [10, 18, 61], respectively). But whereas information flow is a relational property, data dependence is essentially a reachability problem. Also, the latter is more fine-grained, because it is sensitive to the order of memory access.

Static Analysis of Array Programs

Static analysis of array programs is another line of research close to our work. The approach in [20] formulates the problem in terms of reasoning about (partial) access permissions. They analyze the loop body to obtain a permission precondition for a single loop iteration. Permission is defined as the maximum over the variables changed by the loop execution. Unlike our approach, they only record the type of memory accesses and not their order.

Specification Techniques for Capturing Loop Data Dependences

Specification techniques for capturing loop data dependences for correct loop parallelization is another aspect of our work. Blom et. al. [13] suggest to generate iteration contracts to capture data dependences between different loop iterations. Their technique requires a user to provide part of the specification manually that our approach can generate automatically.

7 Future Work and Conclusion

Problems deriving from the HPC area, including dependence analysis, constitute a promising, but so far overlooked, opportunity for deductive verification technology which has immensely matured during the past years [29]. State-of-the-art data dependence analysis tools proposed in the HPC community suffer from over-approximation and, occasionally, under-approximation. In contrast, our approach provides highly precise and always sound results. The challenge is to make use of the power of deductive verification while staying fully automatic.

Our starting point is the formalization of common data dependence properties in an expressive program logic, including the first formally precise semantics of inter-iteration dependence. The logic is the basis for employing a deductive verification tool to reason about these properties. The central focus was on generating loop invariants for loop shapes as they typically occur in HPC applications. These invariants capture existing intra- and inter-loop iteration data dependencies with high precision. Being based on predicate abstraction, our automatic loop invariant generation is sound by design, which means there is no under-

approximation, and hence, no data dependence is missed. This enables safe parallelization based on our analysis.

Our analysis is an ideal starting point for parallelizing loop iterations (for example, applying *DoAll*, reduction, and geometric decomposition patterns [30, 33]). Not to differentiate between intra- and inter-iteration data dependences, as in our earlier work [68], is only appropriate when the goal is to parallelize the body of *each* loop iteration (for example, when applying the pipeline pattern). Otherwise, too many spurious dependences are generated.

But data dependence analysis is one step in the parallelization process. Currently, our approach is stand-alone and needs to be integrated with an existing auto-parallelizer. This will enable us to also evaluate the practical impact of our approach for parallelization.

Regarding other future work, we are investigating further refinements of the lattices (abstract domains) used in the loop invariant generation. Another line of work is to extend the reach of automation with respect to array access using non-linear indexing expressions.

In addition, we are interested in generalizing our approach to synthesize method contracts for reasoning in a modular manner about code data dependences. Specifically, it is promising to automatically generate frame conditions, as used in verification annotation languages [11, 42], that describe which part of the memory a given method reads from or writes to.

A further use of our approach may open up in the area of information flow security. There, programs are analysed for information leaks to ensure confidentiality and privacy. Determining data dependences can rule out certain leaks of confidential data.

Acknowledgements This work was partially funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the DFG projects BU 2924/3-1 and HA 2617/9-1, and by the KiKIT (The Pilot Program for Core-Informatics at the KIT) of the Helmholtz Association. We acknowledge the support of the European Commission and the German Federal Ministry of Education and Research (BMBF) under the EuroHPC Programme DEEP-SEA (GA 955606, BMBF funding No. 16HPC015), which receives support from the European Union's Horizon 2020 programme and DE, FR, ES, GR, BE, SE, GB, CH.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

Appendix A Formal Definition of Memory Access Predicates

The syntax of a *memory access predicate* is

$$\langle fml \rangle ::= \dots l \text{ rPred } ' (' \langle term \rangle, \langle term \rangle ') ' \mid w \text{ rPred } ' (' \langle term \rangle, \langle term \rangle ') '$$

where the term given as first argument denotes a location set and the second term an integer. Its semantics specifies that the i -th most recent memory access of the given kind was limited to locations ls and is defined as follows (the semantics of $w \text{ rPred}$ is analogous):

$$s(\text{rPred}) = \{(ls, i) \mid s = (\sigma, \text{Acc}), 0 \leq i < n,$$

$$\text{Acc} = \langle \text{acc}_1(ls_1) \rangle \circ \dots \circ \langle \text{acc}_{n-i}(ls_{n-i}) \rangle \circ \dots \circ \langle \text{acc}_n(ls_n) \rangle,$$

$$\text{and } \text{acc}_{n-i} = \text{Read and } ls_{n-i} = ls \}$$

Appendix B Anonymization of a Sequence of Memory Access Updates

Analogous to anonymization of memory access updates, for defining the semantics of anonymized *sequence* of memory access updates, we need a helper function:

Definition 21 (*Sequence of sequence of anonymized memory accesses updates*) The function $\widehat{anonAcc} : \mathbb{Z} \rightarrow \text{SequenceOfMemoryAccess}^*$ gives a sequence of sequences of (possibly empty) memory accesses for the specified length.

We extend Definition 14 to contain all the location sets that are read in multiple array iterations.

Definition 22 (*Extended projection*) For $itrSeq = \widehat{acc}_i \circ \dots \circ \widehat{acc}_j$ the extended projection function is defined as

$$\begin{aligned} \downarrow_R(itrSeq) &:= \bigcup_{k=i}^j \bigcup_{l=1}^{n_k} \{ls_{(k,l)} \mid \widehat{acc}_k = acc_{(k,1)} \circ \dots \circ acc_{(k,n_k)} \\ &\text{and } acc_{(k,l)} = \langle Read(ls_{(k,l)}) \rangle\} \end{aligned}$$

collects all memory locations read in loop iterations i to j . We extend the write projection function analogously:

$$\begin{aligned} \downarrow_W(itrSeq) &:= \bigcup_{k=i}^j \bigcup_{l=1}^{n_k} \{ls_{(k,l)} \mid \widehat{acc}_k = acc_{(k,1)} \circ \dots \circ acc_{(k,n_k)} \\ &\text{and } acc_{(k,l)} = \langle Write(ls_{(k,l)}) \rangle\}. \end{aligned}$$

Using the anonymized sequence of memory access updates and the extended projection functions, we define syntax and semantics of anonymized sequence of sequences of memory access updates.

Definition 23 (*Anonymized sequence of sequences of memory access update*) An *anonymized sequence of sequences of memory access update* is defined by extension of the update grammar rule:

$$\langle update \rangle ::= \dots l \setminus \widehat{anonAcc} \text{ ' } (\langle term \rangle, \langle term \rangle, \langle term \rangle) \text{ ' }$$

where the first two arguments are of type LocSet and the third term is of type int . Its *semantics* is as follows:

$$\begin{aligned} val_{\mathcal{K},s,\beta}(\setminus \widehat{anonAcc}(rLocs, wLocs, id)) &= s' \\ \text{where } s' &= (\sigma, \widehat{Acc}') \text{ with } s = (\sigma, \widehat{Acc}), \widehat{Acc}' = \widehat{Acc} \circ \widehat{anonAcc}(val_{\mathcal{K},s,\beta}(id)), \\ rLocs &= \downarrow_R \widehat{anonAcc}(val_{\mathcal{K},s,\beta}(id)), \text{ and } wLocs = \downarrow_W \widehat{anonAcc}(val_{\mathcal{K},s,\beta}(id)). \end{aligned}$$

To use $\setminus \widehat{anonAcc}$ in the calculus, we need to be able to find out the length of an anonymized sequence of sequence of memory access updates.

Definition 24 (*Length of anonymized sequence of sequence of memory accesses updates*) The *length* of anonymized sequence of sequence memory accesses updates is given by $\widehat{anonLength} : \text{int} \rightarrow \text{int}$. Let id denote a term of type int , then

$$val_{\mathcal{K},s,\beta}(\widehat{anonLength}(id)) = n \text{ where } val_{\mathcal{K},s,\beta}(\setminus \widehat{anonAcc}(id)) = \widehat{acc}_1 \circ \dots \circ \widehat{acc}_n$$

Example 20 For the sequent in Example 17 the anonymizing update v is specified as

$$v := v' \parallel \widehat{\backslash \text{anonAcc}}(\text{id})$$

where id is a fresh Skolem constant of type `int` and v' takes care of anonymizing the program variables and array content.

As before, to enable shifting (“priming”), we need an update to compute the strongest postcondition in presence of an unknown sequence of sequences of memory accesses.

Definition 25 (*Renamed anonymized sequence of sequences of memory access update*) The syntax of a renamed anonymized memory access update is:

$$\langle \text{update} \rangle ::= \dots \parallel \widehat{\backslash \text{anonAcc}}' \text{ ' } (\text{term}), \langle \text{term} \rangle, \langle \text{term} \rangle \text{ ' }$$

where the first two arguments are terms of type `LocSet` the last argument term is of type `int` and the semantics is as follows:

$$\begin{aligned} \text{val}_{\mathcal{K},s,\beta}(\widehat{\backslash \text{anonAcc}}'(rLocs, wLocs, id)) &= s' = (\sigma, \widehat{Acc}') \\ \text{with } s &= (\sigma, \widehat{acc}_1 \circ \dots \circ \widehat{acc}_n), \\ \widehat{Acc}' &= \begin{cases} \widehat{acc}_1 \circ \dots \circ \widehat{acc}_{n-\text{val}_{\mathcal{K},s,\beta}(\text{anonLength}(id))} & \text{if } \text{val}_{\mathcal{K},s,\beta}(\text{anonLength}(id)) < n \\ \varepsilon & \text{otherwise} \end{cases} \\ rLocs &= \downarrow_R \text{val}_{\mathcal{K},s,\beta}(\widehat{\backslash \text{anonAcc}}'(id)), \text{ and } wLocs = \downarrow_W \text{val}_{\mathcal{K},s,\beta}(\widehat{\backslash \text{anonAcc}}'(id)). \end{aligned}$$

Figure 20 shows a selection of the rules for the new kinds of updates $\widehat{\backslash \text{anonAcc}}(rLocs, wLocs, id)$ and $\widehat{\backslash \text{anonAcc}}'(rLocs, wLocs, id)$. We explain them briefly below:

- Rule `shiftltrAnonAcc`, like other shifting rules, realizes shifting for an anonymized sequence of memory access updates. In other words, it replaces all formulas in the sequent with their strongest postcondition. Similar to `shiftAnonAcc`, it does not add information on the memory locations accessed in the sequence of anonymized updates, as these are unknown.
- Rule `itrAnonAccApp` applies an anonymized sequence of memory access updates, namely, $\widehat{\backslash \text{anonAcc}}(rLocs, wLocs, id)$, to formula $\widehat{\text{noRaW}}(ls, rLs, wLs, \text{futRLS})$. This formula expresses that there is no read-after-write data dependence on any location in ls ; at the current iteration s location sets rLs and wLs have been read and written, respectively. The processed loop iterations up to the current one have read futRLS . The update has to be applied on all arguments to ensure soundness. Location sets $rLocs$ and $wLocs$ are added to the location sets rLs and wLs that are read and written in the current iteration of the loop.
- Rule `itrAnonAccAppOnHist` applies the anonymized sequence of memory access updates to a formula $\widehat{\text{noRaTHist}}(loc, lb)$, which expresses that there was no read on any location in loc , except possibly for the preceding lb iterations. Applying the update modifies the number of preceding iterations to a shorter sequence in the pre-state. This means instead of going back lb iterations, we only need to go back $lb + \text{anonLength}(id)$ iterations.
- Rule `renamedltrAnonAccApp` acts opposite of rule `itrAnonAccApp` by eliminating $rLocs$ and $wLocs$ from location sets rLs and wLs .
- Rule `renamedltrAnonAccAppOnHist` has the opposite effect of `itrAnonAccAppOnHist`. Instead of decreasing the number of steps going back, we increase them by the number of iterations represented by $\text{anonLength}(id)$.

$$\begin{aligned}
&\text{shiftltrAnonAcc} \\
&\frac{\{\widehat{\text{anonAcc}}'(rLocs, wLocs, id)\} \Gamma \Longrightarrow \phi, \{\widehat{\text{anonAcc}}'(rLocs, wLocs, id)\} \Delta}{\Gamma \Longrightarrow \{\widehat{\text{anonAcc}}(rLocs, wLocs, id)\} \phi, \Delta} \\
&\text{itrAnonAccApp} \\
&\frac{\{\widehat{\text{anonAcc}}(id)\} \widehat{\text{noRaW}}(ls, rLs, wLs, futRLS) \rightsquigarrow}{\widehat{\text{noRaW}}(\{\widehat{\text{anonAcc}}(rLocs, wLocs, id)\} ls, (\{\widehat{\text{anonAcc}}(rLocs, wLocs, id)\} rLs) \cup rLocs,} \\
&\quad (\{\widehat{\text{anonAcc}}(rLocs, wLocs, id)\} wLs) \cup wLocs, futRLS) \\
&\text{itrAnonAccAppOnHist} \\
&\frac{\{\widehat{\text{anonAcc}}(id)\} \widehat{\text{noRaTHist}}(loc, lb) \rightsquigarrow}{\widehat{\text{noRaTHist}}(\{\widehat{\text{anonAcc}}(rLocs, wLocs, id)\} loc,} \\
&\quad (\{\widehat{\text{anonAcc}}(rLocs, wLocs, id)\} lb) - \widehat{\text{anonLength}}(id)) \\
&\text{renamedltrAnonAccApp} \\
&\frac{\{\widehat{\text{anonAcc}}'(rLocs, wLocs, id)\} \widehat{\text{noRaW}}(ls, rLs, wLs, futRLS) \rightsquigarrow}{\widehat{\text{noRaW}}(\{\widehat{\text{anonAcc}}'(rLocs, wLocs, id)\} ls, (\{\widehat{\text{anonAcc}}'(rLocs, wLocs, id)\} rLs) \setminus rLocs,} \\
&\quad (\{\widehat{\text{anonAcc}}'(rLocs, wLocs, id)\} wLs \setminus wLocs, futRLS) \\
&\text{renamedltrAnonAccAppOnHist} \\
&\frac{\{\widehat{\text{anonAcc}}'(id)\} \widehat{\text{noRaTHist}}(loc, lb) \rightsquigarrow}{\widehat{\text{noRaTHist}}(\{\widehat{\text{anonAcc}}'(rLocs, wLocs, id)\} loc,} \\
&\quad (\{\widehat{\text{anonAcc}}'(rLocs, wLocs, id)\} lb) + \widehat{\text{anonLength}}(id))
\end{aligned}$$

Fig. 20 A selection of calculus rules for (renamed) anonymized sequence of memory access updates application

References

- Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H.: In: Ulbrich, M. (ed.) *Deductive Software Verification-The KeY Book: From Theory to Practice*. LNCS, vol. 10001, Springer, Cham (2016). <https://doi.org/10.1007/978-3-319-49812-6>
- Allen, F.E., Burke, M.G., Cytron, R., Ferrante, J., Hsieh, W.C.: A framework for determining useful parallelism. In: Lenfant, J. (ed.) *Proceedings of the 2nd International Conference on Supercomputing, ICS 1988*, pp. 207–215. ACM, New York (1988). <https://doi.org/10.1145/55364.55385>
- Amme, W., Braun, P., Thomasset, F., Zehendner, E.: Data dependence analysis of assembly code. *Intl. J. Parallel Program.* **28**(5), 431–467 (2000)
- Arab, M.N., Wolf, F., Jannesari, A.: Automatic construct selection and variable classification in openmp. In: Eigenmann, R., Ding, C., McKee, S.A. (eds.) *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, 26–28 June 2019*, pp. 330–341. ACM, New York (2019). <https://doi.org/10.1145/3330345.3330375>
- Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R., Frederickson, P.O., Lasinski, T.A., Schreiber, R., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.: The NAS parallel benchmarks - summary and preliminary results. In: Martin, J.L. (ed.) *Proceedings Supercomputing '91, Albuquerque, NM, USA, 18–22 November 1991*, pp. 158–165. ACM, New York (1991). <https://doi.org/10.1145/125826.125925>
- Balaban, I., Pnueli, A., Zuck, L.D.: Shape analysis by predicate abstraction. In: Cousot, R. (ed.) *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris,*

- France, 17–19 January 2005, Proceedings. Lecture Notes in Computer Science, vol. 3385, pp. 164–180. Springer, New York (2005). https://doi.org/10.1007/978-3-540-30579-8_12
7. Banerjee, U.: An introduction to a formal theory of dependence analysis. *J. Supercomput.* **2**(2), 133–149 (1988). <https://doi.org/10.1007/BF00128174>
8. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Boer, F.S., Bonsangue, M.M., Graf, S., Roever, W.P. (eds.) *Formal Methods for Components and Objects*, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, 1–4 November 2005, Revised Lectures. LNCS, vol. 4111, pp. 364–387. Springer, New York (2005). https://doi.org/10.1007/11804192_17
9. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). www.SMT-LIB.org
10. Barthe, G., D’Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop. CSFW, Pacific Grove, pp. 100–114. IEEE Computer Society, Los Alamitos (2004)
11. Baudin, P., Cuoq, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, 1.22 edn. CEA LIST and INRIA (2025). https://frama-c.com/download/acsl_1.4.pdf
12. Beckert, B., Bubel, R., Drod, D., Hähnle, R., Lanzinger, F., Pfeifer, W., Ulbrich, M., Weigl, A.: The Java verification tool KeY: a tutorial. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) *Proc. 26th Intl. Symp. on Formal Methods*, Milan, Italy. LNCS, vol. 14934, pp. 597–623. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-71177-0_32
13. Blom, S., Darabi, S., Huisman, M.: Verification of loop parallelisations. In: Egyed, A., Schaefer, I. (eds.) *Fundamental Approaches to Software Engineering - 18th International Conference, FASE*. LNCS, vol. 9033, pp. 202–217. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-46675-9_14
14. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Gupta, R., Amarasinghe, S.P. (eds.) *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, pp. 101–113. ACM, New York (2008). <https://doi.org/10.1145/1375581.1375595>
15. Bubel, R., Hähnle, R., Tabar, A.H.: A program logic for dependence analysis. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) *Integrated Formal Methods - 15th Intl. Conf. IFM 2019*. LNCS, vol. 11918, pp. 83–100. Springer, New York (2019). https://doi.org/10.1007/978-3-030-34968-4_5
16. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Jr., W.A.H., Somenzi, F. (eds.) *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, 8–12 July 2003, Proceedings. Lecture Notes in Computer Science*, vol. 2725, pp. 420–432. Springer, New York (2003). https://doi.org/10.1007/978-3-540-45069-6_39
17. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) *Conf. Record of the 4th Symp. on Principles of Programming Languages*, pp. 238–252. ACM, New York (1977). <https://doi.org/10.1145/512950.512973>
18. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, 6–8 April 2005, Proceedings. LNCS*, vol. 3450, pp. 193–209. Springer, Berlin (2005). https://doi.org/10.1007/978-3-540-32004-3_20
19. De Gouw, S., De Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK’s sort method for generic collections. *J. Autom. Reason.* **62**(6), 93–126 (2019)
20. Dohrau, J., Summers, A.J., Urban, C., Münger, S., Müller, P.: Permission inference for array programs. In: Chockler, H., Weissenbacher, G. (eds.) *Computer Aided Verification—30th International Conference, CAV. LNCS*, vol. 10982, pp. 55–74. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-96142-2_7
21. Ernst, M.D.: Summary of dynamically discovering likely program invariants. In: 2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, 6–10 November 2001, pp. 540–544. IEEE Computer Society, Washington (2001). <https://doi.org/10.1109/ICSM.2001.972767>
22. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* **69**(1–3), 35–45 (2007). <https://doi.org/10.1016/j.scico.2007.01.015>
23. Feautrier, P.: Dataflow analysis of array and scalar references. *Intl. J. Parallel Program.* **20**(1), 23–53 (1991)
24. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Launchbury, J., Mitchell, J.C. (eds.) *The 29th Symp. on Principles of Prog. Lang.*, pp. 191–202. ACM, New York (2002). <https://doi.org/10.1145/503272.503291>

25. Furia, C.A., Meyer, B., Velder, S.: Loop invariants: analysis, classification, and examples. *ACM Comput. Surv.* **46**(3), 34–13451 (2014). <https://doi.org/10.1145/2506375>
26. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B.: Kremlin: rethinking and rebooting gprof for the multi-core age. In: Hall, M.W., Padua, D.A. (eds.) 32nd ACM SIGPLAN Conf. on Progr. Lang. Design and Implementation, pp. 458–469. ACM, New York (2011). <https://doi.org/10.1145/1993498.1993553>
27. Gentzen, G.: Untersuchungen über das Logische Schliessen. *Math. Z.* **39**, 176–210 (1935). (**English translation in Gentzen69**)
28. Grosser, T., Gröbinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* **22**(4), 1250010 (2012). <https://doi.org/10.1142/S0129626412500107>
29. Hähnle, R., Huisman, M.: Deductive verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) *Computing and Software Science: State of the Art and Perspectives*. LNCS, vol. 10000, pp. 345–373. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_18
30. Hähnle, R., Tabar, A.H., Mazaheri, A., Norouzi, M., Steinhöfel, D., Wolf, F.: Safer parallelization. In: Margaria, T., Steffen, B. (eds.) 9th Intl. Symp. on Leveraging Applications of Formal Methods, ISO/LA, Rhodes, Greece, Part II. LNCS, vol. 12477, pp. 117–137. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61470-6_8
31. Henzinger, T.A., Hottelier, T., Kovács, L.: Valigator: a verification tool with bound and invariant generation. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) *Logic for Programming, Artificial Intelligence, and Reasoning*, 15th International Conference, LPAR 2008, Doha, Qatar, 22–27 November 2008. *Proceedings. Lecture Notes in Computer Science*, vol. 5330, pp. 333–342. Springer, New York (2008). https://doi.org/10.1007/978-3-540-89439-1_24
32. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, 14–16 January 2004*, pp. 232–244. ACM, New York (2004). <https://doi.org/10.1145/964001.964021>
33. Huda, Z.U., Jannesari, A., Wolf, F.: Using template matching to infer parallel design patterns. *ACM Trans. Archit. Code Optim.* **11**(4), 64–16421 (2014). <https://doi.org/10.1145/2688905>
34. Humenberger, A., Jaroschek, M., Kovács, L.: Invariant generation for multi-path loops with polynomial assignments. In: Dillig, I., Palsberg, J. (eds.) *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI. LNCS*, vol. 10747, pp. 226–246. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-73721-8_11
35. Jahr, R., Gerdes, M., Ungerer, T.: A pattern-supported parallelization approach. In: Balaji, P., Guo, M., Huang, Z. (eds.) *Proceedings of the 2013 PPOPP International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2013, Shenzhen, China, 23 February 2013*, pp. 53–62. ACM, New York (2013). <https://doi.org/10.1145/2442992.2442998>
36. Jahr, R., Gerdes, M., Ungerer, T.: On efficient and effective model-based parallelization of hard real-time applications. In: Giese, H., Huhn, M., Phillips, J., Schätz, B. (eds.) *Dagstuhl-Workshop MBEEs: Modellbasierte Entwicklung Eingebetteter Systeme IX, Schloss Dagstuhl, Germany, 24–26 April 2013, Tagungsband Modellbasierte Entwicklung Eingebetteter Systeme*, pp. 50–59. fortiss GmbH, München, Munich (2013)
37. Jhala, R., Podolski, A., Rybalchenko, A., Clarke, E.M., Henzinger, T.A., Veith, H.: Predicate abstraction for program verification. In: Bloem, R. (ed.) *Handbook of Model Checking*, pp. 447–491. Springer, New York (2018). https://doi.org/10.1007/978-3-319-10575-8_15
38. Khatchadourian, R., Tang, Y., Bagherzadeh, M.: Safe automated refactoring for intelligent parallelization of java 8 streams. *Sci. Comput. Program.* **195**, 102476 (2020). <https://doi.org/10.1016/j.scico.2020.102476>
39. Kim, M., Kim, H., Luk, C.: SD3: A scalable approach to dynamic data-dependence profiling. In: 43rd Annual IEEE/ACM Intl. Symp. on Microarchitecture, pp. 535–546. IEEE Computer Society, Washington (2010). <https://doi.org/10.1109/MICRO.2010.49>
40. Kim, M., Kim, H., Luk, C.-K.: Prospector: Discovering parallelism via dynamic data-dependence profiling. In: *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism, HOTPAR*, vol. 10, pp. 395–416. USENIX Association, Berkeley (2010). https://faculty.cc.gatech.edu/~hyesoon/prospector_4page.pdf
41. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) *Fundamental Approaches to Software Engineering*, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, 22–29 March 2009. *Proceedings. LNCS*, vol. 5503, pp. 470–485. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-00593-0_33

42. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual. Draft revision 2344. University of Central Florida (2013). <http://www.eecs.ucf.edu/leavens/JML/OldReleases/jmlrefman.pdf>
43. Liao, C., Quinlan, D.J., Willcock, J., Panas, T.: Semantic-aware automatic parallelization of modern applications using high-level abstractions. *Int. J. Parallel Program.* **38**(5–6), 361–378 (2010). <https://doi.org/10.1007/s10766-010-0139-0>
44. Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: Sarkar, V., Hall, M.W. (eds.) *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pp. 190–200. ACM, New York City (2005). <https://doi.org/10.1145/1065010.1065034>
45. Maydan, D.E., Amarsinghe, S., Lam, M.S.: Data dependence and data-flow analysis of arrays. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) *Lang. and Compilers for Parallel Computing*, 5th Intl. Workshop. LNCS, vol. 757, pp. 434–448. Springer, Berlin (1992)
46. Mendonca, G.S.D., Guimarães, B.C.F., Alves, P.R.O., Pereira, F.M.Q., Pereira, M.M., Araujo, G.: Automatic insertion of copy annotation in data-parallel programs. In: 28th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2016, Los Angeles, CA, USA, 26–28 October 2016, pp. 34–41. IEEE Computer Society, Washington (2016). <https://doi.org/10.1109/SBAC-PAD.2016.13>
47. Morew, N., Norouzi, M., Jannesari, A., Wolf, F.: Skipping non-essential instructions makes data-dependence profiling faster. In: *European Conference on Parallel Processing*, pp. 3–17. Springer, New York City (2020). https://doi.org/10.1007/978-3-030-57675-2_1
48. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Ferrante, J., McKinley, K.S. (eds.) *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, California, USA, June, pp. 89–100. ACM, New York (2007). <https://doi.org/10.1145/1250734.1250746>
49. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) *34th International Conference on Software Engineering*, ICSE, pp. 683–693. IEEE Computer Society, Washington (2012). <https://doi.org/10.1109/ICSE.2012.6227149>
50. Norouzi, M.: Enhancing the speed and automation of assisted parallelization. PhD thesis, Technical University of Darmstadt, Germany (2022). <http://tuprints.ulb.tu-darmstadt.de/22884/>
51. Pouchet, L.-N., Yuki, T.: PolyBench/C 4.1 (2015). <https://sourceforge.net/projects/polybench>
52. Psarris, K.: Program analysis techniques for transforming programs for parallel execution. *Parallel Comput.* **28**(3), 455–469 (2002). [https://doi.org/10.1016/S0167-8191\(01\)00132-6](https://doi.org/10.1016/S0167-8191(01)00132-6)
53. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Martin, J.L. (ed.) *Proc. Supercomputing '91*, pp. 4–13. ACM, New York (1991)
54. Pugh, W., Wonnacott, D.: Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.* **16**(4), 1248–1278 (1994). <https://doi.org/10.1145/183432.183525>
55. Pugh, W., Wonnacott, D., Banerjee, U., Gelernter, D., Nicolau, A.: An exact method for analysis of value-based array data dependences. In: Padua, D.A. (ed.) *Lang. and Compilers for Parallel Computing*, 6th Intl. Workshop. LNCS, vol. 768, pp. 546–566. Springer, Berlin (1993)
56. Ramos, P., Mendonca, G.S.D., Soares, D., Araújo, G., Pereira, F.M.Q.: Automatic annotation of tasks in structured code. In: Evripidou, S., Stenström, P., O’Boyle, M.F.P. (eds.) *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2018, Limassol, Cyprus, pp. 31–13113. ACM, New York City (2018). <https://doi.org/10.1145/3243176.3243200>
57. Royuela, S., Ferrer, R., Caballero, D., Martorell, X.: Compiler analysis for openmp tasks correctness. In: Napoli, C.D., Salapura, V., Franke, H., Hou, R. (eds.) *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF’15, Ischia, Italy, pp. 7–178. ACM, New York City (2015). <https://doi.org/10.1145/2742854.2742882>
58. Rus, S.V., Rauchwerger, L., Hoefflinger, J.P.: Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.* **31**(4), 251–283 (2003). <https://doi.org/10.1023/A:1024597010150>
59. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE J. Sel. Areas Commun.* **21**(1), 5–19 (2003). <https://doi.org/10.1109/JSAC.2002.806121>
60. Sampaio, D.N., Ketterlin, A., Pouchet, L., Rastello, F.: POSTER: hybrid data dependence analysis for loop transformations. In: Zaks, A., Mendelson, B., Rauchwerger, L., Hwu, W.W. (eds.) *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT 2016, Haifa, Israel, 11–15 September 2016, pp. 439–440. ACM, New York (2016). <https://doi.org/10.1145/2967938.2974059>
61. Scheben, C., Greiner, S., Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.: Information flow analysis. In: Ulbrich, M. (ed.) *Deductive Software Verification-The KeY Book: From Theory to Practice*.

- LNCS, vol. 10001, pp. 453–472. Springer, New York (2016). https://doi.org/10.1007/978-3-319-49812-6_13
62. Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D., Reif, W.: Development of a verified flash file system. In: Ameer, Y.A., Schewe, K. (eds.) *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th Intl. Conf., ABZ, Toulouse, France. LNCS*, vol. 8477, pp. 9–24. Springer, Cham (2014). <https://doi.org/10.1007/978-3-662-43652-3>
 63. Scheurer, D., Hähnle, R., Bubel, R.: A general lattice model for merging symbolic execution branches. In: Ogata, K., Lawford, M., Liu, S. (eds.) *18th International Conference on Formal Engineering Method-sICFEM. LNCS*, vol. 10009, pp. 57–73. Springer, New York (2016). https://doi.org/10.1007/978-3-319-47846-3_5
 64. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018(3–8), December 2018, pp. 7762–7773. Canada, Montréal (2018)*
 65. Snelting, G., Giffhorn, D., Graf, J., Hammer, C., Hecker, M., Mohr, M., Wasserrab, D.: Checking probabilistic noninterference using JOANA. *Inf. Technol.* **56**(6), 280–287 (2014)
 66. Subotic, V., Ayguadé, E., Labarta, J., Valero, M.: Automatic exploration of potential parallelism in sequential applications. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) *International Supercomputing Conference ISC. LNCS*, vol. 8488, pp. 156–171. Springer, New York City (2014). https://doi.org/10.1007/978-3-319-07518-1_10
 67. Tabar, A.H.: Automatic data dependence analysis by deductive verification. PhD thesis, Technical University of Darmstadt, Germany (2024). <http://tuprints.ulb.tu-darmstadt.de/26722/>
 68. Tabar, A.H., Bubel, R., Hähnle, R.: Automatic loop invariant generation for data dependence analysis. In: *10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormalISE@ICSE 2022, Pittsburgh, PA, USA, 22–23 May 2022*, pp. 34–45. ACM, New York (2022). <https://doi.org/10.1145/3524482.3527649>
 69. Weiß, B.: Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction. PhD thesis, Karlsruhe Institute of Technology (2011). <https://d-nb.info/1010034960>
 70. Weiß, B.: Predicate abstraction in a program logic calculus. *Sci. Comput. Program.* **76**(10), 861–876 (2011). <https://doi.org/10.1016/j.scico.2010.06.008>
 71. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: a transparent dependence distance profiling infrastructure. In: *The 7th Intl. Symp. on Code Generation and Optimization*, pp. 47–58. IEEE Computer Society, Washington (2009). <https://doi.org/10.1109/CGO.2009.15>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.