FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Pavel Jančík

Efficient Representation of Program States

Department of Distributed and Dependable Systems

Supervisor of the doctoral thesis:	RNDr. Jan Kofroň, Ph.D.
Study programme:	Informatics
Study branch:	Software Systems (4I2)

Prague 2017

Annotation

Title	*Efficient Representation of Program States*

Author	Pavel Jančík
	`pavel.jancik@d3s.mff.cuni.cz`
	`(+420) 221 914 285`

Advisor	RNDr. Jan Kofroň, Ph.D.
	`jan.kofron@d3s.mff.cuni.cz`
	`(+420) 951 554 285`

Department	Department of Distributed and Dependable Systems Faculty of Mathematics and Physics Charles University Malostranske nam. 25, 118 00 Prague, Czech Republic

Abstract

The goal of automated software verification is either to prove the safety or to find erroneous behavior. To achieve this, verification techniques need to efficiently represent program states or sets thereof. In the thesis, we introduce methods to reduce the size of the program state representation for explicit and symbolic model checkers.

In particular, we introduce novel dead-variable analyses for parallel heap-manipulating programs. These analyses are designed for on-the-fly explicit state model checkers.

In contrast to that, in symbolic model checking, sets of program states are almost exclusively represented by logical formulae; these formulae are often obtained from Craig interpolants. In the thesis, we introduce a novel interpolation technique, which uses variables assignments to reduce the size of interpolants. Variable assignments can be used e.g., to block some uninteresting program paths. Interpolants need not to represent the program states from the blocked paths; hence, by using the variable assignment, smaller and more focused interpolants can be obtained.

Keywords

Software verification, Code model checking, State matching, Craig Interpolation

Anotace

Název	*Reprezentace stavů programu*

Autor	Pavel Jančík
	`pavel.jancik@d3s.mff.cuni.cz`
	`(+420) 221 914 285`

Školitel	RNDr. Jan Kofroň, Ph.D.
	`jan.kofron@d3s.mff.cuni.cz`
	`(+420) 951 554 285`

Katedra	Katedra distribuovaných a spolehlivých systémů Matematicko-fyzikální fakulta Univerzita Karlova Malostranské nám. 25, 118 00 Praha, ČR

Abstrakt

Při verifikaci programů se snažíme rozhodnout, zda program obsahuje či neobsahuje chyby Základním předpokladem všech verifikačních postupů je efektivní reprezentace a manipulace se stavy programů. V této práci představujeme techniky pro nalezení nepostatných informací ve stavech programů a pro jejich ostranění. Tato práce obsahuje redukce vhodné pro explicitní i symbolickou reprezentaci stavů.

Naše postupy vhodné pro explicitní reprezentaci byly speciálně navrženy pro vícevláknové programy. Naše analýzy dokáží nalézt takové hodnoty v dynamicky alokovaných objektech, tedy na haldě, které program již nebude v následujících krocích číst.

Logické formule v predikátové nebo výrokové logice jsou převažující symbolickou reprezentací množin stavů programu. Craigovy interpolanty jsou jedním z obvyklých postupů pro získání formulí s požadovanými vlastnostmi. V této práci představujeme nový způsob jejich výpočtu, který používá přiřazení proměnných pro zmenšení jejich velikosti. Pomocí přiřazení proměnných můžeme zablokovat ty cesty v programu, které nechceme, aby interpolant bral v potaz a tím zmenšit jejich velikost.

Klíčová slova

Verifikace software, Model checking kódu, Porovnávání stavů, Interpolanty

Acknowledgments

This work would not be possible without a support from a number of people. I am deeply grateful to my supervisor Jan Kofroň, for his guidance during my study and research work. I thank my colleagues at the department for inspiring ambience they created; a particular thank goes to: Ondřej Šerý, Pavel Parízek, David Hauzar and Jakub Daniel.

Last but not least, many thanks go to my parents and my wife Iva. Only their support and endless patience made this work possible.

This work as well as the related research was partially supported by the Grant Agency of the Czech Republic projects P202/11/0312, 13-12121P and 14-11384S, by Charles University institutional funding SVV-2015-260222, by the Charles University Foundation grant 203-10/253297 and by the European Union project ASCENS 257414.

Contents

1 Introduction

Computers are around all of us and their number increases constantly. Neither hardware nor software failures are admissible in many areas as they can lead to significant financial losses or even endanger our lives. With the increasing complexity of current (not only safety-critical) software there is more space for bugs, which are often hard to discover. This represents the driving force for automated software verification and more importantly for its scalability.

Over the years various techniques for software verification have been introduced. Their goal is either to prove safety or to find violation of a correctness specification. In order to achieve this, all of these techniques need to represent program states or sets thereof. The number of program states increases (often exponentially) with the complexity and size of the software – the well-known *state space explosion* [29] problem. Verification techniques differ in the way they cope with the state space explosion. Hence, efficient representation and manipulation with program states are the key parts of all these techniques.

State representations. Verification techniques represent program states in various ways, which typically depend on the kind of the technique. Explicit state model checkers represent the program state in a similar way as it is represented at run-time. In contrast to the explicit state model checkers which operate on a single program state at a time, the symbolic model checkers store and manipulate a set of states at a point. The sets are often represented as logic formulae or Binary Decision Diagrams (BDDs) [19]. In more details, a logical formula over program variables represent a set of states which satisfy the formula; i.e., the formula represents the states, such that the formula evaluates to $True$ if the values from the state are assigned to the corresponding variables in the formula. The way the formulae are created varies substantially among verification techniques. For example, in symbolic execution [72] the formula is created from program statements (assignments and conditions) of the program path being considered. In abstract interpretation [34] the abstract domains used are often represented symbolically as various forms of inequalities.

In explicit state model checkers such as Jpf [106], Spin [57], Zing [6] and MoonWalker [37] the state holds a single particular value for each variable. For each thread resp. process, the state contains a call-stack storing local values of variables. It also contains heap with dynamically allocated objects and references among objects, if the tool supports dynamic allocation. Explicit representation of program states is precise; however, it tends to be large. The reason can be found in programing languages and their complex semantics. As an example, consider Java and its automatic (un)boxing for variables of primitive types; this together with caching of boxed instances with small values forces the model checker to maintain a large set of instances for boxed values. Moreover collections (i.e., maps, sets, and lists) cannot store the primitive types, the boxed types must be used instead. This contributes to the complex structure of the heap as well as to the overall size of the program state.

A logical formula resp. a predicate is often used to symbolically represent a set of program states. Verification techniques differ substantially in a way formulae are constructed, used, in their meaning, and the properties the formulae should have. However, many state-of-the-art approaches (e.g., IC3 [15, 16], UFO [3, 5], Whale [4]) in more or less visible way try to find safe *inductive invariant*. Inductive invariant is a formula representing an over-approximation of all the reachable states such that after any program step (resp. loop iteration) a program state from the inductive invariant cannot result into a state outside the invariant. Inductive invariant is said to be safe if it does not contain any faulty state; as defined by specification (e.g., states violating program assertions). The safe inductive invariant can be seen as a certificate showing the safety of the verified program; once it is found, the verification terminates. From undecidability of program safety it follows that no technique can guarantee that for a safe program it finds a safe inductive invariant in one step; instead the invariant candidates are iteratively created and checked whether they are real invariants. These checks involve possibly costly solver queries, thus smaller candidates will improve performance of the tools. Note that typically these techniques make also different kinds of solver queries, which could benefit from small formulas as well.

Craig interpolants [35] are often applied in the symbolic techniques. Their properties can be used to guarantee that (a) the interpolant formula is an over-approximation of reachable states, while (b) being precise enough – not reaching faulty states. Moreover (c) there are scalable approaches to compute interpolants for propositional logic [78, 91, 41, 26] as well as various kinds of first order theories (e.g., quantifier-free fragments of linear arithmetic [79, 27]). On the other hand, interpolants are often large. Although various optimizations to reduce their size have been introduced, their size still represents an important problem.

1.1 Goals and Objectives

As we have seen, the representation of program states is complex. The size of the state representation is one of its important properties influencing the performance of the tools. The states are processed in many ways during verification; this includes state matching in explicit state model checking and inductive invariant checks and other solver queries in symbolic techniques. Every inefficiency in the state representation affects scalability of the techniques; not only the verification tools need to process (unnecessarily) large state representation, but they can also increase the state space.

For example, in reference-based languages such as Java, the address of an object is irrelevant for a program; only the structure of the heap should influence the behavior. Thus, two states with isomorphic heaps, which differ only in addresses (resp. identifiers) assigned to objects, should behave in the same way. However, state matching which uses the object addresses directly, will not detect the aforementioned fact and it identifies such states as different. The technique aiming at detection and elimination of this inefficiency is called heap canonicalization.

It is obvious that elimination of such inefficiencies is an important task, since it contributes to the solution of the key problem in the verification field – scalability.

The goal of the thesis is to identify inefficiencies in program states representations and to design and implement novel techniques, which will result in smaller representation of program states.

1.2 Structure of the thesis

The thesis is structured in the following way. Chapter 2 provides the reader with necessary background together with a description of related existing techniques for efficient state representation. Chapter 3 describes the revisited goals as we have identified them based on the current state-of-the-art. Chapters 4, 5 describe our actual contribution, i.e., the techniques to reduce the size of explicit resp. symbolic program state representation based on elimination of their irrelevant parts. Finally, Chapter 6 concludes the thesis. In this chapter, we also propose possible future research directions. We decided to move selected proofs from Chapters 4 and 5 into Appendix A to improve the legibility.

1.3 Publications

In our work, we do not focus only on the efficient representation of program states. We also tackled the broader scalability problem in different ways.

In [68, 65] we have shown how variable assignments can be used to produce smaller Craig interpolants, which are often used to compute symbolic state representations. These papers are closely related to this thesis; they form a major part of Chapter 5.

In [66] we have introduced analyzes to identify dead parts of program states. This work has been extended in the [67]. These papers are closely related to overall goal of this thesis. They form a major part of Chapter 4.

In [85] we have introduced a novel Partial Order Reduction (POR) technique for on-the-fly explicit state model checkers. These model checkers do not have knowledge of the future behavior of the program at a state they have reached during verification. They need to be conservative resulting in exploration of unnecessary many thread interleavings. To cope with this issue, static analysis is often employed to compute an (over-)approximation of the possible future behavior for the reached state. In this paper, we used the currently reached state in the model checker to improve precision of static analyses for POR; the analyses use dynamic information from a reached program state to internally build a happens-before relation among future program actions. This relation is then used to reduce the future behaviors of the currently reached state which need to be considered for POR. This means that more scheduling choices are eliminated and consequently the state space is reduced. This way the approach helps to push the scalability limits.

In [70] we have followed the broad idea of modular verification. The software (especially the larger piece thereof) is not monolithic. Rather it is often split into smaller parts – *software components* – which can be verified separately. Each component comes with a specification which, among other, declare how the component should be used. Often the specification includes also how the component could interact with other ones, i.e., its *behavior*. The verification process is then split into two phases: (1) Behavior specification of connected components is checked for mutual compatibility, and (2) the implementation of each component is verified whether it complies with its behavioral specification. The first phase benefits from behavioral models which abstract away low-level details unrelated to the component behavior; thus, their state space is much smaller compared to the state space of the implementation. The latter phase extends the guarantee from the level of the specification to the implementation. The second phase benefits from modularity; only a single component together with a small automatically-generated environment is verified at a time. The environment represents the rest of the system (i.e., environment simulates all permitted usages of the component). In the paper, we have introduced a checker for the second phase, which is able to verify the compliance of SOFA [22] components with its behavioral specification in the form of Threaded Behavioral Protocols [90].

In [69] we have introduced an extension to JPF which helps users to analyze program states and scheduling choices. It simplifies the process of revealing and understanding the sources of the state space explosion.

I am the main author of [68, 70, 69] where I created implementations resp. formal description of the method (i.e., proofs). I have authored large parts of the text of these papers as well. In [85] I participate mostly in designing the static analyses.

Reviewed papers

[67] Jančík P., Kofroň J. On Partial State Matching, Journal article in *Formal Aspects Of Computing*, ISSN 1433-299X, DOI 10.1007/s00165-016-0413-z, pages 1-27, 2017.

[66] Jančík P., Kofroň J. Dead Variable Analysis for Multi-Threaded Heap Manipulating Programs, In proceedings *ACM Symposium on Applied Computing*, pages 1620-1627, 2016.

[65] Jančík P., Alt L., Fedyukovich G., Hyvärinen A. E. J., Kofroň J. and Sharygina PVAIR: Partial Variable Assignment InterpolatoR, In proceedings

Fundamental Approaches to Software Engineering, pages 419-434, 2016.

[68] Jančík P., Kofroň J., Rollini S. F., and Sharygina N. On Interpolants and Variable Assignments, In Proceedings *Formal Methods in Computer-Aided Design*, pages 123-130, 2014.

[85] Parízek P., and Jančík P. Approximating Happens-Before Order: Interplay between Static Analysis and State Space Traversal. In proceedings of *International Symposium on Model Checking of Software*, SPIN, pages 1-10, 2014

[70] Jančík P., Parízek P., and Kofroň J. BeJC: Checking Compliance between Java Implementation and Behavior Specification, In Proceedings of *Doctoral Symposium on Components and Architecture*, pages 31-36, 2012

[69] Jančík P., Parízek P., and Kofroň J. Advanced Debugging with JPF Inspector, In local proceedings of *MEMICS*, pages 43-50, 2011

1.4 Note on conventions

Parts of this thesis are based on the aforementioned publications. To be able to easily distinguish a text included verbatim from the publications, the corresponding paragraphs are marked with a vertical bar on the left resp. right side of the text.

[X] The following paragraph is an example of the way we mark the text, which is a verbatim copy. It means that the original text appeared in the paper [X].

2 Background

In this chapter, we describe how state-of-the-art explicit and symbolic model checkers represent program states and how they perform state matching. We also describe various optimizations used to reduce the state space and size of the program state representation.

2.1 Explicit state model checking

Explicit state model checkers exhibit in verifications of parallel systems, especially in founding subtle synchronization and concurrency bugs. It is made possible due to cheap computation of successor states, so they are able to explore all the interesting interleavings.

2.1.1 State representation

SPIN model checker uses PROMELA as an input language. The PROMELA was designed as a modeling language; it does not contain function calls (i.e., recursion) and it does not support dynamic allocation. The SPIN can only handle finite models; there are limits on range of variables, length of queues and number of active processes in SPIN. It means that the states have bounded size and SPIN can internally represent states as arrays where (global and process local) variables are mapped to particular indexes in the array. Such a simple representation simplifies a state matching process and makes optimizations easier.

dSpin [38] is an extension of Spin (and its input language Promela) which supports dynamic allocation, references, and function calls. Since the size of the state is no longer bounded, the authors use more complicated (non-linear) representation for states. The run-time representation of program state consists of several growing arrays and support tables. The tables contain sizes and offsets of the elements stored in the corresponding growing array. In particular, heap together with the global variables and global queues occupy one such an array. Due to unbounded nature of call stack (i.e., recursion), each process and its local data is stored in its own array. Because of such a complex structure, the run-time representation is not suitable for state matching. In dSpin the structure is first *serialized* (i.e., linearized) into a single array – *state vector*. This permits to reuse the state matching of Spin and all its the optimizations.

Jpf is an on-the-fly explicit state model checker for Java; Jpf can be seen as a specialized JVM optimized for systematic exploration of the state space. From high level perspective, at run-time Jpf represents program states in a similar way as dSpin. However, Jpf representation is more complicated, since program states are more complex compared to states of Promela language. The representation for a thread holds a list of stack frames which holds parameters, local and temporary variables. Stack frame also contains pointer to a method to which the frame belongs to. In case of instance method (i.e., the method is not static) the frame holds also reference to `this`. In Jpf, the references are represented by integers. Each element on the heap has assigned a unique numeric ID. A heap manager is used to resolve these IDs to representation of the instance. In this representation values of the fields are stored. From technical reasons, the values of static fields are stored separately in a special heap. The run-time representation is designed to be easily accessible by programmers; it resembles the way how a program is represented at run-time in normal JVM. On the other hand, the complex heap graph is extremely inefficient for state comparison, thus the same approach as in dSpin is applied. Before state matching, the run-time representation is serialized in a single integer array (i.e., *state vector*), which holds all relevant data from the program state. State vector is then used in the state matching.

In order to be safe, the serialization process in Jpf as well as in dSpin needs to preserve the following property:

Requirement 1 (Uniqueness) *For any two different program states, a serializer has to create different state vectors.*

2.1.2 State matching

Each time the model checker decides to finish a transition, the state matching is triggered. First, the program state is serialized into state vector (an array of bytes). Then, the state vector is passed into a *state set* which decides whether the given vector is seen for the first time – a new state not being stored in the set, or it has been processed before – visited state already stored in the set.

Full state set. Hash set is one of the easiest ways to efficiently represent a set of items. For each state vector, a hash value is computed. In explicit state model checkers, Jenkins' hash functions [71] are often applied in state matching; in particular, both Jpf and Spin use them. The hash value represents a key (resp. its modulo w.r.t. size of hash table) to a hash table, full state vectors for given key are stored in a linked list.

State vectors are large, especially in the case of programs with rich standard libraries. In Spin state vectors are typically smaller compared to code model checkers. Still, their size represents a major obstacle as to scaling; hence the reduction of memory requirements is the driving force in this research area. In many cases, the memory required to store the state vectors of all the reachable states is by orders of magnitude larger than the available one.

State collapsing. State Collapsing (SC) [58, 105] is another way to reduce a state vector. SC partitions the program state (or state vector) into smaller parts, e.g., stack frames and heap instances. SC matches parts separately; for each kind of parts, it has a specialized state matching (sub)-procedure. Specialized matching procedure holds a numbered list of all seen instances of that kind; given a part it returns its ID. Either the same part is stored in the list and in such a case its index is returned or if not in list it is immediately added and index of the added element is returned. If state vector is constructed, instead of adding whole and possibly large representation of the part into a state vector, the state collapsing will store only the id. That is why this technique is sometimes called *recursive indexing method.*

This technique is still available in Spin. It was also implemented in former versions of Jpf. The collapsing significantly reduces the size of the state vector and space needed to store them, on the other hand additional memory is needed to store indexes for collapsed parts.

Minimized automaton. The set can be also represented by an automaton which accepts exactly the represented words; in case of the state matching, the words are state vectors. Such a representation was proposed in [62] and implemented in Spin. The main idea is to build and update a minimal deterministic finite state automaton, which accepts only the stored visited state vectors. The aim of the authors is to reduce memory used to store visited states; in automaton representation, if two stored states have the same prefix, the prefix can be stored only once. The memory savings comes at a cost of increased run-time; the most performance critical operation is adding new state into the visited set. This operation includes minimization of the automation, which can be costly.

The above state matching techniques are safe; in other words, they cannot mark a state as a visited unless it has been really added into the visited state set. If these methods require more memory then available (a typical case), various unsafe (i.e., approximate) methods can be applied to reduce the required memory. On the other hand, using such techniques, there is some very small probability that new state will be incorrectly marked as visited. Consequently, state space will not be explored completely and it may happen that an error is missed if present only in omitted parts of the state space. These techniques are often based on hashing, so the omitted parts of state space are determined by hash collisions. In other words, the omitted parts are selected randomly and not consistently based on some specific feature used in verified program; this further reduces the probability, that error state will be only in the omitted parts of the state space.

Hash-compact. Hash-compact [108] method can be seen as a modification of the full state set method. The idea of hash-compact is simple, instead of state vectors it uses and stores only their hashed. The hash-compact is the default state matching method used in Jpf. Note that Jpf contains and can be configured to use the full state set method (and any other user provided state matching method).

This method relies on the fact that hash collisions in big enough space are rare; in Jpf and in Spin the state vectors is hashed into a single 64-bit value. The main properties of the hash function being used nearly uniform distribution (i.e., as less collisions as possible) and speed of hash computation; in Jpf Jenkin's LOOKUP3 hash function is used.

Bitstate hashing. Bitstate hashing [56, 59] uses hashes of the state vector differently. Instead of storing hash values of fixed size (i.e., 64-bit) as is done in Hash-compact method, the bitstate hashing allocates a Boolean array of size h. Initially the array has all bits set to 0; it means no state is visited. Bitstate hashing use hash function with range 0 - $h - 1$; the hash of the state vector is used as an address into the Boolean array. If the bit at given address is not set the state is new, otherwise if the bit is set, the state is considered to be visited.

Bitstate hashing suffers from hash collisions, in order to explore whole state space with reasonable high probability, the Boolean array should be filled less than 1% [59]; if ratio of the set bits grows beyond 1% in the Boolean array then the coverage of the state space starts decreasing. To improve resilience to collisions, multiple different hash functions can be used to select bits in a single Boolean array; the state is marked as visited only if bits at all addresses (hash values) are set. The bitstate hashing function is implemented e.g., in SPIN; 2 different hash functions are used.

The optimal state matching algorithm depends a lot on available memory; if enough memory for full state matching (i.e., for storing all state vectors) it is reasonable to use these techniques. In such a case, there is a guarantee that whole state space will be explored. Typically, there is not enough memory for full state matching; then the hash-compact method is worth trying. It provides less collisions compared to the bitstate hashing Only if there is not enough memory to store hashes of state (i.e., for hash-compact method), the bitstate hashing is reasonable option.

Another technique to reduce memory requirements of state matching is to reduce the size of the state vectors which needs to be stored; such techniques nicely matches to safe state matching techniques like a full state set method. For approximate methods, there is typically enough available memor. In this case, shorter state vectors could speed-up verification. However, this is possible only if the reduction (or compression) of the state vector is done faster than the time needed to compute hashes of the omitted parts.

In [58] various compression techniques to reduce size of the state vector has been examined; e.g., Run-Length encoding or Huffman compression [63]. These techniques reduce the overall memory requirements, on the other hand, due to compression they introduce substantial run-time overhead, i.e., slow down verification.

State vectors may contain some fixed or duplicated data (i.e., data which can be derived from other values in the state vector) such data are unnecessary for state matching and can be omitted from state vector. Typical example is values of constant final fields. This technique called *byte-mask* is used as a default compression method SPIN; according [58] the byte masking offers modest compression while having not to big overhead compared to other compression techniques. This technique is applied in Jpf as well.

Dead variable reduction. While the above methods (byte masking and compression) are fully revertible – original full state vector can be computed. The Dead Variable Reduction (DVR) does not have such a property. It first runs an analysis which identifies dead variables. The values of dead variable are not added into state vector. We will discuss this technique in more detail later in this chapter.

2.1.3 Search optimizations

Different way to reduce memory required for state matching is to generate (and consequently store) less program states. Popular optimization for concurrent systems is *Partial Order Reduction* (POR). Other techniques are based on detection of various symmetries; in case of heap – Heap canonicalization.

Partial order reduction. Partial order reduction [49] tries to eliminate unnecessary thread interleavings; i.e., reduce state space explosion caused by concurrency. POR is based on the notion of *independence* among transitions from various threads (resp. processes). Intuitively, if the transitions are independent, the order in which are executed is irrelevant. So, it is enough to explore only one theirs interleaving. A typical example of independent actions are updates of a local variables or two reads of a shared global variable. Over the time various POR algorithms where introduces; among others POR based on *persistent set* [50], *sleep set* [51], *ample set* [88] and *stubborn set* [104].

In on-the-fly model checkers, it is hard to obtain the information about independence of future program actions, because the future program actions were not explored yes, thus are not known. Typically, static analysis is used to create a safe over-approximation of possibly dependent actions. In [85] we have improved precision of the static analysis by utilizing information from currently reached program state. This helped us to create more precise happens-before relation. The relation is used to eliminate interleaving of dependent actions that provably have fixed ordering due to happens-before relation between them.

In [47] different approach is taken. Authors observed that during model-checker run all the dependencies among the actions are resolved once the program trace is explored. So, in the dynamic POR, the thread scheduling choices are added after the trace is explored just before model checker starts backtracking.

Even though various sophisticated POR techniques have been introduced, on-the fly code model checkers typically uses only very simple POR (by default). In JPF independent actions are bytecode instructions accessing over local variables. Simple optimizations and heuristics are used for heap accesses and static fields do determine independence; if exists only a single runnable thread or accesses to final fields are independent. Heuristics based on whether the field is accesses from multiple threads and locks held at the time the fields is accessed are included in JPF. JPF does not apply any kind of static analysis to obtain a knowledge about the future actions and their independence.

Heap canonicalization Heap-canonicalization [64, 83] is another technique to reduce the state space. Heap-canonicalization helps to identify program states, that are behaviorally equivalent, but their representation of the memory differs. In reference-based languages (like Java) the address of the object in the memory (resp. the identifier of the object in the heap assigned be the model checker) cannot influence the behavior of the program. Heap-canonicalization identifies such kind of symmetries between program states; it transforms the state to a canonical representations which is unique for all the equivalent states. In other words, heap-canonicalization assigns addresses to heap objects deterministically based only on the shape of the heap graph and root references – references to the heap from the call stacks.

Heap-canonicalization is tightly connected to state matching. JPF computes canonical addresses during state matching; JPF assigns the canonical addresses while it traverses of the heap and serialize the state into state vector.

2.1.4 Dead variable analysis

Dead variable analysis (or its complement – Live variable analysis) first appeared as a compiler optimization [2]. Informally, the variable is *live* at a program point p if there exists some path from p in the control flow graph (CFG) of the program such that the variable is read on that path. The variable is *dead* if it is not live. The results of the dead variable analysis are used e.g., in register allocation; if new value has to be loaded into a CPU register, the register with dead variable should be used (if exists) as a target of the load. The register with dead variable can be safely overwritten without saving the value of the variable back into memory. The dead variable analysis is commonly used also in the model checking to reduce the state space – *Dead Variable Reduction*. The liveliness information from the analysis is applied in the state matching; in more details, the information is exploited during state serialization when state vector is constructed. If a dead variable is to be stored into the state vector either (i) a well-known constant value (e.g., zero or null) is saved instead of the value of the dead variable or (ii) the dead variables are not stored into the state vector at all. The first approach is used e.g., in SPIN for identified dead local variables. The second approach is used e.g., in JPF for selected `static final` fields.

Over the years various dead variable analysis has been introduced [13, 45, 77, 99, 109]. The first versions were focused only on local variables and exactly followed the above definition of the dead variables from compiler domain. Dead local variables can be obtained by quite simple intra-procedural static analysis. Even these simple analyzes pays-off; they reduce the state space and memory requirements of verified programs [40, 60].

Over the years analyzes specialized on code verification appeared. In [13, 45], Bozga et al. introduce a live variable analysis for an asynchronous parallel composition of processes that communicate via parameterized signal passing using a set of unbounded queues. All program variables are global and shared across processes; heap (i.e., object instantiation) and dynamic process creation are not supported by their formal language. Static analysis over a *control graph* – a parallel composition (i.e., dot product) of control flow graphs of all processes, is used to obtain live variables. Control graphs can have a reasonable size for formal languages where processes can be assumed reasonably small; however, in case of programs the control flow graphs are typically much larger (for example due to used libraries, etc.). In the worst case, the control flow graph can have a comparable size to the overall state space. The applied static analysis a straightforward extension of the analysis for local variables to a global ones and queues. The live variable analysis results are used to define a *live equivalence*, which can be seen as a base for the dead variable reduction. In the papers authors show that their live equivalence is a strong bisimulation. This main result of the paper permits to apply the reduction in tools [14, 46] which are based on a notion of bisimulation.

In [109], Yorav et al. introduce another dead variable analysis. Their DVA operates over a parallel composition of communicating processes; variables are not shared between processes. Heap and recursion are not supported by the analysis. The support for dynamic process creation comes for free, since their computation model does not contain shared global state (i.e., variables) shared among processes. The analysis in [109] is a backward static analysis. In the [109] a notion of *fully* and *partially dead variable* has been introduced. The variable is *fully dead* at given location if on all paths in CFG it is written (i.e., defined) before its first read (i.e., used); this corresponds to the classical DVA as we have introduced it in the above paragraphs. Informally, the variable is *partially dead* if it is dead only on some subset of paths from given location. The subset of paths is determined by a condition (i.e., formula) over program variables. This condition is computed using backward symbolic propagation over CFG; it can be seen as a computation of weakest precondition for the variable being dead. Due to its nature, the analysis loses precision on the loops and array accesses; in the former case, the imprecision comes from unknown number of loop iteration whereas in the latter case, from imprecise identification of the index in the array to which it is accessed.

In [77], Levis et al. comes up with a different kind of dead variable analysis. Their DVA is implemented in Estes [81] which operates on native binaries. The implementation supports arrays (i.e., indirect memory accesses) and programs with interrupts. The key idea of their DVA is to used dynamic information to prune CFG to which the classical static DVA is applied. The pruned CFG parts are those which cannot be reached from the current program state. By removing parts of CFG, the static analysis can provide more precise result, i.e., more variables can be identified as dead. If a variable is read only on the pruned basic blocks, then the variable is identified as dead compared to situation where pruning using is not applied. This dynamic approach typically gets more precise results compared to classical static analysis in case of (a) indirect memory access (arrays and object fields) and (b) in code with complex control structure (if-branches and virtual methods calls) where branches uses different set of variables. Compared to all the above analyzes where the dead variables are computed once as a pre-processing step before the verification run, the Levis's DVA is executed many times on a pruned CFG during the verification run.

The whole analysis works as follows. Before the verification run the Levis's DVA run a forward static analysis which computes *decision paths*; decision path for given basic-block represents decisions (in other words, conditional branches) that have to be taken in order to reach given basic block. The decision paths are used later during the verification run to identify the unreachable parts of CFG. During the verification run the DVA is used in the following way. If the model checker initiates a state matching, a heuristic is used to decide whether to compute more precise DVA information or if the older DVA results from previous analysis runs are to be used. This decision step protects against excessive calls to the costly DVA. If DVA is started it first starts a *forward partial simulation* from the current program state; the simulation involves depth limited execution program from the current program state. During the simulation outcome of the conditional branches is recorded. For each conditional instruction, (i.e., location in the CFG) the simulation records whether `true`-branch or `false`-branch or both branches has been taken. The results of the forward simulation are combined with the decision paths to prune the CFG. The basic block is pruned from the CFG if the forward simulation identifies that a program will take a branch that is inconsistent with the decisions for given basic block. Once pruned CFG is constructed a classical DVA is started and it gets dead variables.

In the paper authors suggests to selectively apply this DVA only on segments of the programs in which many variables are used exclusively along a given path in nested branching structure. The results show that in such cases the analysis performs better compared to classical DVA approach.

In [99], Self et al. introduced a different way how to utilize a dynamic information in dead variable analysis. Instead of using static analysis to over-approximate future behavior of the program from given program state, in this paper authors present a dynamic technique, which first executes a program trace, and once the trace has been fully determined, the analysis computes the dead variables for the states along the path. The trace is known if the analysis is started, thus the analysis knows precisely which branches were taken and which variables were read and written; the approach mitigates these sources of imprecision typical for a static analyses. The backward DVA along the determined trace is quite simple; if the trace ends in the final program state all variables are marked as dead for a state at the end of the trace. If the trace ends in a previously visited state, the dead variables of the visited state, which were computed by the analysis in some previous run, are used at the end of the trace. The analysis goes backwards along the trace, if it reaches a read instruction it removed the read variable from dead variables. In case of write instruction, the variable is added to the set of dead variables. If the analysis reaches a point on the trace with a non-deterministic choice (i.e., a user input) all variables are marked as live (i.e., removed from set of dead variables). This is required since the analysis have no knowledge what the program will do if non-deterministic choice, different from the choice on the analyzed trace, is taken (i.e., what happens if the user provides a different input).

In the paper, the authors introduce a notion of a *DVA maximal reduction*. In short, DVA maximal reduction is a state space derived from a full state space of a program where the program states contain only the live variables. The DVA introduced in the paper produces maximally reduced state space for programs without non-determinism (i.e., for programs containing only a single trace).

This analysis required depth-first-search state space traversal and cannot be easily adapted to other searches (e.g., BFS) compared to the other DVA's introduces above.

The explicit state code model checkers exhibits in verification of the multi-threaded programs which are hard to analyze be symbolic approaches due to numerous thread interleavings. The DVA's introduced above are not well suited for this task.

As pointed above, the approach of [13, 45] creates a control graph, which can be impractically huge in case of programs. The DVA of [109] lacks support of global variables and heap which is essentials feature for code verification, especially for Java language. Typically, the `main` function takes an array (i.e., heap instance) with parameters from command-line. The DVA of [77] is inefficient. It executes the trace multiple times. First it is executed in partial forward simulation, for the second time it is executed during the verification. Moreover, the computation static DVA (i.e., computation of the fixpoint) over pruned CFG is done many times. The technique also requires as an input a limit of length of the forward simulation. In [99] it has been nicely pointed out, that there is no good value which guarantees the best result for all program. Moreover, the DVA's of [77, 99] are not well suited for multi-threaded programs due to the way they cope with non-determinism. In single threaded programs, the transitions are quite long; they are terminated by non-deterministic data choices which simulate random user inputs. In multi-threaded programs, the transitions are short, often only a few of instructions. The prevailing source of non-determinism originates from thread scheduling; the transition terminates on all accesses to shared fields. In [77] the partial forward simulation terminates with the first non-determinism; which means that in case of short transitions, it likely does not obtain useful data. The DVA of [99] derives the dead values from a single trace, thus it case of non-determinism it assumes all variables are live. This makes the analysis unsuitable for multi-threaded programs.

From the above, we can see that we are missing an efficient DVA analysis for verification multi-threaded programs support the global variables and heap.

2.2 Symbolic model checking

The symbolic model checkers manipulate with set of states at a time. Historically, the sets of states have been represented by *Ordered Binary Decision Diagrams* (OBBD) [18, 20]. The logical formulae become predominant representation with the advent of satisfiability solvers.

Basics of symbolic representation. Predicates can be used to express set of states as well as transitions among these set of states.

Let have a program with two integer variables `x` and `y`. Assume the following predicate:

$$p_1 \equiv x >= 0 \land x < 10 \land y > 0$$

The set of states represented by the predicate includes $(x = 1, y = 0)$ and $(x = 9, y = 1)$; if the values of program variables from the state are substituted into the predicate, the predicate evaluates to *true*. The states $(x = 0, y = 0)$ and $(x = 10, y = 1)$ are not in the set of the states represented by the predicate p_1; the predicate evaluates to *false* if the values are substituted.

Transitions can be represented as predicates over input and output variables. Typically, the output variables are primed. To illustrate the concept, assume following program statement:

```
y = x - 1
```

The corresponding predicate will be over variables x, y, x', and y'. Input variables x and y describe program state before the statement. Output variables x' and y' describe the program variables once the statement is executed. The predicate corresponding to the statement is:

$$t_1 \equiv y' = x - 1 \land x' = x$$

The predicate holds if the statement produces the values in output variables from specified input, e.g., $(x = 1, y = 1, x' = 1, y' = 0)$.

Let be given predicate P_I representing a set of input states, predicate T representing a transition, and predicate P_O presenting a set of output states. The satisfiability solvers can be used to check whether there exists a state in the input set P_I which after execution of program statements represented by T will result to some state among the ones in the P_O. The input of the solver will be the conjunction of the predicates is satisfiable.

$$P_I \wedge T \wedge P_O$$

If the predicate P_O is representing the faulty states, then this check can be used to show that faulty states are unreachable, i.e., the safety of programs. To illustrate the concept, let use the predicate p_1 (defined above) to represent a set of input states, and the above predicate t_1 to represent the transition. Assume that the statement `y = x - 1` is followed by `x = x / y`. The division statement can fail if the denominator (i.e., variable y) equals 0. The error states which lead to division be 0 can be described by the predicate p_2

$$p_2 \equiv y' = 0$$

The predicate p_2 represents the (output) program states which yields to the error.

In our example, the conjunction to be passed to satifiability solver is:

$$p_1 \wedge t_1 \wedge p_2 \equiv (x >= 0 \wedge x < 10 \wedge y > 0) \wedge (y' = x - 1 \wedge x' = x) \wedge (y' = 0)$$

The solver outputs that the conjunction is satisfiable and will produce a satisfying assignment, e.g., $(x = 1, y = 10, x' = 1, y' = 0)$. The satisfying assignment corresponds to a program execution from an input state in P_I which will result in faulty state, i.e., will cause division by zero.

On the other hand, if we use a different set of initial states described by the predicate p_1' then the corresponding conjunction passed to solver will be unsatisfiable; so, from these states the program cannot fail on division by zero.

$$p_1' \equiv x >= 2 \wedge x < 10 \wedge y > 0$$

State matching. Logical implication can be used to compare the sets of states represented by predicates, i.e., to perform *symbolic state matching.* Let predicates p and v be sets of states for the same program instruction (i.e., location). Let predicate p represents a set of states that can be reached on currently examined program path and let v represents a set of already processed (i.e., visited) that cannot reach any error states. If it holds that

$$p \Rightarrow v$$

then we know that p is represents a subset of states of v; in other words for all the states that can be reached on the current path, it has been shown that they cannot reach any error state.

Let illustrate this concept on an example and let compare the set of states represented by the above predicates p_1 and p'_1. It does not hold that

$$p_1 \not\Rightarrow p'_1 \equiv (x >= 0 \wedge x < 10 \wedge y > 0) \not\Rightarrow (x >= 2 \wedge x < 10 \wedge y > 0)$$

As we already know that predicate p_1 represents also the state $(x = 1, y = 0)$, however the p'_1 does not represent that state as it requires $x >= 2$. So, this state shows why $p_1 \not\Rightarrow p'_1$.

The satisfiability solvers can be used to check whether $p \Rightarrow v$ or not. As in the example above, we let the solver to search for assignments (i.e., program states) that violates the implication; i.e., the solver input will be:

$$\neg(p \Rightarrow v)$$
$$\neg(p \Rightarrow v) \Leftrightarrow \neg(\neg p \vee v) \Leftrightarrow (p \wedge \neg v)$$

The second line shows the equivalent rewrites of the equations above; it highlights the fact, that SAT solver is looking for a violation. If the solver does not found any satisfying assignment and terminates with the outcome that the formula is unsatisfiable, then we know that there is no counterexample thus the implication holds. As it is the case for the $p'_1 \Rightarrow p_1$.

Inductive step. In the above paragraphs, we have shown that conjunction

$$P_I \wedge T \wedge P_O$$

can be used to check if it is possible from *some* input state (in P_I) to reach an output state (in P_O) after execution of a program statement(s) represented by T. However, in verification it is often important to know that from *all* input states the program will ends-up in output states (i.e., P_O) after execution of states of T. This property is often called *inductive step* it can expressed by the following implication:

$$P_I \wedge T \Rightarrow P_O$$

The implication claims that for any input program state in P_I (i.e., if P_I holds) and any outputs program state such that the output state can be created from the input state by execution of code represented by transition T (i.e., T holds), the output program state need the be in the set of program states represented by P_O (i.e., P_O have to hold).

Let us illustrate how the implication works in the example. Assume we have the following implication:

$$p'_2 \equiv (y' >= 1)$$

$$\begin{aligned} &p'_1 \wedge t_1 \Rightarrow p'_2 \equiv \\ &\qquad \equiv (x >= 2 \wedge x < 10 \wedge y > 0) \wedge (y' = x - 1 \wedge x' = x) \wedge (y' >= 1) \end{aligned}$$

Let take, e.g., an input state $(x = 2, y = 0)$, which is represented by predicate p'_1. For example, let took an output state $(x' = 1, y' = 1)$ that cannot be created from the input. In such a case t_1 does not hold; under given assignment it does not hold that $y' = x - 1$; so, antecedent of the implication is not satisfied and implication holds. Now, let assume and output state $(x' = 2, y' = 1)$; this state is created from assumed input state when statement corresponding transition t_1 is executed. In this case, the t_1 holds, thus the antecedents of the implication are satisfied. Hence, the consequent needs to hold to satisfy the implication. In our example, the output state $(x' = 2, y' = 1)$ belongs to the set of states represented by the predicate p'_2; i.e., consequent p'_2 holds.

Note that the above implication will always hold as the p'_1 requires $x >= 2$ and t will force $y' = x - 1$ and p'_2 expects $y' >= 1$.

2.2.1 Bounded model checking

In the above paragraphs, we illustrated usage of satisfiability solving in program verification. Now we will briefly describe basic concepts of Bounded Model Checking (BMC) [10, 28]. The well-known code bounded model checker are CBMC [30] and ESBMC [32]. Bounded model checkers impose a limit on depth in which the error can be reached. BMC has been successfully applied in model checking of hardware designs; there the depth is typically expressed as number of unwinding of transition relation. In case of programs, the depth can express number of unwinding of the whole transition relation as well. However, more fine-grained approach is typically taken in case of program verification; the depth can be expressed as number of instructions, basic blocks or an iteration of a program loop or recursive calls. This depends on the way transition relation is constructed from a source code.

Naturally, the correctness guarantees are given up to certain depth bound; unless it is shown that given depth is enough to reach all the errors. Exact depth limits typically exist in hardware design or (hard) real time controllers.

If the bound is not known in advance, bonded model checkers interactively increase the depth up-to which the error can occur. This process is stopped if an error trace is found or if a user specified time and depth limit is reached. The advantage of the technique are short error traces.

The idea of BMC can be formalized as follow. Let predicate I be the set of initial states where the program (or system can start). The predicate T be the transition relation. In contrast to the above introduction to symbolic representation, it is not a single program statement. Instead it encodes all statements of the program together with the program locations. The predicate T is over unprimed (i.e., input) and primed (i.e., output variables). We will use upper index to denote number of primes added to each variable occurring in the predicate. This means output variables of T^2 are the input variables for T^3. And let F be the predicate describing the faulty (error) states. Note that we use the same convention for faulty states as for transition relation; predicate F^2 describes the faulty states after two (unwinding) steps.

The BMC works as follows. Initially, the BMC checker verifies that the initial states are safe by the following SAT call:

$$I \wedge F$$

Then the checker iteratively increases the bound up-to which the errors can be found. Let n be the current bound. To determine whether faulty state is reachable in n steps, the following formula is created and passed to SAT solver:

$$I \wedge \bigwedge_{i=0}^{n-1} T^i \wedge \bigvee_{i=0}^{n} F^i$$

Completeness. If any of the above formula is satisfiable the system is not safe and the checker can generate a witness error trace from the satisfying assignment of the formula. However, if the formula is not satisfiable the checker cannot conclude that the program is safe as the error can occur at larger depth. So, the checker increases the bound n and continues the search to be safe.

Over the time there were introduced various techniques to detect if the bound is large enough to cover all the faulty states, so called *completeness threshold.* The natural upper bound on number of iterations is a *diameter* of the state space, which is the longest shortest path between two states in the state space. Other well-known threshold is the largest distance of any state to the set faulty states F.

Other technique adds additional assertion for each loop and recursive call. These assertion checks whether the program can continue in looping/recursion beyond the current bound. If these assertions are not violated, the current bound is large enough, so the properties cannot be violated.

Both above approaches suffer from bounds being too large for general programs. Consider, e.g., a program loop having iteration-count based on user input. In such a case, the above techniques cannot found any reasonable bound, as the loops can iterate arbitrary number of times.

Another technique to detect if all traces to faulty state has been considered is based on Craig interpolants. This technique can solve the above problem with the bound if proper interpolants are generated. In context of BMC the interpolant is a formula that specifies a set of program states; in the same way as e.g., the predicate I specifies initial states. However, from the properties of the interpolants it follows, that it over-approximates reachable states and it does not include states from which a faulty state can be reached in given number of unrolling. Let us introduce Craig interpolants in more details, later we will show Craig interpolants are applied in symbolic model checking to obtain a symbolic representation of program states.

2.2.2 Craig interpolants

There exist different definitions of Craig's interpolation Theorem [35]. Below, we use the definition of Craig interpolants as introduced by McMillan in [78].

Given an unsatisfiable formula $\Phi \equiv A \wedge B$, a *Craig interpolant* is a formula I such that the following holds.

I1. $A \Rightarrow I$ and

I2. $B \wedge I \Rightarrow \bot$ and

I3. free variables, function and predicate symbols in formula I have to occur in both A and B.

The requirement I1 states that the interpolant I over-approximates A and the second requirement I2 states that the interpolants is disjunct with B. The last requirement I3 states that the interpolant is over shared variables and symbols. In case of predicate logic, the I3 means that formula Φ can have only the variables common to A and B.

The Craig's interpolation theorem only guarantees that the interpolant exists, but it is not constructive. However, algorithms to compute the interpolants for various theories have been introduced in recent years. Let us mention well-known Krajicek's [73], and McMillan's [78] interpolation systems for predicate logic. These have been generalized in *Labeled interpolation* *system* [41]. These interpolation systems derive the interpolants from refutation proof which can be obtained from SAT solvers. In contrast to them, in [26] is presented a method which does not require resolution proof. The algorithm is based on model enumeration with generation of models. It uses two SAT solvers corresponding to an A and B part of the Φ; the first solver generates models satisfying A. These models are projected to variables common to A and B and generalized w.r.t. A. Then the second solver is used to check if it generalized model is disjoint with B and then it is generalized w.r.t. B. The model generalized w.r.t. both A and B is then added to the interpolant and its negation is added to the first solver as an additional A clause.

Over the time it has been published a lot of interpolation techniques for first-order theories; let mention some of them. In [91] Pudlak introduced a way to compute interpolants for theory of linear inequalities (LI). An interpolation technique for linear integer arithmetic (LIA) has been introduced in [52]. In [79] McMillan extend its interpolation system to quantifier-free fragment of combined theories of linear inequality and uninterpreted functions (LIUF).

Interpolant properties. In many techniques, multiple interpolants are computed from a single formula (so called interpolant *collectives*) Moreover, these techniques often require special properties to hold for these collectives. Great summary the properties commonly used as well as equivalences and necessary conditions on the interpolation techniques can be found in [54].

The *Path Interpolation property* (PI) belongs to the most commonly required ones. Let be given an unsatisfiable formula $\Phi \equiv I \wedge T \wedge F$. Let interpolant R_1 is computed using $A \equiv I$ and $B \equiv T \wedge F$; we say R_1 is an $(I, T \wedge F)$-interpolant. And let interpolant R_2 is computed for the $A \equiv I \wedge T$ and $B \equiv F$; i.e., R_2 is $(I \wedge T, F)$-interpolant. Note that the T part of the Φ has been moved from the B part in case of the R_1 into the A part in case of R_2.

The interpolants R_1 and R_2 have *path interpolation property* iff it holds that

$$R_1 \wedge T \Rightarrow R_2$$

The property can be extended to arbitrary number of the interpolants in a natural way.

In verification tools, the check of PI property is not done explicitly. Instead, if interpolants with PI property are required, the interpolation system which produces interpolants with given properties is used; e.g., McMillan interpolation system.

The interpolants can be seen as a tool which is applied in various ways (not only) in the verification. In next sections, we will show some of their applications in this field.

2.2.3 Interpolation-based model checking

BMC techniques guarantee to detect all reachable errors up to given bound (e.g., number of steps, loop iterations). The BMC techniques are complete if we can show that given bound is large enough to consider all possible traces. In contrast to BMC, the unbounded model checking techniques do not have such a restriction on the bound. They are complete, i.e., the result includes all the possible program traces.

Many Unbounded Model Checking techniques are based on BMC and use interpolants for a termination check [78, 25, 107, 5]. Let us introduce their basic concept. The BMC tool with bound n passes the following formula to a SAT solver:

$$I \wedge \bigwedge_{i=0}^{n-1} T^i \wedge \bigvee_{i=0}^{n} F^i$$

In case, the formula is satisfiable, the assignment corresponds to real error trace thus the tool reports and error and terminate. So, let assume that the formula is unsatisfiable. In case of standard BMC, the tool would continue with increased the bound. In case of unbounded model checking, the tool would perform a termination check, i.e., it will try to prove the faulty states cannot be reached with any bound.

To perform a termination check, first a sequence of n interpolants R from the above formula is computed. Each interpolant is computed using a different split of the formula into A and B parts; the split is shown in the equation below:

$$\underbrace{I \wedge \bigwedge_{i=0}^{k} T^i}_{A_k} \wedge \underbrace{\bigwedge_{j=k+1}^{n-1} T^j \wedge \bigvee_{i=0}^{n} F^i}_{B_k}$$

For an example assume the bound $n = 3$. The BMC formula and the splits into A and B are as follow:

$$BMC_{n=3} \equiv I \wedge T^0 \wedge T^1 \wedge T^2 \wedge (F^0 \vee F^1 \vee F^2 \vee F^3)$$

$$\begin{aligned} R_1 \equiv& \underbrace{I \wedge T^0}_{A_1} \wedge \underbrace{T^1 \wedge T^2 \wedge (F^0 \vee F^1 \vee F^2 \vee F^3)}_{B_1} \\ R_2 \equiv& \underbrace{I \wedge T^0 \wedge T^1}_{A_2} \wedge \underbrace{T^2 \wedge (F^0 \vee F^1 \vee F^2 \vee F^3)}_{B_2} \\ R_3 \equiv& \underbrace{I \wedge T^0 \wedge T^1 \wedge T^2}_{A_3} \wedge \underbrace{(F^0 \vee F^1 \vee F^2 \vee F^3)}_{B_3} \end{aligned}$$

The set of states represented by the interpolant R_1 over-approximates the states reachable after one step (application of transition relation), i.e., it contains all the reachable states and possibly some additional ones (i.e., unreachable). This follows from property I1. Moreover, the R_1 does not include any faulty states; this follows from the interpolant property I2 and the fact that F_1 is part of the conjunction B_1. The similar reasoning can be done for the remaining interpolants R_2 and R_3 (in general for all R_k interpolants).

Moreover, this approach requires interpolants R to have a path interpolation property, sometimes also called an *inductive-step property*. From the construction of the interpolants the PI property gives us that:

$$\begin{aligned} R_1 \wedge T^1 \Rightarrow& R_2 \\ & R_2 \wedge T^2 \Rightarrow R_3 \end{aligned}$$

As we have shown above in the introduction to symbolic representation, this implication gives us, that from each state represented by R_1 after execution for transition T the program will end-up in a state in set R_2. The similar holds for other transitions.

The interpolants R_i are having variables primed i times. It follows from the properties of Craig interpolants and the way the BMC formula is partitioned into A and B parts. The same notation, which is used to add primes to the variables in transition relation T, is used to remove primes from the variable in the interpolants R_i; to remove the primes from R_i, the notation R_i^{-i} is used. This operation will enable us to compare interpolants from different unwinding.

The termination check can be performed, once all interpolants with required PI property are computed. The check tries to found i and j such that $i < j <= n$ and

$$R_j^{-j} \Rightarrow R_i^{-i}$$

This check requires a number of solver calls quadratic to current bound n. If no such pair of interpolants exists, the BMC needs to continue in next iteration with an increased bound. If such a pair of interpolants exist, we have found a backward loop and we proved that system is safe (for any bound). To illustrate that let us take any path p in the system and we will show that all the state on the path are safe. The state on the path after first step will be in the set of state represented by R_1 and since all R_i represents only safe states, the state is not faulty. The state on the path after second step have to be in R_2. From inductive step property all states from R_i have to terminate in R_{i+1} after one step. Moreover, this state is not faulty. The j-th state on the path is represented by R_j.; from the termination check it follows that this state is represented also by R_i (a superset of R_j).

This fact is used to lower the index of the interpolant; any time the interpolant R_j should be applied, the index is changed to i. In general a k-th step will result in a state represented by $R_{min(k,i+(k-i)\%(j-i))}$.

For predicate logic, this process is complete. The number of variables is finite, i.e., number of distinct interpolants w.r.t. implication is finite as well. Thus, for large enough bound, the BMC formula has to be either satisfiable or required pair of interpolants exist.

The proposed process should illustrate the concept of termination check. Different and more efficient termination checks has been introduced. In [107] the interpolants R_i are preserved and conjoined among BMC iterations.

In [78] the interpolants are used differently. It computes a sequence of P_i formulae which over-approximate reachable states. Initially, the P is set to contain the initial states (the $P_0 \equiv I$). To compute P_{i+1}, first the BMC formula where initial state I is replaced by P_i is created. If the formula is satisfiable, the termination check has failed, and BMC bound has to be increased. If it is unsatisfiable, the interpolant R_{i+1} is computed, such that the A part equals to $P_i \wedge T$. The P_{i+1} is then disjunction of R_{i+1} and P_i. Below we illustrate the formula for bound $n = 2$ and splitting into A and B parts.

$$R_{i+1} \equiv \underbrace{P_i \wedge T^0}_{A} \wedge \underbrace{T^1 \wedge T^2 \wedge (F^0 \vee F^1 \vee F^2 \vee F^3)}_{B}$$

$$P_{i+1} \equiv P_i \vee R_{i+1}^{-1}$$

The termination check is done over formulas P; they have a decreasing logical strength. So, they have to reach a fix-point. The termination check validates if the fixpoint (i.e., $P_{i+1} \Rightarrow P_i$) is reached. If so, the system is safe. The P_i are iteratively computed until either the fix-point is reached or the query for R_{i+1} is satisfiable.

Semantics of interpolants. From the interpolant properties and the way they are computed it follows that they represent all the reachable states (in any location) after one application of transition relation T. This is reasonable for models of logical circuits where all operations are executed in parallel and variables represent the values of registers. In case of programs, typically small set of statements from transition relation can be executed. To be able to determine which statements to be executed, the program state needs to incluede an additional variable representing current program location (an equivalent of instruction pointer register in CPUs). In this case, the interpolants do not represent program state at a single location; the location is encoded in the interpolant formula in a complex unstructured way by the program location variables.

The above follows from (i) the way BMC formula is constructed, (ii) split into A and B parts to compute interpolant, and (iii) the properties of Craig interpolants. If the interpolants are computed from the formula representing multiple program paths, then the interpolant will over-approximate all the states on the *boundary* between A and B parts of the formula.

The above shows, why this monolithic approach is not commonly used in software verification. The state-of-the art tools like, CBMC [30] or UFO [5], are using more fine-grained approaches; e.g., at level of basic block or statements.

2.2.4 Abstract reachability graph

The Abstract Reachability Graph (ARG) are used e.g., in the UFO tool; the similar (non-monolithics) representation of transition system can be found in CBMC and other BMC checkers. The ARGs are used to represent the unwinded transition relation in non-monolithic way. They explicitly represent the locations and information about the paths in the program.

At first let us introduce an *execution tree*. Execution tree is constructed from a set of traces, such that common prefixes of the traces shares the same nodes in the tree. Reachability graph then can be seen as an extension of that concept such that same suffixes of the traces also shared the nodes of the graph.

```
  int main(int max) {
1:  int s = 0;
2:  for (int i = 0; i < max; i++) {
3:    if (i % 2 ==  0) {
4:      s = s - i;
      } else {
5:      s = s + i;
6:      assert s >= 0;
7:    }
    }
8:  assert i >= max;
9:}
```

Figure 2.1: Example code

ARG is a directed acyclic graph. Its nodes relate to location in a program and edges relate to statements in program between corresponding locations. The ARG has designated entry node representing an initial location in the program (i.e., start of the `main` function) and a single error node. Each node in ARG has assigned label which over-approximates a state reachable at given node. There is one node corresponding to an error location.

Let us illustrate this concept on an example from Figure 2.1 which iteratively decreases and increases variable s. In Figure 2.2, there is corresponding ARG without labels. Node n_i correspond to a location at line i in the example program. To ease readability, we annotated its edges with corresponding conditions and statements from the program. Note that in this ARG the loop has been unwinded only once; if more unwinding of the loop are necessary, the sub-graph with nodes $n3$ - $n7$ and $n2'$ would be replicated.

Safety. In Figure 2.3 the nodes and corresponding labels (in the curly braces). The way to obtain labels will be introduced later. Also note, that the labels could be simpler if the loop would be unrolled twice.

Now, let us informally introduce basic properties of ARGs. We say that ARG is *well-labelled* iff (1) label of the initial node is *true* and (2) for each edge in ARG, label of the head node of the edge conjoined with the statement of the program corresponding to the edge implies the label of the tail node of the edge. Let us look on the ARG in Figure 2.3. The edge $n5 \rightarrow n6$ needs

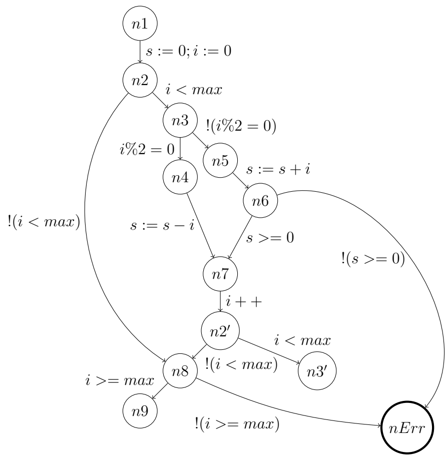


Figure 2.2: Abstract reachability graph

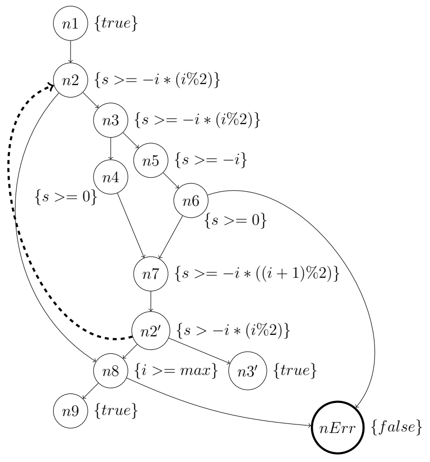


Figure 2.3: ARG with labels

to satisfy:

$$\begin{array}{ccccc} label(n5) & \wedge & [\![stmt(n5 \rightarrow n6)]\!] & \Rightarrow & label(n6) \\ (s >= -i) & \wedge & (s' = s + i) & \Rightarrow & (s' >= 0) \end{array}$$

To avoid naming conflicts, new fresh version (with more primes) of the variables is created each time new value is assigned to the variable. Note, that the well-labelled requirement is similar to the path-interpolation property.

We say that node n is *covered* if there exists predecessor node(s) p in the ARG for the same location as n such that label of n implies the label of the predecessor node p. Let us recall the the introduction about symbolic state matching; this means that states reachable at node n are subset of the states reachable at the predecessor state p. In Figure 2.3 the node $n2'$ is covered (by node $n2$); in the fugure we denoted this using dashed arrow. Both nodes correspond to the same location at line 2 and it holds:

$$\begin{array}{ccc} label(n2') & \Rightarrow & label(n2) \\ (s > -i * (i\%2)) & \Rightarrow & (s >= -i * (i\%2)) >= 0) \end{array}$$

Node is also covered iff all paths to it goes via covered node (i.e., is dominated by covered node). In Figure 2.3 the node $n3'$ is covered, as it is dominated by $n2'$.

We say that ARG is *complete* iff the node is covered or it is completely unwinded (i.e., has the successors and edges corresponding to all its the successors in the program). The ARG in Figure 2.3 is complete. If we would remove covered node $n3'$ and corresponding edge, it would be still complete (node $n2'$ is covered). On the other hand, if we remove node $n4$ the ARG would not be complete.

We say that ARG is *safe* if the error location is unreachable, i.e., label of the error node $nErr$ is $false$. The ARG in Figure 2.3 is safe.

Theorem 1 in [5] claims that if we create a *safe, complete*, and *well-labeled* ARG for given program, then the program is safe.

Computing labels. The label for the ARG nodes can be computed using various techniques, e.g., interpolation, and predicate abstraction. In this section, we focus on interpolation.

To compute interpolants and to derive labels from them, an ArgCond formulate is created. It contains only the nodes and edges via which it is possible to reach an error node. A helper variable c_{n_i} is introduced for each node n_i. Moreover, for each node a formula μ_i in the following form is created:

$$\mu_i \equiv c_{n_i} \Rightarrow \bigvee_{i \to j} (c_{n_j} \wedge [\![stmt(n_i \to n_j)]\!])$$

The formula μ_i can be expressed in a way that if a program reaches a node i (i.e., the premise c_{n_i} of the implication holds) then some out-going edge is taken (i.e., $\bigvee i \to j$ in conclusion of the implication have to hold). In such a case, program reaches tail node of the edge (i.e., c_{n_j} must hold) and executes the action on the edge (i.e., $[\![stmt(n_i \to n_j)]\!]$). In the BMC case, we have been adding primes to all variables (i.e., created fresh variables) in the formula after each unrolling step. In contrast to BMC, in this case fresh variables are created when necessary. In upper part of Figure 2.4 there are μ_i for all the nodes from our sample ARG. There are no μ_9 and $\mu_{n'_3}$ as there is no path via these nodes to an error node. The ArgCond is then a conjunction of all μ_i and c_{n_1}, which force all executions to start in the initial program state.

The ArgCond formula is passed to the solver. If ArgCond is satisfiable, the satisfying assignment gets an error trace. In case of unsatisfiable ArgCond, the refutation proof can be used to compute interpolants and to derive the labels.

For example let compute label for node $n6$. To compute the interpolans we have to split the formula into A and B part. In this case, the node $n6$, resp. μ_6 corresponding to that node, and all its successors, i.e., $n7$, $n2'$, and $n8$ have to be in the B part. All its predecessors, resp. their corresponding

$$
\begin{aligned}
&\mu_1 \equiv c_{n_1} \Rightarrow (c_{n_2} \land s = 0 \land i = 0)\\
&\mu_2 \equiv c_{n_2} \Rightarrow ((c_{n_3} \land i < max) \lor (c_{n_8} \land !(i < max) \land (i = i'')))\\
&\mu_3 \equiv c_{n_3} \Rightarrow ((c_{n_4} \land (i\%2) = 0) \lor (c_{n_5} \land !((i\%2) = 0)))\\
&\mu_4 \equiv c_{n_4} \Rightarrow (c_{n_7} \land s' = s - i)\\
&\mu_5 \equiv c_{n_5} \Rightarrow (c_{n_6} \land s' = s + i)\\
&\mu_6 \equiv c_{n_6} \Rightarrow ((c_{n_7} \land s' >= 0) \lor (c_{n_{Err}} \land !(s' >= 0)))\\
&\mu_7 \equiv c_{n_7} \Rightarrow (c_{n'_2} \land i' >= i + 1)\\
&\mu_{2'} \equiv c_{n'_2} \Rightarrow (c_{n_8} \land !(i' < max) \land (i'' = i'))\\
&\mu_8 \equiv c_{n_8} \Rightarrow (c_{n_{Err}} \land !(i'' >= max))\\
&\quad \textsc{ArgCond} \equiv c_{n_1} \land \mu_1 \land \mu_2 \land \mu_3 \land \mu_4 \land \mu_5 \land \mu_6 \land \mu_7 \land \mu_{2'} \land \mu_8
\end{aligned}
$$

Figure 2.4: The ArgCond formula for ARG from Figure 2.2

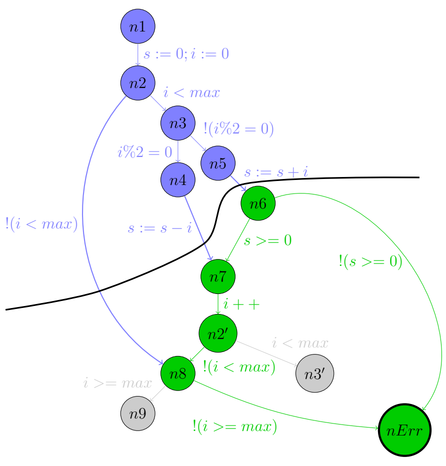
μ formulae, have to be in the A part; i.e., $n1$, $n2$, $n3$, $n5$. The node μ_4 can either in A or B part, let assume it belong to A.

The interpolant over-approximates states on the boundary between A and B parts. This follows from the semantics of the interpolants. In Figure 2.5 we highlighted the split of the ArgCond formula as well as the corresponding ARG. The A parts are in blue, while the B parts are green. The black line highlights the boundary between the A and B parts; the interpolant over-approximates the set of states reachable in at the target node of the crossed edges. The boundary in this case includes node $n6$, as well as node $n7$ (due to the edge $n4 \rightarrow n7$), and node $n8$ (due to $n2 \rightarrow n8$). The information about the reachable states at node $n7$ and $n8$ are irrelevant for the state $n6$ and it only make the interpolant larger.

At node $n6$ the value of variable s has to be non-negative, while at state $n7$ also negative values are permitted. This information about the possibility of negative value needs is the additional fact that have to be present in the interpolant itself.

To get a label from the interpolant, all helper variables (i.e., c_{n_i}) and all out-of-scope variables needs to be removed. The labels should describe the reachable states at node, the above variables do not exist in the state. All the above variables are quantified out, except of the helper variables for a node being considered. This variable can be set to $true$, as all considered paths goes via that node.

The quantification is a well know bottleneck in the verification as processing of the quantified. In case of Boolean helper variables their elimination will increase even more the size of the resulting formula. In case of out-of-scope variables we may end-up with labels having quantifies.



$$
\begin{aligned}
\mu_1 &\equiv c_{n_1} \Rightarrow (c_{n_2} \wedge s = 0 \wedge i = 0)\\
\mu_2 &\equiv c_{n_2} \Rightarrow ((c_{n_3} \wedge i < max)\vee\\
&\qquad (c_{n_8} \wedge !(i < max) \wedge (i = i'')))\\
\mu_3 &\equiv c_{n_3} \Rightarrow ((c_{n_4} \wedge (i\%2) = 0)\vee\\
&\qquad (c_{n_5} \wedge !((i\%2) = 0)))\\
\mu_4 &\equiv c_{n_4} \Rightarrow (c_{n_7} \wedge s' = s - i)\\
\mu_5 &\equiv c_{n_5} \Rightarrow (c_{n_6} \wedge s' = s + i)
\end{aligned}
$$

$$
\begin{aligned}
\mu_6 &\equiv c_{n_6} \Rightarrow ((c_{n_7} \wedge s' >= 0)\vee\\
&\qquad (c_{n_{Err}} \wedge !(s' >= 0)))\\
\mu_7 &\equiv c_{n_7} \Rightarrow (c_{n'_2} \wedge i' >= i + 1)\\
\mu_{2'} &\equiv c_{n'_2} \Rightarrow (c_{n_8} \wedge !(i' < max) \wedge (i'' = i'))\\
\mu_8 &\equiv c_{n_8} \Rightarrow (c_{n_{Err}} \wedge !(i'' >= max))
\end{aligned}
$$

$$
\begin{aligned}
\textsc{ArgCond} \equiv\ & c_{n_1} \wedge \mu_1 \wedge \mu_2 \wedge \mu_3 \wedge \mu_4 \wedge \mu_5 \wedge\\
& \mu_6 \wedge \mu_7 \wedge \mu_{2'} \wedge \mu_8
\end{aligned}
$$

Figure 2.5: Splitting of ARG and ArgCond into A (blue) and B (green) parts to compute labels for node $n6$.

Let assume the interpolant for node $n6$ is I_6. In you case they are no out-of-scope variables, however the helper variables c_{n_6}, c_{n_7} and c_{n_8} have to appear in the interpolant. To produce a label for node $n6$, we have to remove c_{n_7} and c_{n_8} using quantifiers, and c_{n_6} assign to *true*. The label will be $label(n6) = \forall c_{n_7}, \forall c_{n_8}\ I_6[c_{n_6} \leftarrow \top]$.

Summary. As we have seen, the interpolant over-approximates reachable program states at the boundary between the A and B parts. If we are interested in program states at a location (e.g., at an ARG node), then states from the other location are irrelevant and only increase the size of the formula. This problem is not specific to ARG; it originates in the properties of the interpolants. If the input formula (i.e., transition relation) represents multiple program paths and the interpolant for a selected location computed, the problem will occur.

2.2.5 Interpolant size reduction

Craig interpolants in propositional logics are typically derived from refutation proofs [73, 91, 78, 74, 41, 98]. Interpolant has, to some extent, the same structure as the refutation proof. The interpolant size is linear in the size of the proof. The main limitation is the need for a proof of a manageable size. The resolution proof can be of exponential size compared to the size of the input formula; moreover, the solvers are designed to produce neither minimal nor small proofs, but to decide satisfiability quickly.

There are two principle ways to obtain smaller interpolants; (i) reduce the size of a proof used to compute the interpolant and (ii) reduce the size of the interpolant formula.

The former techniques are applied once a proof is constructed and before interpolants are computed; this way interpolant collectives having certain properties can be computed. The later techniques are either applied as a post-processing step or are applied as an optimization during the interpolant computation. As these techniques modify single interpolant, logically equivalent formula have to be produces if interpolant collectives should be derived. Moreover, in case of collectives the optimization needs to be applied on each interpolant whereas in case of proof reduction techniques, only one run is required.

2.2.5.1 Proof reduction techniques

Proof reduction techniques identify and eliminate redundancies in the proofs. Many techniques have been introduced [8, 53, 12, 94, 48]; below, we first define a resolution proof and later, we introduce the reduction techniques.

Proofs. Proof can be seen as a directed acyclic graph (DAG) with a single root node, where the empty clause (i.e., $\bot$) is derived in case of refutation proof.

Before we formally define a resolution proof, a few auxiliary definitions are needed. A *literal* is a Boolean variable l or its negation $\bar{l}$, a *clause* is a finite disjunction of literals. In this thesis, we use angle brackets $\langle\Theta\rangle$ to denote the clause built over the literals in Θ. A clause can be also built from multiple sets of literals; denoted as in $\langle\Theta,\Theta'\rangle$ or $\langle\Theta,\{l\}\rangle$. The latter is often abbreviated as $\langle\Theta,l\rangle$.

Let $\langle\Theta,p\rangle$ and $\langle\Theta',\overline{p}\rangle$ be clauses. Using variable p as the *pivot*, their *resolution* yields the clause $\langle\Theta,\Theta'\rangle$ – *resolvent.* We write $Res(\langle\Theta,p\rangle,\langle\Theta',\overline{p}\rangle,p)$ for the resolvent of clauses $\langle\Theta,p\rangle$ and $\langle\Theta',\overline{p}\rangle$ using variable p as the pivot. The clauses $\langle\Theta,p\rangle$ and $\langle\Theta',\overline{p}\rangle$ are called *antecedents*.

For literal l or a set of propositional formulae A, $\mathsf{Var}(l)$ and $\mathsf{Var}(A)$ denote the variable of l and the set of variables in the formulae of A, respectively.

We adopt the definition of the resolution proof from [41]: a *resolution proof* R for a CNF formula Φ is a tuple (V,E,cl,piv,s), where V is a set of vertices in the proof, $E\subset V\times V$ is a set of edges forming a full binary DAG (i.e., all the vertices except for the leaves have the in-degree 2). The sink vertex s has the out-degree 0. Each vertex $v\in V$ is associated to a vertex-clause specified by the $cl(v)$ function. Each vertex clause of a leaf vertex v corresponds to a clause from the input formula Φ (i.e., $cl(v)\in\Phi$). Each inner vertex v represents resolution of its antecedent vertex-clauses (specified by cl) using the pivot $piv(v)$; formally, for each inner vertex v, there exist edges $(v_1, v), (v_2, v) \in E$ such that $cl(v) = Res(cl(v_1), cl(v_2), piv(v))$. A *refutation* derives the empty clause in the sink vertex s; formally $cl(s) = \bot$.

Proofs are often considered as trees for purposes of interpolation. Similar simplification cannot be done for proof reduction techniques introduced below.

RecyclePivot. We say that a proof is *regular* if each variable is used at most once as a pivot on each path from a leave to the sink vertex; the proof is irregular, otherwise.

Let us consider a proof represented as a tree. In such a case, on paths having multiple resolutions with same pivot variable, it is possible to remove all these resolutions except for the one closest to the sink. After the removal, it is required to reconstruct the proof; the reconstruction further reduces the size of the proof. This way, an irregular proof can be converted into a regular one.

In case of a proof represented as a DAG, more attention is needed. In a DAG, there can be more paths from a vertex to the sink. To remove vertex v (with pivot variable p) from the proof, it is required that on *all* the paths from v to the sink, there is another resolution with the pivot p. Then, the proof has to be reconstructed to obtain a valid resolution proof; the same procedure as for trees can be used. In case a non-tree proof, regular proofs cannot be always created this way. The reduction can be performed in a single pass via the resolution proof, because the proof is acyclic.

This reduction has been introduced in [8] as RecyclePivot operation and it considers only tree-like segments of the proofs. It has been improved to take whole proof into account in all-RmPivots [53] and RecyclePivotsWithIntersection [48, 94].

RecycleUnits. The DPLL solver increases its knowledge during the search via so called *learnt clauses*; in some cases, a learnt clause is a unit clause (e.g., $\langle x \rangle$ or $\langle \overline{x} \rangle$). The RecycleUnits reduction applies this clauses to prune a proof. First, it identifies the proof vertices having a unit vertex-clause. Then, it finds the vertices with resolution having the same pivot variable as the variable on some unit vertex-clause. The antecedents of vertices identified in the latter step with the literal of the unit clause, are replaced by the vertex with the unit clause.

To illustrate it on the example, let us have a vertex v_u with unit clause $\langle \overline{x} \rangle$ (found in the first step), the vertex v representing resolution with pivot variable x (identified in the latter step). Antecedents of the vertex v are v^+ and v^- with vertex clause $cl(v^-) = \langle C, \overline{x} \rangle$. After the transformation, the vertex v will have the antecedents v^+ and v_u (instead of v^-). Note that the vertex-clause of v^- is subsumed by the unit clause of v_u (i.e., $\overline{x} \Rightarrow (\overline{x} \vee C)$). The above transformation can be applied only if it does not introduce a cycle in the proof; i.e., if the vertex being modified (i.e., v) is not (transitive) antecedent of the vertex with the unit clause (i.e., v_u). The proof needs to be reconstructed (in the same way as in RecyclePivot) to obtain a valid resolution proof.

Reduction via subsumption. In [11] a reduction method based on subsumption is introduced. First, for each vertex v, a set of literals σ_v is computed which are resolved on all paths from the given vertex to the sink. Then, it attempts to find a pair of vertices v and u such that the vertex-clause of u subsumes the vertex-clause of v disjoint with σ_v; i.e., $cl(u) \Rightarrow (cl(v) \vee \langle \sigma_v \rangle)$. If the vertex v is not a transitive antecedent of u, then v is replaced by u; and the proof is reconstructed. If that was the only usage of the vertex v in the proof, then it is removed and the size of the proof is reduced.

RecycleUnits can be seen as a special case of this reduction; the RecycleUnits considers only the unit clauses for the subsumption. Also, the RecyclePivot and its extensions are a special case of this reduction.

In [11], the authhors study the effect of the reduction on the interpolants; the authors show that RecyclePivot may add variables into the interpolant. Not all shared variables have to appear in the interpolant, but RecyclePivot can cause that additional shared variables will appear in it. Moreover, authors introduced additional requirements which guarantee, that no additional shared variables will appear in the interpolant if the reduction is applied.

LowerUnits. The LowerUnits [48] and PushdownUnits [94] reductions use unit vertex-clauses in a different way than RecycleUnits. This reduction picks a unit clause and detaches its vertex from the proof; then the proof is reconstructed. Since the resolution with the unit vertex-clause has been removed, the negated literal of the unit clause needs to be inserted into the vertex-clauses of the vertices on the paths from the removed vertex to the sink. If a negated literal has been added into the vertex-clause of the sink, then the vertex with the unit clause is attached to the sink vertex; i.e., new sink vertex is created having the original sink vertex and the unit-clause vertex as its antecedents. This way the refutation is restored. The unit clause is pushed down the proof closer to the sink vertex.

This transformation pays off if the unit clause is applied in the proof many times, since these resolutions are removed and replaced with only one. Also, if the variable of the unit clause is resolved on all paths from the detached vertices to the sink, then the clause of the original sink vertex after the proof reconstruction will remain $False$ and the unit vertex will not be reattached. Moreover, re-attaching a unit-vertex to the sink adds the resolution of the variable to all paths in the proof; this helps RecyclePivot to identify more resolutions to be removed.

This reduction has been generalized in LowerUnivalents [12] which is able to move vertices having more complex clauses.

Local proof transformations. The aforementioned reductions operate and analyze the whole proof. In [94], a set of local rules to transform the proof is introduced; the rule is defined on a local context and it specifies condition allowing the application of the rule, and a way the proof should be modified. The local context is represented by the vertex being considered and its predecessors (up to depth 2). Local transformation rules are of two kinds – *swapping* rules, which are revertible and do not change the clause derived in the root vertex of the local context, and *reducing* rules, which yields a stronger (i.e., smaller) clause in the root vertex of the context. The local transformations as well as some of the global reductions have been implemented in Periplo [93].

Structural hashing. In resolution proofs, all derived clauses are determined by their antecedents. Thus, it is possible to use the input clauses together with the structure of the resolution sub-proof to determine a vertex clause. Instead of computing and comparing vertex clauses, *structural hashing* [33, 94] can be used to identify the vertices with the same vertex clauses.

The structural hashing algorithm processes the proof from the leaves to the sink; i.e., it sorts the vertices topologically. It assigns a unique id to each vertex; for a leaf vertex, a new id is assigned immediately. In case of an inner vertex, the algorith decides whether the vertex having provably the same vertex-clause have not been already processed. To identify the above case, the algorithm maintains a hash-map; each entry in the map corresponds to a vertex. The map key is a pair of ids corresponding to the left and the right antecedent of the vertex; the value is the vertex and its id. If the structural hashing algorithm visits an inner vertex, it gets ids of its antecedents; then it checks the hash-map if the pair is already stored. If the hash-map contains the key, the equivalent vertex is used instead of the current one; the structure of the proof is changed immediately. If the hash-map does not contain the given pair of ids, a new id is assigned to the current vertex and it is inserted into the map.

2.2.5.2 Interpolant reduction techniques

Craig interpolant is a logical formula having certain properties; thus, methods used to reduce the size of formulae are often applied on them – constant propagation, structural hashing, BDD-based sweeping. Not much research effort has been devoted specifically to reduction of interpolant formulae.

And-Inverter Graphs. Often, And-Inverter Graphs (AIGs) are used to represent interpolants. Some interpolant reduction techniques operate on AIGs' in the following, we introduce them.

AIG can be seen as a directed acyclic graph having four types of nodes: (1) source nodes, (2) sink node(s), (3) internal nodes, and (4) the constant $\top$ node. *Primary input* (PI) (i.e., an input variable) corresponds to a source node. *Primary output* (PO) is a sink node which has exactly one predecessor. An internal node represents a 2-input AND gate. An edge in an AIG can be either the *inverting* or non-inverting (i.e., normal).

Each node and edge in AIG is associated with a Boolean function. The function for PI is a Boolean variable. A non-inverting edge has a function of its source node; an inverting edge represents a function complementary to its source node. For an internal node, the function is the conjunction of the functions of its incoming edges. The Boolean function of a PO equals to its incoming edge and it is the formula represented by AIG.

Constant propagation. An interpolant is typically computed from resolution proof in the following iterative process. For each vertex of the proof, a partial interpolant is computed from the partial interpolants of the antecedent vertices. The process starts in the leave vertices and continues until the sink vertex is reached; in case of the sink vertex, Craig interpolant is computed.

In the leave vertices of a proof, partial interpolants are quite often a logical constant (i.e., $True$ or $False$). The constant can be used in the successor vertices to simplify partial interpolants. To demonstrater how constant propagation works on an example, let partial interpolants of the antecedents of vertex v be I^+ and $I^- \equiv \bot$ (i.e., I^- be a constant) and the partial interpolant for the vertex v be $I_v \equiv I^+ \wedge I^-$. Since I^- is known to be a constant $False$, it is possible to conclude that $I_v \equiv I^+ \wedge \bot \Leftrightarrow \bot$; so the vertex interpolant I_v is a constant as well.

Observability don't cares. In [23, 24], *Observability don't cares* (ODC) are used to reduce the size of interpolants. The reduction attempts at identifying an input of a formula, which cannot influence its satisfiability and thus it can be removed (replaced by a constant). In case of interpolants, the reduction is applied to their sub-formulae.

In [23], the ODC relies on the following identities.

$$f(X,a) = a \wedge g(X,a) \equiv a \wedge g(X,\top)$$
$$f(X,a) = a \vee g(X,a) \equiv a \vee g(X,\bot)$$

The reduction attempts at finding a (sub-)formula representing function $f(X,a)$ in which the variable a is used twice; first, directly in the top level operation $\wedge$ (resp. $\vee$) and then in a sub-formula $g(X,a)$ (which represents partial interpolant of an antecedent vertex). Let us assume a sub-formula in the first form (i.e., $f(X,a) = a \wedge g(X,a)$). Observe, if variable a is assigned a constant $False$, then function $f(X,a)$ has to evaluate to $False$ as well; the value of $g(X,a)$ is not important in this case. Thus, it is possible to replace variable a in sub-formula $g(X,a)$ by a constant $True$ and propagate the constants as in the reduction technique above.

Note that in [23] ODC reduction is applied onto AIGs representation of an interpolant formula.

Structural hashing. Structural hashing can be applied onto the interpolants as well. In case of proofs, all proof vertices represent the same operation (i.e., resolution) and the order of operands is important. In case of interpolants, structural hashing is applied on a parse tree of the formula, where internal vertices represent logical operations $\wedge$ and $\vee$. In contrast to proofs, hash-map keys have to include the operation as well as the operands. Moreover, since logical operations $\wedge$ and $\vee$ are commutative, it is possible to normalize the order of the operands in the key (e.g., sort operands by their ids).

BDD-based sweeping. The sweeping methods have been used in equivalence checking [76]. The reduction uses (reduced ordered) binary decision diagrams (BDDs) to detect functionally equivalent parts of a formula (resp. circuit); equivalent Boolean functions yield the same BDDs, thus equivalence checking is reduced to comparison of the BDD structure. The main disadvantage of BDDs is their size, which can blow up exponentially for certain formulae. To cope with a given problem, BDDs are not applied to the whole (interpolant) formula, but only to its sub-formulae.

In [76], the BDD-based sweeping algorithm maintains a work-list of BDDs ordered by their size. The algorithm operates on the AIG representation of the formula; each BDD belongs to some AIG node (i.e., a Boolean function). The algorithm starts from the leaves representing a single variable (i.e., primary inputs), and it creates a BDD representation for them. Then, it iteratively picks the smallest BDD from the work list and it takes associated AIG node to the smallest BDD. Then, the algorithm goes via the outgoing edges of the node (in other words, it attempts at finding all usages of the associated sub-formula) and for each successor node, it tries to creates a BDD representing its Boolean function.

If some previously visited node has the same BDD as the successor (i.e., represent the same Boolean function), then the AIG graph is immediately reconstructed, the previously visited node is used instead of the successor and the edges are reconnected (i.e., the redundant sub-formula is replaced by the equivalent one discovered in previous steps). If no node with the same BDD exists, then the size of the BBD is checked; if it is below the threshold then the BDDt is added into the work-list. The use of the threshold prevents from the blow-up of the BDDs. In [24] the BBD-based sweeping as introduced above has been applied onto interpolants.

Other variants of the sweeping algorithms include SAT-based [75] and cut-based sweeping [42]. Instead of BDDs, the former technique uses SAT solver to decide whether Boolean functions of some AIG nodes are equivalent or not. The cut-based sweeping is using truth-tables to represent Boolen functions (i.e., for functions of k variables, it uses a vector of 2^k bits).

In [55], the above sweeping strategies are iteratively applied in various order onto formulae and the effect on their size as well as on performance of the IC3 [15] verifier is studied. Authors also studied various heuristics to cut AIGs into layers.

Summary. All the aforementioned reductions (i.e., the proof reductions as well as the interpolant reductions) are independent of the way the formula will be used later in the verification process. On one hand, this means that they are general and their applicability in not restricted to some verification technique. On the other hand, these reductions cannot target inefficiencies specific only a to given verification problem; in particular, for Abstract Reachability Graphs, these reductions are not able to focus interpolant to a selected ARG node, and to remove program states from other nodes at the boundary between A and B parts.

3

Goals revisited

The overall goal of the thesis is to identify inefficiencies in program state representation and to design and implement novel techniques, which will result in smaller representation of program states.

In the previous chapter, we identified shortcomings of state representation in the state-of-the-art model checking techniques. As we have shown, these techniques include unimportant pieces of information into their program state representation. This makes the state space larger and model checker slower, since larger state representations have to be processed.

In particular, (i) there is no efficient technique to identify dead program variables in the heap of multi-threaded programs. In case of symbolic representation, we have shown that (ii) the commonly used Craig interpolants, due to their semantics, include states from multiple program locations if applied on a formula representing multiple program paths or the whole transition system. If Craig interpolants are used to compute states for a single ARG node, the included states from other locations only increase the size of the interpolants.

G1 *Design techniques to eliminate dead variables on the heap.*

G1a *Dead variable analysis for heap.*

Design sound dead variable analysis to identify fields of the objects on the heap whose value will not be read anymore. The analysis should be tailored for explicit state representation and should support multi-threaded programs.

G1b *Dead variable reduction with state matching.*

Introduce a dead variable reduction based on the above analysis. The reduction should identify dead values from state vectors. Design a safe technique for matching state vectors ignoring dead parts.

G2 *Design technique to focus Craig interpolants to single program location.*

G2a *Interpolation technique.*

Introduce a way to specify the ARG nodes (i.e., program locations) being considered. Develop a novel interpolation technique that computes interpolants focused on a considered ARG node.

G2b *Interpolant properties.*

Verification tools often require additional properties that the interpolants have to satisfy. Show the properties of the interpolants computed by the proposed method.

G2c *Interpolant sizes.*

The size of the interpolants computed from a refutation proof is one of their well-known bottlenecks. The goal is to evaluate the introduced technique w.r.t. the size of the interpolants and to compare it with other techniques.

With respect to the differences among explicit state and symbolic representation of program states, the thesis describing our contribution is divided into two chapters. The following chapter focuses on explicit state model checking, while in the chapter 5, we present a novel interpolation technique, which solves the goals for the symbolic representation.

4 Dead variable reduction

The on-the-fly model checkers do not construct full state space and then check the properties on the generated state space. Instead, they generate successor states from the current one and for each reached state they check, whether the property of interest holds or not. Applying optimizations is challenging in such a setting. The optimizations used by these model checkers are typically very conservative. The successor states (i.e., the future behavior) of a currently reached state are unknown at the time the state is reached. Sound optimizations have to assume that the program can do anything the programming language permits. This makes optimizations such as Partial-Order-Reduction unnecessarily imprecise. In case of dead variable analysis, information about future behavior of a state have to be obtained in a different way, e.g., using static analysis. Unless additional analysis is performed, a model checker does not know the variables read by the program in the future executions from the current state. It had to assume that all variables can be read (as permitted by the programming language), thus no variable can be identified as dead.

Program states are complex. They contain a large amount of highly structured data. The complexity comes mostly from the expressiveness of programming languages (dynamic object and thread creation) and the size the standard libraries. Memory footprint of a minimal run-time environment is non-trivial. States of the simplest Java program – *HelloWorld* – have to contain representation of the `System` class and streams (`PrintStreams`) used to print out characters. Explicit state model checkers typically do not model the system libraries on the API level, instead, they model only lower-level parts i.e., system-calls and native methods. This pragmatic approach (1) saves the amount of work needed to create a usable model checker, since the libraries are complex and feature-rich, (2) helps to meet precise semantics of the library code and (3) permits to detect invalid usage of the library code. As to (3) consider parallel usage of a single `HashMap`[1] from multiple threads. If the `HashMap` would be modeled and all operations were atomic (as it is a typical for modeled methods), it would be impossible to detect errors occurring when the inserting thread causes resizing of the internal hashing table while another reader threads still uses the map.

Program state consists of call-stacks, global variables (in case of Java represented by static fields) and heap. It is not surprising that the heap comprises the largest part of program state; in the Java language all local variables and static fields, which are not of a primitive type, point to some instance on the heap (unless holding `null`).

As we have identified in the background section; there is no DVA technique suitable from multi-threaded programs which are extensively using heap – a domain of the highest interest for explicit state code model checkers. Existing techniques either rely on static analysis and do not consider heap [13, 45, 109, 77] or are based on dynamic observation of program state (so they support heap) but do not support parallelism [99].

In this thesis, we present two dead variable analyses which were designed to (1) identify *Dead fields of Heap instances* in multi-threaded programs and (2) to be compatible with state matching as used in explicit state model checkers. We implemented both analysis in JPF and evaluate them on non-trivial multi-threaded java benchmarks. The first analysis is called *Hybrid DVA* is based on static analysis, the second analysis is called *Dynamic DVA* as it purely relays on information observed during model checker run-time.

4.1 Overview

[67] The contribution of DVR is two-fold. First, it reduces the state space, since states differing only in values of the identified dead variables are matched. This means that only a single representative of each set of matched states is explored. Second, state matching (i.e., canonicalization, hashing) can process only the live parts of state representation (ignoring dead parts), thus making the whole state matching process faster. This is also of a particular importance, since explicit-state model checkers spend a large amount of their runtime (approx. 30%) by state matching [84]. The former effect applies both for DVAs over local variables as well as for those focusing on

[1]Note that the standard `java.util.HashMap` is not thread-safe, the thread safe-implementations can be found in `java.util.concurrent` package.

Thread $T1$

```java
public void run() {
  ...
  assert(tree.contains(5));
}
```

Thread $T2$

```java
public void run() {
  ...
  assert(tree.contains(1));
}
```

Figure 4.1: Java program composed of two threads where the `tree` variable contains red-black tree from Fig. 4.2.

the whole program state. On the other hand, the second effect can be observed only if a large enough portion of a program state is identified as dead, which can be expected only for the heap. Instead of ignoring dead variables, some DVR implementations set their value to a predefined constant (e.g., 0 or `null`). In such cases, the former effect is eliminated, since the size of the program state is not reduced at all. Note that the more precise DVA is, the more these effects manifest themselves.

Let us illustrate the aforementioned effects on the Java program in Fig. 4.1 and the red-black trees in Fig. 4.2 stored in the `tree` variable. The corresponding class is listed in Fig. 4.3. The tree is shared among threads and we assume that the program can generate either of them. No assertion is violated irrespective of whether either the left or the right tree has been generated. While the colours in red-black trees are used only in modifying operations (insertions and deletions), the `contains` operation does not access them and thus the variables (fields) representing the colours of nodes are dead. The same holds for the right descendant of the root node holding value 5. Since operation `tree.contains(1)` reads only the left descendant of the root node, and operation `tree.contains(5)` accesses only the value of the root node, the whole right sub-tree is dead. It means that the program states where the `tree` variable holds the left resp. right tree of Fig. 4.2 are

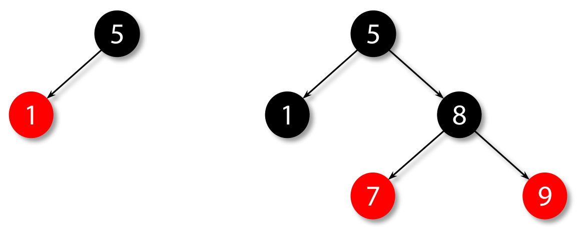


Figure 4.2: Two different red-black trees, which can be identified as equivalent using our DVA for program in Fig. 4.1.

equivalent w.r.t. dead variables; the model checker can explore the successors only for the state that is reached first and does not need to re-explore the successors of the (equivalent) state reached later. This way the state space is reduced.

To illustrate the second effect, let us inspect the parts of the tree (in other words the parts of the program states) which are processed by the state matching (in the case of JPF, the data appearing in the state vector). Note that the code in Fig. 4.1 does not modify the tree below the lines 3 resp. 7. First of all, the `colours` are dead and thus can be ignored by state matching. More importantly, the right descendant of the root node (`tree.right`) is also dead and can thus be omitted from state matching as well. This simple fact causes the whole sub-tree `tree.right` (composed of nodes 7, 8, and 9) not to be accessed and thus it can be omitted from state matching. The latter effect speeds-up verification also if the state space is not reduced (i.e., the former effect does not apply). A prominent example of this effect is representation of environment variables, which form a non-negligible part of the state, however are seldom accessed by the program.

In order to be useful, DVR itself and DVA as its part need to be fast enough to pay off, while still having a modest memory demand. The DVR implementation should be also compatible with all other state matching optimizations.

Another important question is how to cope with the states having different sets of dead variables; in our example, each tree has a different set of them. The tree at the right-hand side has dead variables in nodes 7, 8, and 9, which do not exist in the tree at the left-hand side.

4.2 Running example

We have developed two versions of DVA for multi-threaded Java programs; the first one, called *Dynamic DVA* (DDVA), aims at precision. The second one, called *Hybrid DVA*, uses static analysis (for multi-threaded programs), and combines the results of the analysis with the knowledge from the dynamic (runtime) program state. This lightweight analysis is designed to be fast and easy to integrate into state matching.

We illustrate our analyses using the example from Fig. 4.1 and the `tree` variable holding the right-hand-side tree from Fig. 4.2. The tree nodes are stored using the data structure from Fig. 4.3. We add a suffix holding the stored value to distinguish the instances of `TreeNode`s from each other; e.g., the root tree node is denoted as `TreeNode@5`.

```
public class TreeNode {
  public int value;
  public Color color;
  public TreeNode left;
  public TreeNode right;
  ...
}
```

```
  boolean contains(int i) {
    int v = this.value;
    TreeNode child;
    if (v == i) return true;
    if (v < i) child = this.left;
    else child = this.right;
    if (child == null) return false;
    return child.contains(i);
  }
}
```

Figure 4.3: Red-black tree node representation

Before we show how our analyses work, let us focus on the state space a bit. Thread $T1$ will call the `TreeNode.contains` method, which will read `TreeNode@5.value` only. The $T2$ also calls the `TreeNode. contains` method, which in this case reads `TreeNode@5.value`, `TreeNode@5.left`, and `TreeNode@1.value`. Although both threads are independent of each other and thus it is enough to explore only one of their interleavings, on-the-fly model checkers cannot be aware of this fact (since this requires knowledge about future behaviour of the program). To be safe, they assume, e.g., a possible conflicting write to a given shared field by another thread and create a non-deterministic choice at each of the field reads [86]. In other words, the model checker will generate 15 unique states out of which 8 states contain a non-deterministic scheduling choice deciding whether $T1$ or $T2$ will be scheduled for execution (Fig. 4.5).

4.2.1 Dynamic analysis

Our dynamic analysis tracks field reads and writes executed by the program and based on them it identifies live parts of the program state (i.e., live addresses). During the depth-first search (DFS), the model checker maintains two data structures related to visited states: (i) a stack of states currently present at the DFS stack, for which full state vectors are stored to enable detection of state-space cycles, and (ii) a set of visited states, for which the pairs $\langle$*live addresses*, *reduced state vector*$\rangle$ are stored; these are used for state matching. The reduced state vectors, which are stored after a state and its successors are fully explored, do not include dead variables.

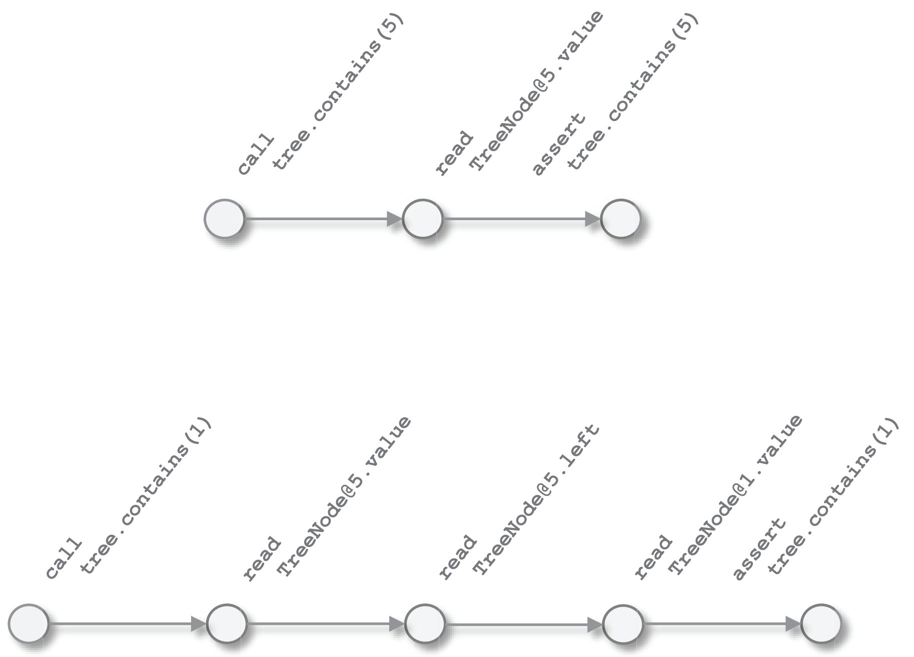


Figure 4.4: Future transitions of thread $T1$ resp. $T2$.

To illustrate this process on our motivation example from Fig. 4.1, let us assume that the model checker uses the DFS exploration strategy and that thread $T1$ has a higher priority than $T2$; in other words, if both threads $T1$ and $T2$ can be scheduled at a state, the successor where $T1$ is executed is explored first. The corresponding state space is shown in Fig. 4.5.

The model checker is in state $S1$ just before the `Tree.contains` calls where both threads can be scheduled, thus thread $T1$ is executed first. The model checker executes the call to the `Tree.contains` and continues execution until it encounters the read of `TreeNode@5.value` field. Just before the field read (in state $S2$) it creates a non-deterministic scheduling choice (to be able to read the value possibly modified by other threads) and the execution of thread $T1$ continues. The first executed instruction is the actual read of `TreeNode@5.value`, which caused the scheduling choice; the dynamic analysis stores the fact that at the instance `TreeNode@5`, its field `value` has been read. First, note that our dynamic analysis distinguishes among object instances. Also note that our implementation stores this information

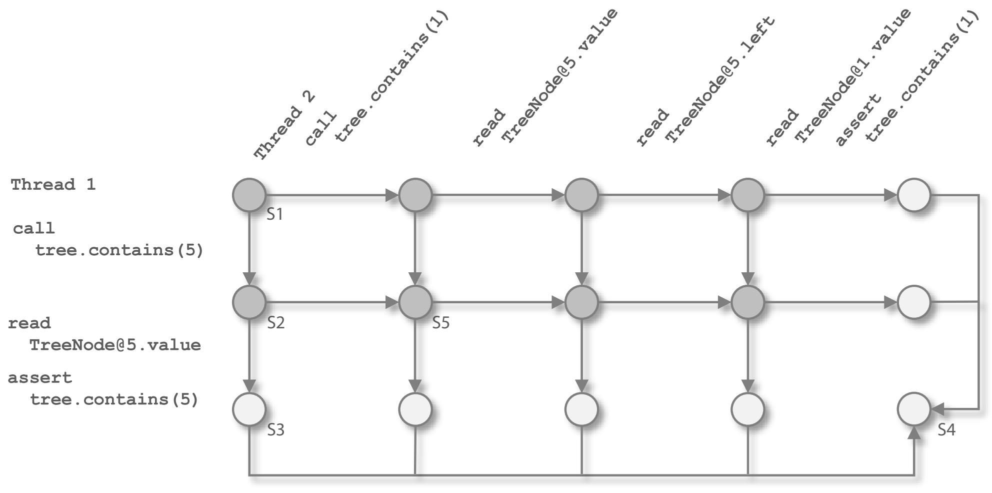


Figure 4.5: State space of T1 and T2. States with decision choices are in dark grey.

during forward exploration (because JPF does not provide another way to obtain this piece of information). The model checker then continues in forward execution of thread $T1$, it executes the assert statement and thread $T1$ terminates (state $S3$). Since in JPF, a transition has to be executed by a single thread, a new transition is created and, without any non-determinism, the only remaining thread $T2$ is scheduled. The model checker then executes instructions of thread $T2$; among others, the reads of `TreeNode@5.value`, `TreeNode@5.left` and `TreeNode@1.value`. Our analysis stores all these field reads (in the execution order) as in the previous case; the field writes are stored in the same way. All these reads are executed in a single transition, since $T2$ is the only runnable thread; the transition ends in state $S4$, where thread $T2$ terminates.

If the model checker reaches the visited or final state and it starts backtracking, our dynamic analysis uses the stored information to compute live parts of program states. In the example, the model checker has reached a final state and thus dynamic analysis marks all fields in all instances as dead (the program cannot read any of them). The model checker then backtracks from $S4$ to $S3$; during backtracking the dynamic analysis processes the stored information for backtracked transition in the reverse order. Thus, due to field reads, the analysis will mark `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value` as live at the beginning of the transition $S3 \rightarrow S4$, while all other data remain dead. This includes, if the right-hand side tree is stored in the `tree` variable, the instances (and their fields) `TreeNode@7`,

State	TreeNode@5.value	TreeNode@5.left	TreeNode@5.right	TreeNode@1.value	TreeNode@1.left	TreeNode@1.right
$S1$	✓	✓		✓		
$S2$	✓	✓		✓		
$S3$	✓	✓		✓		
$S4$						
$S5$	✓	✓		✓		

Table 4.1: Live parts of selected states in Fig. 4.5.

`TreeNode@8`, and `TreeNode@9` as well as the colors of all `TreeNode` instances. Since there is only a single outgoing transition from state $S3$, the live addresses of state $S3$ are those at the beginning of the transition $S3 \to S4$. After the part of the entire state space in Fig. 4.5 is explored, the live addresses of $S1$ are stored. During exploration of another execution (most notably when the other tree from Fig. 4.2 is created by the program), the state corresponding to the beginning of $T1$ and $T2$ is matched with the one already explored earlier, because the live part of the stored state matches the corresponding part of the state being explored. If no DVR had been performed, the corresponding part of the state space (having the same shape but differing in the `tree.right` field) would have had to be explored.

State matching. Once the live resp. dead addresses are known at a state, dynamic analysis computes a reduced state vector and stores the pair consisting of the set of live addresses and the reduced state vector in the set of visited states. The reduced state vector contains only live values from the static and instance fields, and call-stacks of the threads. Since local variables are not considered by our analysis (i.e., none of them is removed), the complete call-stacks (without any reduction) are stored. In order to speed up state matching, the same optimization as in [99] is applied – for each call-stack, we store a list of live addresses used to compute the reduced state vectors corresponding to this call-stack.

The reduced state vectors enable to detect those states which differ in dead parts of the heap only. In our example, if the model checker backtracks over state $S1$, where the example begins, and generates another state with the tree from the right-hand side of Fig. 4.2, our state matching procedure will match this state with $S1$.

If the model checker reaches a state, it has to check whether the state (or an equivalent one) has been already visited or not, so the state matching is initiated; the model checker uses the call-stack of the current state to find a list of live addresses associated with the call-stack. For each set of live addresses from the list, a reduced state vector of the current state is computed and it is checked whether the pair of live addresses and the reduced state vector is among the visited ones. If so, the state is reported as visited.

4.2.2 Hybrid analysis

In contrast to our DDVA, where we addressed precision in the case of multi-threaded programs, our hybrid analysis targets scalability and speed, instead.

Our hybrid analysis consists of two phases. The first phase is executed before the model checker runs. Cheap simple inter-procedural, context-insensitive static analysis is performed, identifying the fields that may be read (i.e., *used* in the terminology of DVA) by a given method before it terminates (up to the end of its body). The analysis gets only a partial view on the liveness; on the other hand, it is cheap to compute. In other words, this phase provides an under-approximation (i.e., a subset) of the live fields at a state. Additional pieces of information are obtained in the second phase, performed at model-checker runtime just before each state matching. Note that the hybrid analysis does not distinguish among object instances; a field is live either at all instances or at none. Therefore, field writes (i.e., *def*s) are not tracked.

In the second phase, the analysis uses information from the call-stacks of the threads; i.e., the locations (instruction pointers) in the methods at the call-stacks. It traverses the call-stacks in the top-down manner and extends the scope for which the liveness is considered. Using the location in the current method (top stack frame), the analysis uses the results from the first phase to obtain fields read before the current method terminates; these fields will be marked as live. Going down the call-stack, the analysis adds the live fields from the caller using the results from the first analysis computed for the location right after the call. This way, we obtain the fields which can be read by a thread before the considered method terminates. Once the whole call-stack is processed, we know which fields the thread may read before it terminates. The above processing of the call-stack is done for each thread and the results are joined together, thus obtaining the final result – the fields which can be read at a given program state. These fields are marked as live and only these are considered in state matching.

Note that even though the second phase is executed many times (before each state matching), it is extremely fast and thus introducing only a minimal overhead.

Using this approach, we obtain more precise results compared to purely static DVA. In particular, the information from call-stack determines the methods that are called; contrary to purely static DVA, which has to consider all possibly called methods and reads in them. Also, the analysis does not have to construct the possibly huge cross-product of thread locations, as it is (typically) done in flow-sensitive analyses of multi-threaded programs.

In state matching, the information is used in a straight-forward way. If the analysis identifies a field as possibly live, the value of the field is included into the state vector, otherwise the value is ignored and not stored. Note that the results of the hybrid analysis intentionally do not distinguish particular instances more precisely (e.g., using allocation sites). While the analysis itself can use a better pointer analysis (e.g., to construct the call graph) to distinguish among instances of the same type, a slower and more complicated state matching algorithm, similar to the one used in dynamic analysis, would be needed to preserve safety in such a case.

4.3 Formalization

In this section, we first introduce parallel heap-manipulating programs as a parallel composition of guarded-action transition systems and their state space (using the typical small-step-big-step approach). Later, we formally describe our dynamic DVR and show its correctness. In contrast to static DVAs, which identify variables which are dead at a location, our dynamic analysis identifies the addresses on the heap which contain a dead value at a program state.

The memory (*Mem*) is modelled as a function taking an address and returning the value stored at this address. As usual, writing to the memory produces a new function Mem' being equivalent to the original one for all but the written address.

Syntax. A *program* $P = (T_1, \dots, T_n)$ is a tuple of concurrently executed threads $T_1, \dots, T_n$ operating over shared memory. For the sake of presentation simplicity, we omit dynamic thread creation in the formalization, since it can be added in a straight-forward way not affecting DVA.

Each *thread* can be seen as a labeled transition system $T_i = (L_i, Tr_i \subseteq L_i \times Guard \times Act \times L_i, l_i^{init} \in L_i)$, where L_i is a set of locations, l_i^{init} is the initial location, and Tr_i a transition relation (can be seen as CFG) with transitions labeled by *Guard*ed *Act*ions. We refer to edges in Tr_i as to *steps*; the steps are executed atomically. Guards and actions can be arbitrary, provided that they satisfy the following conditions (later in this section we show how to extract the information important to DVA): (α) The guards are assumed to be side-effect-free (i.e., they do not modify the memory), and (β) the actions and guards are deterministic (i.e., they act as functions; the same input values result in the same output). Data non-determinism (e.g., user input) is modelled by branching – one branch for each possible user input. The actions encode statements of the source code. They typically represent direct or indirect writes into memory. Without loss of generality, an action can include at most one write into the memory ending up the action; more complex constructs of programming languages are split into several actions. Guards are intended to express conditions and synchronization primitives from the source code. The memory is shared by all the threads; it holds all the thread-local as well as global (shared) data and states of locks (locked/unlocked).

We choose this representation, since we believe that it is best suited for multi-threaded programs. In contrast to *while* language [109] in MURPHI, our memory is shared among all threads, thus our approach permits complex interactions involving multiple threads over shared object instances which is common in multi-threaded code. Our formalization does not include primitives for message passing and queues as in the IF model checker [45] (unbounded FIFO queues) and PROMELA [61] (bounded blocking and lossy channels). The motivation for this decision was that queues are usually not an inherent part of programming languages. They are typically included in the language libraries with various semantics – in the form of priority queues, LIFOs, FIFOs, etc.

Semantics. A *program state* $S = (IP, Mem)$ is a pair consisting of function IP (instruction pointers), which for each thread i returns the current thread location $l_i \in L_i$, and function Mem representing the content of the memory. In the rest of the chapter, IP_S and Mem_S refer to the corresponding components of state S, respectively. We use $\mathcal{S}$ to denote the set of all possible program states.

For a function f, we write $f[d := val]$ to denote the function which is equivalent to original f for all inputs but d, for which it returns value val; we use this syntax to denote execution of a step in a particular thread. We write $Mem[A]$ to denote original memory Mem, whose content is modified by the write of action A.

Definition 1 (State space) *The state space (i.e., behaviour) of a concurrent program P is the transition system $A_P = (\mathcal{S}, \triangle \subseteq \mathcal{S} \times Guard \times Act \times \mathcal{S}, S^{init} \in \mathcal{S})$ over the program states $\mathcal{S}$, where the initial state S^{init} consists of the initial states of all the threads and the initial state of the memory: $S^{init} \equiv (IP(t) = l_t^{init}\ \forall t \in 1...n, Mem^{init})$.*

There is a transition $(S, G, A, S') \in \triangle$, *where $S \equiv (IP_S,\ Mem_S)$ and $S' \equiv (IP_{S'}, Mem_{S'})$ iff there exists a thread t and its step $(l_t, G, A, l'_t) \in L_t$ such that:*

- *$IP_S(t) = l_t$ and $IP_{S'} = IP_S[t := l'_t]$, and*
- *guard G holds in state S, and*

- *$Mem_{S'} = Mem_S[A]$ (i.e., action A transforms Mem_S into $Mem_{S'}$).*

According to the definition, the guards for all transitions in $\triangle$ hold, hence a transition cannot be blocked by its guard. We include guards into the transitions, because they read the memory, thus are important for DVA. The state space (i.e., $\triangle$) is non-deterministic; two different threads may, at a state, execute the same action guarded by the same guard.

Definition 2 (Transition equivalence) *Let $p \equiv S_1 \xrightarrow{G_1,A_1} \cdots$ and $p' \equiv S'_1 \xrightarrow{G'_1,A'_1} \cdots$ be (finite or infinite) traces. We say that p and p' are* transition-equivalent *if they consist of the same sequence of actions and guards: $\forall i : G_i = G'_i, A_i = A'_i$.*

Since actions are deterministic, the content of memory in all states along a trace is determined by the initial state of memory and by the actions on the trace. The sets P and P' of traces are equivalent if $\forall p \in P$ there exists a transition-equivalent trace $p' \in P'$ and vice versa. We use $\mathcal{P}$ to denote the set of all traces in the transition system A_P.

Let $read\_addr : \triangle \to 2^{dom(Mem)}$ be a function, which for a given transition $(S, G, A, S') \in \triangle$ returns the set of memory addresses that are read from the memory by either guard G or action A or both. Similarly, function $write\_addr : \triangle \to 2^{dom(Mem)}$ for a given transition returns the set of memory addresses to which action A of the transition writes. Recall that guards do not modify the memory thus are not considered by this function. Moreover, from the definition of the action, it follows that the set of written addresses is either empty or it contains a single address; without this restriction, an inter-transition data-flow analysis would be needed if multiple writes occur in a transition to obtain results with the same precision.

In our formalization of the program, we do not exactly specify the form of guards and actions. Instead, we use $read\_addr$ and $write\_addr$ to extract their behaviour relevant to DVA (i.e., the read and written addresses). Also note that these functions do not take the guard or action itself, but the whole transition; the purpose is to precisely model indirect memory accesses and arrays, where the address which is read or written depends on the content of the memory (in the initial state of the transition).

Let f be a function and $S \subseteq dom(f)$ be a set. We write $f\restriction_S$ to denote the equivalent partial function with domain restricted to S. We use partial functions to represent program states omitting dead values.

Live addresses. The *read/write_addr* functions can be easily generalized for traces. In a similar way, one can define function *tr_live_addr* which returns the set of addresses that are read before they are written on a given trace, i.e., the set of live variables for the given trace.

Definition 3 (Live trace addresses) *Let* $p \equiv S_1 \xrightarrow{G_1,A_1} \cdots$ *be a trace. Function* $tr\_live\_addr : \mathcal{P} \to 2^{dom(Mem)}$ *is defined as:*

$$\begin{aligned} tr\_live\_addr(p) = \{\ a \mid \exists i : a \in read\_addr((S_i, G_i, A_i, S_{i+1})) \wedge \\ \forall j < i : a \notin write\_addr((S_j, G_j, A_j, S_{j+1}))\} \end{aligned}$$

Note that in contrast to Static DVA, the analysis operates on real program states and state space instead of program locations in CFG. The values stored at live addresses at state S_1 (initial state of the trace) fully determine the behaviour of the trace, i.e., the outcome of the actions on the trace as well as fulfilment of the guards. The *tr_live_addr* function points to the memory addresses whose values are important to follow the trace, and fully determines the computations done (results of the actions) along the trace. If a state S_1' differs from S_1 in values at dead addresses only (i.e., S_1 and S_1' equals on $tr\_live\_addr(p)$), then there exists a trace p' from S_1' being transition-equivalent to p. Moreover, the corresponding states along these equivalent traces differ only in the memory content which is dead w.r.t. the suffices of these traces. Note that in general, two transition-equivalent traces can have different live addresses, however if the initial states of the traces differ at dead addresses only, then $tr\_live\_addr(p) = tr\_live\_addr(p')$.

In the verification, one is typically not interested in a single possible future behaviour starting at a state (i.e., a single trace), but rather in all possible future behaviours (i.e., all traces). To reflect this, we extend *tr_live_addr*. We say that an address is live at a state if there is a trace from the state at which the address is live:

Definition 4 *Let a be an address and $S \in \mathcal{S}$ be a program state. Function* $live\_addr : \mathcal{S} \to 2^{dom(Mem)}$ *is defined as follows:*

$$live\_addr(S) = \{\ a \mid \exists\ trace\ p = S \xrightarrow{*} \cdots\ such\ that\ a \in tr\_live\_addr(p)\}$$

The *live_addr* function refers to the memory addresses on whose values the future behaviour of the program depends (once program reaches the given state). Similarly, to the trace-based counterpart, if states S and S' equal w.r.t. $live\_addr(S)$ (and of course refer to the same program point IP), then for any trace p from S there is a transition-equivalent trace from S' (i.e., the future behaviours of S is a subset of the future behaviours of S').
This claim is formally expressed in Lemma 1 which is stated later in this section.

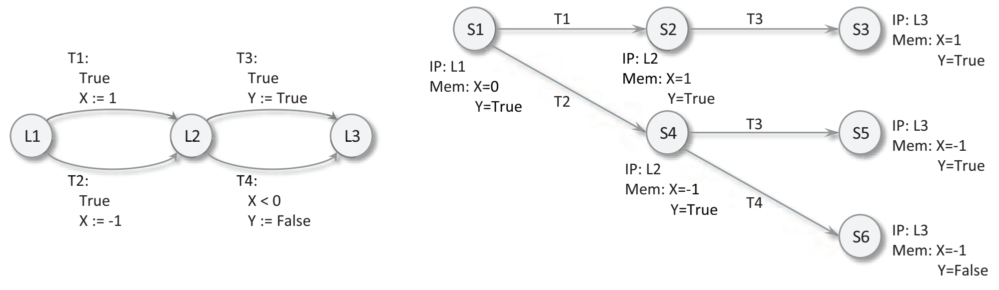


Figure 4.6: Thread specification (left) and its state space (right)

In the context of DVA, if two program states equal on their live addresses, their future behaviour is assumed to be the same; informally stated, the sets of states reachable from either one are the same w.r.t. their live addresses. This is generally not true for Guarded-Action LTS, and also for programs, this needs to be proved (e.g., thread synchronization is to be considered).

Consider a single-threaded program $P = (T)$, where thread T is shown in Figure 4.6; it consists of three locations $L1 - L3$ and four steps $T1 - T4$. All steps except for $T4$ contain guard $True$, step $T4$ contains guard $X < 0$. The actions are assignments to (global) variables X and Y.

The corresponding state space is depicted in Fig. 4.6 on the right-hand side; it consists of six states $S1 - S6$ and five transitions. Note that for the sake of readability, we used variable names instead of addresses in the example; while this is doable for global variables, it is not suitable for heap instances.

To illustrate that the claim above does not hold for LTSs in general, let us focus on states $S2$ and $S4$. State $S2$ has an empty set of live addresses, since only the assignment $Y := \mathit{True}$ of step $T3$ is executed before the final state is reached; thus only location IP is considered in state matching. State $S4$ has the same location as $S2$, and its set of live addresses equals to that of $S2$. However, the future behaviour of these states is different – from $S4$ there is an additional trace taking step $T4$ into $S6$.

First, note that problem is caused by unsatisfied guards; guard $X < 0$ does not hold in $S2$, thus there is no transition which could make the variable X live. Later in the subsection about *guard restrictions*, we focus on the problem in more detail.

Also note that the claim below the *live_addr* definition holds; from state $S2$ there is a trace $S2 \xrightarrow{T3} S3$. From state $S4$ there is a transition-equivalent trace $S4 \xrightarrow{T3} S5$.

Safety properties. Out of the transition-equivalent traces p and p' one may end up in an error state while the other in a safe state. This is because the values on which the decision whether the state is safe or not depends are ignored. For example, consider Figure 4.6; transition-equivalent traces $S2 \xrightarrow{T3} S3$ and $S4 \xrightarrow{T3} S5$ and the safety property $(IP = L3 \wedge X > 0)$. While the first trace leads to the safe state $S3$, the second one leads to the error state $S5$.

Let ϕ be (an externally specified) safety property for program P. We use $addr(\phi)$ to denote all the memory addresses which ϕ reads (i.e., the addresses of variables ϕ contains). To be able to distinguish between safe and error states, the addresses $addr(\phi)$ have to be a (live) part of each program state (even in cases the address is not read on any trace). Note that for many common types of program errors, such as assertion violation, the $addr(\phi) = \emptyset$, since the property is encoded in the program itself.

Live parts of state. The aforementioned definitions allow us to introduce a function which omits dead (i.e., irrelevant) parts of program states. For a full state T we define a function $reduce\_state_T$ which reduces full states w.r.t. the future behaviour of the state T.

Definition 5 (Reduced states) *Let $S \equiv (IP_S, Mem_S)$ be a full state. The $reduce\_state_T(S)$ function is defined as follows:*

$$reduce\_state_T(S) = (IP_S, Mem_S\restriction_{live\_addr(T) \cup addr(\phi)})$$

The $reduce\_state_*$ functions eliminates dead addresses from the domain of memory function Mem_S, while instruction pointers IP_S are not modified.

An optimal state matching algorithm matches (currently reached) state S with previously (fully) explored state S' if the future behaviours of S are a subset of behaviours from S'. In other words, the states should be matched if (i) they both contain the same value of the instruction pointer and (ii) equal on the data stored at the live addresses of S; formally if $reduce\_state_S(S') = reduce\_state_S(S)$.

This is however hard to implement efficiently, especially if just hashes of the visited states are stored. At the time when state S' was reached, it was not known which states visited later on would be considered in the state matching with S', thus the $reduce\_state_S$ was unknown at that time. And symmetrically at the time when S is reached, from the previously visited state S' only the hash value (of the full state) is known and there is no easy way to get the hash value considering only the live parts of S', i.e., to obtain $reduce\_state_S(S')$.

Alternatively, it is possible to omit the checking for subsets and match states with exactly the same future behaviour; formally expressed:

$$reduce\_state_S(S) = reduce\_state_{S'}(S')$$

This check is easier to implement, but the states whose future behaviour is a proper subset of the other are considered different due to different domains of their *Mem* functions. This matching approach should be comparable to the optimal one since for programs, a state does not exhibit a subset of the behaviours (i.e., traces) of another one; either the sets of traces are the same or completely different.

Note that even this alternative way of state matching still needs to precisely know the future behaviour of the currently reached state S. To detect whether state S has been already visited or not, it is not a good idea first to explore the successors of S and then to compute the live addresses from the observed behaviour; such an approach would eliminate one of the benefits of DVA – reduction of the state space. In our hybrid analysis, an over-approximation of the future behaviour origins from the static phase; so there is no need to explore the successors. Our dynamic analysis utilizes the observed behaviour to compute live addresses, however this is done only for states which are *known to be new* (i.e., visited for the first time). During state matching, the approach uses the live addresses of previously visited states which are tested against those of S; this is possible due to Theorem 2 below.

Reduced state space. As stated above, for state matching it is possible to consider states w.r.t. their own future behaviour. For a full state $S \equiv (IP_S, Mem_S) \in \mathcal{S}$, we define a *reduced state* R as $R \equiv reduce\_state_S(S)$. We use *reduce_state* without subscript to denote reduction of the state w.r.t. its own future behaviour. The symbol $\mathcal{R}$ is used to denote the set of all reduced states. Upon the reduced states, it is possible to build a state space in a similar way as for full states.

Definition 6 (Reduced state space) *Let P be a program with the state space $A_P = (\mathcal{S}, \triangle, S^{init})$. The* reduced state space R_P of P *is tuple $R_P = (\mathcal{R}, \triangle_R \subseteq \mathcal{R} \times Guard \times Act \times \mathcal{R}, R^{init})$, where $R^{init} = reduce\_state_{S^{init}}(S^{init})$.*

There is a transition $(R, G, A, R') \in \triangle_R$ iff there exists a full state transition $(S, G, A, S') \in \triangle$ such that:

$$reduce\_state_S(S) = R \text{ and}$$
$$reduce\_state_{S'}(S') = R'.$$

We now justify the correctness of this definition. In the reduced state space, it is easy to see that (1) there are no reads of the undefined values from the memory, and (2) there are no undetermined values in the memory. For a transition $t \equiv ((IP_R, Mem_R),\ G, A, (IP_{R'}, Mem_{R'})) \in \triangle_R$, (1) can be formally expressed as $read\_addr(t) \in dom(Mem_R)$, while (2) can be expressed as $dom(Mem'_R) \subseteq dom(Mem_R) \cup write\_addr(t)$.

As to (1), for any transition $(R, G, A, R') \in \triangle_R$ if some memory address is read by guard G and/or action A, then from the definition of the $read\_addr$ function (using the trace containing only this single transition) these addresses are preserved in the reduced state. So, both G and A read defined values. Note that the potential memory write of A can be performed to an irrelevant address. In such a case the write is ignored, and the state of the memory is unchanged. This happens if the given memory address is dead w.r.t. the property being verified and if the address is not read later in the program.

As to (2), for any transition $(R, G, A, R') \in \triangle_R$, all values (in the memory) of state R' have to be computed from the values in predecessor state R. For any address $a \in dom(Mem_{R'})$, there are three options: either (i) a is read by the property ϕ (i.e., $a \in addr(\phi)$) and thus the address is in the domain of Mem for all reduced states, in particular Mem_R, or (ii) a is written by action A, thus the value is determined in R', or (iii) a is not written by action A. From the definition of the reduced state space it follows that if there is a reduced transition $(R, G, A, R') \in \triangle_R$, then there is a full transition $(S, G, A, S') \in \triangle$ such that $reduce\_state(S) = R$ and $reduce\_state(S') = R'$. Then from the definition of $live\_addr(S')$, there must be a trace p' starting at S' which (reads a and thus) causes a to be live in R'. This trace p' can be prefixed by the considered transition; and $p = S \xrightarrow{G,A} p'$ can be created. Trace p shows that a is live in S and thus $a \in dom(Mem_R)$.

First, we will show a lemma which formally expresses the above claim transition-equivalent traces. Then, we will show one of the main results, the Theorem 1, i.e., bisimulation between reduced and original (full) state space.

Lemma 1 (Trace-equivalent paths) *Let $S_1 \equiv (IP_{S_1}, Mem_{S_1}) \in \mathcal{S}$ and $S'_1 \equiv (IP_{S'_1}, Mem_{S'_1}) \in \mathcal{S}$ be two states such that $reduce\_state_{S_1}(S_1) = reduce\_state_{S_1}(S'_1)$.*

Then for any trace $p = S_1 \xrightarrow{A_1,G_1} S_2 \cdots \cdots S_i \xrightarrow{A_i,G_i} S_{i+1} \cdots$ there exists a transition-equivalent trace $p' = S'_1 \xrightarrow{A_1,G_1} S'_2 \cdots$ such that $IP_{S_i} = IP_{S'_i}$ for all i (i.e., the corresponding states along equivalent traces differ only in the memory content).

Proof sketch. We show here the main idea of the proof. The formal proof is presented later in the appendix A.1.

The proof is constructive and inductive on the path length. The induction will iteratively append equivalent transitions to the end of path p' and preserve the following inductive invariant:

(1) $reduce\_state_{S_i}(S_i) = reduce\_state_{S_i}(S'_i)$ and
(2) $IP_{S_i} = IP_{S'_i}$ and
(3) the path p' is transition equivalent to the prefix of p of length i.

The appended transition is taken from the path p. It can be appended to p' since it ends-up in the location as the corresponding part of the path p. The first induction invariant holds because the guard and the action of the action reads the same data on the transition of path p and p' thus yielding the same outcome. The second induction invariant holds from the construction of p' the added transition is from the same for both state (i.e., same thread, guard, and action).

Such a path p' satisfied all requirements of the lemma. □

Below, we show that in our case there is a bisimulation between the full and reduced state spaces (as it is typically done for static DVA). To our best knowledge this is the first proof for Dynamic DVA.

Theorem 1 (Bisimulation) *For a program P and a property ϕ, transition system $A_P = (\mathcal{S}, \triangle, S^{init})$ and corresponding reduced transition system $R_P = (\mathcal{R}, \triangle_R, R^{init})$ are bisimilar.*

Since reduced states and corresponding full states are equivalent w.r.t. ϕ, it directly follows that the state space of the program is safe w.r.t. ϕ iff the reduced state space is safe (and similarly for error traces). So, it is enough to check the safety on the reduced state space. Bisimulation also preserves liveness properties, so these can be safely checked just on reduced state space as well [7].

Proof sketch. Here we sketch the idea of the proof, while a formal proof is presented later in the appendix A.1. We will show that $reduce\_state_*$ functions define the relation among the states which is a bisimulation. We assume states $S \in \mathcal{S}$ and $R \in \mathcal{R}$ to be in the relation iff $reduce\_state_S$ $(S) = R$.

The direction from full states to reduced states follows directly from the definition. The only non-trivial step is the other direction. For a reduced transition $(R, G, A, R') \in \triangle_R$ executed by thread i and related full state S we will show that there is full transition $(S, G, A, S') \in \triangle$ such that full state S' reduces to R'.

From Def. 6 we obtain a *b*ase full transition $(S_b, G, A, S'_b) \in \triangle$. Using this transition we show that there is also a transition $(S, G, A, S') \in \triangle$ since S and S_b equal on all values that G and A use.

We will show that S' and S'_b have the same live addresses: Since S and S_b reduce to the same state R, they have (pair-wise) transition-equivalent sets of outgoing traces. Hence, the sets of traces via S' or S'_b are transition-equivalent as well. It directly follows that S' and S'_b have the same live addresses and, in turn, S'_b and S' reduce to the same reduced state (R'). $\square$

Guard restriction. While the bisimulation theorem guarantees soundness of the approach, the definition of the *reduce_state* function is not suitable for efficient state matching. On-the-fly explicit state model checkers can get a precise set of live addresses for a state, however they can be computed only *after* the state is fully explored, which is too late to directly reduce the state space. Below, we articulate Lemma 2, which permits us to use *live_addr* (i.e., the $reduce\_state_*$ function) of one state to safely reduce another (still unexplored) one. The theorem, however, cannot be applied on general guarded-action LTS. Hence, we first introduce a syntax restriction for guards.

Definition 7 (Guard-restriction) *The program $P = (T_1, ...T_n)$ satisfies* guard-restriction *if for each thread $T_i = (L_i, Tr_i, l^{init})$ and for each step $(l_i, G, A, l_j) \in Tr_i$ such that guard G is non-trivial (i.e., different from $True$) there exists also a step $(l_i, \neg G, A', l_k) \in Tr_i$, (in the same thread t from the same location l_i) with the complementary (negated) guard $\neg G$.*

Both guards read the same data and in any case exactly one of the guards is satisfied. The guard-restriction is inspired by the way source code is converted into transition systems. The guards are used to encode conditions (`if-then-else`) and synchronization primitives. The conditions are encoded this way naturally. Synchronization primitives (such as synchronized blocks, thread joins, and wait-notify) can also be easily modelled this way by adding an (active-waiting) self-loop for the blocked state. This means that source code (e.g., Java) yields guard-restricted transition systems.

Let us now focus on the purpose of the guard restriction. Without the guard restriction, if two states C and V equal w.r.t. relevant addresses of V, we know that the future behaviours of C includes all the future behaviours of V. They are in the subset relation, because there can exist a trace from C which cannot be followed from V; it means if the trace is followed from V, the corresponding transition to follow the trace can be missing due to an unsatisfied guard. Note that unsatisfied guards do not contribute to relevant addresses. It means that C and V do not behave equally w.r.t. (unsatisfied) guards. Exactly this case is illustrated in examples from Figure 4.6, which do not satisfy the guard restrictions. Guard restrictions, on the other hand, eliminate this issue. Consequently, if two states C and V equal w.r.t. relevant addresses of V, they have (pair-wise) transition-equivalent sets of traces and also the same set of relevant addresses. This is formally expressed in the following lemma:

Lemma 2 *Let P be a program that satisfies guard restriction, and $A_P = (\mathcal{S}, \triangle, S^{init})$ be its state space. Let $C \in \mathcal{S}$ and $V \in \mathcal{S}$ be two states such that $reduce\_state_V(C) = reduce\_state_V(V)$. Then it holds that $reduce\_state_C = reduce\_state_V$.*

Proof sketch. We show that for each trace from C a transition-equivalent trace with the same set of *tr_live_addr* from V exists and vice-versa; thus, the states have the same set of live addresses and hence the *reduce_state* functions. The direction from V to transition-equivalent trace from C is informally stated just below the definition of (trace) relevant addresses, being exactly the claim of the Lemma 1. As to this part, the guard restriction is not applied here.

The other direction, i.e., for any trace from C, a transition-equivalent trace from V exists, is shown by contradiction. Assume that p is the shortest trace from C, such that there is no transition-equivalent trace from state V and that states C and V satisfy the requirements of the lemma. Since $p \equiv C \xrightarrow{G,A} C'$ is the shortest such a trace, it follows that the first transition cannot be followed from V. In the opposite case, we could have moved along the common prefix of traces p and p' (thus finding shorter traces from states C_i and V_i).

Now we exploit the guard restriction to show that trace p cannot exist. Guard G cannot be trivial (in such a case transition $\xrightarrow{G,A}$ would exist from V). Moreover, due to the guard restriction, there is a transition $V \xrightarrow{\neg G,A'} V'$ thus all the addresses that $\neg G$ (as well as G) reads are live in V. Because C and V equal w.r.t. the relevant addresses of V (formally expressed as $reduce\_state_V(C) = reduce\_state_V(V)$), and all the addresses that guard G reads are relevant, it follows that in C guard G is not satisfied (in fact its negation holds). Thus there is no transition $C \xrightarrow{G,A} C'$ and trace p does not exist. In turn, in the whole state space, there is no contradicting p (i.e., a trace without a transition-equivalent counterpart). □

4.4 Implementation

We have developed two independent DVAs to identify live fields of the heap instances and implemented them in Java PathFinder (Jpf). The analyses differ in precision, computation complexity as well as in the way live addresses are obtained. To make our approach clear, we present the algorithms in pseudo-code below. To emphasize our extensions to classical model checking, we present the standard algorithm we build upon in Alg. 1.

Algorithm 1 Common DFS model checking algorithm

```
1: procedure Main
2:     ModelCheck(P, $s\_init$, [$s\_init$])

3: procedure ModelCheck(program $P$, state $s$, trace $t$)
4:     if IsErrorState($s$) then throw Unsafe($s, t$)
5:     if Visited($s$) then return VISITED
6:     SetVisited($s$)
7:     for transition $alfa \in$ EnabledTransitionsIn($s$) do
8:         $s\_succ \leftarrow alfa(s)$
9:         ModelCheck($P$, $s\_succ$, $t$ + ($alfa$, $s\_succ$))
10:    return SAFE
```

Our Dynamic DVA tracks field reads and writes executed by the program. The live addresses for a given program state are computed once all the future behaviours of the program state are fully explored. Thus, our DDVA can be used only with the default Depth-First Search (DFS) with no test-like heuristics (those traversing only a part of the state space). The analysis is exact for loop-less state space, it marks particular field as live if and only if there exists a real trace from that state on which the value of the field is read (before it is overwritten). On the other hand, the analysis requires additional memory and has a bigger computational cost compared to our Hybrid DVA. DDVA has to store the observed live addresses (i.e., *reduce_state* function) together with the program state (i.e., its state vector or its hash).

Our Hybrid DVA combines static analysis and dynamic information from program states; hence its name. Static analysis provides an over-approximation of future program behaviour (in terms of field reads), thus the information about live addresses is computed once the state is reached (before its successors will be explored). So, the analysis can be safely used together with various test-like heuristics and state space exploration strategies. It is less precise than our DDVA, however, it offers very low computational and memory overhead; only small amount of memory is required to store the results of static analysis.

In our work, we aim at identifying irrelevant content stored in the heap. Note that we disregard irrelevant data stored in the local variables, which is a subject of other analyses that can be combined with ours.

Heap of Java programs. Heap is typically formally modelled as an array of values where references (pointers) are indexes into the array. While this straightforward approach is generic and can express any usage of the heap, it is too low-level for object-oriented languages. JPF (as well as other explicit-state model checkers) typically represent heap in a more complex way. This helps to better express the semantics, and it makes various optimizations (e.g., heap canonicalization) easier. It leads us not to represent the DVA related information as addresses, but to follow a higher-level abstraction used in model checkers. It also provides us with an easier integration into state matching. Below, we describe how the elements of heap are represented; such a representation is directly used in our DDVA. Our Hybrid DVA uses a simpler representation, which is described later.

In object-oriented languages, the objects are allocated on the heap. In Java, the objects are of two types – arrays and instances of classes. Instead of addresses for referring to objects, we use pairs consisting of a unique object ID (i.e., the representation of an address) and a field name (its index since the name may not be unique) for class instances.

The class instances do not contain only fields, but also a type. The type of each instance is stored in its headers. The type can be accessed via introspection (reflection), by the `instanceof` Java operator, and indirectly via virtual method calls. The instance cannot be marked as dead (i.e., completely ignored by state matching), even if no field is read from the instance, if a virtual method is called on the instance; its type is live, since if another instance with a different type had been used instead of the original one, the program would behave differently; it will execute a different (virtual) method. Thus, for each instance (i.e., unique instance ID), the analysis stores whether its type is live or not.

In Java, monitors are other properties of instances; each instance has associated a monitor (i.e., a lock), upon which the threads can synchronize. The state of monitors (locked, unlocked, notified, etc.) is of course an important part of the program state and thus has to be considered by state matching. The monitors realize the `synchronized` blocks and methods; each monitor needs to be accessed at the beginning and at the end of the corresponding synchronized section, and so each monitor is live. That is why our analysis assumes all monitors to be live.

Arrays are other objects stored on the heap. They are handled in a similar way as class instances (since, in Java, arrays are instances as well, but with no fields and methods defined by user), however, instead of field names, we store which particular indices in the array are live – we store pairs consisting of a unique object ID and an index. The arrays are more difficult to handle than normal instances, since they can vary in length. Instances of a single class have a fixed set of fields and thus a fixed size. For arrays, we also store whether their length has been accessed, in other words whether the length attribute is live or dead. In rare cases, when an array is not accessed at all, but still the instance is important (e.g., it is compared to another reference), we even omit the size of the array in state matching. Of course, if any index of the array is live, then the array length is live as well; if the length of corresponding arrays differs, an attempt to access an index may result in the `IndexOutOfBoundsException` only in one state with the shorter array. In Java, the arrays are in fact objects, which however cannot have any user defined methods and fields. It means that we also have to store the type of each array as for a normal class instance.

Our analysis also handles static fields. In JPF, static data are stored in a (special) heap, where each class has its own instance. Because of that, we handle the static fields in the same way as object instances; for each such a field, we store a pair consisting of a unique class identifier and a field name for identification. Note that a particular class can be loaded into JVM several times (each time with a different defining classloader) and each such loading results in a different and incompatible type, featuring its own static fields (sharing the names). Even though these classes have the same name, we support this in our approach, since JPF assigns them different identifiers.

Analysis of JVM bytecode. Above, we focused on specific properties of the heap. We need to track not only addresses, but also, e.g., types and array lengths.

In the following paragraphs, we move from LTS to JVM bytecode. Our analyses, as described, operate over LTS, however, the program is not executed as guarded-action LTS. Java programs are represented by JVM bytecode, which is directly executed by the JPF model checker and observed for purposes of our analyses. The mapping from LTS to bytecode is quite straightforward; however, guards need special attention. Moreover, DDVA requires a guard-restricted transition system; so below (1) we focus on guards and show that JVM bytecode can express only guard-restricted transition systems. Later, we show (2) how to obtain *read_addr* and *write_addr* resp. their counterparts for programs with heap.

In LTS, evaluation of guards influences which transitions the program may take; if LTS encodes a source code, then guards are used to express conditions and synchronization primitives. In JVM, there are no guards. Instead, bytecode (assemblers) have *compare instructions* to evaluate conditions, and *conditional jump* instructions, to take a particular branch (i.e., transition). In particular, JVM bytecode contains the compare `[D/F]CMP[G/F]` instructions, and conditional jumps specialized for a specific condition (e.g., the `IF_ICMPLE` instruction which compares two top integer values on the stack and jumps if the one-but-top value is less than or equal to the top one).

Let us focus on the guard restriction. The restriction has been introduced to ensure that the reduced states behave in the same way w.r.t. both satisfied and unsatisfied guards; in other words, the restriction makes the reads of unsatisfied guards visible. Informally, in order to enable JVM to decide whether a given transition can be taken, it first has to evaluate the condition (i.e., the guard); thus, the reads of (satisfied as well as unsatisfied) guards are visible. Therefore, for JVM bytecode, we can assume that it satisfies the guard restriction.

Let us now focus on the conditions and synchronization primitives in more detail. In case of conditions (e.g., `if-then-else` and loops), first, the condition (i.e., the guard) is evaluated (so its reads are visible) and based on the result, either the `then` or the `else` branch (i.e., the transition) is taken. In case of synchronization primitives, in particular the `MONITORENTER` and `MONITOREXIT` bytecode instructions, which correspond to the beginning and end of `synchronized` sections, respectively, we always consider the corresponding monitor to be live, as argued above. To focus more on the bytecode level, independently of the fact whether the `MONITORENTER` would block the thread or not, the execution of the instruction is started and thus the read of the monitor can be observed in either case – (blocking the thread or acquiring of the monitor) as needed for the guard restriction.

Below, we focus on extraction of relevant information for JVM bytecode. First, we identify the bytecode instructions that need to be considered by the analysis, later we focus on more specific JVM properties.

In Tab. 4.2, there is a list of bytecode instructions which manipulate references and accesses the heap; for each instruction, we mark whether it depends on or modifies values in the interesting parts of the heap mentioned above. Not all instructions which manipulates with references are interesting for DVA; e.g., `aconst_null`, `if[not]null`, `areturn`, `aload` do not directly access the heap, so they can be safely ignored by DVA.

Field accessing (`[PUT/GET][field/static]`) instructions do not make

bytecode instruction	Object instances		Arrays			Static
	Fields	Type	Index	Length	Type	Fields
`getfield`	✓					
`putfield`	✓					
`getstatic`						✓
`putstatic`						✓
`checkcast`		✓			✓	
`instanceof`		✓			✓	
`[a/b/c/d/f/i/l/s]aload`			✓	✓		
`[a/b/c/d/f/i/l/s]astore`			✓	✓		
`arraylength`				✓		
`invokeinterface`		✓				
`invokevirtual`		✓				
`athrow`		✓				
`monitor[enter/exit]`						

Table 4.2: Heap manipulating JVM bytecode instructions

the type of the instance live. The particular field that is read or written is determined during compilation; there is no dynamic-dispatch as in case of virtual methods. Field accesses are thus independent of the actual runtime type of the corresponding instance. Similarly, the `*aload` resp. `*astore` instructions read resp. write data from/to arrays; the proper instruction is determined at compile-time using static types; its behaviour is independent of the actual runtime type (in case of `aaload`), thus these instructions do not mark type information as live.

As to the invoke instructions, it is obvious that their behaviour does not depend on the values of the fields. However, as mentioned above, since the runtime type of the instance on which the method is invoked influences which particular method is executed, the dynamic dispatch is applied (`invokeinterface`, `invokevirtual`). Note that `invokestatic` and `invokespecial` do not involve dynamic dispatch, the called method is determined statically during compilation, thus runtime type information is not used by them.

The `athrow` instruction throws an exception; its behaviour is independent of the type of the exception, however, the exception handling, i.e., the decision which exception handler matches the exception depends on the type of the exception instance.

The `invokedynamic` instruction is the most complex; its purpose is to support dynamic languages using JVM. Our implementation of the DVA analyses do not consider this instruction, because the particular JPF we used does not support this instruction.

The behaviour of the locking instructions (as well as calls to synchronized methods) has been described above. JPF stores a state of all locked monitors into the state vector; our DVR's do not modify the way monitors are serialized. Moreover, in order to lock/unlock the monitor, it needs to be accessed. This guarantees that the monitor instance is live and thus not omitted from program state.

The DVA does not have to consider the JVM instructions which create new objects (i.e., `new` and `ldc`). The created objects do not exist in predecessor states; so obviously these objects cannot be live in them.

Not only bytecode can manipulate with the heap content; native functions (in JPF realized by modelled functions) can access it as well. The most obvious example is the introspection, which allows JPF to read or modify fields and call methods. Another example is the output to the console (i.e., `System.out`); the method `print(String)` is modelled in JPF and its native body reads the whole content of the provided string.

Furthermore, in addition to the types of non-determinism present during normal Java programs execution, JPF features methods for efficient modelling of the user or environment behaviour. These methods include, e.g., `Verify.getBoolean()`, `Verify.getInt()`. JPF process these methods by creating a temporal variable storing the selected value in them and creating a choice for each possible one (i.e., branches in the state space). Even when using these methods, the program satisfies the guard-restriction property.

4.4.1 Dynamic DVA

Our DDVA monitors interesting instructions described above and based on their appearances, it computes live addresses for each state. The algorithm of DDVA is listed in Alg. 2.

Algorithm 2 Dynamic DVR

```
11: procedure Main
12:     ModelCheck($P$, $s\_init$, [$s\_init$])

13: procedure ModelCheck(program $P$, state $s$, trace $t$)
14:     if is_error_state($s$) then throw Unsafe($s$, $t$)
15:     ($visited$, $live\_addrs$) ← IsVisited($s$)
16:     if $visited$ then return (VISITED, $live\_addrs$)
17:     if $s \in trace$ then return (LOOP, {$all\_addrs$})
18:     $live\_addr\_s$ ← {}
19:     for transition $alfa \in$ EnabledTransitionsIn($s$) do
20:         ($s\_succ$, $heap\_accesses$) ← $alfa(s)$
21:         (_, $live\_addrs\_s\_succ$) ← ModelCheck($P$, $s\_succ$, $t$ + ($alfa$, $s\_succ$))
22:         $live\_addrs\_s\_alfa$ ← update_live_addrs ($live\_addrs\_s\_succ$, $heap\_accesses$)
23:         $live\_addrs\_s$ ← $live\_addrs\_s \cup live\_addrs\_s\_alfa$
24:     SetVisited(ReduceState(($s$, $live\_addrs\_s$), $live\_addr$))
25:     $map\_stacks2live\_addrs$[$s.threads.callStacks$] += $live\_addrs\_s$
26:     return (SAFE, $live\_addrs\_s$)

27: procedure IsVisited(state $s$)
28:     for $live\_addrs\_v \in map\_stacks2live\_addrs$[$s.threads.callStacks$] do
29:         $r\_s\_candidate$ ← ReduceState($s$, $live\_addrs\_v$)
30:         if Visited($r\_s\_candidate$, $live\_addrs\_v$) then return (TRUE, $live\_addrs\_v$)
31:     return (FALSE, {})
```

Live addresses. While JPF executes the program forwards, for each transition (i.e., each forward step) (line 19–23), the analysis records all relevant instructions (line 20–21); to be more specific, the addresses (a part of the heap) to mark resp. unmark as live. JPF stops advancing forward once it reaches either the end of the program or a visited state (line 16). In the former case, there are no live addresses, so no address is marked as live (this corresponds to the empty set in our algorithm) (line 18). In the latter case, the live addresses stored for the matched (previously visited) state are used (line 15). When JPF backtracks a transition (i.e., goes backwards along a program trace), the analysis computes a union of live addresses for the target state of the transition and the reads and writes recorded for the transition itself (line 23); this way, live addresses for the source of the transition are computed. To obtain live addresses for state s, the live addresses from the beginning of all the outgoing transitions from s are merged; an address is live at state s if and only if it is live at the beginning of (at least one) transition starting in s.

Due to the DFS exploration strategy, all outgoing transitions are explored before the state is backtracked from. Thus, the live addresses for state s (i.e., the $reduce\_state_s$ function) as well as its reduced state r_s can be computed and stored for the state matching purposes after JPF backtracks over s. The only exception are cycles in the state space (not just in CFG). If a cycle is detected after reaching (an already visited) state s, the live addresses for it (being in the stack of unprocessed states) are not known. In order to preserve correctness, we have to mark all addresses in s as live (line 17). Note that it does not necessarily mean that the program has to loop forever; after a few iteration of the cycle, a non-deterministic choice can be taken, which will exit the loop. Alternatively, there is a more elaborated approach, which does not lose precision. It is possible to postpone the computation of live addresses for the cycle states. First, all the traces leading out of the cycle (remaining non-deterministic choices in the states forming the cycle) have to be explored, and later, after all of them are explored and their live addresses are known, the sets of live addresses for the cycle states can be computed – their live addresses are propagated along the states forming the cycle to simulate any number of loop iterations. This way, the sets of live addresses can be obtained even for state spaces with loops in a precise way. Nonetheless, since the practical outcome of this approach is not clearly visible, we postpone the experiments with it as future work.

State matching. After JPF reaches a state (after a forward step), state matching is initiated to decide whether the state has already been visited or it is a new one (line 27–31). Java programs satisfy the guard-restriction property, so Lemma 2 can be applied here; the state matching process attempts to find (line 28) a state V and a corresponding function $reduce\_state_V$ (i.e., $live\_addrs\_v$) satisfying the requirements of the lemma (line 30).

For the purpose of state matching, a helper map (*map_stacks2live_addrs*) is maintained (line 28). For each call-stack, the map holds live addresses (in other words *reduce_state* functions) used to reduce the states with the particular call-stack. This map is used as an optimization reducing the number of live addresses to be examined.

State matching proceeds as follows: First, the set of possible live addresses (i.e., *reduce_state* functions) is obtained using the call-stack of the current state s (line 28). These live addresses are examined one-by-one whether their live values are the same as in s (line 29–30). In other words, a candidate reduced state is created using the current state s and its live addresses are examined (formally $\mathit{reduce\_state\_v}(s)$ is computed). Then, using the standard state matching function, the algorithm determines if a previously visited state v exists such that $\mathit{reduce\_state\_v}(s) = \mathit{reduce\_state\_v}(v)$ (line 30). If so, due to Lemma 2, a state equivalent to s has been already visited (line 30); if not, s is a new state (line 31).

4.4.2 Hybrid DVA

The other analysis of ours – Hybrid DVA – focuses on scalability instead of precision. It is designed to be fast in presence of threads and to allow for efficient state matching. The algorithm of our hybrid analysis is listed in Alg. 3.

For each program state, it identifies the fields which can be read (on any object of the given type) before the program terminates by any trace from that state. In its nature, the analysis is similar to the analyses proposed in [86], but we adapted it for the purpose of state matching. It consists of two phases: (1) static data-flow analysis, whose results are combined with (2) the (verification time) information from the currently reached state. The first phase is done once before a JPF run, while the second one is executed on demand during state-space exploration just before state matching.

Static phase. Backward flow-sensitive context-insensitive data-flow analysis over full inter-procedural control flow graph (ICFG) is used to obtain information about future behaviour of the program (line 33). For a given location (i.e., bytecode instruction) l, the analysis computes an over-approximation of the set of all fields which may be read before *returning from the method* containing l (including potential reads from the nested method calls and spawned threads). The data flow facts are pairs `ClassName.FieldName` which unambiguously identify all program fields. In contrast to [86], the facts do not include allocation sites of objects; even though potentially less precise, this

Algorithm 3 Hybrid DVR

```
32: procedure MAIN
33:     $partial\_liveness \leftarrow$ HdvrComputePartialLiveFields($P$)
34:     ModelCheck($P$, $s\_init$, [$s\_init$])

35: procedure MODELCHECK(program $P$, state $s$, trace $t$)
36:     if IsErrorState($s$) then throw Unsafe($s$, $t$)
37:     $live\_fields \leftarrow$ HdvrComputeLiveFields($s.callstacks$)
38:     $reduced\_s \leftarrow$ ReduceState($s$, $live\_fields$)
39:     if IsVisited($reduced\_s$) then return $VISITED$
40:     SetVisited($reduced\_s$)
41:     for transition $alfa \in$ EnabledTransitionsIn($s$) do
42:         $s\_succ \leftarrow alfa(s)$
43:         ModelCheck($P$, $s\_succ$, $t$ + ($alfa$, $s\_succ$))
44:     return SAFE

45: procedure HDVRCOMPUTELIVEFIELDS(state $s$)
46:     $live\_fields \leftarrow \{\}$
47:     for Thread $t$ : $s.threads$ do
48:         for StackFrame $f$ : $t.callstack$ do
49:             $live\_fields \leftarrow live\_fields \cup partial\_liveness[f.instruction\_pointer]$
50:     return $live\_fields$
```

allows for fast state matching, used together with heap canonicalization.

The transfer functions are defined in Fig. 4.7. The result of the static phase is the least fixpoint over the equations determined by these functions. If static analysis encounters the field-read instruction, it adds the field into the resulting set (see the second rule). Because the analysis summarizes all instances into a single data-flow fact, it does not treat field writes in any special way (the last rule applies).

The field reads are not propagated via exit-return edges (see the third rule). This blocks propagation along infeasible paths in ICFG, in particu-

Instruction	Transfer function
(branching point)	$\text{after}[\ell] = \bigcup_{\ell' \in \text{succ}(\ell)} \text{before}[\ell']$
ℓ: v = o.f	$\text{before}[\ell] = \text{after}[\ell] \cup \{\text{ClassName(o).f}\}$
ℓ: return	$\text{before}[\ell] = \emptyset$
ℓ: call M	$\text{before}[\ell] = \text{before}[\text{M.entry}] \cup \text{after}[\ell]$
ℓ: other instr.	$\text{before}[\ell] = \text{after}[\ell]$

Figure 4.7: Transfer functions for the static phase of Hybrid DVA.

lar those where a related call-entry edge starts and exit-return edge leads to a different method. For a program state, the complete picture about future reads is computed at the dynamic phase. This technique gets precision of a context-sensitive analysis at the computational cost of a fast context-insensitive analysis.

Dynamic phase. In the dynamic phase, the results from the static phase are utilized to get complete information about future field reads, first for each thread, then for a program state. The call-stack (of each thread) is processed in the top-down manner (line 48). The static analysis result for the current instruction on the top stack frame contains the possible field reads, before the current method is exited. Going down the call-stack, for each stack frame, the analysis joins (i.e., adds) the field reads computed in the static phase for the instruction just after the call instruction (current instruction pointer in the stack frame) (line 49). This way, future behaviour being considered is extended to the end of the given stack frame. Once the entire call-stack is processed, the result contains all the fields the thread can read before it terminates. At the end, future reads of all threads are joined together to obtain the future field reads for the program state (line 50).

State matching. Hybrid DVA provides us with an over-approximation of live addresses. The dynamic phase depends only on information from the call-stacks, thus for the same call-stacks the analysis yields the same live addresses (irrespective of the content of the heap). So, there is no need for complex state matching as in the case of aforementioned Dynamic DVR; put simply, a reduced state (in a representation suitable for state matching) is created, which contains only the live fields of object instances. For the state matching purposes, only the reduce state is used. In contract to Dynamic DVR, the live fields (i.e., a Hybrid-DVR equivalent of live addresses) does not need to be stored in addition to reduced states; each reduced state already contains all the information required to reconstruct the live fields.

We found that an efficient implementation is crucial in order to speed-up the verification. We heavily employ bit-vectors and block-operations in our hybrid DVR implementation.

4.5 Related work

In this section, we compare the presented DVRs to related approaches both in general and showing the differences on our running example from Sect. 4.2.

Dynamic DVA. Let us continue with the example after exploring the $S3$ state. After state $S3$ is fully processed, the model checker will backtrack the transition $S2 \rightarrow S3$. The dynamic analysis uses the records stored for this transition and it adds `TreeNode@5.value` among live variables; because this field is live in state $S3$ (i.e., in the target state of the transition) adding it has no effect and at the beginning of transition $S2 \rightarrow S3$, the live variables are the same as in $S3$, i.e., `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value`. Because this is not the only outgoing transition from $S2$, the model checker has to explore also the transitions where thread $T2$ is scheduled before thread $T1$ – transition $S2 \rightarrow S5$. Using the same approach as before it discovers that only `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value` are live at the beginning of transition $S2 \rightarrow S5$. To compute live variables for $S2$, our analysis merges the live variables at the beginning of all outgoing transitions; the live variables again include the `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value`. Since the model checker has already explored all outgoing transitions from $S2$, it backtracks transition $S2 \rightarrow S1$ to state $S1$ and our analysis computes that at the beginning of the transition, the live variables are again `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value`.

In [77], Lewis et al. introduce DVA, which uses dynamic information obtained at runtime to improve the precision of static analysis. Their analysis supports heap and interrupts; however, it lacks support for multi-threaded programs. Once a state is reached, Lewis's DVA spawns an additional forward simulation to get a depth-limited knowledge about future behaviour of the state. The results from the simulation are used in subsequent static analysis. In particular, the information of the simulation is used for elimination of the edges in CFG that cannot be taken. Our approach differs from that of Lewis in three aspects: our DDVA (i) has full knowledge about future behaviour, (ii) computes this information cheaply during state space exploration, and (iii) is thus more precise. Consider Lewis's DVA with a bound of two transitions. In this case, the simulation also finds the trace $S1 \rightarrow S2 \rightarrow S5$ where both threads stop at the read of `TreeNode@5.value`. Hence, using this bound, the analysis is not able to block the path in CFG that accesses the `TreeNode.right` fields. In turn, Lewis's analysis will imprecisely mark `TreeNode.right` fields in $S1$ as live, in contrast to our dynamic analysis which correctly identifies these fields as dead and ignores their values.

The DDVA introduced by Self and Mercer [99] is similar to our approach. Live addresses are computed from observed reads and writes in the same way. The approaches differ in the way non-determinism is handled; our DDVA explores all possible future executions from a state (i.e., all outgoing traces) and then merges their live variables, while DDVA of [99] operates only on a single trace. It lacks knowledge about reads and writes on other traces, hence, at non-deterministic (branching) states, it marks all the variables as live. In our running example, both analyses get the same result for $S3$, however in $S2$, where a non-deterministic choice is present, DDVA of [99] marks all fields as live in contrast to our analysis. The dynamic DVA in [99] seems to be more reasonable for sequential programs, where it provides (some) dead variables even for states which are not fully processed. However, our approach is better suited for multi-threaded programs, where transitions are quite short (often only a few instructions), and where DDVA of [99] becomes considerably less precise.

In the case of programs without non-determinism both methods equal and yield the *DVA maximal reduction* [99], while in the other case, our approach provides more precise results (as shown above). Moreover, for our Dynamic DVR, we also proved bisimulation between the original transition system and the transition system over reduced states (without dead variables). However, in [99], their definition permits additional transitions in DVA abstract state space which do not have pre-image transitions in the full state space. From this point of view, it is obvious that the original state space and the corresponding reduced one cannot be bisimilar.

Hybrid DVA. Similar DVA focusing just on local variables is used in, e.g., the SPIN model checker [61], MURPHI [82, 109], and BANDERA [31]. To obtain complete results for global variables and static fields, the effects of other threads need to be considered; in [13], a *control-graph* – cross-product of all locations of all threads is created and static DVA is done over this structure; for programs, this can be a bottleneck. Each thread (its ICFG) is composed of a large number of locations, thus their cross-product can be potentially of the same size as the state space, e.g., if values of variables are determined by the location as in the classical Dining Philosophers problem. Moreover, the number of parallel threads can be unbounded, and in such cases the approach of [13] cannot be used at all.

To our best knowledge, the analysis presented in [13] is the only (Static) DVA, which supports concurrency and global variables. However, it does not support heap (instances), since it was designed for verification of specification languages. In contrast to our Hybrid DVA, which uses cheap ICFG analysis, the analysis of [13] creates so called *control graph* – a cross-product of all possible locations in all threads, whose size can be comparable to the one of the corresponding (full) state space. The live addresses are computed by static analysis over the control graph, i.e., as the least fixpoint using reads and

Benchmark	**LOC**	**Threads**
AlarmClock	250	4
CLIF-BladeInsertAdapter	2780	3
Cache4J	600	3
CoCoME	3500	4
Deos	2160	3
Elevator	400	3
FTDemo	1300	3
LinkedList	291	3
Producer-Consumer	180	3
RepWorkers	630	3
Simple JBB	2300	3

Table 4.3: Sizes and numbers of threads for each benchmark.

writes of the actions. This implies that their analysis is expensive for multi-threaded programs, which have thousands of locations (i.e., instructions) per thread.

4.6 Evaluation

Implementation of both DVRs together with an experimental setup, and related data are accessible at `http://d3s.mff.cuni.cz/software/jpf-psm/`.

4.6.1 Benchmarks

We evaluated our DVRs on 11 benchmarks – CoCoME [21], FTDemo [1], CLIF [39], the Cache4j and the Elevator (both from PJBench suite [89]), Simple JBB, and a set of small benchmarks taken from the CTC repository [36] (AlarmClock, LinkedList, Deos, Producer-Consumer, Replicated-Workers). The information on LOC and number of threads created during the programs' runs are listed in Tab. 4.3. All the benchmarks are multi-threaded, use heap, are error-free, and their state space (not the CFG) is acyclic, thus the reported data are related to their complete state spaces. The reason for not including the benchmarks with cyclic state spaces is that our current implementation does not support them. However, there is no principle obstacle – we just have not implemented the support for them so far. The benchmarks were taken from [87]; we have chosen them for several reasons. We already knew they worked with Jpf, they generate a relatively small yet non-trivial state space, and since we worked with them before, we could compare the results with other techniques.

As to the default Jpf configuration, we disabled *heap canonicalization*, because it is not compatible with our analyses. The reason is that in our implementation, we rely on one-to-one mapping of unique ids to objects, which does not exist if heap canonicalization is used. Theoretically, it is possible to overcome this restriction of our analyses, but it would imply an additional overhead for maintaining a one-to-one mapping by ourselves. On the other hand, the following default optimizations were left turned on: *final fields filtering*, *live threads*, and *dynamic lock analysis*.

Cache4j is a simple framework for in-memory caching of Java objects. We use the configuration with LRU eviction algorithm and a blocking cache. Usage of the cache is modelled by two parallel threads accessing it.

CLIF is an open-source stress-testing platform, which is able to generate various kinds of traffics and measure resource usage for the system under test. The core of CLIF is based on the Fractal component model [17]. Our benchmark consists of one of its internal component, which is responsible for adding measured blade servers, and a generated environment, which simulates its usage. The environment was generated from a behaviour specification using [70].

The *CoCoME* benchmark is a prototype of a cash-desk system for supermarkets. It consists of an inventory management sub-system (storing a product database) and a cash-desk line formed by a set of cash desks. The application consists of a test driver, which simulates two clients served in parallel.

The *Elevator* benchmark is a simulator of elevators in a building. We use the configuration with two elevators and four actions executed by a simulated person.

FTDemo is a high-level component-based prototype of a software system providing Wi-Fi internet access at airports. The demo consists of around twenty software components, handling, e.g., user authentication, payment for network access, and IP address allocation. We ran the system with two simulated users in parallel.

Simple JBB is a simplified version of the SPEC JBB 2005 benchmark, which is a model of an enterprise information system for concurrent processing of clients' requests. It models several databases (e.g., orders and stock) and transactions that operate upon these databases. Some simplifications were necessary to make the benchmark run inside Jpf and to reduce the size of the state space to a reasonable level.

4.6.2 Experimental objectives

The main question to answer by our experiments was whether and to which extent the DVRs improve the performance of software model checking in the case of Java PathFinder, both in terms of the runtime and memory consumption. In particular, this involves both contribution to reduction of the state vectors and the reduction of the number of explored states. We were also interested in the overhead of our analyses, that is the memory and time consumption of Dynamic DVR and the static part of Hybrid DVR to see whether the analyses actually pay off. In other words, the aim was to determine whether employing dead variable analyses of heap data can broaden the set of Java programs that can be verified by Jpf.

4.6.3 Results

All the experiments were run on a Linux machine with an Intel(R) Xeon(R) X5687 (3.60GHz) CPU and 192GB memory with the Jpf filtering serializer. Tab. 4.4 and Tab. 4.5 summarize the results of the benchmarks.

In Tab. 4.4, the number of states and the running time (in hour:min:sec format) for each type of DVR is reported for each benchmark. The percentages are related to the values obtained by original Jpf. We ran each benchmark ten times and report the average run times. The standard deviation of Jpf run times was negligible with the average of 0.67% and maximum of 2.95%, which practically means few seconds in the case of the most complex benchmarks. The information about memory (state space, vector sizes, etc.) were constant across the runs. On average, Hybrid DVR yields a reduction of 12% in the verification time, while using Dynamic DVR results in a 20% speed-up and a reduction of 30% in the number of states. Note that the DeOS benchmark took in the case of Hybrid DVR five times longer to complete than in the original Jpf settings. This is due to static analysis performed at the beginning of the run, which, even though it reduced the state space to 59%, did not pay off. Since the original state space consists of 625 states only, we do not consider this case an issue.

Tab. 4.5 lists the information on memory consumption of the state matching using full state vectors. We report the memory consumed solely by state matching [2] – we summed up the sizes of state vectors used (they are all arrays of `int`). The default state matching in Jpf stores only hashes of state

[2] At the beginning, we used the *-Xmx* Java option to limit the maximal memory consumption of the Java process to estimate the approximate amount of memory needed for each benchmark.

Benchmark	original JPF		JPF with Hybrid DVR				JPF with Dynamic DVR			
	states	time	states		JPF time + SA time		states		time	
AlarmClock	573 362	2:15	573 362	100%	1:58 + 0:03	89%	573 362	100%	2:08	95%
CLIF	50 627	0:21	43 512	86%	0:14 + 0:04	86%	31 491	62%	0:47	224%
Cache4J	5 106 128	17:53	5 105 482	100%	17:03 + 0:04	96%	2 880 839	56%	14:18	80%
CoCoME	2 213 005	23:19	2 103 729	95%	19:44 + 0:05	85%	1 319 894	60%	13:36	58%
DeOS	625	0:01	370	59%	0:01 + 0:04	500%	211	34%	0:01	100%
Elevator	7 304 096	29:34	7 256 407	99%	27:12 + 0:03	92%	6 947 527	95%	33:44	114%
FTDemo	59 354	0:40	56 308	95%	0:31 + 0:06	92%	53 646	90%	0:35	88%
LinkedList	2 038 840	5:51	2 038 840	100%	5:10 + 0:04	89%	1 974 486	97%	6:15	107%
ProdConsumer	6 074 085	19:35	6 074 055	100%	16:46 + 0:04	86%	1 237 457	20%	3:46	19%
RepWorkers	15 363 223	56:42	15 362 801	100%	48:26 + 0:05	86%	12 052 301	78%	50:11	88%
SimpleJBB	109 861	2:30	85 655	78%	1:37 + 0:05	68%	60 222	55%	1:10	47%
Overall	38 893 206	2:38:42	38 700 521	100%	2:18:42 + 0:47	88%	27 131 436	70%	2:06:31	80%

Table 4.4: Experimental results – runtime and state space size. "JPF time" represents the time spent by Jpf at runtime. The "SA time" is the time spent in static analysis of HDVR. "Time" is the overall running time, if there is no static-analysis phase.

Benchmark	original JPF		JPF with Hybrid DVR				JPF with Dynamic DVR					
	memory	vector size	memory		vector size		memory		DVA memory	vec. size		DVA size
AlarmClock	2 802.82 MB	5 117.84	557.12 MB	20%	1 010.88	20%	593.42 MB	21%	246.81 MB	625.88	12%	451.37
CLIF	907.05 MB	18 778.60	198.48 MB	22%	4 775.00	25%	21.28 MB	2%	7.80 MB	440.76	2%	259.72
Cache4J	30 351.17 MB	6 224.80	12 300.51 MB	41%	2 518.32	40%	3 727.82 MB	12%	517.32 MB	1 160.56	19%	188.30
CoCoME	32 290.57 MB	15 292.08	16 097.64 MB	50%	8 015.64	52%	4 157.39 MB	13%	2 299.96 MB	1 467.60	10%	1 827.18
DeOS	3.62 MB	6 064.60	0.81 MB	22%	2 287.80	38%	0.41 MB	11%	0.25 MB	799.88	13%	1 242.39
Elevator	44 217.47 MB	6 339.88	19 410.64 MB	44%	2 796.92	44%	7 661.87 MB	17%	1 894.32 MB	862.48	14%	285.91
FTDemo	1 437.22 MB	25 382.60	455.65 MB	32%	8 477.16	33%	178.55 MB	12%	92.27 MB	1 678.36	7%	1 803.53
LinkedList	10 347.93 MB	5 313.96	1 887.42 MB	18%	962.68	18%	1 057.93 MB	10%	7.09 MB	550.08	10%	3.77
ProdConsumer	30 995.96 MB	5 342.88	8 675.65 MB	28%	1 489.68	28%	1 084.43 MB	3%	368.92 MB	598.32	11%	312.61
RepWorkers	87 001.07 MB	5 930.04	34 958.10 MB	40%	2 378.04	40%	9 302.25 MB	11%	695.21 MB	740.84	12%	60.48
SimpleJBB	5 452.43 MB	52 033.12	2 112.35 MB	39%	25 851.12	50%	366.94 MB	7%	205.17 MB	2 808.68	5%	3 572.39

Table 4.5: Experimental results – memory consumption. The vector sizes are averages over all vectors in a particular analysis. The "DVA memory" is the memory used solely by the DVA analysis. The "DVA size" is the average amount of bytes needed to store the DVA information for each state vector.

vectors, thus its memory requirements are negligible. In the table, both the absolute amount of memory and the percentage reduction in the case of our analyses are presented. We also report reduction of the state vector size; the reduction is smaller compared to reduction of the memory requirements, however, it is still significant. It is caused by fewer program states being visited. The reduction of the vector size is larger for Dynamic DVR than Hybrid DVR, because dynamic DVR is more precise.

We consider the vector shrinking particularly interesting. In the case of Hybrid DVR, memory consumption is reduced to one third on average, both in terms of the total consumption and vector sizes, while the memory consumption for Dynamic DVR is an order of magnitude lower.

An interesting example is the Alarm clock benchmark, where neither Hybrid nor Dynamic DVR reduce the number of states, but the verification is faster (cf. Tab. 4.4). This shows that the overhead of our DVAs is low; the analyses identify dead parts of program states faster than the state matching processes these dead parts (e.g., compute state hashes). In this particular case, the state vectors reduce to 20 and 12 per cent of the original size for Hybrid and Dynamic DVR, respectively (cf. Tab 4.5). The main source of dead variables is the `System.properties` object.

Hybrid DVR. In the case of Hybrid DVR, we present both the JPF runtime and the static phase duration. Typically, static analysis of live field takes less than five seconds (with a maximum of six seconds). If the Java standard library uses introspection to create class instances of a dynamic class (i.e., it calls `Class.forName(*).newInstance()`), it has a major impact on the static-analysis runtime. This is due to processing a large number of classes from standard libraries. Since static analysis considers also the used standard Java libraries, the whose size of processed Java bytecode is considerably larger than the size of the verified benchmarks, this phase does not represent a bottleneck of our approach.

Dynamic DVR. Our dynamic analysis needs an additional amount of memory to store live addresses of program states. This additional amount of memory is always required, independently of whether full state vectors or only their hashes are used in state matching. In Tab. 4.5, we report the memory used to store live addresses in the DVA memory column. On average, a single set of live addresses takes 910 bytes; the consumed memory ranges from 256KB for DeOS, to 2.3GB for CoCoME. Since multiple states often have the same set of live addresses (and differ only in the values at these addresses), the amount of memory required to store them is one to two orders of magnitude less than the memory needed to store and match full program states (instead of their hashes – JPF default). In Tab. 4.5, the column DVA size reports how much additional memory per program state is needed to store DVA information; due to sharing of the live addresses, the size is smaller than the average of 910 bytes.

4.6.4 Summary

The results of the benchmarks demonstrate that the contribution to performance in both the Hybrid and Dynamic DVR cases is significant. Moreover, according to our experience with software model checking, the most frequent reason why model checking fails is an insufficient amount of memory available, whereas the runtime is usually acceptable (even though it can take up to several hours). Regarding this, we consider the reduction in memory consumption when using our DVRs especially valuable.

Let us have a look at the aspects influencing the decision which analysis to choose for model checking of a particular input. In general, Hybrid DVR usually pays off if the benchmark runtime takes just several minutes. In this case, the speed up of the entire process is higher than the time required for the static analysis phase. The only situations when HDVR would not be suitable (from the practical point of view) is when many new (not implemented inside JPF) native methods are used. This requires creating models for them in JPF first (this is not DVR specific) and then create models for static analysis itself (HDVR specific), which represents a significant additional amount of work. The usefulness of dynamic DVR is hard to predict in general, but we foresee some heuristics based on random state space search and sampling the stack traces, which would provide an insight as to the DDVR benefits for a specific Java program (without model checking it). In particular, the size the set of live addresses at particular stack-trace influences the overall runtime in terms of number of comparisons when searching for an equivalent state.

It would be also useful to see the results if our techniques are combined with DVRs for local variables. We have not implemented a combined analysis mainly for the following reasons: The corresponding techniques are already known, so there would not be a scientific contribution in that. Next, the local DVRs only reduce the state space, not the state vectors. Finally, it would not be clear what is the effect of the local DVRs and what is the effect of the global (heap) ones' reduction (turning on and off each of them would not give a clear insight), but, on the other hand, we agree that it would show the practical contribution.

The selected benchmarks share some properties that, in general, might slightly bias the final observation about their results. First of all, all of them are multi-threaded. Even though this seems to be more general than including also single-threaded benchmarks, the (data) non-determinism appearing in single-threaded programs may cause the reductions to work slightly worse or better. Another aspect of the benchmarks is that their state spaces do not contain cycles. Unlike in the previous case, we do not expect programs with cyclic state space to have significant impact on reduction, regardless of the way it is handled, i.e., simplified or precise (see Sect. 4.4.1). Last aspect we would like to mention here is that we restricted ourselves, due to implementation platform, to explicit model checking. In principle, there is no obstacle in applying our reductions also in the context of symbolic model checking, but it is worth mentioning that a number of technical details have to be addressed there.

5 PVA Interpolants

As we have shown in the background chapter, Craig interpolants are heavily applied in symbolic model checking techniques. They are typically used as a means of abstraction and to refine the abstraction. The techniques differ in a way the interpolants are utilized as well as in additional properties the interpolants have to satisfy, e.g., path interpolation property, state-transition interpolation property, tree-interpolation property. These properties are essential for the safety of the corresponding approaches. The size and the logical strength of interpolants are other important attributes [24, 93], which are orthogonal to the aforementioned ones. Below, we present a technique to reduce the size of interpolants while preserving the ability to control their logical strength.

We have observed that in many cases, the techniques permit to compute interpolants under *assumption* and that the assumptions can be expressed in form of variable assignments (i.e., by focusing only to particular models of the formula).

For example, let consider Abstract Reachability Graphs (ARG). For each node in ARG, the label holds an (over)-approximation of reachable program states at that node; the over-approximation needs to be strong enough to block all the traces via given node to the faulty states. If Craig interpolants are used to compute these over-approximations (i.e., the current approach), the interpolant over-approximates the states reachable at any ARG node at the boundary between A and B parts. Depending on the structure of the ARG, the boundary could include not only the considered node but also many additional ones. Moreover, these additional nodes could introduce shared variables which do not exist at the considered node. Such variables could occur in the Craig interpolant and need to be subsequently eliminated. Based on the way the ARG is encoded into a formula, the variable assignment can be used to block paths via particular ARG node. This permits us to create an assignment which blocks all paths in the ARG that do not go via selected node, such paths in ARG cannot influence the reachable states at that node. Effectively such assignment reduces the boundary between A and B parts to a single ARG node (via which all the considered paths go). The assignment is then used as an assumption under which the interpolant (i.e., an over-approximation of reachable states) for given node is computed. This process and the way assignments are generated is described in more details in section 5.1.

Very similar idea can be applied to function summaries (e.g., in FunFrog). Function summary is a formula over input and output variables of the function such that the formula holds if the summarized function for given input values can return the specified output. Function summaries in FunFrog are computed as follows: first the BMC formula which encodes all possible (bounded) executions of the program is created. The BMC formula is given to the SAT solver; if the formula is unsatisfiable, the program is safe (w.r.t. considered assertions) and function summaries are computed as Craig interpolants from the resolution proof of unsatisfiability. To compute the summary for the function f, the BMC formula is partitioned such that the body of summarized function f and bodies of all transitively called functions (resp. their representation in BMC formula) belongs to the A partition. The rest of the BMC formula belongs to B partition. Based on the way BMC formula is created, the assignment could be used to eliminate all the traces in the B partition which do not call the summarized function. The assignment can be assumed when the summary is computed.

In the background chapter, we have introduced other techniques used to reduce size of interpolants. These techniques have in common that they either (i) reduce the size of the refutation proofs from which the interpolants are computed, or (ii) size of the interpolant formula. The former techniques identify common inefficiencies present in the refutation. They are applied once the refutation proof is constructed and before the interpolant computation; the interpolants are computed from reduced refutation proof. These techniques cannot be targeted to a specific interpolant or to a specific verification problem. The similar holds for the latter techniques (ii) which operation on the interpolant formula; in general, these techniques have to preserve a logical strength of interpolant. In the opposite case, the interpolants would lose additional properties required by the verification techniques.

In contrast to this, our technique is aware of the interpolation problem and it utilizes this information to reduce the interpolant size; in particular, the variable assignment is used to identify part of the refutation proof unimportant for given sub-problem. So, compared to the aforementioned techniques, we utilize different and goal specific inefficiencies, i.e., these techniques should be orthogonal and should complement each other.

The related techniques in general do not require additional inputs. In contrast to them, our technique depends on properly created assignments which limits applicability of the approach.

In this chapter, we introduce *Partial Variable Assignment Interpolants (PVAI)* – a generalization of Craig interpolants – which, in addition to the standard subdivision of an unsatisfiable formula (the *interpolation problem*) into A and B parts, is parametric in a *partial variable assignment (PVA)* – an assumption. A PVA defines a *sub-problem* on which a PVAI is focused. A sub-problem is obtained from the interpolation problem by removing the clauses (constraints) satisfied by the assignment. Due to this specialization, (1) the interpolants for the sub-problem can be of smaller size, compared to Craig interpolants computed for the interpolation problem. Moreover, since the satisfied constraints (those not occurring in the sub-problem) need not to be considered by interpolation, (2) it is possible to restrict the variables occurring in an interpolant to those relevant to the sub-problem, i.e. those shared between the A and B parts of the sub-problem.

We also propose the new framework of *Labeled Partial Assignment Interpolation Systems* (LPAIS) – a generalization of LIS, which computes PVAIs for propositional logic. We define the notion of logical strength for LPAISs and show how introducing a partial order over LPAISs allows to systematically compare the strength of the computed interpolants (a feature intuitively relevant to verification since it affects the coarseness of the over-approximations realized by interpolants [93]). We also show how LPAISs can be used to generate collections of interpolants which enjoy the path interpolation property. We evaluated our approach on a set of unsatisfiable benchmarks from SAT competition to see, how the assignments (and in particular LPAIS) can be used to reduce the size of interpolants.

5.1 Motivation

In the following, we illustrate a possible application of PVAIs; nonetheless, the proposed PVAIs are not limited to this context. As an example, consider the source code in Figure 5.1 and the corresponding ARG in Figure 5.2. Node i is associated with location i in the program. Node 1 is the initial node, [68]

```
1: int max(int i, int j) {
2:   if (i > j)
3:     return i;
       else
4:     return j;
5: }
   // The main function
6: assert(max(random(), 0) >= 0);
```

Figure 5.1: Motivating example

$$\begin{tikzcd}
& 1 \arrow[d, "\tau_{12} \equiv j = 0"] & \\
& 2 \arrow[dl, "\tau_{23} \equiv i > j"'] \arrow[dr, "\tau_{24} \equiv \neg(i > j)"] & \\
3 \arrow[dr, "\tau_{35} \equiv result = i"'] & & 4 \arrow[dl, "\tau_{45} \equiv result = j"] \\
& 5 \arrow[d, "\tau_{56} \equiv \neg(result >= 0)"] & \\
& 6 &
\end{tikzcd}$$

Figure 5.2: Abstract reachablity graph

while node 6 is the node representing an error location. The *edge constraints* τ_{ij} encode the semantics of the corresponding program statements. Note that τ_{12} originates from the call to the `max` function in `main`, on line 6. Further, in node 3, the parameter i is the only in-scope variable; similarly, in node 4 the parameter j is the only in-scope variable. A variable is in-scope at a given node, if there is a path through the node where the variable is used before as well as after the node.

In the context of software verification, an important question is whether

$$\begin{array}{ll}
\mu_1 \equiv (n_1 \Rightarrow n_2) & \wedge\, ((n_1 \wedge n_2) \Rightarrow \tau_{12}) \\
\mu_2 \equiv (n_2 \Rightarrow (n_3 \vee n_4)) & \wedge\, ((n_2 \wedge n_3) \Rightarrow \tau_{23}) \wedge \\
 & \wedge\, ((n_2 \wedge n_4) \Rightarrow \tau_{24}) \\
\mu_3 \equiv (n_3 \Rightarrow n_5) & \wedge\, ((n_3 \wedge n_5) \Rightarrow \tau_{35}) \\
\mu_4 \equiv (n_4 \Rightarrow n_5) & \wedge\, ((n_4 \wedge n_5) \Rightarrow \tau_{45}) \\
\mu_5 \equiv (n_5 \Rightarrow n_6) & \wedge\, ((n_5 \wedge n_6) \Rightarrow \tau_{56})
\end{array}$$

$$\mathsf{Cond} \equiv n_1 \wedge \mu_1 \wedge \mu_2 \wedge \mu_3 \wedge \mu_4 \wedge \mu_5$$

Figure 5.3: The Cond formula

an error location is actually reachable from the initial location of a program – this is known as the *reachability problem*. The question can be answered by computing, for each node i, the set of states reachable at i via paths in the program ARG [5, 80]. Typically, it is enough to compute an over-approximation of these states, i.e. a *node interpolant*. To this end, the ARG is converted into a Cond formula, which represents all execution paths in the ARG. The Cond have the same meaning as ArgCond in [3]. An auxiliary *structure-encoding* Boolean variable n_i is introduced for each node i in the ARG; for each i (except for the error node), a *node formula* μ_i is created, which encodes the labels on the outgoing edges (Figure 5.3).

For illustration, we describe the meaning of μ_2. The first conjunct $n_2 \Rightarrow (n_3 \vee n_4)$ expresses that after reaching node 2, a path has to proceed to a successor node (3 or 4). The second conjunct $(n_2 \wedge n_3) \Rightarrow \tau_{23}$ guarantees that if a path goes via the edge $2 \rightarrow 3$, the semantics of the edge is preserved (i.e., the constraint τ_{23} is satisfied). Similarly, the third conjunct enforces the semantics of the edge $2 \rightarrow 4$.

The Cond formula is satisfiable if and only if a feasible path exists that leads from node 1 to node 6 in the ARG. Suppose now that Cond is unsatisfiable; then a node interpolant for each node i can be computed. First, the ARG needs to be partitioned into A and B – so that A corresponds to the antecedents of i, B to all the other nodes in the ARG – and then a Craig interpolant I is generated as an over-approximation of the states reachable at i. For instance, in the case of node 3, A would be set to $n_1 \wedge \mu_1 \wedge \mu_2$ and B to $\mu_3 \wedge \mu_4 \wedge \mu_5$. However, employing standard Craig interpolation in this manner to compute a node interpolant I is not sufficient; out-of-scope variables might in fact belong to both A and B, they could therefore appear in I, and should be consequently eliminated. Variable j, in particular, could appear in the interpolant for node 3. Even though out-of-scope variables can be eliminated by resorting to quantification followed by a quantifier-elimination phase, this phase is a well-known bottleneck in verification [102].

Computing node interpolants using PVAIs effectively solves the problem of out-of-scope program variables. Assume that a node interpolant is to be computed for a node k; a suitable PVA assigns False to all the structure-encoding variables corresponding to the nodes not lying on the paths through k. By setting a variable n_j to False, the paths via node j are blocked; moreover, the whole node formula μ_j is satisfied and thus μ_j is not a part of the sub-problem for node k. On the other hand, the PVA can assign n_k to True to express that each considered path has to pass through k (the node for which the interpolant is computed). In particular, to compute an interpolant for node 3, consider Fig. 5.4; we assign n_3 to True and n_4 to False to block the path through node 4 (π_3); the rest of variables remain unassigned. This

$$\begin{aligned}
\pi_3 \equiv\ & n_3 \wedge \overline{n_4} \\
A_3 \equiv\ & n_1 \wedge \\
& (n_1 \Rightarrow n_2) \wedge ((n_1 \wedge n_2) \Rightarrow j = 0) \wedge \\
& \qquad\qquad \wedge ((n_2 \wedge n_3) \Rightarrow i > j) \\
\\
B_3 \equiv\ & (n_3 \Rightarrow n_5) \wedge ((n_3 \wedge n_5) \Rightarrow \mathit{result} = i) \wedge \\
& (n_5 \Rightarrow n_6) \wedge ((n_5 \wedge n_6) \Rightarrow \neg(\mathit{result} >= 0))
\end{aligned}$$

Figure 5.4: The A and B parts of the sub-problem for node 3

assignment satisfies (and thus removes) $n_2 \Rightarrow (n_3 \vee n_4)$, $(n_2 \wedge n_4) \Rightarrow \tau_{24}$ and μ_4 (Fig. 5.3) from the sub-problem (see Fig. 5.4). In the A part, the sub-problem for node 3 contains the edge labels (and consequently the program state variables) related to the path from node 1 to node 3, and in the B part, information related to the path from node 3 to node 6. The program state variables shared by the A and B parts of the sub-problem are the in-scope variables, which are exactly those that may appear in PVA interpolants.

5.2 Preliminaries

Let us recall the most important notation from the background section.

Resolution proof R for a CNF formula Φ is a tuple (V, E, cl, piv, s), where V is a set of vertices in the proof, $E \subset V \times V$ is a set of edges forming a full binary DAG (i.e., all the vertices except for the leaves have the in-degree 2). The sink vertex has the out-degree 0. Each vertex $v \in V$ is associated to a vertex-clause specified by $cl(v)$ function. Each vertex clause of a leaf vertex v corresponds to a clause from input formula Φ (i.e., $cl(v) \in \Phi$). Each inner vertex v represents resolution of its antecedent vertex-clauses (specified by cl) using the pivot $piv(v)$; formally, for each inner vertex v there exist edges $(v_1, v), (v_2, v) \in E$ such that $cl(v) = Res(cl(v_1), cl(v_2), piv(v))$. A *refutation* derives the empty clause in the sink vertex s; formally $cl(s) = \bot$.

Since the resolution proofs take the set of clauses as input, the input formula is first converted into a conjunction of clauses. Therefore, in the following we use the terms formula and set of clauses interchangeably.

Craig Interpolants Let us briefly recall the definition of Craig interpolant and interpolant sequence from the background section. Given an unsatisfiable formula Φ and its (A,B)-partitioning into $A \wedge B$ parts, a *Craig interpolant* [35] is a formula I such that (1) $A \Rightarrow I$, (2) $B \wedge I \Rightarrow \bot$, and (3) $\mathsf{Var}(I) \subseteq$

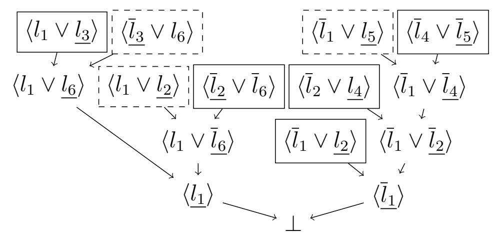

$\langle l_1 \vee \underline{l_3}\rangle$
$\langle \underline{\bar{l}_3} \vee l_6\rangle$
$\langle \bar{l}_1 \vee \underline{l_5}\rangle$
$\langle \bar{l}_4 \vee \underline{\bar{l}_5}\rangle$
$\langle l_1 \vee \underline{l_6}\rangle$
$\langle l_1 \vee \underline{l_2}\rangle$
$\langle \underline{\bar{l}_2} \vee \bar{l}_6\rangle$
$\langle \bar{l}_2 \vee \underline{l_4}\rangle$
$\langle \bar{l}_1 \vee \underline{\bar{l}_4}\rangle$
$\langle l_1 \vee \underline{\bar{l}_6}\rangle$
$\langle \bar{l}_1 \vee \underline{l_2}\rangle$
$\langle \bar{l}_1 \vee \underline{\bar{l}_2}\rangle$
$\langle \underline{l_1}\rangle$
$\langle \underline{\bar{l}_1}\rangle$
$\perp$

Figure 5.5: Refutation resolution proof; the clauses from A-part and B-part are in dashed and full boxes, respectively.

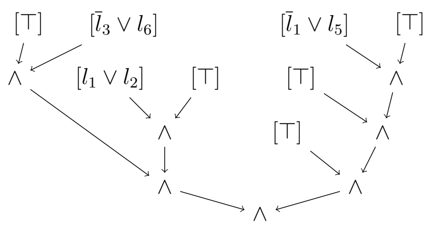


Figure 5.6: Derivation of McMillan's interpolant $(l_1 \vee l_2) \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$.

$\mathsf{Var}(A) \cap \mathsf{Var}(B)$.

An *interpolant sequence* for the unsatisfiable formula $A_1 \wedge A_2 \wedge ... \wedge A_n$ is a tuple of formulas $(I_0, I_1, ...I_n)$, where I_i is an interpolant for partitioning $(A_1 \wedge ... \wedge A_i, A_{i+1} \wedge ... \wedge A_n)$. If for all i, $I_i \wedge A_i \Rightarrow I_{i+1}$, then $(I_0, I_1, ...I_n)$ is said to satisfy the *path interpolation* (PI) property. In [54], it was proved that the path interpolation property holds for any LISs, including the well-known McMillan's and Pudlák's systems, whenever the interpolant sequence is computed from the same proof.

D'Silva et al. introduced the *labeled interpolation systems* (LIS) [41], which allows constructing interpolants of different logical strengths. The system is parametrized by a labeling function assigning a label (color) to each node of the refutation tree; the label of the pivot variable in particular nodes then defines the rule used to compute the corresponding partial interpolant. We build upon their system, using an additional parameter in the form of a partial variable assignment.

Example 1: Figure 5.5 shows a resolution refutation proof for formula $\Phi = \langle l_1 \vee l_2\rangle \wedge \langle \bar{l}_1 \vee l_5\rangle \wedge \langle \bar{l}_3 \vee l_6\rangle \wedge \langle l_1 \vee l_3\rangle \wedge \langle \bar{l}_2 \vee \bar{l}_6\rangle \wedge \langle \bar{l}_4 \vee \bar{l}_5\rangle \wedge \langle \bar{l}_2 \vee l_4\rangle \wedge \langle \bar{l}_1 \vee l_2\rangle$. [65] Assume an (A,B)-partitioning with A consisting of the conjunction of the first free clauses and B of the remaining five clauses. There might not be just a single interpolant for an unsatisfiable formula; many different ones of various strengths can exist. Figure 5.6 shows how McMillan's interpolant $I_1 \equiv (l_1 \vee l_2) \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$ can be derived (after constant propagation) from the proof in Figure 5.5, e.g., by LIS. Note that for convenience we write the partial interpolant associated to a particular node of the proof into brackets. Formula $I_2 \equiv (l_1 \vee [(l_6 \vee \bar{l}_3) \wedge (\bar{l}_6 \vee l_2)]) \wedge (\bar{l}_1 \vee l_5)$ is another interpolant which can be computed by LIS from the proof; Figure 5.9 shows the labels used to compute I_2 (minimal labeling).

Variable assignments. Let A be a set of clauses. A *variable assignment* assigns either True ($\top$) or False ($\bot$) to each variable in the $\mathsf{Var}(A)$ set. The variable assignment can be seen as a conjunction of literals. A *partial variable assignment* (PVA) π assigns values only to a subset of variables in $\mathsf{Var}(A)$. A PVA π can be used as an assumption w.r.t. A (i.e., $\pi \models A$) to restrict the set of models of A to those compatible with π.

Definition 8 (Clauses under assignment) *Let A be a set of clauses and π be a PVA over $\mathsf{Var}(A)$. We define the sets of*

satisfied *clauses*	$A_\pi = \{\langle\Theta\rangle \mid \langle\Theta\rangle \in A \text{ and } \pi \models \langle\Theta\rangle\}$ *and*
unsatisfied *clauses*	$A_{\overline{\pi}} = \{\langle\Theta\rangle \mid \langle\Theta\rangle \in A \text{ and } \pi \not\models \langle\Theta\rangle\}$.

Satisfied clauses contain at least one literal evaluated to $\top$ under π, while, for unsatisfied clauses, every literal is either unassigned or falsified. The unsatisfied clauses $A_{\overline{\pi}}$ determine the sub-problem. We use $\pi \models l$ to express that a literal l evaluates to $\top$ in a given PVAπ.

[65] **Example 1 (cont.):** Let us assume assignment $\pi \equiv \bar{l}_2$ (i.e., assigning $False$ to variable l_2) and the set of clauses from our previous example. Given the assignment, B can be split into $B_\pi \equiv \langle \bar{l}_2 \vee \bar{l}_6\rangle \wedge \langle \bar{l}_2 \vee l_4\rangle$ and $B_{\overline{\pi}} \equiv \langle l_1 \vee l_3\rangle \wedge \langle \bar{l}_4 \vee \bar{l}_5\rangle \wedge \langle \bar{l}_1 \vee l_2\rangle$. A_π is empty thus $A_\pi \equiv \top$ and $A_{\overline{\pi}} \equiv A$.

5.3 Partial Variable Assignment Interpolants

[68] In this section, we formally define *Partial Variable Assignment Interpolation*, which, in addition to the division of an unsatisfiable formula into A and B parts, requires specification of a PVA.

Definition 9 (PVA Interpolant) *Let R be a refutation of formula $A \wedge B$ and π be a partial variable assignment over $\mathsf{Var}(A \wedge B)$. A* partial variable assignment interpolant *(PVAI) is a formula I such that:*

(D9.1) $\pi \models A \Rightarrow I$

(D9.2) $\pi \models B \wedge I \Rightarrow \bot$

(D9.3) $\mathsf{Var}(I) \subseteq \mathsf{Var}(A_{\overline{\pi}}) \cap \mathsf{Var}(B_{\overline{\pi}})$

(D9.4) $\mathsf{Var}(I) \cap \mathsf{Var}(\pi) = \emptyset$

In the following, we use (A, B, π) to denote a refutation of formula $A \wedge B$ and partial variable assignment π. that a PVAI is computed using (A, B)-partitioning and the partial assignment π.

Since for any set of clauses X, $\pi \models (X \Leftrightarrow X_{\overline{\pi}})$, D9.1 and D9.2 can be equivalently rewritten as $\pi \models A_{\overline{\pi}} \Rightarrow I$ and $\pi \models B_{\overline{\pi}} \wedge I \Rightarrow \bot$, respectively; in other words, I is a Craig interpolant for the sub-problem, i.e., for $A_{\overline{\pi}} \wedge B_{\overline{\pi}}$, which is formed by removing the literals falsified by the assignment π from $A \wedge B$. Note that even after removing (the satisfied) clauses, the sub-problem remains unsatisfiable (assuming π).

On the other hand, a PVAI cannot be obtained from standard interpolants by application of a partial assignment (denoted by $I[\pi]$). The reason is that, in addition to assigned variables (disallowed by D9.4), rule D9.3 excludes from the PVAI also all unassigned (out-of-scope) variables that occur in satisfied clauses only, which can still appear in $I[\pi]$.

Example 1 (cont.): Craig and PVA interpolants differ in the variables which can occur in the interpolant. The shared variables between A and B (i.e., those that can appear in a Craig interpolant) are l_1, l_2, l_3, l_5, and l_6. Since PVAI considers (for the shared variables) only unsatisfied parts of A resp. B (i.e., $A_{\overline{\pi}}$ and $B_{\overline{\pi}}$), fewer variables are shared; in our example, assuming $\pi \equiv \bar{l}_2$, only l_1, l_3, and l_5 can appear in a PVA interpolant, which are those that can appear in a Craig interpolant for the sub-problem.

Given an assignment ($\pi \equiv \bar{l}_2$) and a Craig interpolant, an alternative way to reduce the interpolant size is to assign the values inside the interpolant formula and propagate the Boolean constants. In this case, the interpolants from the example above result in $I_1[\pi] \equiv l_1 \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$ and $I_2[\pi] \equiv (l_1 \vee [(l_6 \vee \bar{l}_3) \wedge \bar{l}_6]) \wedge (\bar{l}_1 \vee l_5)$. None of them is a PVA interpolant since each one contains variable l_6. Both of them can be further simplified (i.e., equivalently rewritten) as $I_1^s[\pi] \equiv l_1 \wedge l_5 \wedge (\bar{l}_3 \vee l_6)$ resp. $I_2^s[\pi] \equiv l_1 \vee (\bar{l}_3 \wedge \bar{l}_6) \wedge (\bar{l}_1 \vee l_5)$. In general, such a transformation requires a complex analysis and sometimes the out-of-scope variables can be eliminated from the interpolant by this technique. However, as we shown above, variable l_6 cannot be eliminated from the interpolants by these transformations. This means that the aforementioned techniques can be used to reduce the size of the formula, but it cannot guarantee producing interpolants without the variables appearing just in satisfied clauses – the information is not present in the interpolant formula.

Calls of a solver are resource demanding. A refutation is independent of PVAs; this important fact allows us to call the solver only once on the overall [68] problem Φ, and, later, to compute various PVAs (representing relevant sub-problems) for which the PVAI can be efficiently computed. This follows the idea of having a refutation and several partitionings, for which several (related) interpolants are computed.

Although Craig interpolation has many applications in program verification, verification tools often require interpolation sequences with specific properties [54]. The PVAI for all the sub-problems are computed from the same proof, thus they are related to each other. The existence of a single proof permits the application of a standard proving technique in the area of interpolation – structural induction over a refutation – to show various properties of PVA interpolant sequences. All the techniques where interpolants for different sub-problems are computed using different proofs (e.g., applying a solver directly on each sub-problem, or incremental solving with assumptions) do not, per se, guarantee any properties of their sequences. The price to pay is an additional assumption in the form of a partial assignment.

5.4 Labeled Partial Assignment Interpolation System

To show that PVAIs are not just a theoretical concept, we present the framework of *Labeled Partial Assignment Interpolation Systems*, a generalization of LISs [41], which computes PVAIs for propositional logic, and prove its soundness. Next, in order to prove the path interpolation property, we introduce the concept of logical strength on LPAISs, which allows one to systematically compare the strength of the generated interpolants.

In order to define LPAISs, first we have to extend the definitions of labeling functions and locality from LISs to take variable assignments into account. Note that if no variable is assigned, LPAISs are equivalent to LISs.

A labeling function assigns labels to literals in a refutation tree; the labeling drives the computation of an interpolant from the proof and determines its strength (Fig. 5.7). Note that in the following, if not stated otherwise, we assume just a single refutation being re-used for computing many interpolants.

Definition 10 (Labeling function) *Let $L = (S, \sqsubseteq, \sqcap, \sqcup)$ be the lattice in Figure 5.7, where $S = \{\bot, a, b, ab, d^+\}$ and $\bot$ is the least element, and let $R = (V, E, cl, piv, s)$ be a resolution proof over a set of literals* Lit. *Function*

$\mathsf{Lab}_{R,L} : V \times \mathsf{Lit} \to S$ *is called* labeling function *for refutation R iff $\forall v \in V$ and $\forall l \in \mathsf{Lit}, \mathsf{Lab}_{R,L}$ satisfies the following conditions:*

(D10.1) $\mathsf{Lab}_{R,L}(v, l) = \bot$ *if and only if $l \notin cl(v)$, and*

(D10.2) $\mathsf{Lab}_{R,L}(v, l) = \mathsf{Lab}_{R,L}(v_1, l) \sqcup \mathsf{Lab}_{R,L}(v_2, l)$*, where v_1, v_2 are the antecedent vertices.*

From condition D10.2 it follows that the labeling function is fully determined once the labels in the leaves have been specified. We omit subscripts R and L if clear from the context.

$$\begin{tikzcd}
 & d^+ & \\
 & ab \arrow[u] & \\
a \arrow[ur] & & b \arrow[ul] \\
 & \bot \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 5.7: Lattice of labels ($\sqcup$)

Naming conventions. Let us assume a pair of sets of clauses (A, B) and a PVA π. The clause sets are split into four groups, the unsatisfied clauses $A_{\overline{\pi}}$ and $B_{\overline{\pi}}$ which specify the sub-problem and are taken into account during interpolation, and the satisfied clauses A_π and B_π, which are disregarded.

We distinguish among the following kinds of variables, depending on the standard notions of locality and sharedness, as well as on where the variables appear in the four groups of clauses. We say that a variable k is *unassigned* if $k \notin \mathsf{Var}(\pi)$. An unassigned variable k is:

$A_{\overline{\pi}}$*-local*	if $k \in \mathsf{Var}(A_{\overline{\pi}})$ and $k \notin \mathsf{Var}(B_{\overline{\pi}})$
$B_{\overline{\pi}}$*-local*	if $k \notin \mathsf{Var}(A_{\overline{\pi}})$ and $k \in \mathsf{Var}(B_{\overline{\pi}})$
$A_{\overline{\pi}}B_{\overline{\pi}}$*-shared*	if $k \in \mathsf{Var}(A_{\overline{\pi}})$ and $k \in \mathsf{Var}(B_{\overline{\pi}})$
$A_{\overline{\pi}}B_{\overline{\pi}}$*-clean*	if $k \notin \mathsf{Var}(A_{\overline{\pi}})$ and $k \notin \mathsf{Var}(B_{\overline{\pi}})$

The properties above are independent of the occurrence of k in $\mathsf{Var}(A_\pi)$ and $\mathsf{Var}(B_\pi)$. The "clean" variables occur only in the satisfied clauses, thus are out-of-scope and cannot appear in PVA interpolants.

Definition 11 *We say that variable k is* McMillan-labeled *if, whenever k is $A_{\overline{\pi}}B_{\overline{\pi}}$-shared or $A_{\overline{\pi}}B_{\overline{\pi}}$-clean, it is labeled b.*

Note that the labels of the other variables are not limited to b. If all variables are McMillan-labeled, LIS reduces to McMillan's interpolation system [41], which yields the strongest interpolant that LISs (and LPAISs) can produce from a given refutation.

$$\begin{tikzcd}
\langle l_1^{\mathrm{b}} \vee l_3^{\mathrm{b}} \rangle \arrow[d] & \langle \bar{l}_3^{\mathrm{b}} \vee l_6^{\mathrm{a}} \rangle \arrow[dl] & & & \langle \bar{l}_1^{\mathrm{b}} \vee l_5^{\mathrm{b}} \rangle \arrow[dr] & \langle \bar{l}_4^{\mathrm{b}} \vee \bar{l}_5^{\mathrm{b}} \rangle \arrow[d] \\
\langle l_1^{\mathrm{b}} \vee l_6^{\mathrm{a}} \rangle \arrow[ddrr] & \langle l_1^{\mathrm{b}} \vee l_2^{\mathrm{b}} \rangle \arrow[dr] & \langle \bar{l}_2^{\mathrm{d}^+} \vee \bar{l}_6^{\mathrm{a}} \rangle \arrow[d] & & \langle \bar{l}_2^{\mathrm{d}^+} \vee l_4^{\mathrm{b}} \rangle \arrow[dr] & \langle \bar{l}_1^{\mathrm{b}} \vee \bar{l}_4^{\mathrm{b}} \rangle \arrow[d] \\
 & & \langle l_1^{\mathrm{b}} \vee \bar{l}_6^{\mathrm{a}} \rangle \arrow[d] & & \langle \bar{l}_1^{\mathrm{b}} \vee l_2^{\mathrm{b}} \rangle \arrow[dr] & \langle \bar{l}_1^{\mathrm{b}} \vee \bar{l}_2^{\mathrm{d}^+} \rangle \arrow[d] \\
 & & \langle l_1^{\mathrm{b}} \rangle \arrow[dr] & & & \langle \bar{l}_1^{\mathrm{b}} \rangle \arrow[dll] \\
 & & & \bot & &
\end{tikzcd}$$

Figure 5.8: McMillan's labeling for (A, B, π).

$$\begin{tikzcd}
\langle l_1^{\mathrm{b}} \vee l_3^{\mathrm{b}} \rangle \arrow[d] & \langle \bar{l}_3^{\mathrm{a}} \vee l_6^{\mathrm{a}} \rangle \arrow[dl] & & & \langle \bar{l}_1^{\mathrm{a}} \vee l_5^{\mathrm{a}} \rangle \arrow[dr] & \langle \bar{l}_4^{\mathrm{b}} \vee \bar{l}_5^{\mathrm{b}} \rangle \arrow[d] \\
\langle l_1^{\mathrm{b}} \vee l_6^{\mathrm{a}} \rangle \arrow[ddrr] & \langle l_1^{\mathrm{a}} \vee l_2^{\mathrm{a}} \rangle \arrow[dr] & \langle \bar{l}_2^{\mathrm{b}} \vee \bar{l}_6^{\mathrm{b}} \rangle \arrow[d] & & \langle \bar{l}_2^{\mathrm{b}} \vee l_4^{\mathrm{b}} \rangle \arrow[dr] & \langle \bar{l}_1^{\mathrm{a}} \vee \bar{l}_4^{\mathrm{b}} \rangle \arrow[d] \\
 & & \langle l_1^{\mathrm{a}} \vee \bar{l}_6^{\mathrm{b}} \rangle \arrow[d] & & \langle \bar{l}_1^{\mathrm{b}} \vee l_2^{\mathrm{b}} \rangle \arrow[dr] & \langle \bar{l}_1^{\mathrm{a}} \vee \bar{l}_2^{\mathrm{b}} \rangle \arrow[d] \\
 & & \langle l_1^{\mathrm{ab}} \rangle \arrow[dr] & & & \langle \bar{l}_1^{\mathrm{ab}} \rangle \arrow[dll] \\
 & & & \bot & &
\end{tikzcd}$$

Figure 5.9: Minimal labeling for (A, B) and empty assignment.

Definition 12 *Variable k is labeled* consistently *if all occurrences of k in a refutation have the same label:*

$$\forall x, x' \in V, l \in cl(x), l' \in cl(x') : \mathsf{Var}(l) = \mathsf{Var}(l') \Rightarrow \mathsf{Lab}(v, l) = \mathsf{Lab}(v', l').$$

Example 1 (cont.): Figure 5.8 shows how a labeling function assigns labels to literals; the label of a literal is shown in superscript. We choose the strongest possible labeling (which for an empty assignment would produce McMillan's interpolant I_1); in particular $A_{\overline{\pi}}B_{\overline{\pi}}$-shared and $A_{\overline{\pi}}B_{\overline{\pi}}$-clean variables are labeled b. Note that variable l_6 is $A_{\overline{\pi}}$-local and thus has to be labeled a, the $A_{\overline{\pi}}B_{\overline{\pi}}$-shared variables are l_1, l_3, and l_5, no variable is $A_{\overline{\pi}}B_{\overline{\pi}}$-clean, and variable l_4 is $B_{\overline{\pi}}$-local. Figure 5.9 shows another example of labeling; note clauses $\langle l_1 \rangle$ and $\langle \bar{l}_1 \rangle$ which illustrate how the labels are merged from the labels in antecedents.

Not all labeling functions can be used to generate interpolants; in LPAIS, interpolants are computed if a locality preserving labeling is used.

Definition 13 *Let R be a refutation of formula $A \wedge B$ and π be a PVA. Labeling function* Lab *for refutation R is* locality preserving *iff $\forall v \in V, \forall l \in cl(v)$ all the following locality constraints are satisfied:*

(D13.1) $\mathsf{Lab}(v, l) = d^+ \Leftrightarrow \pi \models l$

(D13.2) $\mathsf{Var}(l)$ *is unassigned and A_π-local* $\Rightarrow \mathsf{Lab}(v, l) = a$

(D13.3) $\mathsf{Var}(l)$ *is unassigned and B_π-local* $\Rightarrow \mathsf{Lab}(v, l) = b$

(D13.4) $\mathsf{Var}(l)$ *is unassigned and $A_\pi B_\pi$-clean $\Rightarrow$ it is consistently labeled a or b.*

Locality constraints provide freedom in labeling $A_\pi B_\pi$-shared and $A_\pi B_\pi$-clean variables; the choice of labels directly affects the strength of the computed interpolants. The label of $A_\pi B_\pi$-shared variables can be set freely to a, b, or ab. The same holds for falsified literals; their labels are irrelevant since they are removed by the assignment filter (defined below).

D13.2 and D13.3 are equivalent to the locality requirements of LIS, where A-local and B-local variables must be labeled a and b, respectively. D13.1 concerns the satisfied literals. Label d^+ is used in the interpolation process to identify resolutions with an assigned pivot and parts of the proof which are not relevant to the sub-problem. D13.4 is specific to PVAI and deals with variables which occur in the satisfied clauses only. The requirement guarantees that such variables do not occur in the interpolant, because Res-ab (see Tab. 5.1) cannot be applied. Further, note that for the empty assignment, the locality constraints reduce to those of LISs, since D13.1 and D13.4 do not apply to any literal.

Filters. For a clause $\langle\Theta\rangle$, a labeling function Lab, a resolution-proof vertex $v \in V$, and a label c, we define the *match filter* $|$, which preserves only the literals with the specified label, as follows:

$$\langle\Theta\rangle|_{c,v,\mathsf{Lab}} = \{l \in \langle\Theta\rangle \mid c = \mathsf{Lab}(v, l)\}$$

Similarly, we define the *upward filter* $\upharpoonright$, which preserves the literals with labels greater than c (Fig. 5.7), as:

$$\langle\Theta\rangle\upharpoonright_{c,v,\mathsf{Lab}} = \{l \in \langle\Theta\rangle \mid c \sqsubseteq \mathsf{Lab}(v, l)\}$$

The subscripts Lab, v are omitted if clear from the context. Given a partial assignment π and a clause $\langle\Theta\rangle$, we also define the *assignment filter*,

Leaf v:	$\langle\Theta\rangle, [I]$	
$I = \langle\Theta\rangle[\pi]\|_{b,v,\mathsf{Lab}}$	if $\langle\Theta\rangle \in A_{\overline{\pi}}$	Hyp-$A_{\overline{\pi}}$
$I = \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}}$	if $\langle\Theta\rangle \in B_{\overline{\pi}}$	Hyp-$B_{\overline{\pi}}$
$I = \top$	if $\langle\Theta\rangle \in A_{\pi} \cup B_{\pi}$	Hyp-A_{π}, Hyp-B_{π}
Inner vertex v:	$\dfrac{v_1 : \langle p, \Theta_1\rangle, [I_1] \qquad v_2 : \langle \bar{p}, \Theta_2\rangle, [I_2]}{\langle\Theta_1, \Theta_2\rangle, [I]}$	
$I = I_1 \vee I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a$	Res-a
$I = I_1 \wedge I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$	Res-b
$I = (I_1 \vee p) \wedge (I_2 \vee \overline{p})$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$	Res-ab
$I = I_2$	if $\mathsf{Lab}(v_1, p) = d^+$	Res-d^+
$I = I_1$	if $\mathsf{Lab}(v_2, \overline{p}) = d^+$	Res-d^+

Table 5.1: Hypothesis and resolution rules for Labeled Partial Variable Assignment Interpolation System

which removes all the assigned literals (satisfied and falsified ones), as follows:

$$\langle\Theta\rangle[\pi] = \{l \in \langle\Theta\rangle \mid \mathsf{Var}(l) \notin \mathsf{Var}(\pi))\}$$

Moreover, in our notation, we assume that filters have a higher precedence than negation. E.g., $\neg\langle\Theta\rangle[\pi]\upharpoonleft_a$ can be equivalently rewritten as $\neg((\langle\Theta\rangle[\pi])\upharpoonleft_a)$.

Interpolation system. An interpolation system is a procedure for computing an interpolant given a refutation. It assigns partial *vertex-interpolant* to each vertex of the refutation, yielding the final interpolant at the sink vertex.

Definition 14 *Let R be a refutation of $A \wedge B$, π be a PVA, and* Lab *be a corresponding locality preserving labeling function. Then, Tab. 5.1 defines the* Labeled Partial Assignment Interpolation System $\mathsf{Lpaltp}(\mathsf{Lab}, R, A, B, \pi)$.

LPAIS produces interpolants in the following way: First, the vertex-interpolants for leaves of the refutation are computed using the rules in the upper part of Table 5.1 (*Hyp*othesis rules). Depending on the occurrence of the vertex-clause $\langle\Theta\rangle$ in the A or B sets, the corresponding rule describes the transformation of the vertex-clause into a partial vertex-interpolant. Later, going down through the proof from the leaves to the sink, the vertex-interpolants for inner vertices are computed using the resolution rules in the lower part of Table 5.1. The labels assigned to the pivots determine how vertex-interpolants of both antecedents are combined. This process ends at

$$\begin{tikzcd}
\overset{\text{Hyp-}B_{\overline{\pi}}}{\langle l_1^{\mathrm{b}} \vee l_3^{\mathrm{b}}\rangle} \arrow[d, "\text{Res-}b"'] & \overset{\text{Hyp-}A_{\overline{\pi}}}{\langle \bar{l}_3^{\mathrm{b}} \vee l_6^{\mathrm{a}}\rangle} \arrow[dl] & & \overset{\text{Hyp-}A_{\overline{\pi}}}{\langle \bar{l}_1^{\mathrm{b}} \vee l_5^{\mathrm{b}}\rangle} \arrow[dr] & \overset{\text{Hyp-}B_{\overline{\pi}}}{\langle \bar{l}_4^{\mathrm{b}} \vee \bar{l}_5^{\mathrm{b}}\rangle} \arrow[d, "\text{Res-}b"] \\
\langle l_1^{\mathrm{b}} \vee l_6^{\mathrm{a}}\rangle \arrow[ddrr] & \overset{\text{Hyp-}A_{\overline{\pi}}}{\langle l_1^{\mathrm{b}} \vee l_2^{\mathrm{b}}\rangle} \arrow[dr] & \overset{\text{Hyp-}B_{\pi}}{\langle \bar{l}_2^{\mathrm{d}^+} \vee \bar{l}_6^{\mathrm{a}}\rangle} \arrow[d, "\text{Res-}d^+"] & \overset{\text{Hyp-}B_{\pi}}{\langle \bar{l}_2^{\mathrm{d}^+} \vee l_4^{\mathrm{b}}\rangle} \arrow[dr] & \langle \bar{l}_1^{\mathrm{b}} \vee \bar{l}_4^{\mathrm{b}}\rangle \arrow[d, "\text{Res-}b"] \\
& & \langle l_1^{\mathrm{b}} \vee \bar{l}_6^{\mathrm{a}}\rangle \arrow[d, "\text{Res-}a"] & \overset{\text{Hyp-}B_{\overline{\pi}}}{\langle \bar{l}_1^{\mathrm{b}} \vee l_2^{\mathrm{b}}\rangle} \arrow[dr] & \langle \bar{l}_1^{\mathrm{b}} \vee \bar{l}_2^{\mathrm{d}^+}\rangle \arrow[d, "\text{Res-}d^+"] \\
& & \langle l_1^{\mathrm{b}}\rangle \arrow[dr, "\text{Res-}b"] & & \langle \bar{l}_1^{\mathrm{b}}\rangle \arrow[dl] \\
& & & \perp &
\end{tikzcd}$$

Figure 5.10: Rules applied at proof vertices if McMillan's labeling of Figure 5.8 is used.

the sink vertex where the PVAI is derived. The interpolants are computed in time linear to the size of the proof.

The main difference compared to LISs are the additional d^+ rules. For instance, consider the last rule $\mathsf{Lab}(v_2, \overline{p}) = d^+$ in Table 5.1. In contrast to the original LIS rules, the partial vertex-interpolant is simpler, because it does not contain I_2, omitted due to the variable assignment. Generally, these rules *cut out* the satisfied sub-tree of the proof. Usually, the later in the refutation the assigned variable is resolved, the larger sub-tree is pruned and the smaller the resulting interpolant is.

The differences between LPAISs and LISs are motivated by the way variable assignments work. The new d^+ rules can be seen as a specialization of the *ab* resolution rule if PVA π is assumed. A similar relationship holds for the hypothesis rules in the leaves of a refutation. These rules are equivalent to LIS hypothesis rules if applied on a clause under the assumed assignment. The changes we introduce w.r.t. LISs are of two kinds: (i) those in LPAIS rules force specialization of the interpolant on a sub-problem, and (ii) the changes in the locality constraints remove unassigned out-of-scope variables from the interpolant.

Example 1 (cont.): Figures 5.10 and 5.11 show the rules, which applies to a vertex; the labels are taken from Figures 5.8 resp. 5.9.

Figure 5.12 shows how LPAIS produces interpolant $I_\pi \equiv l_1 \vee \bar{l}_3$ for our example using labeling of Figure 5.10. Note the dotted arrows at vertices corresponding to Res-d^+ resolutions; they highlight the antecedents whose partial vertex-interpolants are ignored and their sub-trees do not contribute [65]

$$\begin{tikzcd}
\overset{\text{Hyp-}B_{\overline{\pi}}}{\langle {l_1}^{\mathrm{b}} \vee {l_3}^{\mathrm{b}} \rangle} \arrow[d, "\text{Res-}ab"'] & \overset{\text{Hyp-}A_{\overline{\pi}}}{\langle {\bar{l}_3}^{\mathrm{a}} \vee {l_6}^{\mathrm{a}} \rangle} \arrow[dl] & & & \overset{\text{Hyp-}A_{\overline{\pi}}}{\langle {\bar{l}_1}^{\mathrm{a}} \vee {l_5}^{\mathrm{a}} \rangle} \arrow[dr] & \overset{\text{Hyp-}B_{\overline{\pi}}}{\langle {\bar{l}_4}^{\mathrm{b}} \vee {\bar{l}_5}^{\mathrm{b}} \rangle} \arrow[d, "\text{Res-}ab"] \\
\langle {l_1}^{\mathrm{b}} \vee {l_6}^{\mathrm{a}} \rangle \arrow[ddrr] & \overset{\text{Hyp-}A_{\overline{\pi}}}{\langle {l_1}^{\mathrm{a}} \vee {l_2}^{\mathrm{a}} \rangle} \arrow[dr] & \overset{\text{Hyp-}B_{\overline{\pi}}}{\langle {\bar{l}_2}^{\mathrm{b}} \vee {\bar{l}_6}^{\mathrm{b}} \rangle} \arrow[d, "\text{Res-}ab"] & & \overset{\text{Hyp-}B_{\overline{\pi}}}{\langle {\bar{l}_2}^{\mathrm{b}} \vee {l_4}^{\mathrm{b}} \rangle} \arrow[dr] & \langle {\bar{l}_1}^{\mathrm{a}} \vee {\bar{l}_4}^{\mathrm{b}} \rangle \arrow[d, "\text{Res-}b"] \\
& & \langle {l_1}^{\mathrm{a}} \vee {\bar{l}_6}^{\mathrm{b}} \rangle \arrow[d, "\text{Res-}ab"] & \overset{\text{Hyp-}B_{\overline{\pi}}}{\langle {\bar{l}_1}^{\mathrm{b}} \vee {l_2}^{\mathrm{b}} \rangle} \arrow[drr] & & \langle {\bar{l}_1}^{\mathrm{a}} \vee {\bar{l}_2}^{\mathrm{b}} \rangle \arrow[d, "\text{Res-}b"] \\
& & \langle {l_1}^{\mathrm{ab}} \rangle \arrow[dr, "\text{Res-}ab"] & & & \langle {\bar{l}_1}^{\mathrm{ab}} \rangle \arrow[dll] \\
& & & \bot & &
\end{tikzcd}$$

Figure 5.11: Rules applied at proof vertices if minimal labeling of Figure 5.9 is used.

to final PVA interpolant. Also note that the PVA interpolant I_π is smaller compared to both $I_1[\pi]$ and $I_2[\pi]$ from examples above.

Assignment applied onto (interpolant) formula (i.e., if $I[\pi]$ is computed) can reduce the size of the formula only if the assigned variable appears in the formula (i.e., the variable has to be shared). However, LPAIS reduce the size of the interpolants even if the assigned variable does not appear in the interpolant, since the reduction is done as a part of interpolant computation and not as a post-processing step.

5.4.1 Correctness

$$\begin{tikzcd}
{[\top]} \arrow[d] & {[\bar{l}_3]} \arrow[dl] & & & {[\bar{l}_1 \vee l_5]} \arrow[dr] & {[\top]} \arrow[d] \\
\wedge \arrow[ddrr] & {[l_1]} \arrow[dr] & {[\top]} \arrow[d, dotted] & & {[\top]} \arrow[dr] & \wedge \arrow[d] \\
& & {[I_4]} \arrow[d] & & {[\top]} \arrow[dr] & \wedge \arrow[d, dotted] \\
& & \vee \arrow[dr] & & & {[I_3]} \arrow[dll] \\
& & & \wedge & &
\end{tikzcd}$$

$I_3 \equiv \top$
$I_4 \equiv l_1$

Figure 5.12: PVA interpolant $I_\pi \equiv l_1 \vee \bar{l}_3$, using labeling of Figure 5.10.

$$\begin{tikzcd}[row sep=small]
b \arrow[d, dash] \\
ab = d^{+} \arrow[d, dash] \\
a \arrow[d, dash] \\
\bot
\end{tikzcd}$$

Figure 5.13: Strength ordering ($\preceq$)

Theorem 2 (Correctness) *Let R be a refutation of $A \wedge B$, π be a PVA, and* Lab *be a locality preserving labeling function. Then,* $\mathsf{LpaItp}(\mathsf{Lab}, R, A, B, \pi)$ *generates a partial variable assignment interpolant at the sink vertex s.*

Proof sketch. For clarity of presentation highlight the overall idea of the proof here. The complete proof can be found in the appendix of the thesis.

By structural induction over R we show that for each vertex v of a resolution proof the following invariants hold:

$$\pi \models A \wedge \neg\langle\Theta\rangle\restriction_{a,v,\mathsf{Lab}} \Rightarrow I_v$$
$$\pi \models B \wedge \neg\langle\Theta\rangle\restriction_{b,v,\mathsf{Lab}} \Rightarrow \neg I_v$$

I_v is the partial vertex-interpolant and $\langle\Theta\rangle$ is a vertex-clause of v ($cl(v) = \langle\Theta\rangle$). These invariants yield the PVAI constraints (D9.1, D9.2) at the sink vertex, where $\neg\langle\Theta\rangle = \top$. $\square$

Symmetry. Notice that the locality constraints, as well as the way LPAISs compute interpolants, are *symmetric* in terms of presence formulas in the A_π and B_π sets of satisfied clauses. It reflects the fact that these clauses are not a part of the sub-problem under consideration, thus irrelevant for PVAI interpolants. Given a fixed π, the satisfied clauses can be moved freely between the A and B sets; both computed interpolants and locality of the labeling functions are not affected if satisfied clauses are moved. This fact allows us to articulate the *strength* theorem in an elegant way.

5.4.2 Interpolant strength

Interpolation systems based on labeling provide some freedom in the choice of labels (e.g., for shared variables); this choice affects the resulting interpolants, in particular their logical strength. In the following, we investigate this relationship in more detail.

Definition 15 (Strength order) *Let $\preceq$ be a pre-order relation defined over set of labels $S = \{\bot, a, b, ab, d^+\}$ as: $b \preceq ab = d^+ \preceq a \preceq \bot$ (Figure 5.13). Let* Lab *and* Lab' *be labeling functions for a refutation R. We say* Lab is stronger than Lab', *denoted as* $\mathsf{Lab} \preceq \mathsf{Lab}'$, *if for all vertices $v \in V$ and for all literals $l \in cl(v)$ it holds that* $\mathsf{Lab}(v, l) \preceq \mathsf{Lab}'(v, l)$.

Note that labels ab and d^+ are of the same strength and can be exchanged if the locality requirements permit it; b is the strongest label, while a is the weakest one a literal can get. The following theorem states that the introduced strength order on labeling functions also induces ordering of the produced interpolants by logical strength.

Theorem 3 (Interpolant strength) *Let R be a refutation of $A \wedge B$, π and π' be partial variable assignments, and* Lab *and* Lab' *be corresponding locality preserving labeling functions. Let I be a partial variable assignment interpolant for* $\mathsf{LpaItp}(\mathsf{Lab}, R, A, B, \pi)$ *and I' be a PVA interpolant for* $\mathsf{LpaItp}(\mathsf{Lab}', R, A, B, \pi')$.

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ *then* $\pi, \pi' \models I \Rightarrow I'$.

Note that if π and π' are *empty* assignments, we obtain exactly the theorem on interpolant strength from [41]. Also note that the theorem permits different variable assignments for the interpolants. Thus, it relates the interpolants generated for different sub-problems (e.g., interpolants considering different sets of paths through a given ARG node). Since both π and π' are assumptions of the formula $I \Rightarrow I'$, the theorem applies to cases common to both sub-problems (i.e., to the shared paths). Both interpolants (I and I') have to be computed using the same A, B partitioning, thus interpolants for different ARG nodes cannot be compared using this theorem; we present a generalization in this direction later.

Weakened-labels filter. To be able to relate interpolants computed using different labeling functions (as in the above Theorem), we need to introduce a new type of filter, which preserves the literals whose label is weaker in Lab' than in Lab. Let Lab and Lab' be labeling functions. Let $v \in V$ be a vertex, $\langle\Theta\rangle$ be a clause and $C_1, C_2 \subseteq L$ be sets of labels. The *label change filter* $||$ is defined as follows:

$$\langle\Theta\rangle||^{\mathsf{Lab},\mathsf{Lab}'}_{v,C_1 \Rightarrow C_2} = \{l \in \Theta \mid \mathsf{Lab}(v, l) \in C_1 \text{ and } \mathsf{Lab}'(v, l) \in C_2\}$$

For a preserved literal, set C_1 specifies permitted labels for Lab and set C_2 specifies permitted labels for labeling function Lab'.

We define *weakened-labels filter* $\Downarrow_{v}^{\mathsf{Lab},\mathsf{Lab}}$ as follows:

$$\Downarrow_{v}^{\mathsf{Lab},\mathsf{Lab}'} = ||_{v,\{b,ab,d^+\}\Rightarrow\{ab,d^+,a\}}^{\mathsf{Lab},\mathsf{Lab}'}$$

It preserves all the literals whose label is weaker in the primed labeling function according to the strength ordering $\preceq$. Note that from technical reasons the weakened-labels filter also preserves some equally strong literals, i.e., those labeled ab or d^+ by both labeling functions. E.g., the filter preserves a literal l if the strongest labels b (i.e., $\mathsf{Lab}(v, l) = b$) is weakened into label a or ab in $\mathsf{Lab}'(v, l)$, while it filters-out a literal if both functions assign label a to it. The vertex resp. labeling functions are omitted if clear from the context.

In [68] we prove above theorem using the invariant shown in the proof sketch below. In this thesis, we choose a different approach; we only show the main idea of the Theorem 3 proof. In next section, we show stronger Theorem 4. Theorem 3 directly follows from Theorem 4 (using empty set of clauses S).

Proof sketch (Theorem 3). By structural induction over R, we show that for each vertex v of the resolution proof, the following invariant holds:

$$\pi, \pi' \models I_v \wedge \neg\langle\Theta\rangle\Downarrow_v \Rightarrow I'_v$$

$\langle\Theta\rangle$ is the vertex-clause of v (i.e., $cl(v) = \langle\Theta\rangle$), I_v and I'_v are the partial vertex-interpolants for the vertex v as generated by LPAIS using the labeling functions Lab and Lab', respectively. In the proof, we show that the invariant holds for all possible combinations of the rules that can be used to define the partial vertex-interpolants I_v and I'_v. $\square$

As in LISs, for a fixed variable assignment there is a lattice of LPAISs ordered according to the strength of labeling functions. The top element of the lattice involves the strongest labeling function, which assigns label b to $A_{\overline{\pi}}B_{\overline{\pi}}$-shared and $A_{\overline{\pi}}B_{\overline{\pi}}$-clean variables, while the labeling function of the bottom element assigns label a to them. Theorem 3 claims that LPAISs produce interpolants ordered by strength according to the lattice.

5.4.3 Path interpolation property

Several verification approaches such as [3, 80, 107] depend on the *path interpolation* property (PI). In [96], the authors show that LISs can be employed to generate path interpolants by providing a sequence of labeling functions that are decreasing in terms of strength. In this subsection, we generalize this property for LPAIS. We study conditions that the labeling functions have to satisfy in order to obtain a sequence of interpolants with the PI property.

The following theorem states the main result:

Theorem 4 (PI property) *Let* Lab *and* Lab' *be locality preserving labeling functions, let* R *be a refutation of* $A \land S \land B$*, and* π *and* π' *be PVAs. Let* $I = \mathsf{LpaItp}(\mathsf{Lab}, R, A, S \cup B, \pi)$ *and* $I' = \mathsf{LpaItp}(\mathsf{Lab}', R, A \cup S, B, \pi')$.

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ *then* $\pi, \pi' \models I \land S \Rightarrow I'$.

Proof sketch (Theorem 4). By structural induction over refutation R we show that for each vertex $v \in V$ of the refutation, the following invariant holds:

$$\pi, \pi' \models I_v \land S \land \neg\langle\Theta\rangle|\downarrow_v \Rightarrow I'_v$$

where $cl(v) = \langle\Theta\rangle$ is the vertex clause and I_v and I'_v are the partial vertex-interpolants for vertex v as generated by LPAIS using labeling functions Lab and Lab', respectively. In the proof, we show that the invariant holds for all possible combinations of the rules that can be used to define partial vertex-interpolants I_v and I'_v.

The full proof can be found in the appendix of the thesis. □

A tentative reader may ask whether it exists locality preserving labeling satisfying the requirements of the above theorem. The answer to this question can be found in our paper [68] where we stated and proved weaker version of the above theorem; the proof of the theorem is based construction a strongest possible labeling Lab' from Lab that satisfies requirements of the theorem; i.e., is a locality preserving and weaker then Lab.

5.5 Evaluation

We implemented LPAIS in a tool called *Partial Variable Assignment InterpolatoR* (PVAIR). It is built on top of the open-source tool PeRIPLO [93] which provides resolution proofs and is able to optimize the proofs for interpolation through transformations. PeRIPLO has been used in various verification projects, including function summarization in eVolCheck [44] and FunFrog [100], both as an interpolation engine and as a SAT solver.

[65] The PVAIR architecture is shown in Figure 5.14. It takes a propositional formula Φ, its (A, B)-partitioning, and a partial variable assignment π as input and produces PVA interpolant if the input formula is unsatisfiable.

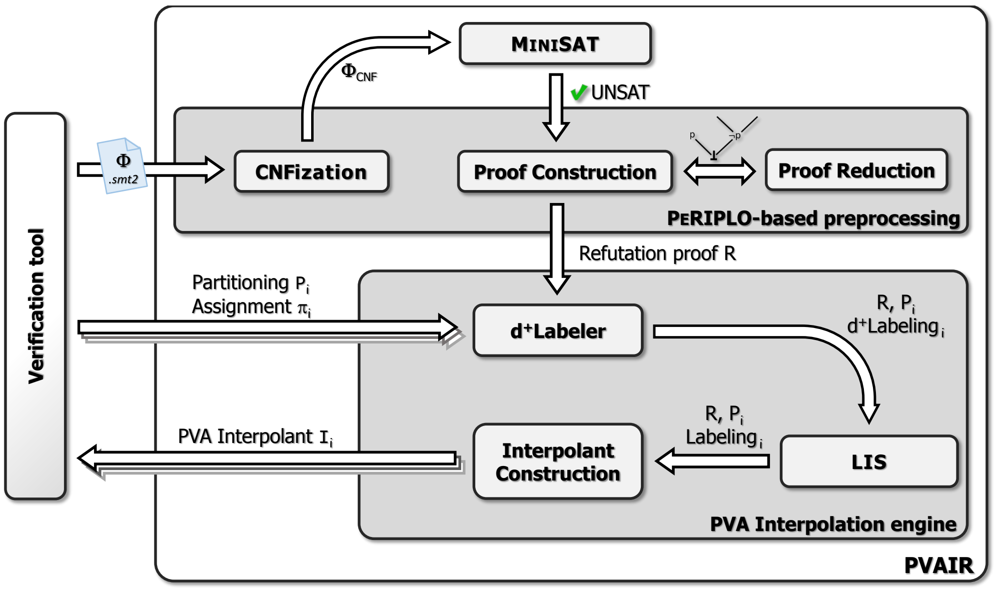


Figure 5.14: PVAIR architecture.

The input can be provided either in a file in the SMT-LIB 2.0 [9] format or via a C++ API.

The workflow of the PVAIR tool is as follows. First, the input formula is passed to the PeRIPLO-based preprocessing module. Since the formula can be in arbitrary form, it is transformed into CNF (the top box in Figure 5.14) using an efficient, structure-sharing version of the Tseitin encoding [103]. Its satisfiability is then determined using the MiniSAT 2.2.0 solver [43].

In the case of an unsatisfiable input, an initial refutation is extracted from the solver in the compact MiniSAT internal proof format. The format is then transformed into a resolution DAG to allow more efficient handling of the proof (ProofConstruction). In particular, using the resolution DAG form, the proof can be compressed using well-known proof reduction techniques such as structural hashing or pivot recycling [95, 92] available in PeRIPLO (ProofReduction). The proof reduction techniques can be enabled/disabled via a configuration file or API.

Once the resolution proof R is computed, it is passed together with the partitionings and variable assignments to the interpolation engine (the bottom box in Figure 5.14). From this point on, any number of partial variable assignments π_i and partitionings P_i (into $A_i \wedge B_i$) can be given as input to the tool and used to construct the corresponding interpolants I_i. Note that in any case only one SAT-solver call will be made during the entire execution. Then the proof is labeled; the labels are assigned to literals in the proof based on the partitioning and the assignment and selected LIS-based interpolation algorithm (which can be chosen in the configuration file or via API). When the labeling is complete, it is used together with the partitioning and resolution proof R to compute interpolants (InterpolantConstruction).

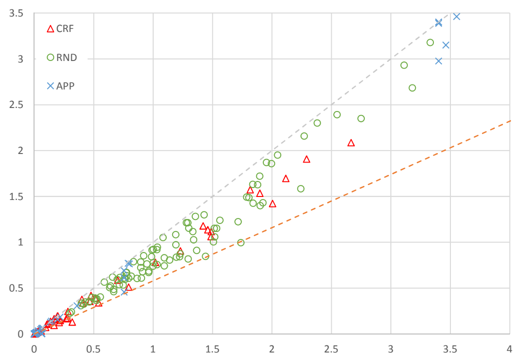
The construction starts by computing partial vertex-interpolants (according to the upper part of Table 5.1) for the leaf nodes of the refutation. The computation then proceeds from the leaves to the root node. During the interpolant construction, partial interpolants are optimized using Boolean constant propagation and structural sharing (hashing). The final interpolant is computed in the root node.

In symbolic model checking Craig interpolants are used as a tool to compute sets of program states with required properties; their usage varies a lot among verification techniques. Thus, we decided to choose unsatisfiable benchmarks from SAT Competition [97]. They provide us with large and heterogeneous kinds of benchmarks. An alternative approach would be integration into a particular verification technique; on one hand, this would get results tightly related to verification, on the other hand the results would be more influenced by the chosen technique and the way assignment is generated. Such an evaluation would also require deep analysis of the technique in order to introduce assignments preserving the safety of the approach.

We also employed the PVAIR tool in software verification process; we applied it on computational problems generated by the eVolCheck tool during verification procedure. To demonstrate the tool performance, we measured the size of produced interpolants and its effect on the total verification time. These experiments can be found in [65].

SAT competition. We used 47 unsatisfiable benchmarks from the SAT Competition from all categories – 12 from the *Application* (APP), 11 from the *Crafted* (CRF), and 24 from the *Random* (RND) sets. Since the benchmarks are not partitioned, we generated six partitionings for each benchmark; we simulated the typical way the path interpolants are computed, i.e., we randomly choose n, first n clauses of the benchmark belong to the A part, the remaining clauses to the B part. No partition is empty. No assignment is given by authors of the benchmarks, thus for each partitioning, we generate five random variable assignments consisting of a single, five, resp. twenty assigned variables. Assignments of various sizes indicate how the reduction scales w.r.t. the number of assigned variables.

For comparison, we use McMillan's interpolants – a widely used approach. Experiments were run on a Linux server with Intel Xeon X5687 CPU using the timeout of 60 minutes and the memory limit of 20GB using the GNU



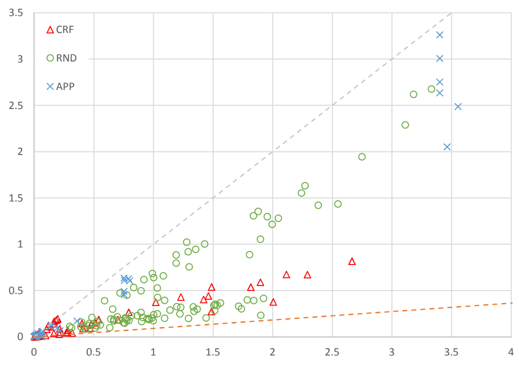


Table 5.2: Comparison of interpolant sizes computed without variable assignment [x] and with one variable assigned [y] (left) and five variables assigned (right).

Parallel environment [101]. The proof reduction techniques were disabled; we used the default PeRIPLO settings.

Table 5.2 compares the sizes of the computed interpolants. Each point in the graph corresponds to a single partitioning of a benchmark; the x-axis represents the interpolant size if no assignment is provided (Craig interpolant), while the y-axis represents the size of the PVA interpolant with a single (resp. five) assigned variable(s). For presentation clarity, the y-axis is the average size of all five random assignments generated for a given partitioning. The values on axes represent millions of nodes if an interpolant is represented as DAG (counting literals and Boolean operators). The orange dashed line shows the average size of Craig interpolants for sub-problems; i.e., if all satisfied clauses and falsified literal are removed from input formula and then SAT solver is called on this simplified input and standard Craig interpolant are computed from new refutation proof. This illustrates what price is paid by PVA interpolants for the path interpolation property and a single SAT solver call. All graphs show interesting reduction in the size for PVA interpolants as well as substantially larger reduction in case of five assigned variables. In all graphs, the same partition of the same benchmark shares the same x-value, thus it is possible, especially for the larger ones, to compare their reductions.

Table 5.3 summarizes the results shown in the graphs, reporting precise numbers. The first table compares the sizes of PVA interpolants to Craig interpolants. The *No assignment* row shows the average size of Craig interpolants for a given benchmark type. The remaining rows show the relative sizes of PVA interpolants w.r.t. the *No assignment* row. The application benchmarks exhibit a smaller reduction compared to the other types, and

PVA Itp.	APP	RND	CRF	All
No Assignment	344 298.7	1 308 750.1	489 469.1	776 573.9
1 var	92.8 %	83.0 %	78.1 %	83.7 %
5 vars	76.2 %	45.2 %	31.5 %	47.6 %
20 vars	48.3 %	10.1 %	4.8 %	15.0 %

Itp. from sub-prob.	APP	RND	CRF	All
No Assignment	344 298.7	1 308 750.1	489 469.1	776 573.9
1 var	69.5 %	55.0 %	65.5 %	58.8 %
5 vars	24.4 %	5.7 %	9.7 %	9.1 %
20 vars	0.12 %	0.01%	0.39%	0.09%

Table 5.3: Average interpolant sizes by category and number of assigned variables.

even for twenty assigned variables, the interpolants are half in the size of the Craig interpolants.

As we have mentioned, a PVA interpolants can be seen as Craig interpolants for the corresponding sub-problem. The second table compares the sizes of Craig interpolants to Craig interpolants for the corresponding sub-problem. For each pair of partitioning and assignment, we created the sub-problem instance and used PVAIR to compute the standard Craig interpolant for it without assignment; in this case the assignment is used only during construction of the sub-problem. Sub-problems are simpler compared to the benchmark from which they were generated; the satisfied clauses and falsified literals are removed, so interpolants for sub-problems are typically smaller compared to Craig interpolants of the benchmark. However, the interpolant for each sub-problem is computed from a different refutation; in contrast to PVA interpolants which, for a particular benchmark, are all computed from the same proof. This means that the sequence of interpolants for sub-problems may not have the path interpolation property [107].

As to the interpretation of the results: The *No assignment* reflects the state-of-the-art approaches, where Craig interpolants are used directly. PVA interpolants (the first table) show how the size of the interpolants can be reduced if the model checker (i.e., a tool generation the input) provides a reasonable assignment together with a partitioning. The interpolants for a sub-problem (the second table) can be seen as an alternative to PVAinterpolants because of their similar meaning, however these interpolants lack the properties of the PVA ones.

Time and memory demands are crucial properties of each interpolation tool. The reduction in overall running time and required memory roughly correspond to the reduction of interpolant sizes; e.g., on average PVAIR is 11% faster and requires 9% less memory if a single variable is assigned. The time and memory savings occur during the interpolant computation phase due to smaller interpolants being handled.

6 Conclusion

In the thesis, we presented techniques to eliminate irrelevant information from program state representations. These techniques reduce the size of the representation used by the verification tools. First, we sum-up our contribution related to explicit representation of program states. Later, we focus on symbolic representation and our interpolation method, which can be used to compute smaller interpolants.

In chapter 4, we focused on goals G1a and G1b. We have introduced two novel dead variable analyses – hybrid dead variable analysis and the dynamic one. Both analyses focus on heap of multi-threaded programs. The hybrid DVA is fast; it can be combined with various heuristic search strategies. On the other hand, it does not distinguish among different heap instances, which negatively influence its precision. The dynamic DVA requires a complete depth-first-search to work as it observes the reads and writes of the program to identify dead variables. The analysis aims at precise detection of dead variables; it distinguishes among object instances. We formally described our DVA over labeled transition systems and proved its soundness in Theorem 1.

Based on the above analyses, we designed two dead variable reduction techniques. We created a custom state matching technique to preserve the safety of the dynamic DVR. The DVR takes into account a set of live addresses together with live values. The state matching is based on Lemma 2. In case of the hybrid DVR, we could use classical state matching.

We developed a prototype implementation in Java PathFinder [106] and evaluated it on set of benchmarks. In the case of hybrid DVR, the state vector size is reduced to one third on average, while in case of dynamic DVR it is an order of magnitude smaller. This leads to reduction of 12% in the verification time in case of hybrid DVR, while using Dynamic DVR results in a 20% speed-up and a reduction of 30% in the number of states on average.

In chapter 5, we described a novel interpolation technique. First, we introduced a notion of Partial Variable Assignment Interpolants, which use variable assignment to specify a sub-problem. The sub-problem defines the important parts of the input formula, while remaining parts of the input formula can be ignored during interpolant computation.

Later, we introduced a technique to compute PVA interpolants for predicate logic – Labeled Partial Assignment Interpolation System – an extension of Labeled Interpolation System [41]. LPAIS computes interpolants from a refutation proof, while using assignment to omit unimportant parts of the proof. This way the size of the interpolant is reduced.

In Theorem 4, we showed that LPAIS yields interpolants with the path interpolation property – a property especially useful for verification.

We created tool PVAIR, which implements LPAIS, and evaluated it on a set of benchmarks from the SAT Competition. For assignments with 5 variables, the interpolant size was reduced by one half on average.

6.1 Future work

The most obvious future work in case of explicit state representation is a combination of our heap analysis with DVR for local variables. This should further reduce the size of state spaces.

In case of dynamic DVR, we observed that the run-time is more than twice as long compared to standard JPF in the CLIF benchmark despite nearly 40% reduction of the program state. This is caused by the state matching, which have to try large number of possible sets of live addresses. In the future, we would like to introduce a new heuristic selecting a subset of live addresses to be tried. It will result into larger state space; however, the model checker will explore the state space faster.

We also envision a possibility beyond the dead variable analysis; it lies in introducing a property-driven definition of important (i.e., live) values that should be considered during state matching. The dead variable analysis (from its definition) considers all the reads of a program, but not all reads can influence the properties being verified. If the value is not used in an expression which can (indirectly) influence (i) validity of an assertion and (ii) conditions in the program, its value is unimportant for the verification.

The dynamic analysis should be able to track usage of the values in a similar way as our dynamic analysis tracks their reads.

In case of symbolic representation, we plan to focus on proving other interpolations properties, especially the tree-interpolation property. That would permit to use PVAI in more verification tools, for example, eVolCheck.

We would like to extend our interpolation method to support various theories in first order logic, especially linear arithmetic, which is commonly used in verification. However, that would require using a different approach; the labeled interpolation system, a base upon which our method is built, works only for the propositional logic.

A Proofs

A.1 Dead variable reduction

Lemma 1 (Trace-equivalent paths) Let $S_1 \equiv (IP_{S_1}, Mem_{S_1}) \in \mathcal{S}$ and $S'_1 \equiv (IP_{S'_1}, Mem_{S'_1}) \in \mathcal{S}$ be two states such that $reduce\_state_{S_1}(S_1) = reduce\_state_{S_1}(S'_1)$. [67]

Then for any trace $p = S_1 \xrightarrow{A_1,G_1} S_2 \cdots \cdots S_i \xrightarrow{A_i,G_i} S_{i+1} \cdots$ there exists a transition-equivalent trace $p' = S'_1 \xrightarrow{A_1,G_1} S'_2 \cdots$ such that $IP_{S_i} = IP_{S'_i}$ for all i (i.e., the corresponding states along equivalent traces differ only in the memory content).

Proof (Lemma 1 – Trace-equivalent paths). he proof is constructive and inductive on the trace length. The induction will iteratively append an equivalent transition to the end of trace p' and preserve the following inductive invariant: $reduce\_state_{S_i}(S_i) = reduce\_state_{S_i}(S'_i)$ and $IP_{S_i} = IP_{S'_i}$ and there is a trace p'_i which is transition equivalent to the prefix of p of length i.

The base case (traces with no transitions): In such a case, trace p contains just one state: $p = S_1$ (i.e., the trace of length 0 starting at S_1). A transition-equivalent trace $p' = S'_1$ (a trace staring in S'_1 with no transition) exists trivially. Moreover $IP_{S_1} = IP_{S'_1}$, because $reduce\_state_{S_1}(S_1) = reduce\_state_{S_1}(S'_1)$ and the fact that $reduce\_state$ functions do not modify the instruction pointer (IP).

Inductive step: First, for the initial states S_1 and S'_1 the inductive invariant holds. Let the inductive invariant hold for a trace p of length i. If p is of length i, the inductive invariant is exactly the claim of the lemma. So, let us assume trace p to be longer. The inductive invariant gives us transition-equivalent trace $p'_i = S'_1 \longrightarrow \cdots S'_i$ such that $\mathit{reduce\_state}_{S_i}(S_i) = \mathit{reduce\_state}_{S_i}(S'_i)$ and $IP_{S_i} = IP_{S'_i}$. Moreover, there is a transition $p = \cdots S_i \xrightarrow{G_i,A_i} S_{i+1} \cdots$ (i.e., (S_i, G_i, A_i, S_{i+1})); we show that p'_i can be extended with this transition.

Let $S'_{i+1} = (IP_{S_{i+1}}, \mathit{Mem}_{S'_i}[A_i])$. First, we show that there is a transition $(S'_i, G_i, A_i, S'_{i+1}) \in \triangle$ that can be appended to p'_i to form p'_{i+1}. The first requirement of Def. 1 is satisfied since $IP_{S_i} = IP_{S'_i}$ (from the induction invariant) and $IP_{S_{i+1}} = IP_{S'_{i+1}}$ (from the definition of S'_{i+1}), thus the step showing the existence of the transition (S_i, G_i, A_i, S_{i+1}) satisfies the condition for transition $(S'_i, G_i, A_i, S'_{i+1})$ as well. The second condition (i.e., guard G_i holds in S'_i) is satisfied, because G_i holds in state S_i and S_i and S'_i equal on live addresses ($\mathit{reduce\_state}_{S_i}(S_i) = \mathit{reduce\_state}_{S_i}(S'_i)$ from the induction invariant). Due to trace $p_{G_i} = S_i \xrightarrow{G_i,A_i} S_{i+1}$ (i.e., a trace with a single transition), all the addresses on which G_i depends are live in S_i. The last condition is satisfied directly by the definition of S'_{i+1}.

Now we show that $\mathit{reduce\_state}_{S_{i+1}}(S_{i+1}) = \mathit{reduce\_state}_{S_{i+1}}(S'_{i+1})$ (i.e., the states S_{i+1} and S'_{i+1} equal on live addresses). From the definition of S'_{i+1} we know that $IP_{S_{i+1}} = IP_{S'_{i+1}}$. Let us look at the memory. In the following, we use $\mathit{Mem}_{R_{i+1}}$ to denote the memory of state $\mathit{reduce\_state}_{S_{i+1}}(S_{i+1})$ (which is exactly the memory of the reduced state for full state S_{i+1}) and $\mathit{Mem}_{R'_{i+1}}$ to denote the memory of state $\mathit{reduce\_state}_{S_{i+1}}(S'_{i+1})$ (i.e., the memory of states S'_{i+1} reduced w.r.t. future behaviour of S_{i+1}). We show that the above memory functions are the same (i.e., $\mathit{Mem}_{R_{i+1}} = \mathit{Mem}_{R'_{i+1}}$); in other words, these functions have (i) the same domain and (ii) they evaluate to the same values for any address from their domain. As to (i), because the function $\mathit{reduce\_state}_{S_{i+1}}$ is used to create both memory functions, their domains are the same (i.e., $\mathit{dom}(\mathit{Mem}_{R_{i+1}}) = \mathit{dom}(\mathit{Mem}_{R'_{i+1}})$).

Now we show (ii) i.e., $\forall a \in \mathit{dom}(\mathit{Mem}_{R_{i+1}})$ it holds that $\mathit{Mem}_{R_{i+1}}(a) = \mathit{Mem}_{R'_{i+1}}(a)$. Let us pick an address $a \in \mathit{dom}(\mathit{Mem}_{R_{i+1}})$; this address can be either assigned (written) by action A_i or not. In the former case the values equal (i.e., $\mathit{Mem}_{R_{i+1}}(a) = \mathit{Mem}_{R'_{i+1}}(a)$), because in both cases they are computed by the same action A_i using the same values; the trace $p_{G_i} = S_i \xrightarrow{G_i,A_i} S_{i+1}$ shows that all the addresses read by action A_i are live and thus preserved by $\mathit{reduce\_state}_{S_{i+1}}$ and the induction invariant gives us that Mem_{S_i} and $\mathit{Mem}_{S'_i}$ equal on these addresses.

Let us focus on the latter case, i.e., when address a is not written by action A_i and its value is taken from the predecessor. Formally, it holds that $\mathit{Mem}_{R_{i+1}}(a) = \mathit{Mem}_{S_i}(a)$ and $\mathit{Mem}_{R'_{i+1}}(a) = \mathit{Mem}_{S'_i}(a)$. Address a is preserved by the $\mathit{reduce\_state}_{S_{i+1}}$ function, thus either $a \in \mathit{addr}(\phi)$ or $a \in \mathit{live\_addr}(S_{i+1})$. In both cases address a is preserved by $\mathit{reduce\_state}_{S_i}$; in the first case, the reason comes from the definition of $\mathit{reduce\_state} - \mathit{addr}(\phi)$

addresses are always a part of reduced state. In the second case, the definition of $\mathit{reduce\_state}$ gives us a trace p'_a from state S_{i+1} which demonstrates that address a is live. Let trace p_a be p'_a prefixed by (S_i, G_i, A_i, S_{i+1}) (i.e. $p_a \equiv S_i \xrightarrow{G_i, A_i} p'_a$), because A_i does not write to address a, the trace p_a shows that the address a is live in S_i and thus preserved by $\mathit{reduce\_state}_{S_i}$. We know that $\mathit{Mem}_{R_{i+1}}(a) = \mathit{Mem}_{S_i}(a)$ and $\mathit{Mem}_{R'_{i+1}}(a) = \mathit{Mem}_{S'_i}(a)$, and the inductive invariant gives us that $\mathit{Mem}_{S_i}(a) = \mathit{Mem}_{S'_i}(a)$; thus, we have shown that $\mathit{Mem}_{R_{i+1}}(a) = \mathit{Mem}_{R'_{i+1}}(a)$.

Put together, we have proved that:

$\mathit{reduce\_state}_{S_{i+1}}(S_{i+1}) = \mathit{reduce\_state}_{S_{i+1}}(S'_{i+1})$ – the inductive invariant for $i+1$.

Theorem 1 (Bisimulation) For a program P and a property ϕ, transition system $A_P = (\mathcal{S}, \triangle, S^{init})$ and corresponding reduced transition system $R_P = (\mathcal{R}, \triangle_R, R^{init})$ are bisimilar.

Proof (Theorem 1 – Bisimulation). Let $\mathit{reduce\_state}$ be a relation among the full and reduced states, such that full state S and reduced state R are in the relation iff $\mathit{reduce\_state}_S(S) = R$. We show that $\mathit{reduce\_state}$ is a bisimulation, i.e., $\mathit{reduce\_state}$ satisfies the following three conditions:

(1) The initial states S^{init} and R^{init} are in the relation (which comes directly from the definition).

(2) Let there is a full transition $(S, G, A, S') \in \triangle$, and let state R be in the relation with S, i.e., $R = \mathit{reduce\_state}_S(S)$. Then there has to be a reduced transition $(R, G, A, R') \in \triangle_R$ such that $R' = \mathit{reduce\_state}_{S'}(S')$. This comes also trivially from the definition of the reduced state space.

(3) Let there is a reduced transition $(R, G, A, R') \in \triangle_R$ executed by a thread i. Then for any full state S such that $\mathit{reduce\_state}_S(S) = R$ (a pre-image of reduced state R), there exists a full transition $(S, G, A, S') \in \triangle$ such that $\mathit{reduce\_state}_{S'}(S') = R'$.

The only condition remaining to be proved is (3). From the definition of reduced state there exists a *b*ase full transition $(S_b, G, A, S'_b) \in \triangle$ due to which the reduced transition (R, G, A, R') exists. From the definition of $\mathit{reduce\_state}$ it follows that it does not modify the instruction pointer, thus $IP_S = IP_{S_b} = IP_R$ (i.e., the corresponding states S, S_b, and R point to the same program location) and $IP_{S'_b} = IP_{R'}$ (i.e., the base transition is executed by the same thread i as the reduced transition). Moreover the trace $p_b \equiv S_b \xrightarrow{G,A} S'_b$ (i.e., the trace of length 1 consisting exactly of the base transition) guarantees that all the addresses that guard G and action A read are live and thus preserved by $\mathit{reduce\_state}_{S_b}$. States S and S_b are both reduced to state R thus, first, $\mathit{reduce\_state}_{S_b} = \mathit{reduce\_state}_S$ (i.e., the

$reduce\_state$ functions for both states are the same and both states have the same live addresses) and second, the values at the live addresses are the same in both states (i.e., $\forall a \in dom(Mem_R) : Mem_S(a) = Mem_{S_b}(a)$).

Def. 1 gives us that (i) there is step $(l_i, A, G, l'_i) \in L_i$ of thread i such that $IP_{S_b}(i) = l_i$ and $IP_{S'_b} = IP_{S_b}[i := l'_i]$ and (ii) guard G holds in S_b.

Let $S' \equiv (IP_{S'_b}, Mem_S[A])$ be a full state, i.e., it is created from state S by a step of thread i into the same program location as the base (and reduced) transition by executing action A. In order to prove the theorem, we need to show that there is a transition $(S, G, A, S') \in \triangle$ and that S' reduces into R'.

According to Def. 1, there exists transition $(S, G, A, S') \in \triangle$, because $IP_S = IP_{S_b}$, $IP_{S'} = IP_{S'_b}$ so (i) satisfies the first requirement of Def. 1 and guard G holds in S (due to (ii) and because S_b and S equal on all the values that guard G reads). The last requirement of Def. 1 follows directly from the definition of state S'.

We now show that S' reduces to R' (i.e., $reduce\_state_{S'}(S') = R'$); first we show that (iii) states S' and S'_b have the same live addresses

(i.e., $reduce\_state_{S'} = reduce\_state_{S'_b}$, in other words the functions are the same) and then that (iv) these states have the same values at the live addresses.

Before moving to (iii), we first show that (v) S' and S'_b have (pair-wise) transition-equivalent sets of traces (using Lemma 1). Let $p_{S'}$ be a trace from state S' (i.e., $p_{S'} = S' \xrightarrow{*} \cdots$). Then there is also a trace p which is formed by $p_{S'}$ prefixed by transition (S, G, A, S') (i.e., $p \equiv S \xrightarrow{G,A} S' \xrightarrow{*} \cdots$). Then $reduce\_state_S(S) = reduce\_state_S(S_b) = R$, because from assumption of the condition it holds that $reduce\_state_S(S) = R$, and $reduce\_state_S = reduce\_state_{S_b}$, which was shown above, and $reduce\_state_{S_b}(S_b) = R$ (from definition of S_b). Lemma 1 can now be applied on states S and S_b and trace p resulting in a transition-equivalent trace $p' = S_b \xrightarrow{G,A} S_x \xrightarrow{*} \cdots$ such that $IP_x = IP_{S'}$. Now we show that $S_x = S'_b$; the actions are deterministic, so $Mem_x = Mem_{S'_b}$ (more rigorously by the definitions, both equal to $Mem_{S_b}[A]$). Since $IP_x = IP_{S'_b}$ (via $IP_{S'}$), we have shown that the states S_x and S'_b are the same (i.e., $S_x = S_b$). Thus, we can remove the first transition of p' to obtain $p_{S'_b}$ from S'_b, which is transition equivalent to $p_{S'}$. Exactly the same reasoning can be used to show that for any trace from S'_b there is a transition-equivalent trace from S'.

As to (iii), it remains to show that the function $reduce\_state_{S'}$ and the fucntion $reduce\_state_{S'_b}$ are the same. From the definition of the $reduce\_state$ functions it follows that they modify only the Mem part of the program state by restricting its domain. Assume that $reduce\_state_{S'}$ preserves an address $a \in dom(Mem_{S'})$; then it holds either $a \in addr(\phi)$ or $a \in live\_addr(S')$. In the former case, the address a is also preserved by $reduce\_state_{S'_b}$ from the same reason (the safety property ϕ is the same for the whole program). In the latter case, we use the definition of $live\_addr$, which gives us a trace $p_{S'}$ – a witness why a is a live address (i.e., $a \in tr\_live\_addr(p_{S'})$). Above we have shown that there is a trace $p_{S_b'}$ transition equivalent to $p_{S'}$ (see (v)). Trace $p_{S_b'}$ (having the same live addresses as $p_{S'}$, since the traces equal on tr_live_addr in their initial states) is a witness that address a is live in S_b' (i.e., $a \in LA(S_b')$). Thus a is also preserved by $reduce\_state_{S_b'}$. The same reasoning can be applied to show that all addresses preserved by $reduce\_state_{S_b'}$ are also preserved by $reduce\_state_{S'}$, which together gives us that $reduce\_state_{S'} = reduce\_state_{S_b'}$.

The only remaining part is (iv). $IP_{S'}$ and $IP_{R'}$ equal, because we know that $IP_{R'} = IP_{S_b'}$ and $IP_{S'} = IP_{S_b'}$ from the definition. Let us focus on the memory parts; we show that for any address a preserved by $reduce\_state_{S'}$, the states S' and S_b' have the same value (i.e., $Mem_{S'}(a) = Mem_{S_b'}(a)$).

Address a can be either written by action A (taken from the transition (S, G, A, S')) or not. In the former case, the value written by A is the same in $Mem_{S'}$ and $Mem_{S_b'}$, because the outcome of the actions is deterministic and the values used by action A to compute its result are the same (S and S_b both reduce into R thus equal on live addresses and due to the single transition trace $\xrightarrow{G,A}$, the addresses read by A are live in these states). Let us focus on the latter case in which address a is not modified by A and thus the values (in states S' and S_b') are the same as in their predecessor (in states S and S_b respectively); formally $Mem_{S'}(a) = Mem_S(a)$ and $Mem_{S_b'}(a) = Mem_{S_b}(a)$. The predecessors have the same value at this address (i.e., $Mem_S(a) = Mem_{S_b}(a)$); because S and S_b both reduce into R and thus equal on live addresses and a is live in both S and S_b. Address a is live in S, because it is live address in S'; so there exists a trace p' – witness that the address is live. The trace $p = S \xrightarrow{G,A} p'$ (i.e., prefixed by transition (S, G, A, S')) shows that the address is also live in S as we need.

We have shown (iii) and thus $reduce\_state(S') = R$. It means we have created state S' such that there exists a transition (S, G, A, S') and moreover $reduce\_state(S') = R$. Thus, we have shown (3). This proves the bisimulation theorem.

A.2 LPAIS

A.2.1 Correctness

Theorem 2 (Correctness) Let R be a refutation of $A \wedge B$, π be a PVA, and

Lab be a locality preserving labeling function. Then, $\mathsf{LpaItp}(\mathsf{Lab}, R, A, B, \pi)$ generates a partial variable assignment interpolant at the sink vertex s.

Proof (Theorem 2 – Correctness). In the proof, we follow the proof idea of LIS. By structural induction, we show that for each vertex v of a resolution proof the following invariants hold:

(T2.Inv1) $\pi \models A \wedge \neg\langle\Theta\rangle\!\upharpoonright_{a,v,\mathsf{Lab}} \Rightarrow I$

(T2.Inv2) $\pi \models B \wedge \neg\langle\Theta\rangle\!\upharpoonright_{b,v,\mathsf{Lab}} \Rightarrow \neg I$

(T2.Inv3) $\mathsf{Var}(I) \subseteq \mathsf{Var}(A_{\overline{\pi}}) \cap \mathsf{Var}(B_{\overline{\pi}})$

where I is the partial interpolant of vertex v and $cl(v) = \langle\Theta\rangle$.

These invariants are equivalent to the PVAI constraints for the sink node (where the $\neg\langle\Theta\rangle = \top$). We omit the labeling function Lab from subscripts (since it is unique in the proof) and the vertex if clear.

Base cases. The base cases apply to the leaf vertices of the proof where the hypotheses operations are applied.

Hyp-$A_{\overline{\pi}}$: $\langle\Theta\rangle \in A_{\overline{\pi}}$ so $I = \langle\Theta\rangle[\pi]\!\rrbracket_b$

(T2.Inv1) $\pi \models A \wedge \neg\langle\Theta\rangle\!\upharpoonright_a \Rightarrow \langle\Theta\rangle[\pi]\!\rrbracket_b$ holds because $A \Rightarrow \langle\Theta\rangle$ and $\langle\Theta\rangle \Leftrightarrow (\langle\Theta\rangle\!\upharpoonright_a \vee \langle\Theta\rangle\!\rrbracket_b)$, so $\langle\Theta\rangle \wedge \neg\langle\Theta\rangle\!\upharpoonright_a \Rightarrow \langle\Theta\rangle\!\rrbracket_b$. Moreover, it holds that $\pi \models \langle\Theta\rangle\!\rrbracket_b \Leftrightarrow \langle\Theta\rangle[\pi]\!\rrbracket_b$ because the clause $\langle\Theta\rangle$ (thus even $\langle\Theta\rangle\!\rrbracket_b$) is not satisfied by the partial assignment π, so all the assigned literals (i.e., those removed by the filter $[\pi]$) evaluate to $\bot$.

(T2.Inv2) $\pi \models B \wedge \neg\langle\Theta\rangle\!\upharpoonright_b \Rightarrow \neg\langle\Theta\rangle[\pi]\!\rrbracket_b$ holds because $\neg\langle\Theta\rangle\!\upharpoonright_b \Rightarrow \neg\langle\Theta\rangle\!\rrbracket_b$. Moreover, it holds that $\pi \models \neg\langle\Theta\rangle\!\rrbracket_b \Leftrightarrow \langle\Theta\rangle[\pi]\!\rrbracket_b$; the reason is the same as above, all the assigned literals evaluate to $\bot$.

(T2.Inv3) $\mathsf{Var}(\langle\Theta\rangle[\pi]\!\rrbracket_b) \subseteq \mathsf{Var}(A_{\overline{\pi}}) \cap \mathsf{Var}(B_{\overline{\pi}})$. Label b implies that such variables are $A_{\overline{\pi}}B_{\overline{\pi}}$-shared. Otherwise, the locality-preserving requirement D13.2 is violated. Moreover, the assignment filter is applied, so the partial vertex-interpolant does not contain any assigned variable.

Hyp-$B_{\overline{\pi}}$: $\langle\Theta\rangle \in B_{\overline{\pi}}$ so $I = \neg\langle\Theta\rangle[\pi]\!\rrbracket_a$. The situation is symmetric to Hyp-$A_{\overline{\pi}}$ case.

(T2.Inv1) $\pi \models A \wedge \neg\langle\Theta\rangle\!\upharpoonright_a \Rightarrow \neg\langle\Theta\rangle\!\rrbracket_a$ holds because $\neg\langle\Theta\rangle\!\upharpoonright_a \Rightarrow \neg\langle\Theta\rangle\!\rrbracket_a$. Moreover, $\pi \models \langle\Theta\rangle\!\rrbracket_a \Leftrightarrow \langle\Theta\rangle[\pi]\!\rrbracket_a$, because all the assigned literals in the clause $\langle\Theta\rangle$ evaluate to $\bot$ under the assignment π.

(T2.Inv2) $\pi \models B \wedge \neg\langle\Theta\rangle\lceil_b \Rightarrow \langle\Theta\rangle\rfloor_a$ holds because $B \Rightarrow \langle\Theta\rangle$ and $\langle\Theta\rangle \Leftrightarrow (\langle\Theta\rangle\lceil_b \vee \langle\Theta\rangle\rfloor_a)$ so $\langle\Theta\rangle \wedge \neg\langle\Theta\rangle\lceil_b \Rightarrow \langle\Theta\rangle\rfloor_a$. Moreover, as shown above, $\pi \models \langle\Theta\rangle\rfloor_a \Leftrightarrow \langle\Theta\rangle[\pi]\rfloor_a$.

(T2.Inv3) $\mathsf{Var}(\neg\langle\Theta\rangle[\pi]\rfloor_a) \subseteq \mathsf{Var}(A_{\overline{\pi}}) \cap \mathsf{Var}(B_{\overline{\pi}})$. The label a implies that these variables are $A_{\overline{\pi}}B_{\overline{\pi}}$-shared. Otherwise, the locality preserving requirements D13.3 is violated. Moreover, the assignment filter is applied, so the partial vertex-interpolant does not contain any assigned variable.

Hyp-A_π, Hyp-B_π: $\langle\Theta\rangle \in A_\pi \cup B_\pi$ so $I = \top$.

(T2.Inv1) $\pi \models A \wedge \neg\langle\Theta\rangle\lceil_a \Rightarrow \top$ holds trivially.

(T2.Inv2) $\pi \models B \wedge \neg\langle\Theta\rangle\lceil_b \Rightarrow \bot$.

We show that the antecedents of the implication are unsatisfied. The reason is that $\neg\langle\Theta\rangle\lceil_b$ evaluates (is equivalent) to $\bot$ under assignment π.

From $\langle\Theta\rangle \in A_\pi$ (resp. $\langle\Theta\rangle \in B_\pi$) it follows that exists literal $l \in \Theta$ such that $\pi \models l$; the literal l makes the clause $\langle\Theta\rangle$ satisfied under π. The label of l is d^+ (locality of labeling function – D13.1) so the literal is preserved by the upward-filter $\lceil_b$.

Thus $\pi \models \neg\langle\Theta\rangle\lceil_b \Leftrightarrow \bot$.

(T2.Inv3) $\mathsf{Var}(\top) \subseteq \mathsf{Var}(A) \cap \mathsf{Var}(B)$ holds trivially.

Before the proof of Theorem 2 continues (i.e., moves from leaves to inner vertices), we introduce auxiliary lemmas. The first one introduces upward-filter for pivot variables. The second lemma connects the antecedents of the invariant implications of the current vertex and the antecedent vertices.

Lemma 3 (Introducing upward-filters) *Let p be a variable, v be a vertex, and c be a label ($c \in L$). It holds:*

$$\models p \Rightarrow \neg\langle\overline{p}\rangle\lceil_{c,v} \qquad \textit{and} \qquad \models \overline{p} \Rightarrow \neg\langle p\rangle\lceil_{c,v}$$

Proof (Lemma 3). The upward-filter $\lceil_{c,v}$ can either preserve the literal $\overline{p}$ or filter it out. In the first case, the filter evaluates to $\neg\langle\overline{p}\rangle$ which is equivalent to p and the implication $\models p \Rightarrow p$ holds trivially. In the second case, the filter evaluates to the empty clause, i.e., False and the implication $\models p \Rightarrow \neg\mathsf{False}$ holds trivially.

The same reasoning applies to the second formula. □

Lemma 4 (Filters in antecedent vertices) *Let $R \equiv (V, E, cl, piv, s)$ be a resolution proof and $\mathsf{Lab}_{R,L}$ be a labeling function for proof R. Let $v \in V$ be inner vertex of the proof with vertex clause $cl(v) = \langle \Theta_1, \Theta_2 \rangle$. Let vertices v_1 and v_2 be the antecedents of vertex v and their vertex clauses be $cl(v_1) = \langle p, \Theta_1 \rangle$ resp. $cl(v_2) = \langle \overline{p}, \Theta_2 \rangle$. Let c be a label ($c \in L$). Then it holds:*

$$\neg\langle p\rangle\restriction_{c,v_1} \wedge \neg\langle \Theta_1, \Theta_2\rangle\restriction_{c,v} \Rightarrow \neg\langle p, \Theta_1\rangle\restriction_{c,v_1} \qquad \textit{and}$$
$$\neg\langle \overline{p}\rangle\restriction_{c,v_2} \wedge \neg\langle \Theta_1, \Theta_2\rangle\restriction_{c,v} \Rightarrow \neg\langle \overline{p}, \Theta_2\rangle\restriction_{c,v_2}$$

Proof (Lemma 4). The upward-filter $\restriction$ preserves all the literals whose label equals to or is greater than the given label (e.g., $\restriction_a$ preserves literals with labels a, ab, d^+). From the definition of labeling function (in particular from the conditions D10.1 and D10.2) it follows that $\forall l \in \langle \Theta_1, \Theta_2 \rangle$: $\mathsf{Lab}(v_1, l) \sqsubseteq \mathsf{Lab}(v, l)$. So, the literals preserved by the upward filter in the vertex v_1 (excluding the pivot) are also preserved by the upward filter in the successor vertex v. Thus, it follows that $\langle \Theta_1, \Theta_2\rangle\restriction_{c,v_1} \Rightarrow \langle \Theta_1, \Theta_2\rangle\restriction_{c,v}$, which can be equivalently rewritten into contrapositive implication $\neg\langle \Theta_1, \Theta_2\rangle\restriction_{c,v} \Rightarrow \neg\langle \Theta_1, \Theta_2\rangle\restriction_{c,v_1} \Rightarrow \neg\langle \Theta_1\rangle\restriction_{c,v_1}$.

The implication $\neg\langle p\rangle\restriction_{c,v_1} \wedge \neg\langle \Theta_1, \Theta_2\rangle\restriction_{c,v} \Rightarrow \neg\langle p, \Theta_1\rangle\restriction_{c,v_1}$ holds, because the same filter is applied onto literal p (it is either filtered out or preserved by both filters).

The same reasoning applies to the second formula. □

Proof (Theorem 2 – Correctness – cont.).
Induction hypothesis. Now, we will focus on the inductive step. Let v be an inner vertex of the proof and let variable p be the pivot of the resolution at vertex v (i.e., $p = piv(v)$). Let vertex v_1 be the antecedent of v with the vertex-clause containing the pivot positively (i.e., $cl(v_1) = \langle p, \Theta_1 \rangle$) and let vertex v_2 be the antecedent of v having negated pivot in its vertex-clause (i.e., $cl(v_2) = \langle \overline{p}, \Theta_2 \rangle$). From the induction hypothesis, we know that for the antecedent vertices, the following invariants hold:

$$\begin{array}{lclcl} \pi \models A \wedge \neg\langle p, \Theta_1\rangle\restriction_{a,v_1} \Rightarrow I_1 & \text{and} & \pi \models B \wedge \neg\langle p, \Theta_1\rangle\restriction_{b,v_1} \Rightarrow \neg I_1 & \text{and} \\ \pi \models A \wedge \neg\langle \overline{p}, \Theta_2\rangle\restriction_{a,v_2} \Rightarrow I_2 & \text{and} & \pi \models B \wedge \neg\langle \overline{p}, \Theta_2\rangle\restriction_{b,v_2} \Rightarrow \neg I_2 & \end{array} \tag{IH}$$

For each type of the resolution, we establish the induction invariants for vertex v.

Res-a: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a$ so $I = I_1 \vee I_2$.
In this case the pivot variable p has the label a in both antecedents v_1 and v_2.

(T2.Inv1) It follows that:

$$\begin{aligned}\pi \models \overline{p} \wedge A \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{a,v} &\overset{\text{(L3)}}{\Rightarrow} \neg\langle p\rangle\lceil_{a,v_1} \wedge A \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{a,v} \overset{\text{(L4)}}{\Rightarrow}\\ &\Rightarrow A \wedge \neg\langle p,\Theta_1\rangle\lceil_{a,v_1} \overset{\text{(IH)}}{\Rightarrow} I_1\\ \pi \models p \wedge A \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{a,v} &\overset{\text{(L3)}}{\Rightarrow} \neg\langle \overline{p}\rangle\lceil_{a,v_2} \wedge A \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{a,v} \overset{\text{(L4)}}{\Rightarrow}\\ &\Rightarrow A \wedge \neg\langle \overline{p},\Theta_2\rangle\lceil_{a,v_2} \overset{\text{(IH)}}{\Rightarrow} I_2\end{aligned}$$

The first implication is application of Lemma 3. The second implication is application of Lemma 4 and the last one is the induction hypothesis.

From the previous implications, it directly follows that:

$$\pi \models A \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{a,v} \Leftrightarrow (\overline{p} \vee p) \wedge A \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{a,v} \Rightarrow (I_1 \vee I_2)$$

The first equivalence is a simple logical consequence of $p \vee \overline{p} \Leftrightarrow \top$. The second implication follows from the two equations above.

(T2.Inv2) Because the label of the pivot in the antecedents is a, it follows that $\neg\langle p\rangle\lceil_{b,v_1} \Leftrightarrow \neg\langle\overline{p}\rangle\lceil_{b,v_2} \Leftrightarrow \top$. Thus, Lemma 4 can be applied directly without any additional assumptions:

$$\begin{aligned}\pi \models B \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{b,v} &\Leftrightarrow \neg\langle p\rangle\lceil_{b,v_1} \wedge B \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{b,v} \overset{\text{(L4)}}{\Rightarrow}\\ &\Rightarrow B \wedge \neg\langle p,\Theta_1\rangle\lceil_{b,v_1} \overset{\text{(IH)}}{\Rightarrow} \neg I_1\\ \pi \models B \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{b,v} &\Leftrightarrow \neg\langle \overline{p}\rangle\lceil_{b,v_2} \wedge B \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{b,v} \overset{\text{(L4)}}{\Rightarrow}\\ &\Rightarrow B \wedge \neg\langle \overline{p},\Theta_2\rangle\lceil_{b,v_2} \overset{\text{(IH)}}{\Rightarrow} \neg I_2\end{aligned}$$

$$\pi \models B \wedge \neg\langle\Theta_1,\Theta_2\rangle\lceil_{b,v} \Rightarrow (\neg I_1 \wedge \neg I_2) \Leftrightarrow \neg(I_2 \vee I_2)$$

The first implication follows from the two equations above. The second equivalence is factoring out the negation.

(T2.Inv3) The third requirement (shared variables only) holds trivially. No new variable is added into the partial vertex-interpolant.

Res-b: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$ so $I = I_1 \wedge I_2$.
The proof is symmetric to the Res-a case. In this case the pivot variable has the label b in both antecedents v_1 and v_2.

(T2.Inv1) The label of the pivot in the antecedents is b so it holds: $\neg\langle p\rangle\restriction_{a,v_1} \Leftrightarrow \neg\langle\overline{p}\rangle\restriction_{a,v_2} \Leftrightarrow \top$.

$$\begin{aligned}
\pi \models A \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{a,v} &\Leftrightarrow \neg\langle p\rangle\restriction_{a,v_1} \wedge A \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{a,v} \overset{(L4)}{\Rightarrow} \\
&\Rightarrow A \wedge \neg\langle p,\Theta_1\rangle\restriction_{a,v_1} \overset{(IH)}{\Rightarrow} I_1 \\
\pi \models A \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{a,v} &\Leftrightarrow \neg\langle\overline{p}\rangle\restriction_{a,v_2} \wedge A \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{a,v} \overset{(L4)}{\Rightarrow} \\
&\Rightarrow A \wedge \neg\langle\overline{p},\Theta_2\rangle\restriction_{a,v_2} \overset{(IH)}{\Rightarrow} I_2
\end{aligned}$$

The equations above directly yield the result:

$$\pi \models A \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{a,v} \Rightarrow (I_1 \wedge I_2)$$

(T2.Inv2) It follows that:

$$\begin{aligned}
\pi \models \overline{p} \wedge B \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{b,v} &\overset{(L3)}{\Rightarrow} \neg\langle p\rangle\restriction_{b,v_1} \wedge B \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{b,v} \overset{(L4)}{\Rightarrow} \\
&\Rightarrow B \wedge \neg\langle p,\Theta_1\rangle\restriction_{b,v_1} \overset{(IH)}{\Rightarrow} \neg I_1 \\
\pi \models p \wedge B \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{b,v} &\overset{(L3)}{\Rightarrow} \neg\langle\overline{p}\rangle\restriction_{b,v_2} \wedge B \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{b,v} \overset{(L4)}{\Rightarrow} \\
&\Rightarrow B \wedge \neg\langle\overline{p},\Theta_2\rangle\restriction_{b,v_2} \overset{(IH)}{\Rightarrow} \neg I_2
\end{aligned}$$

From the previous implications, it directly follows that:

$$\begin{aligned}
\pi \models B \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{b,v} &\Leftrightarrow (\overline{p} \vee p) \wedge B \wedge \neg\langle\Theta_1,\Theta_2\rangle\restriction_{b,v} \Rightarrow \\
&\Rightarrow (\neg I_1 \vee \neg I_2) \Leftrightarrow \neg(I_1 \wedge I_2)
\end{aligned}$$

The first equivalence is a simple logical consequence of $p \vee \overline{p} \Leftrightarrow \top$. The second implication follows from the two equations above, while the last equivalence factors out the negation.

(T2.Inv3) The third requirement (shared variables only) holds trivially as no new variable is added into the partial vertex-interpolant.

Res-ab: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$, so $I = (p \vee I_1) \wedge (\overline{p} \vee I_2)$.

(T2.Inv1) It follows that:

$$\begin{aligned}
\pi \models A \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{a,v} &\Rightarrow p \vee (\overline{p} \wedge A \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{a,v}) \overset{(L3)}{\Rightarrow} \\
&\Rightarrow p \vee (\neg\langle p\rangle\!\restriction_{a,v_1} \wedge A \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{a,v}) \overset{(L4)}{\Rightarrow} \\
&\Rightarrow p \vee (A \wedge \neg\langle p, \Theta_1\rangle\!\restriction_{a,v_1}) \overset{(IH)}{\Rightarrow} (p \vee I_1) \\
\pi \models A \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{a,v} &\Rightarrow \overline{p} \vee (p \wedge A \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{a,v}) \overset{(L3)}{\Rightarrow} \\
&\Rightarrow \overline{p} \vee (\neg\langle \overline{p}\rangle\!\restriction_{a,v_2} \wedge A \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{a,v}) \overset{(L4)}{\Rightarrow} \\
&\Rightarrow \overline{p} \vee (A \wedge \neg\langle \overline{p}, \Theta_2\rangle\!\restriction_{a,v_2}) \overset{(IH)}{\Rightarrow} (\overline{p} \vee I_2)
\end{aligned}$$

The first implication is a logical consequence of $p \vee \overline{p} \Leftrightarrow \top$. The second implication is application of Lemma 3. The third implication is application of Lemma 4 and the last one is the induction hypothesis. From the implications above, it directly follows that:

$$\pi \models A \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{a,v}) \Rightarrow (p \vee I_1) \wedge (\overline{p} \vee I_2)$$

(T2.Inv2) Similarly to the previous case:

$$\begin{aligned}
\pi \models \overline{p} \wedge B \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{b,v} &\overset{(L3)}{\Rightarrow} \overline{p} \wedge (\neg\langle p\rangle\!\restriction_{b,v_1} \wedge B \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{b,v}) \overset{(L4)}{\Rightarrow} \\
&\Rightarrow \overline{p} \wedge (B \wedge \neg\langle p, \Theta_1\rangle\!\restriction_{b,v_1}) \overset{(IH)}{\Rightarrow} \\
&\Rightarrow \overline{p} \wedge (\neg I_1) \Leftrightarrow \neg(p \vee I_1) \\
\pi \models p \wedge B \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{b,v} &\overset{(L3)}{\Rightarrow} p \wedge (\neg\langle \overline{p}\rangle\!\restriction_{b,v_2} \wedge B \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{b,v}) \overset{(L4)}{\Rightarrow} \\
&\Rightarrow p \wedge (B \wedge \neg\langle \overline{p}, \Theta_2\rangle\!\restriction_{b,v_2}) \overset{(IH)}{\Rightarrow} \\
&\Rightarrow p \wedge (\neg I_2) \Leftrightarrow \neg(\overline{p} \vee I_2)
\end{aligned}$$

In the first implication, the conjunct $\overline{p}$ is duplicated and then, Lemma 3 is applied. The last implication is simple logical equality.

$$\begin{aligned}
\pi \models B \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{b,v} &\Leftrightarrow (\overline{p} \vee p) \wedge B \wedge \neg\langle\Theta_1, \Theta_2\rangle\!\restriction_{b,v} \Rightarrow \\
&\Rightarrow \neg(p \vee I_1) \vee \neg(\overline{p} \vee I_2) \Leftrightarrow \neg((p \vee I_1) \wedge (\overline{p} \vee I_2))
\end{aligned}$$

The same reasoning as in the Res-a (T2.Inv1) case is used. The first equivalence is a simple logical consequence of $p \vee \overline{p} \Leftrightarrow \top$. The second implication follows from the two equations above. The last equivalence just factors out the negation.

(T2.Inv3) Variable p is the only new variable added into the interpolant. Variable p is shared (because of its label ab), thus the requirements are met. Moreover, variable p is not assigned. If it would be assigned, it would be labeled d^+ in one of the antecedents, which would lead to the Res-d resolution.

Res-d: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = d^+$ so $I = I_1$ resp. $I = I_2$.

In this case, the pivot variable is assigned by PVA π. Labeling function Lab is locality preserving and constraint D13.1 give us that there is exactly one antecedent where the pivot is labeled d^+. Assume that $\mathsf{Lab}(v_1, p) = d^+$, so it holds that $\pi \models p$; the case $\mathsf{Lab}(v_2, \overline{p}) = d^+$ is symmetric.

(T2.Inv1) It follows that:

$$\pi \models A \wedge \neg\langle\Theta_1, \Theta_2\rangle\restriction_{a,v} \Leftrightarrow \neg\langle\overline{p}\rangle\restriction_{a,v_2} \wedge A \wedge \neg\langle\Theta_1, \Theta_2\rangle\restriction_{a,v} \overset{(L4)}{\Rightarrow}$$
$$\Rightarrow A \wedge \neg\langle\overline{p}, \Theta_2\rangle\restriction_{a,v_2} \overset{(IH)}{\Rightarrow} I_2$$

The first equivalence holds because $\pi \models \neg\langle\overline{p}\rangle\restriction_{a,v_2}$; the $\neg\langle\overline{p}\rangle\restriction_{a,v_2}$ is either directly $\top$ if the $\overline{p}$ literal is not preserved by the upward-filter or it is p if the $\overline{p}$ literal is preserved by the filter $\restriction_{a,v_2}$. In the latter case, p is satisfied under π.

(T2.Inv2) Similarly to the previous case, it holds:

$$\pi \models B \wedge \neg\langle\Theta_1, \Theta_2\rangle\restriction_{b,v} \Leftrightarrow \neg\langle\overline{p}\rangle\restriction_{b,v_2} \wedge B \wedge \neg\langle\Theta_1, \Theta_2\rangle\restriction_{b,v} \overset{(L4)}{\Rightarrow}$$
$$\Rightarrow B \wedge \neg\langle\overline{p}, \Theta_2\rangle\restriction_{b,v_2} \overset{(IH)}{\Rightarrow} \neg I_2$$

(T2.Inv3) This condition holds trivially from the induction hypothesis.

To sum-up, we have shown that all the resolutions and hypotheses establish the inductive invariant for its partial vertex-interpolant. Thus, the inductive invariant also holds for the sink vertex s, where $cl(s) = \langle\emptyset\rangle = \bot$; the inductive invariant establishes Theorem 2 for the sink vertex. $\square$

A.2.2 Path interpolation property

Theorem 4 (Path interpolation property) Let Lab and Lab' be locality preserving labeling functions, let R be a refutation of $A \wedge S \wedge B$, and π and π' be PVAs. Let $I = \mathsf{Lpaltp}(\mathsf{Lab}, R, A, S \cup B, \pi)$ and $I' = \mathsf{Lpaltp}(\mathsf{Lab}', R, A \cup S, B, \pi')$.

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ then $\pi, \pi' \models I \wedge S \Rightarrow I'$.

Before we prove Theorem 4, we introduce auxiliary lemmas; the lemmas are of similar purpose as lemmas used in the proof of the correctness (i.e., the proof of Theorem 2). Lemma 5 about the weakened-labels filters is similar to Lemma 4. The latter one is used to introduce assignment filters.

Lemma 5 (Weakened-labels filters in antecedent vertices) *Let $R \equiv (V, E, cl, piv, s)$ be a resolution proof and let Lab and Lab' be labeling functions for proof R. Let $v \in V$ be an inner vertex of the proof with vertex clause $cl(v) = \langle \Theta_1, \Theta_2 \rangle$. Let vertices v_1 and v_2 be the antecedents of vertex v and their vertex clauses be $cl(v_1) = \langle p, \Theta_1 \rangle$ resp. $cl(v_2) = \langle \overline{p}, \Theta_2 \rangle$. Then it holds:*

$$\neg\langle p \rangle|\!\lfloor_{v_1} \wedge \neg\langle \Theta_1, \Theta_2 \rangle|\!\lfloor_{v} \Rightarrow \neg\langle p, \Theta_1 \rangle|\!\lfloor_{v_1} \qquad \textit{and}$$
$$\neg\langle \overline{p} \rangle|\!\lfloor_{v_2} \wedge \neg\langle \Theta_1, \Theta_2 \rangle|\!\lfloor_{v} \Rightarrow \neg\langle \overline{p}, \Theta_2 \rangle|\!\lfloor_{v_2}$$

Proof (Lemma 5). First, we show that if a literal $l \in \langle \Theta_1, \Theta_2 \rangle$ is preserved by the weakened-labels filter in antecedent vertex v_1, then it is also preserved by the weakened-labels filter in vertex v where its label is a result of the join (i.e., $\sqcup$) operation (see D10.2). The sets used by the filter $|\!\lfloor$, in particular $\{b, ab, d^+\}$ and $\{ab, d^+, a\}$, are closed under the join operation (i.e., $\sqcup$); formally, $\forall c \in L$ and $\forall c' \in \{b, ab, d^+\}$ it holds that $c \sqcup c' \in \{b, ab, d^+\}$.

If literal l is preserved by the weakened-labels filter in vertex v_1, the first labeling function assigns to literal l a label from set $\{b, ab, d^+\}$ (formally, $\mathsf{Lab}(v_1, l) \in \{b, ab, d^+\}$); for the other labeling function, it holds that $\mathsf{Lab}'(v_1, l) \in \{ab, d^+, a\}$. Because these sets are closed under the join operation (which is used to compute the labels at vertex v from the labels at vertex v_1), the same holds even in vertex v (formally $\mathsf{Lab}(v, l) \in \{b, ab, d^+\}$ and $\mathsf{Lab}'(v, l) \in \{ab, d^+, a\}$). It means that literal l is also preserved by the weakened-labels filter in vertex v.

It follows that $\langle \Theta_1, \Theta_2 \rangle|\!\lfloor_{v_1} \Rightarrow \langle \Theta_1, \Theta_2 \rangle|\!\lfloor_{v}$; the implication can be equivalently rewritten into the contrapositive form:

$$\neg\langle \Theta_1, \Theta_2 \rangle|\!\lfloor_{v} \Rightarrow \neg\langle \Theta_1, \Theta_2 \rangle|\!\lfloor_{v_1} \Rightarrow \neg\langle \Theta_1 \rangle|\!\lfloor_{v_1}$$

The claim of the lemma directly follows from the above implication:

$$\neg\langle p\rangle\|\lfloor_{v_1} \wedge \neg\langle\Theta_1,\Theta_2\rangle\|\lfloor_{v} \Rightarrow \neg\langle p,\Theta_1\rangle\|\lfloor_{v_1}$$

Note that the same filter is applied on pivot p; the pivot is either filtered or preserved in both cases. The same reasoning applies to the second formula. □

Lemma 6 (Introducing assignment filter) *Let π be a partial variable assignment and $\langle\Theta\rangle$ be a clause not satisfied by the partial assignment, i.e., $\pi \not\models \langle\Theta\rangle$.*

Then it holds: $\pi \models \langle\Theta\rangle \Leftrightarrow \langle\Theta\rangle[\pi]$.

Proof (Lemma 6). It is possible to split the set of literals Θ into two disjoint sets; set of literals Θ_1 containing the literals over the assigned variables (these literals will be filtered-out by the assignment filter) and set Θ_2 containing the remaining literals over the non-assigned variables. So $\langle\Theta\rangle \Leftrightarrow \langle\Theta_1\rangle \vee \langle\Theta_2\rangle$.

From the assumption that $\pi \not\models \langle\Theta\rangle$, it follows that all the literals over assigned variables evaluate to $\bot$ under the assignment π, thus: $\pi \models \langle\Theta_1\rangle \Leftrightarrow \bot$. From the definition of the assignment filter, it directly follows that $\langle\Theta_2\rangle \equiv \langle\Theta_2\rangle[\pi]$ and $\langle\Theta_1\rangle[\pi] \equiv \langle\emptyset\rangle \Leftrightarrow \bot$. So, it holds:

$$\pi \models \langle\Theta\rangle \Leftrightarrow \langle\Theta_1\rangle \vee \langle\Theta_2\rangle \Leftrightarrow \bot \vee \langle\Theta_2\rangle \Leftrightarrow \langle\Theta_1\rangle[\pi] \vee \langle\Theta_2\rangle[\pi] \Leftrightarrow \langle\Theta\rangle[\pi]$$

□

Note that the claim of Lemma 6 is trivial; we formulate the claim as a lemma to be able to refer to it in the proofs.

Proof (Theorem 4 – Path interpolation property). By structural induction over refutation R we show that for each vertex $v \in V$ of the refutation, the following invariant holds:

$$\pi,\pi' \models I_v \wedge S \wedge \neg\langle\Theta\rangle\|\lfloor_{v} \Rightarrow I'_v$$

where $cl(v) = \langle\Theta\rangle$ is the vertex clause and I_v and I'_v are the partial vertex-interpolants for vertex v as generated by LPAIS using labeling functions Lab and Lab', respectively. In the proof, we show that the invariant holds for all possible combinations of the rules that can be used to define partial vertex-interpolants I_v and I'_v.

Bases cases. The base cases correspond to the leaves of the proof where the hypotheses operations are applied. Theoretically, there are 16 possible combinations of hypotheses, however, not all of them are possible due to the assumptions of the theorem; below we discuss each of the combinations in more detail. As to the naming conventions, we call each case either as Hyp or Res, followed by the kind of the rule used to compute the first partial vertex-interpolant I and by the kind of the rule used to compute the second partial vertex-interpolant I'. Note that for the first interpolant, the partitioning is $(A,\ S \cup B)$, while for the second interpolant the partitioning is $(A \cup S, B)$; we use these names of partitions in names of the Hyp rules.

Hyp-$A_{\overline{\pi}}$-$(A_{\overline{\pi}'} \cup S_{\overline{\pi}'})$: $I_v = \langle\Theta\rangle[\pi]]_{b,v,\mathsf{Lab}}$ and $I'_v = \langle\Theta\rangle[\pi']]_{b,v,\mathsf{Lab}'}$.

First, we show the following:

$$\langle\Theta\rangle|_{b,v,\mathsf{Lab}} \land \neg\langle\Theta\rangle|\lfloor_v \Rightarrow \langle\Theta\rangle|_{b,v,\mathsf{Lab}'}$$

Let literal l be labeled b by labeling function Lab (i.e., it is preserved by the match filter $|_{b,v,\mathsf{Lab}}$). It can either get a label b by labeling function Lab', thus l is preserved by the match filter $|_{b,v,\mathsf{Lab}'}$ in the consequent of the implication, or it gets a different label, which is necessarily weaker then b. In the latter case, the literal is preserved by the weakened-labels filter $|\lfloor_v$, which is negated in the antecedent of the implication above. This means that if clause $\langle\Theta\rangle|_{b,v,\mathsf{Lab}}$ is satisfied due to literal l, then either the consequent of the implication is satisfied (the former case) and the implication holds, or the negation of the literal is in the antecedent of the implication (due to weakened-labels filter), so the antecedent of the implication is not satisfied (and the whole implication holds).

We can add the assignments as assumptions. Then it holds:

$$\pi, \pi' \models \langle\Theta\rangle|_{b,v,\mathsf{Lab}} \land \neg\langle\Theta\rangle|\lfloor_v \Rightarrow \langle\Theta\rangle|_{b,v,\mathsf{Lab}'}$$

Clause $\langle\Theta\rangle$ is neither satisfied by π nor by π', so Lemma 6 can by used to remove the falsified literals. Then it holds:

$$\pi, \pi' \models \quad \langle\Theta\rangle[\pi]]_{b,v,\mathsf{Lab}} \land \neg\langle\Theta\rangle|\lfloor_v \Rightarrow \langle\Theta\rangle[\pi']]_{b,v,\mathsf{Lab}'}$$

$$\begin{aligned}\pi, \pi' \models \quad & I_v \land S \land \neg\langle\Theta\rangle|\lfloor_v \Leftrightarrow \\ & \langle\Theta\rangle[\pi]]_{b,v,\mathsf{Lab}} \land S \land \neg\langle\Theta\rangle|\lfloor_v \Rightarrow \langle\Theta\rangle[\pi']]_{b,v,\mathsf{Lab}'} \Leftrightarrow I'_v\end{aligned}$$

Hyp-$A_{\overline{\pi}}$-$(A_{\overline{\pi}'} \cup S_{\overline{\pi}'})$: $I_v = \langle\Theta\rangle[\pi]]_{b,v,\mathsf{Lab}}$ and $I'_v = \top$.

Note that in contrast to the previous case, vertex clause $\langle\Theta\rangle$ is satisfied under assignment π'. In this case, the invariant holds trivially, since anything implies $\top$.

Hyp-$A_{\overline{\pi}}$-$B_{\overline{\pi}'}$: $I_v = \langle\Theta\rangle[\pi]\rrbracket_{b,v,\mathsf{Lab}}$ and $I'_v = \langle\Theta\rangle[\pi']\rrbracket_{a,v,\mathsf{Lab}'}$.

This combination is impossible due to our partitionings; it would require clause $\langle\Theta\rangle$ to move from the A-part of the first partitioning into the B-part of the second partitioning. However, the partitionings permit only moves of clauses from the B-part of the first partitioning into the A-part of the second partitioning; the moved clauses are those forming S.

Hyp-$A_{\overline{\pi}}$-$B_{\overline{\pi}'}$: $I_v = \langle\Theta\rangle[\pi]\rrbracket_{b,v,\mathsf{Lab}}$ and $I'_v = \top$. The same reasoning as above applies; such combination is impossible due to our partitionings. Note that the reasoning above is independent of the assignment.

Hyp-$(B_{\overline{\pi}} \cup S_{\overline{\pi}})$-$(A_{\overline{\pi}'} \cup S_{\overline{\pi}'})$: $I_v = \neg\langle\Theta\rangle[\pi]\rrbracket_{a,v,\mathsf{Lab}}$ and $I'_v = \langle\Theta\rangle[\pi']\rrbracket_{b,v,\mathsf{Lab}'}$. In this case, clause $\langle\Theta\rangle$ is moved from the B-part of the first partitioning into the A-part of the second partitioning; it means the clause belongs to the set of clauses S ($\langle\Theta\rangle \in S$). First, we show that in this case, the following holds:

$$\langle\Theta\rangle \Leftrightarrow \langle\Theta\rangle|_{a,v,\mathsf{Lab}} \vee \langle\Theta\rangle|_{b,v,\mathsf{Lab}'} \vee \langle\Theta\rangle|\downharpoonleft_v$$

The direction from right to left (i.e., the implication $\Leftarrow$) is trivial, since filters only remove literals. So, if the right-hand side of the equivalence holds, the unfiltered clause $\langle\Theta\rangle$ must also hold. The direction from left to right (i.e., the implication $\Rightarrow$) is shown below. We consider all the combinations of labels the literal $l \in \langle\Theta\rangle$ can get by labeling functions Lab and Lab'.

- If $\mathsf{Lab}(v,l) = a$ then the match filter $|_{a,v,\mathsf{Lab}}$ preserves the literal l.
- If $\mathsf{Lab}(v,l) \in \{ab, d^+\}$ and $\mathsf{Lab}'(v,l) \neq b$ then the weakened-label filter $|\downharpoonleft_v$ preserves the literal l.
- If $\mathsf{Lab}(v,l) \in \{ab, d^+\}$ and $\mathsf{Lab}'(v,l) = b$ then the assumption $\mathsf{Lab} \preceq \mathsf{Lab}'$ is violated.
- If $\mathsf{Lab}(v,l) = b$ and $\mathsf{Lab}'(v,l) = b$ then the match filter $|_{b,v,\mathsf{Lab}'}$ preserves the literal l.
- If $\mathsf{Lab}(v,l) = b$ and $\mathsf{Lab}'(v,l) \neq b$ then the weakened-label filter $|\downharpoonleft_v$ preserves the literal l.

The clause $\langle\Theta\rangle$ is satisfied neither by π nor by π' so Lemma 6 can be used to remove the falsified literals. Then it holds:

$$\pi, \pi' \models \langle\Theta\rangle \Leftrightarrow \langle\Theta\rangle[\pi]\rrbracket_{a,v,\mathsf{Lab}} \vee \langle\Theta\rangle[\pi']\rrbracket_{b,v,\mathsf{Lab}'} \vee \langle\Theta\rangle|\downharpoonleft_v$$

The invariant is shown by the following:

$$\begin{array}{rlcr}
\pi, \pi' \models & I_v \quad \wedge & S & \wedge \neg\langle\Theta\rangle|\downharpoonleft_v \equiv \\
& \equiv \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}} \wedge & S & \wedge \neg\langle\Theta\rangle|\downharpoonleft_v \Rightarrow \\
& \Rightarrow \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}} \wedge & \langle\Theta\rangle & \wedge \neg\langle\Theta\rangle|\downharpoonleft_v \Leftrightarrow \\
& \Leftrightarrow \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}} \wedge & (\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}} \vee \langle\Theta\rangle[\pi']\|_{b,v,\mathsf{Lab}'} \quad \vee \langle\Theta\rangle|\downharpoonleft_v) & \wedge \neg\langle\Theta\rangle|\downharpoonleft_v \Leftrightarrow \\
& \Leftrightarrow \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}} \wedge & \langle\Theta\rangle[\pi']\|_{b,v,\mathsf{Lab}'} & \wedge \neg\langle\Theta\rangle|\downharpoonleft_v \Rightarrow \\
& & \Rightarrow \langle\Theta\rangle[\pi']\|_{b,v,\mathsf{Lab}'} & \equiv I'_v
\end{array}$$

The first implication follows from the fact that $\langle\Theta\rangle \in S$ and S is a conjunction (or equivalently a set) of clauses, so $S \Rightarrow \langle\Theta\rangle$. The second equivalence is shown above. The third equivalence is a logical consequence. The following pattern is used twice: $\neg A \wedge (A \vee B) \Leftrightarrow (\neg A \wedge A) \vee (\neg A \wedge B) \Leftrightarrow \neg A \wedge B$, where we use $A \equiv \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}}$ resp. $A \equiv \neg\langle\Theta\rangle|\downharpoonleft_v$. The last implication is a trivial logical consequence.

Hyp-($B_{\overline{\pi}} \cup S_{\overline{\pi}}$)-($A'_{\pi} \cup S_{\pi'}$): $I_v = \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}}$ and $I'_v = \top$.

As in all the other cases where the vertex clause is satisfied by the second assignment π', the invariant holds trivially, because anything implies $I'_v \equiv \top$.

Hyp-($B_{\overline{\pi}} \cup S_{\overline{\pi}}$)-$B_{\overline{\pi}'}$: $I_v = \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}}$ and $I'_v = \neg\langle\Theta\rangle[\pi']\|_{a,v,\mathsf{Lab}'}$.

This case is similar to the Hyp-$A_{\overline{\pi}}$-($A_{\overline{\pi}'} \cup S_{\overline{\pi}'}$) case. First, we show that:

$$\neg\langle\Theta\rangle|_{a,v,\mathsf{Lab}} \wedge \neg\langle\Theta\rangle|\downharpoonleft_v \Rightarrow \neg\langle\Theta\rangle|_{a,v,\mathsf{Lab}'}$$

Let literal l be labeled a by labeling Lab' (so it is preserved by the match filter $|_{a,v,\mathsf{Lab}'}$). The literal is either labeled a by labeling Lab or not (in which case its label can be b or ab). In the former case, the literal is preserved by the match filter $|_{a,v,\mathsf{Lab}}$. In the latter case, the literal is preserved by the weakened-labels filter $|\downharpoonleft_v$. To sum it up, all the literals in the consequent of the implication occur even in the antecedent of the implication; this shows that the implication holds.

Clause $\langle\Theta\rangle$ is neither satisfied by π nor by π', so Lemma 6 can be used to remove the falsified literals. Then it holds:

$$\pi, \pi' \models \neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}} \wedge \neg\langle\Theta\rangle|\downharpoonleft_v \Rightarrow \neg\langle\Theta\rangle[\pi']\|_{a,v,\mathsf{Lab}'}$$

The implication above is even stronger than the invariant; it does not require S to be a part of the antecedent of the implication.

Hyp-$(B_{\overline{\pi}} \cup S_{\overline{\pi}})$-$B_{\pi'}$: $I_v = \neg\langle\Theta\rangle[\pi]\!]_{a,v,\mathsf{Lab}}$ and $I'_v = \top$.

As in all the other cases where the vertex clause is satisfied by the second assignment π' the invariant holds trivially, because anything implies $I'_v \equiv \top$.

Hyp-A_π-$(A_{\overline{\pi}'} \cup S_{\overline{\pi}'})$: $I_v = \top$ and $I'_v = \langle\Theta\rangle[\pi']\!]_{b,v,\mathsf{Lab}'}$.

Vertex clause $\langle\Theta\rangle$ is satisfied under π, thus there exists literal $l \in \Theta$ such that $\mathsf{Lab}(v,l) = d^+$; literal l makes the clause satisfied under π. From the assumption (of the theorem that) $\mathsf{Lab} \preceq \mathsf{Lab}'$, it follows that $\mathsf{Lab}'(v,l) \neq b$; thus literal l is preserved by weakened-labels filter $|\!|\!\downharpoonright_v$. It means that $\pi, \pi' \models \neg\langle\Theta\rangle\!|\!\downharpoonright_v \Leftrightarrow \bot$, so the antecedent of the invariant implication is falsified by the assumed assignments and the whole invariant implication holds.

Exactly the same reasoning applies to all the remaining hypotheses where the assignment π satisfies the vertex clause; in particular, to the Hyp-A_π-$(A_{\pi'} \cup S_{\pi'})$, Hyp-A_π-$B_{\overline{\pi}'}$, Hyp-A_π-$B_{\pi'}$, Hyp-$(B_\pi \cup S_\pi)$-$(A_{\overline{\pi}'} \cup S_{\overline{\pi}'})$, Hyp-$(B_\pi \cup S_\pi)$-$(A_{\pi'} \cup S_{\pi'})$, Hyp-$(B_\pi \cup S_\pi)$-$B_{\overline{\pi}'}$, and Hyp-$(B_\pi \cup S_\pi)$-$B_{\pi'}$.

Induction hypothesis. Now, we will focus on the inductive step. Let v be an inner vertex of the proof and let variable p be the pivot of the refutation at vertex v (i.e., $p = piv(v)$). Let vertex v_1 be the antecedent of v with the vertex-clause containing the pivot positively (i.e., $cl(v_1) = \langle p, \Theta_1\rangle$) and let vertex v_2 be the antecedent of v having negated pivot in its vertex-clause (i.e., $cl(v_2) = \langle \overline{p}, \Theta_2\rangle$). From the induction hypothesis, we know that for the antecedent vertices the following invariants hold:

$$\begin{aligned} \pi, \pi' &\models I_{v_1} \wedge S \wedge \neg\langle\Theta\rangle\!|\!\downharpoonright_v \Rightarrow I'_{v_1} \\ \pi, \pi' &\models I_{v_2} \wedge S \wedge \neg\langle\Theta\rangle\!|\!\downharpoonright_v \Rightarrow I'_{v_2} \end{aligned} \tag{PIH}$$

For each possible combination of the resolutions we establish the induction invariant for the vertex v. Theoretically, there are 16 possible combinations of resolutions, however, not all of them are possible due to the assumptions of the theorem; below we discuss each of the combinations in more detail.

Res-a-a': $\mathsf{Lab}(v_1,p) \sqcup \mathsf{Lab}(v_2,\overline{p}) = a$ and $\mathsf{Lab}'(v_1,p) \sqcup \mathsf{Lab}'(v_2,\overline{p}) = a$. It means that $I_v \equiv I_{v_1} \vee I_{v_2}$ and $I'_v \equiv I'_{v_1} \vee I'_{v_2}$.

The label of pivot p in both antecedent vertices v_1 resp. v_2 must be a (in both labeling functions), so it is not preserved by the weakened-labels filters $|\!|\!\downharpoonright_{v_1}$ and $|\!|\!\downharpoonright_{v_2}$; thus, it holds $\neg\langle p\rangle\!|\!\downharpoonright_{v_1} \Leftrightarrow \top$.

It holds that:

$$\pi,\pi' \models I_{v_1} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v \Leftrightarrow \neg\langle p\rangle\!|\lfloor_{v_1} \wedge I_{v_1} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v \overset{L5}{\Rightarrow}$$
$$\Rightarrow I_{v_1} \wedge S \wedge \neg\langle p,\Theta_1\rangle\!|\lfloor_{v_1} \overset{PIH}{\Rightarrow} I'_{v_1}$$
$$\pi,\pi' \models I_{v_2} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v \Leftrightarrow \neg\langle \overline{p}\rangle\!|\lfloor_{v_2} \wedge I_{v_2} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v \overset{L5}{\Rightarrow}$$
$$\Rightarrow I_{v_2} \wedge S \wedge \neg\langle \overline{p},\Theta_2\rangle\!|\lfloor_{v_2} \overset{PIH}{\Rightarrow} I'_{v_2}$$

From the implications above, the invariant for vertex v directly follows:

$$\pi,\pi' \models ((I_{v_1} \vee I_{v_2})) \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v \Rightarrow (I'_{v_1} \vee I'_{v_2})$$

Res-a-ab', Res-a-$d^{+'}$ and Res-a-b': All these cases violate the assumption of the theorem that $\mathsf{Lab} \preceq \mathsf{Lab}'$. Pivot p gets a stronger label than a by Lab' in at least one of the antecedent vertices (i.e., v_1 and v_2); otherwise, this will become the previous Res-a-a' case. Variable p, at that vertex, is the witness that assumption $\mathsf{Lab} \preceq \mathsf{Lab}'$ is violated.

Res-ab-ab': $\mathsf{Lab}(v_1,p) \sqcup \mathsf{Lab}(v_2,\overline{p}) = ab$ and $\mathsf{Lab}'(v_1,p) \sqcup \mathsf{Lab}'(v_2,\overline{p}) = ab$. It means that $I_v \equiv (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$ and $I'_v \equiv (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$.

Note that in this case, the proof is independent of the labels of the pivot variable. Also note that the proof works regardless of whether p is assigned by assignment π (resp. π') or not. So, it can be safely used to show other cases, such as Res-ab-a', as well.

We have to show the following:

$$\pi,\pi' \models (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$$

First, we introduce two auxiliary implications. The following holds:

$$\pi,\pi' \models I_{v_1} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v \Rightarrow p \vee (\overline{p} \wedge I_{v_1} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v) \Rightarrow$$
$$\Rightarrow p \vee (\neg\langle p\rangle\!|\lfloor_{v_1} \wedge I_{v_1} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v) \overset{L5}{\Rightarrow}$$
$$\Rightarrow p \vee (I_{v_1} \wedge S \wedge \neg\langle p,\Theta_1\rangle\!|\lfloor_{v_1}) \overset{PIH}{\Rightarrow} p \vee I'_{v_1}$$
$$\pi,\pi' \models I_{v_2} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v \Rightarrow \overline{p} \vee (p \wedge I_{v_2} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v) \Rightarrow$$
$$\Rightarrow \overline{p} \vee (\neg\langle \overline{p}\rangle\!|\lfloor_{v_2} \wedge I_{v_2} \wedge S \wedge \neg\langle\Theta_1,\Theta_2\rangle\!|\lfloor_v) \overset{L5}{\Rightarrow}$$
$$\Rightarrow \overline{p} \vee (I_{v_2} \wedge S \wedge \neg\langle \overline{p},\Theta_1\rangle\!|\lfloor_{v_2}) \overset{PIH}{\Rightarrow} \overline{p} \vee I'_{v_2}$$

The first implication stems from the fact that $p \vee \overline{p} \Leftrightarrow \top$. The second implication holds because $\overline{p} \Rightarrow \neg\langle p\rangle\!|\lfloor_{v_1}$; either the literal $\overline{p}$ is preserved by the filter $|\!|\downharpoonright_{v_1}$ and then it holds that $\overline{p} \Leftrightarrow \neg\langle p \rangle\!|\!|\downharpoonright_{v_1}$, or the literal $\overline{p}$ is not preserved by the weakened-labels filter and then it holds that $\neg\langle p \rangle\!|\!|\downharpoonright_{v_1} \Leftrightarrow \top$.

The proof can be split into two cases. It holds that:

$$(p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \Leftrightarrow (p \wedge I_{v_2}) \vee (\overline{p} \wedge I_{v_1}))$$

We show that each of the two cases above leads to $(p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$.

$$\begin{aligned} \pi, \pi' \models (\overline{p} \wedge I_{v_1}) \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!|\!|\downharpoonright_v \quad &\Rightarrow \overline{p} \wedge (p \vee I'_{v_1}) \quad &\Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \\ \pi, \pi' \models (p \wedge I_{v_2}) \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!|\!|\downharpoonright_v \quad &\Rightarrow p \wedge (\overline{p} \vee I'_{v_2}) \quad &\Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \end{aligned}$$

The first implication comes from the auxiliary implications above. The second implication is a simple logical consequence.

We have shown that:

$$\pi, \pi' \models (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!|\!|\downharpoonright_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$$

Res-*ab*-*a*′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = a$. It means that $I_v \equiv (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$ and $I'_v \equiv I'_{v_1} \vee I'_{v_2}$.

It holds that:

$$\begin{aligned} \pi, \pi' \models (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!|\!|\downharpoonright_v \Rightarrow \\ \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Rightarrow I'_{v_1} \vee I'_{v_2} \end{aligned}$$

The first implication comes from Res-*ab*-*ab*′. The second one is a trivial logical consequence.

Res-*ab*-*d*$^+$′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = d^+$. So, partial vertex-interpolant I_v is defined as follows: $I_v \equiv (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$. Assume that $\mathsf{Lab}'(v_1, p) = d^+$ thus $I'_v \equiv I'_{v_2}$. The situation is symmetric if $\mathsf{Lab}'(v_2, \overline{p}) = d^+$.

It holds that:

$$\pi, \pi' \models (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!|\!|\downharpoonright_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Leftrightarrow I'_{v_2}$$

The first implication comes from Res-*ab*-*ab*′. It holds that $\pi' \models p$ (the locality constraint D13.1 and $\mathsf{Lab}'(v_1, p) = d^+$); the second equivalence is a trivial logical consequence of the fact that variable p is assigned $\top$ by π'.

Res-ab-b': This case violates the assumption of the theorem that $\mathsf{Lab} \preceq \mathsf{Lab}'$. The pivot variable p gets the strongest label b by Lab' in both antecedent vertices (i.e., v_1 and v_2). However, the first labeling function Lab have to assign a weaker label a resp. ab to the pivot variable in at least one of the antecedent vertices; otherwise this case will become Res-b-b'. Variable p, at that vertex, is the witness that the assumption $\mathsf{Lab} \preceq \mathsf{Lab}'$ is violated.

Res-b-a': $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = a$. It means that $I_v \equiv I_{v_1} \wedge I_{v_2}$ and $I'_v \equiv I'_{v_1} \vee I'_{v_2}$.

It holds that:

$$\begin{aligned} \pi, \pi' \models \quad & (I_{v_1} \wedge I_{v_2}) \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\downharpoonright_v \Rightarrow \\ \Rightarrow & (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\downharpoonright_v \Rightarrow \\ \Rightarrow & (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Rightarrow I'_{v_1} \vee I'_{v_2} \end{aligned}$$

The first and third implications are simple logical consequences. The second implication comes from Res-ab-ab'.

Res-b-ab': $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = ab$. It means that $I_v \equiv I_{v_1} \wedge I_{v_2}$ and $I'_v \equiv (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$.

It holds that:

$$\begin{aligned} \pi, \pi' \models \quad & (I_{v_1} \wedge I_{v_2}) \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\downharpoonright_v \Rightarrow \\ \Rightarrow & (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\downharpoonright_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \end{aligned}$$

The first implication is simple logical consequence. The second implication comes from Res-ab-ab'.

Res-b-$d^{+\prime}$: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = d^+$. So, partial vertex-interpolant I_v is defined as $I_v \equiv I_{v_1} \wedge I_{v_2}$. Assume that $\mathsf{Lab}'(v_1, p) = d^+$, thus $I'_v \equiv I'_{v_2}$. The situation is symmetric if $\mathsf{Lab}'(v_2, \overline{p}) = d^+$. We will use Res-$ab$-$ab'$ in the same way as in the above cases.

It holds that:

$$\begin{aligned} \pi, \pi' \models \quad & (I_{v_1} \wedge I_{v_2}) \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\downharpoonright_v \Rightarrow \\ \Rightarrow & (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\downharpoonright_v \Rightarrow \\ \Rightarrow & (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Leftrightarrow I'_{v_2} \end{aligned}$$

The first implication is a simple logical consequence. The second implication comes from Res-ab-ab'. It holds that $\pi' \models p$ (the locality constraint D13.1 and $\mathsf{Lab}'(v_1, p) = d^+$); the second equivalence is a trivial logical consequence of the fact that variable p is assigned $\top$ by π'.

Res-b-b': $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = b$. It means that $I_v \equiv I_{v_1} \wedge I_{v_2}$ and $I'_v \equiv I'_{v_1} \wedge I'_{v_2}$.

The label of pivot p in both antecedent vertices v_1 and v_2 must be b (in both labeling functions), so it is not preserved by the weakened-labels filters $|\!\lfloor_{v_1}$ and $|\!\lfloor_{v_2}$; thus, it holds $\neg\langle p\rangle\!|\!\lfloor_{v_1} \Leftrightarrow \top$.

The same auxiliary implications as in the previous Res-a-a' case hold:

$$\begin{aligned}
\pi, \pi' \models I_{v_1} \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\lfloor_v \Leftrightarrow \neg\langle p\rangle\!|\!\lfloor_{v_1} \wedge I_{v_1} \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\lfloor_v &\overset{L5}{\Rightarrow} \\
\Rightarrow I_{v_1} \wedge S \wedge \neg\langle p, \Theta_1\rangle\!|\!\lfloor_{v_1} &\overset{PIH}{\Rightarrow} I'_{v_1} \\
\pi, \pi' \models I_{v_2} \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\lfloor_v \Leftrightarrow \neg\langle \overline{p}\rangle\!|\!\lfloor_{v_2} \wedge I_{v_2} \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\lfloor_v &\overset{L5}{\Rightarrow} \\
\Rightarrow I_{v_2} \wedge S \wedge \neg\langle \overline{p}, \Theta_2\rangle\!|\!\lfloor_{v_2} &\overset{PIH}{\Rightarrow} I'_{v_2}
\end{aligned}$$

The first equivalence is shown above. The following implication is application of Lemma 5, while the last one is the induction hypothesis.

From the previous implications, the invariant for the vertex v directly follows:

$$\pi, \pi' \models (I_{v_1} \wedge I_{v_2}) \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\lfloor_v \Rightarrow (I'_{v_1} \wedge I'_{v_2})$$

Res-d^+-a': $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = d^+$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = a$. It means that partial vertex-interpolant I'_v is defined as follows: $I'_v \equiv I'_{v_1} \vee I'_{v_2}$. Assume that $\mathsf{Lab}(v_1, p) = d^+$, thus $I_v \equiv I_{v_2}$. The situation is symmetric if $\mathsf{Lab}(v_2, \overline{p}) = d^+$.

It holds that $\pi \models p$ (the locality constraint D13.1 and $\mathsf{Lab}(v_1, p) = d^+$); thus, the following equivalence holds: $\pi \models I_{v_2} \Leftrightarrow (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$. So, the invariant for vertex v can be established using the Res-ab-ab resolution:

$$\begin{aligned}
\pi, \pi' \models \qquad & I_{v_2} \wedge S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\lfloor_v \Leftrightarrow \\
\Leftrightarrow (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge & S \wedge \neg\langle\Theta_1, \Theta_2\rangle\!|\!\lfloor_v \Rightarrow \\
& \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Rightarrow I'_{v_1} \vee I'_{v_2}
\end{aligned}$$

Res-d^+-ab': $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = d^+$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = ab$. It means that partial vertex-interpolant I'_v is defined as follows: $I'_v \equiv (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$. Assume that $\mathsf{Lab}(v_1, p) = d^+$, thus $I_v \equiv I_{v_2}$. The situation is symmetric if $\mathsf{Lab}(v_2, \overline{p}) = d^+$.

It holds that $\pi \models p$ (the locality constraint D13.1 and $\mathsf{Lab}(v_1, p) = d^+$); thus the following equivalence holds: $\pi \models I_{v_2} \Leftrightarrow (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$.

The invariant for vertex v can be established using the Res-*ab*-*ab* resolution:

$$\begin{aligned}\pi, \pi' \models \quad & I_{v_2} \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!\rangle|_v \Leftrightarrow \\ \Leftrightarrow\ & (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!\rangle|_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})\end{aligned}$$

Res-d^+-$d^{+\prime}$: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = d^+$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = d^+$.

This rule splits into two different sub-cases; the pivot variable gets assigned either the same value by π and π' or one assignment assigns $\top$, while the other assignment assigns $\bot$ to variable p.

In the latter case, the invariant for vertex v holds trivially. The assumptions (i.e., assignments) contradict, so any formula, e.g., the invariant, holds. Let us focus on the former case. Assume that $\mathsf{Lab}(v_1, p) = d^+$, thus it holds that $\mathsf{Lab}'(v_1, p) = d^+$ and $I_v \equiv I_{v_2}$, $I'_v \equiv I'_{v_2}$. The situation is symmetric if $\mathsf{Lab}(v_2, \overline{p}) = d^+$.

The invariant for vertex v can be established from the invariant of vertex v_2:

$$\begin{aligned}\pi, \pi' \models I_{v_2} \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!\rangle|_v \Leftrightarrow p \wedge I_{v_2} \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!\rangle|_v \Leftrightarrow \\ \Rightarrow \neg\langle \overline{p} \rangle\!\rangle|_{v_2} \wedge I_{v_2} \wedge S \wedge \neg\langle \Theta_1, \Theta_2 \rangle\!\rangle|_v \overset{L5}{\Rightarrow} \\ \Rightarrow I_{v_2} \wedge S \wedge \neg\langle \overline{p}, \Theta_1 \rangle\!\rangle|_{v_2} \overset{PIH}{\Rightarrow} I'_{v_2}\end{aligned}$$

It holds that $\pi \models p$ (the locality constraint D13.1 and $\mathsf{Lab}(v_1, p) = d^+$); this shows the first equivalence. It holds that $\pi \models \neg\langle \overline{p} \rangle\!\rangle|_{v_2}$; either the filter preserves the literal $\overline{p}$, which means that due to the assignment $\neg\overline{p} \equiv p$ evaluates to $\top$ or the filter removes the literal and the negated empty clause is equivalent to $\top$ without any assumptions. The above reasoning shows the second equivalence.

Note that alternatively, this case can be shown via Res-*ab*-*ab* resolution as well.

Res-d^+-b': This case violates the assumption of the theorem that $\mathsf{Lab} \preceq \mathsf{Lab}'$. Pivot p gets the strongest label b by Lab' in both antecedent vertices (i.e., v_1 and v_2). However, the first labeling function Lab has to assign a weaker label d^+ to the pivot variable in one of the antecedent vertex. Variable p, at that vertex, is the witness that assumption $\mathsf{Lab} \preceq \mathsf{Lab}'$ is violated. $\square$

References

[1] Jiří Adámek, Tomáš Bureš, et al. Component Reliability Extensions for Fractal Component Model. `http://d3s.mff.cuni.cz/projects/formal_methods/ft/`, 2006.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From Under-Approximations to Over-Approximations and Back. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2012*, volume 7214 of *Lecture Notes in Computer Science*, pages 157–172, 2012.

[4] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 39–55, 2012.

[5] Aws Albarghouthi, Yi Li, et al. UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In *Computer Aided Verification - 24th International Conference, CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 672–678, 2012.

[6] Tony Andrews, Shaz Qadeer, et al. Zing: Exploiting Program Structure for Model Checking Concurrent Software. In *CONCUR 2004 - Concurrency Theory, 15th International Conference*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15, 2004.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, Cambridge, Mass, 2008.

[8] Omer Bar-Ilan, Oded Fuhrmann, et al. Reducing the size of resolution proofs in linear time. *STTT*, 13(3):263–272, 2011.

[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[10] Armin Biere, Alessandro Cimatti, et al. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207, 1999.

[11] Roderick Bloem, Sharad Malik, et al. Reduction of Resolution Refutations and Interpolants via Subsumption. In *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC*, volume 8855 of *Lecture Notes in Computer Science*, pages 188–203, 2014.

[12] Joseph Boudou and Bruno Woltzenlogel Paleo. Compression of Propositional Resolution Proofs by Lowering Subproofs. In *Automated Reasoning with Analytic Tableaux and Related Methods - 22th International Conference, TABLEAUX*, volume 8123 of *Lecture Notes in Computer Science*, pages 59–73, 2013.

[13] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. State Space Reduction Based on Live Variables Analysis. In *Static Analysis, 6th International Symposium, SAS '99*, volume 1694 of *Lecture Notes in Computer Science*, pages 164–178, 1999.

[14] Marius Bozga, Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Protocol Verification with the ALDÉBARAN Toolset. *STTT*, 1(1-2):166–184, 1997.

[15] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87, 2011.

[16] Aaron R. Bradley. Understanding IC3. In *Theory and Applications of Satisfiability Testing - SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 1–14, 2012.

[17] Eric Bruneton, Thierry Coupaye, et al. An Open Component Model and Its Support in Java. In *Component-Based Software Engineering, 7th International Symposium, CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22, 2004.

[18] Randal E. Bryant. Symbolic manipulation of Boolean functions using a graphical representation. In *22nd ACM/IEEE conference on Design automation, DAC*, pages 688–694, 1985.

[19] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[20] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[21] Lubomír Bulej, Tomáš Bureš, et al. CoCoME in Fractal. In *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, pages 357–387, 2007.

[22] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Software Engineering, Research, Management and Applications (SERA)*, pages 40–48, 2006.

[23] Gianpiero Cabodi, Paolo Camurati, et al. Reducing Interpolant Circuit Size by Ad-Hoc Logic Synthesis and SAT-Based Weakening. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 25–32. IEEE, 2016.

[24] Gianpiero Cabodi, C. Loiacono, and D. Vendraminetto. Optimization techniques for Craig Interpolant compaction in Unbounded Model Checking. In *Design, Automation and Test in Europe, DATE 13*, pages 1417–1422. EDA Consortium, 2013.

[25] Gianpiero Cabodi, Marco Murciano, et al. Stepping forward with Interpolants in Unbounded Model Checking. In *International Conference on Computer-Aided Design, ICCAD 2006*, pages 772–778. ACM, 2006.

[26] Hana Chockler, Alexander Ivrii, and Arie Matsliah. Computing Interpolants without Proofs. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC*, volume 7857 of *Lecture Notes in Computer Science*, pages 72–85, 2012.

[27] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *ACM Transactions on Computational Logic*, 12(1):7, 2010.

[28] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[29] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[30] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, 2004.

[31] James C. Corbett, Matthew B. Dwyer, et al. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE*, pages 439–448, 2000.

[32] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *International Conference on Automated Software Engineering, ASE*, pages 137–148, 2009.

[33] Scott Cotton. Two Techniques for Minimizing Resolution Proofs. In *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 306–312, 2010.

[34] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[35] William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[36] Concurrency Tool Comparison repository. `https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison`.

[37] Niels H. M. Aan de Brugh, Viet Yen Nguyen, and Theo C. Ruys. MoonWalker: Verification of .NET Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2009*, volume 5505 of *Lecture Notes in Computer Science*, pages 170–173, 2009.

[38] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A Dynamic Extension of SPIN. In *Theoretical and Practical Aspects of SPIN Model*

Checking, 5th and 6th International SPIN Workshops, volume 1680 of *Lecture Notes in Computer Science*, pages 261–276, 1999.

[39] Bruno Dillenseger. CLIF, a framework based on Fractal for flexible, distributed load testing. *Annales des Télécommunications*, 64(1-2):101–120, 2009.

[40] Yifei Dong and C. R. Ramakrishnan. *An Optimizing Compiler for Efficient Model Checking*, pages 241–256. Springer US, 1999.

[41] Vijay D'Silva, Daniel Kroening, et al. Interpolant Strength. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145, 2010.

[42] Niklas Eén. Cut Sweeping. Technical Report CDNL-TR-2007-0510, Cadence Technical Report, 2017.

[43] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75, 2005.

[44] Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. eVolCheck: Incremental Upgrade Checker for C. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 292–307, 2013.

[45] Jean-Claude Fernandez, Marius Bozga, and Lucian Ghirvu. State Space Reduction based on Live Variables Analysis. *Sci. Comput. Program.*, 47(2-3):203–220, 2003.

[46] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Sci. Comput. Program.*, 29(1-2):123–146, 1997.

[47] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 110–121, 2005.

[48] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of Propositional Resolution Proofs via Partial Regularization. In *Automated Deduction - CADE-23 - 23rd International Conference*

on Automated Deduction, volume 6803 of *Lecture Notes in Computer Science*, pages 237–251, 2011.

[49] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[50] Patrice Godefroid and Didier Pirottin. Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract). In *Computer Aided Verification, 5th International Conference, CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449, 1993.

[51] Patrice Godefroid and Pierre Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Computer Aided Verification, 3rd International Workshop, CAV*, volume 575 of *Lecture Notes in Computer Science*, pages 332–342, 1991.

[52] Alberto Griggio, Thi Thieu Hoa Le, and Roberto Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 20011*, volume 6605 of *Lecture Notes in Computer Science*, pages 143–157, 2011.

[53] Ashutosh Gupta. Improved Single Pass Algorithms for Resolution Proof Reduction. In *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 107–121, 2012.

[54] Arie Gurfinkel, Simone Fulvio Rollini, and Natasha Sharygina. Interpolation Properties and SAT-Based Model Checking. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 255–271, 2013.

[55] Zyad Hassan, Yan Zhang, and Fabio Somenzi. A Study of Sweeping Algorithms in the Context of Model Checking. In *First International Workshop on Design and Implementation of Formal Tools and Systems*, volume 832 of *CEUR Workshop Proceedings*, 2011.

[56] Gerard J. Holzmann. An Improved Protocol Reachability Analysis Technique. *Software: Practice and Experience*, 18(2):137–161, 1988.

[57] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[58] Gerard J. Holzmann. State Compression in SPIN: Recursive Indexing And Compression Training Runs. In *Proceedings of Third International SPIN Workshop*, 1997.

[59] Gerard J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.

[60] Gerard J. Holzmann. The Engineering of a Model Checker: The Gnu i-Protocol Case Study Revisited. In *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 232–244, 1999.

[61] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[62] Gerard J. Holzmann and Anuj Puri. A Minimized Automaton Representation of Reachable States. *STTT*, 2(3):270–278, 1999.

[63] D.A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

[64] Radu Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *16th IEEE International Conference on Automated Software Engineering, ASE*, pages 254–261, 2001.

[65] Pavel Jancík, Leonardo Alt, Grigory Fedyukovich, Antti E. J. Hyvärinen, Jan Kofron, and Natasha Sharygina. PVAIR: Partial Variable Assignment InterpolatoR. In *Fundamental Approaches to Software Engineering FASE, part of the ETAPS*, volume 9633 of *Lecture Notes in Computer Science*, pages 419–434, 2016.

[66] Pavel Jancík and Jan Kofron. Dead Variable Analysis for Multi-threaded Heap Manipulating Programs. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016*, pages 1620–1627, 2016.

[67] Pavel Jančík and Jan Kofroň. On Partial State Matching. *Formal Aspects of Computing*, pages 1–27, 2017.

[68] Pavel Jancik, Jan Kofron, Simone Fulvio Rollini, and Natasha Sharygina. On Interpolants and Variable Assignments. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 123–130, 2014.

[69] Pavel Jancik, Pavel Parízek, and Jan Kofroň. Advanced Debugging with JPF Inspector. In *Local proceedings of Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 43–50, 2011.

[70] Pavel Jancik, Pavel Parízek, and Jan Kofroň. BeJC: Checking Compliance Between Java Implementation and Behavior Specification. In *Proceedings of the 17th International Doctoral Symposium on Components and Architecture*, WCOP '12, pages 31–36, 2012.

[71] Bob Jenkins. Algorithm Alley: Hash Functions. *"Dr. Dobb's Journal of Software Tools"*, 22(9):107–109, 115–116, September 1997.

[72] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[73] Jan Krajicek. Interpolation Theorems, Lower Bounds for Proof Systems, and Independence Results for Bounded Arithmetic. *J. Symb. Log.*, 62(2):457–486, 1997.

[74] Daniel Kroening and Georg Weissenbacher. Verification and falsification of Programs with Loops Using Predicate Abstraction. *Formal Aspect of Computing*, 22(2):105–128, 2010.

[75] Andreas Kuehlmann. Dynamic Transition Relation Simplification for Bounded Property Checking. In *International Conference on Computer-Aided Design, ICCAD*, pages 50–57, 2004.

[76] Andreas Kuehlmann and Florian Krohm. Equivalence Checking Using Cuts and Heaps. In *Proceedings of the 34st Conference on Design Automation*, pages 263–268, 1997.

[77] Micah Lewis and Michael Jones. A Dead Variable Analysis for Explicit Model Checking. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 48–57, 2006.

[78] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13, 2003.

[79] Kenneth L. McMillan. An Interpolating Theorem Prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[80] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Computer Aided Verification, 18th International Conference, CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136, 2006.

[81] Eric Mercer and Michael D. Jones. Model Checking Machine Code with the GNU Debugger. In *Model Checking Software, 12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265, 2005.

[82] MURPHI Model Checker. `http://formalverification.cs.utah.edu/Murphi/`.

[83] Madanlal Musuvathi and David L. Dill. An Incremental Heap Canonicalization Algorithm. In *Model Checking Software, 12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 28–42, 2005.

[84] Viet Yen Nguyen and Theo C. Ruys. Memoised Garbage Collection for Software Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 201–214, 2009.

[85] Pavel Parizek and Pavel Jancik. Approximating Happens-Before Order: Interplay between Static Analysis and State Space Traversal. In *Model Checking of Software, SPIN*, pages 1–10, 2014.

[86] Pavel Parizek and Ondrej Lhoták. Identifying Future Field Accesses in Exhaustive State Space Traversal. In *International Conference on Automated Software Engineering, ASE*, pages 93–102, 2011.

[87] Pavel Parizek and Ondrej Lhoták. Model Checking of Concurrent Programs with Static Analysis of Field Accesses. *Science of Computer Programming*, 98, Part 4:735–763, 2015.

[88] Doron Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *Computer Aided Verification, 6th International Conference, CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390, 1994.

[89] Parallel Java Benchmarks. `https://bitbucket.org/pag-lab/pjbench`.

[90] Tomas Poch, Ondrej Sery, Frantisek Plasil, and Jan Kofron. Threaded Behavior Protocols. *Formal Aspects of Computing*, 25(4):543–572, 2013.

[91] Pavel Pudlák. Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[92] Simone Rollini, Roberto Bruttomesso, and Natasha Sharygina. An Efficient and Flexible Approach to Resolution Proof Reduction. In *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC*, volume 6504 of *Lecture Notes in Computer Science*, pages 182–196, 2010.

[93] Simone Fulvio Rollini, Leonardo Alt, et al. PeRIPLO: A Framework for Producing Effective Interpolants in SAT-Based Software Verification. In *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR-19*, volume 8312 of *Lecture Notes in Computer Science*, pages 683–693, 2013.

[94] Simone Fulvio Rollini, Roberto Bruttomesso, et al. Resolution Proof Transformation for Compression and Interpolation. *Formal Methods in System Design*, 45(1):1–41, 2014.

[95] Simone Fulvio Rollini, Roberto Bruttomesso, Natasha Sharygina, and Aliaksei Tsitovich. Resolution Proof Transformation for Compression and Interpolation. *Formal Methods in System Design*, 45(1):1–41, 2014.

[96] Simone Fulvio Rollini, Ondrej Sery, and Natasha Sharygina. Leveraging Interpolant Strength in Model Checking. In *Computer Aided Verification - 24th International Conference, CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 193–209, 2012.

[97] SAT Competition. `http://www.satcompetition.org/`.

[98] Matthias Schlaipfer and Georg Weissenbacher. Labelled Interpolation Systems for Hyper-Resolution, Clausal, and Local Proofs. *Journal of Automated Reasoning*, 57(1):3–36, 2016.

[99] Joel P. Self and Eric G. Mercer. On-the-Fly Dynamic Dead Variable Analysis. In *Model Checking Software, SPIN Workshop*, volume 4595 of *Lecture Notes in Computer Science*, pages 113–130, 2007.

[100] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization. In *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 203–207, 2012.

[101] O. Tange. GNU Parallel - The Command-Line Power Tool. *The USENIX Magazine*, 36(1):42–47, Feb 2011.

[102] Stefano Tonetta. Abstract Model Checking without Computing the Abstraction. In *FM 2009: Formal Methods*, volume 5850, pages 89–105. Springer, 2009.

[103] Grigori S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II, Volume 8 of Seminars in Mathematics, V. A. Steklov Mathematical Institute*, Leningrad, 1969. Consultants Bureau.

[104] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515, 1989.

[105] Willem Visser. Memory Efficient State Storage in SPIN. In *Proceedings of Second International SPIN Workshop*. American Mathematical Society, 1996.

[106] Willem Visser, Klaus Havelund, et al. Model Checking Programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[107] Yakir Vizel and Orna Grumberg. Interpolation-Sequence based Model Checking. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 1–8, 2009.

[108] Pierre Wolper and Denis Leroy. Reliable Hashing without Collosion Detection. In *Computer Aided Verification, 5th International Conference, CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70, 1993.

[109] Karen Yorav and Orna Grumberg. Static Analysis for State-Space Reductions Preserving Temporal Logics. *Formal Methods in System Design*, 25(1):67–96, 2004.