Static Analysis of mixed-granularity Locking Schemes using Abstract Interpretation

LUCA THOMAS, TU Braunschweig, Germany

Low-level programs in the context of heavily parallelized software architecture, commonly found in operating systems and device drivers, are notoriously hard to reason about and debug for humans. Locks can interact on multiple dependance levels, can be dynamically allocated at locations not statically known and can depend on the values of variables or relative array indices. This paper presents an offline approach to soundly analyzing complex interleaved locking schemes frequently used in the Linux kernel and associated device drivers using abstract semantics. The system is parametric in the analyses used to reduce false positives, so handling of yet unsupported synchronization patterns and concepts can be added in the future. The main reference is the work of Vojdani, Apinis, Rõtov, Seidl, Vene and Vogler [10], which was realized using the Goblint static analysis tool by people at the Technical University of Munich and University of Tartu. We will explore practical code examples in C, the mathematical foundations of the analyzer, the modeling and abstraction process, how the implementation combines these concepts and what limitations exist in practical application.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Software Safety**.

Additional Key Words and Phrases: Race Detection, Static Analysis, Concurrency, Abstract Interpretation

ACM Reference Format:
Luca Thomas. 2026. Static Analysis of mixed-granularity Locking Schemes using Abstract Interpretation. *J. ACM* 1, 1, Article 1 (February 2026), 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Static analysis is the process of reasoning about the semantic space of a program without the need of actually running them. Sound safety guarantees are possible. As opposed to that, dynamic analyses are not able to prove the absence of bugs. A program, regardless of the language used, can be modeled as a set of memory states, transitions between them that are defined by fundamental operations that all other operations can be derived from (assignments, mathematical operations, branchings, boolean logic, etc.), the instruction pointer indicating the current program point and some set of valid initial configurations. Programs, although people like to hide this principle behind all kinds of different programming paradigms, are fundamentally data transformers. Procedural instructions are performed on some input to produce the desired output.

If we widen this model to concurrent programs, transitions between states can now be taken non-deterministically at any time. Because program correctness depends on an ordering of instructions, the operations of different threads need to be synchronized if they interact. A lack of such synchronizations leads to so-called *race conditions*, which have always been one of the most subtle and hard to debug types of bugs in multi-threaded software. They occur when some data is accessed by more than one thread at the same time and at least one of those is a mutating access. Race detection tools like Goblint try to verify their absence by analyzing the semantic domain of a program.

Author's Contact Information: Luca Thomas, TU Braunschweig, Germany, luca.thomas@tu-braunschweig.de.

ACM 2475-1421/2026/2-ART1
https://doi.org/XXXXXXX.XXXXXXX

Device drivers provide a standardized interface to the operating system for interacting with the corresponding hardware. It typically consists of file operations, memory mapping and event polling [1]. These functions can be called by the operating system at any time by any thread and are expected to already handle synchronization concerns of their shared data correctly. That includes threaded access and hardware interrupt handling.

2 Background

Linked list implementations used in the Linux Kernel do not allocate individual nodes for every list. Instead, they allow for the same node to be part of multiple lists at the same time by placing the list headers inside the container struct of the payload, not the other way around. We can access the base struct by subtracting the byte offset of the field from the field pointer. In the kernel implementation this is done with the `container_of` macro. Nodes in different lists are backed by the same allocation and can be compared using the result of `container_of` to determine their memory region. Example 1, that we will explore in Section 3.2, contains a list structure similar to the one used in the kernel.

2.1 Locking Schemes and Challenges

Now imagine that multiple threads operate on such lists. Because the lists only exist due to connections between the containers, data from the same container can be relevant for different lists. That means that, if access to different lists is limited to different threads, fields of the same struct will be protected by separate locks. It might be the case, that different threads operate on different parts of the same list simultaneously. This leads to locking schemes of varying granularity – access to data is not just protected by a single lock, but through the interaction between locks on different levels. That means: *race freedom* might only hold as a result of logic-dependant limitations, although the space of valid configurations in the program would allow for them. Therefore, we have to analyze and approximate the reachable configurations of a program. This is done using so-called *abstract interpretation*, a concept that will be introduced later in Section 2.3 and that is the foundational theoretical concept that Goblint is built upon.

There are a lot of obstacles in our way when trying to find ways to validate concurrent data access in a low-level programming language like C that allows for complex manual memory management schemes like the linked lists we discussed. Traditional shape analysis [5] is not helpful, because objects do not belong to well-defined heap regions. The list nodes are dynamically allocated, which means that the memory locations of the locks are not statically known. Additionally, the correspondence of locks and protected data might be dependant on runtime values like array indices or hash table keys. We will track which of them *may* alias, and what locks *must* be held during access [10]. The latter are tracked using *locksets* that correspond to a program point or memory location. This is non-trivial as the values of variables and locks held during access might be interlinked. That leads to the possibility of state invariants and race freedom only being able to exist together, because races might be invariant-dependant and invariants race-freedom-dependant; they have to be inferred at the same time and result in constraints that are mutually recursive in their definition.

When trying to reason about conditional locking, the number of possible branching combinations is also a problem. The number of execution paths grows exponentially in the number of branchings. Our analysis has to account for that while being computationally feasible. Goblint only processes a "safe" subset of C that excludes, for example, arbitrary pointer arithmetic. The assumption is also that locking follows the commonly used idiom that potentially racing accesses to a variable are at least protected by a single common lock [9]. More complex idioms, such as requiring that all accesses to a global have pair-wise overlapping locksets, are theoretically possible to check for with the computed lockset information, but not our primary concern in this paper.

2.2 Memory Consistency Models

There is a semantic gap between the mentioned program models and the actual operations of the real CPU that executes the program. CPU threads use caches of different levels that need to be synchronized with each other, main memory (the source of truth) and with the registers that are used for computations. Other threads might be accessing the cached data that is modified. That is how the race-conditions previously mentioned happen; loads and stores in our theoretical model do not correspond to atomic instructions on a real CPU.

The memory consistency model is the specification of when/how memory operations become visible to other threads. Some consistency models give more guarantees than others. This is relevant for our analysis, because memory consistency models further constrain the amount of cases that have to be accounted for. This is, however, architecture-dependant and operating systems like Linux that are designed to work on a wide variety of CPUs are unable to give strong guarantees. Linux only provides *release consistency* in its memory model [2]. That means that acquires and releases are the only operations that deterministically control visibility of memory operations to other threads. They are one-way-barriers in terms of when an instruction is considered "complete" and thus readable by other threads. All prior writes become visible before a release is visible and no following reads/writes will happen before an acquire is visible. Memory operations before those instructions might be executed after them. Modern CPUs often do real-time instruction re-ordering and speculative caching for performance reasons while adhering to the constraints of the memory consistency model. In our model, we need to handle these cases adequately. Synchronization operations of the memory subsystem should be allowed to run at any time. Acquires/releases of locks need to invalidate/flush cached memory locations. Advanced synchronization patterns, like atomic operations, are not supported, but theoretically could be if the required abstract domains and transfer functions are provided, because the system is parametric in the analyses used.

2.3 Abstract Interpretation

In order to reason about the semantic space of a program, we do not want to concern ourselves with concrete language-specific syntax and properties of the program text source. We want to obtain a universal understanding of a program and approximate its behavior based on an abstraction of that model. Using that approximation, we can then make sound statements about the behavior of the program – specifically about a condition/invariant that is supposed to hold during execution. In the field of program verification there are different categories of properties one could investigate: There are structural properties of the control flow graph are about reachability of "bad" states and temporal properties under some behavioral assumptions of the execution environment, often in the context of concurrent programs, like program termination under certain scheduling assumptions. In our case, we care about the reachability of states, where some memory location is accessed by more than one thread without a common lock held. Our abstraction is *property-directed* [4], meaning that we are not interested in explicit states and soundness wrt. state transitions, but soundness wrt. the property holding when taking abstract transitions. That important detail manifests itself in our abstract domain and transition relation and will be discussed in Section 3.1.

There are a few components that are required for this analysis. First, we need to create an abstract representation, almost like a categorization, of the elements of the domain of the program. Then, the transition relation between the states needs to be simulated, such that, when the result is finally transformed back to the concrete domain, it is sound. Dealing with false positives is left to the algorithm using this concept. The pair of abstraction and concretization functions, α and γ, is called a Galois connection iff they are monotonic and they satisfy $c \sqsubseteq \gamma(\alpha(c)) \wedge \alpha(\gamma(a)) \sqsubseteq a$ with $a \in A$ and $c \in C$ for an abstract domain A and a concrete domain C. Now let f be a function that operates on

C. If programs are well-structured, there is a simulation relation wrt. f that operates on the abstract domain. We call it $f^\sharp$. In our approximation of the reachable states we want to compute a fixpoint of the transition relation; an *inductive invariant* of the program. An inductive invariant is a set of program states that contains the initial configurations and is closed under taking transitions. For a sound over-approximation the simulation and the Galois connection need to be conditioned such that $\mathrm{lfp}.f \sqsubseteq \gamma\big(\mathrm{lfp}.f^\sharp\big)$. We can then perform a fixpoint transfer over the prefix points of a monotonic function using Knaster-Tarski [8]. That means that we essentially want to prove that $\gamma\big(\mathrm{lfp}.f^\sharp\big)$ is an inductive invariant for a sensible lifting of f that is based on the abstract interpretation of program commands. One can imagine a hostile environment, where all possible actions are taken.

3 Formal Model of the Problem

So-called thread templates $t \in \mathcal{T}$ denote the function-local control flow graphs a thread can execute. Issuing and completion of instructions inside of a template follows the consistency model. A thread instance $i \in \mathcal{I} = \mathcal{T} \times \mathbb{N}$ is identified by its template and an index. The control flow graphs are modeled as the non-deterministic transition systems mentioned in Section 1 containing configuration nodes. Transition edges between them are labeled with the commands that resemble the corresponding concrete program instructions. The semantics of a command s is given by a state mapping $[\![s]\!] : \mathcal{D} \longrightarrow \mathcal{D}$. A memory state in $\mathcal{D} = \mathcal{L} \longrightarrow \mathbb{N}$ is an address space mapping to integer values. A current configuration $d_j = \big(\vec{u}, \vec{\sigma}, \vec{\mu}, \vec{w}, \varphi\big)$ is defined by each thread's program counter $u_i \in N$, local privatized memory state $\sigma_i : \mathcal{D}$, currently held locks $\mu_i \in 2^{\mathcal{L}}$ and cached dirty memory locations $w_i \in 2^{\mathcal{L}}$ and the state of global memory $\varphi : \mathcal{D}$. The initial configuration of a thread i is $d_{0i} = (n_t, \varphi_0, \emptyset, \emptyset, \varphi_0)$. n_t denotes the start node of a thread template t. For the sake of simplicity, for now, we reduce the memory state $\mathcal{D}$ to integer variable assignments and local variables are treated as thread-local globals. Keep in mind that this model is not used for actual analysis, but is required for the abstraction in Section 3.1 that illustrates how Goblint performs its analyses. [10]

Labeled transition edges between program points.	$(u, l, v) \in E$
The grammar of transition labels.	$l ::= s \mid \mathrm{lock}(m) \mid \mathrm{unlock}(m)$
Control flow graph of the program.	$G_t = (N, E, n_t)$
Reads and writes of a command.	$[\![s]\!]_a : \mathcal{D} \longrightarrow 2^{\mathcal{L}} \mid a \in \{r, w, rw\}$

Table 1. Additional definitions for the mathematical model [10].

At any moment during execution any thread $i \in \mathcal{I}$ can update the state. The usable transition rules consist of the basic instructions and branchings that correspond to the label grammar in Table 1:

Statements Regular commands modify the thread-local memory, so $\sigma_i' = [\![s]\!]\sigma_i$ and modified locations are marked as dirty: $w_i' = w_i \cup [\![s]\!]_w \sigma_i$.

Aquire If the lock m is available, meaning $m \notin \bigcup_j \mu_j$, we set $\mu_i' = \mu_i \cup \{m\}$ and the rule **Invalidate** is triggered.

Release If the lock is held, meaning $m \in \mu_i$, we release the lock $\mu_i' = \mu_i \setminus \{m\}$ and all pending writes are flushed according to **Flush**.

and the memory subsystem operations that synchronize local cache with main memory according to the consistency model and that were mentioned before [10]:

Flush Modified locations $x \in w_i$ are flushed to main memory such that $\varphi' = \sigma_i(x)$ and $w_i' = w_i \setminus \{x\}$.

Invalidate Clean locations $x \notin w_i$ are invalidated. The value in the cache is immediately updated, so $\sigma_i'(x) = \varphi(x)$.

The latter could be executed by the environment at any time in addition to the automatic triggers. A race is detected at some point, when two threads may both issue instructions accessing some location $l \in \mathscr{L}$, meaning $(u_i, s_1, \_) \in E \wedge (u_j, s_2, \_) \in E$ with $l \in [\![s_1]\!]_{rw}(\sigma_i) \cap [\![s_2]\!]_{rw}(\sigma_j)$ [10]. For simplicity's sake, we do not distinguish between reading and writing accesses in the actual analysis; another form of over-approximation. Deadlocks on reacquisitions of the same lock and failures on releases of a non-held lock are accounted for in the definitions to respect the behavior of the Linux kernel previously discussed in Section 2.2.

3.1 Abstraction

Concrete Model		**Abstract Analysis**	
$\vec{u} : \mathscr{T} \longrightarrow N$	each thread's program point	$u \in N$	(contextual) program points
$\vec{\sigma} : \mathscr{T} \longrightarrow \mathscr{D}$	each thread's memory view	$\psi : N \longrightarrow \mathbb{D}$	privatized state mapping
$\vec{\mu} : \mathscr{T} \longrightarrow 2^{\mathscr{L}}$	each thread's locks held	$\lambda : N \longrightarrow 2^{\mathscr{M}_s}$	(symbolic) locks definitely held
$\vec{w} : \mathscr{T} \longrightarrow 2^{\mathscr{L}}$	each thread's pending writes	$\Lambda : \mathbb{G} \longrightarrow 2^{\mathscr{M}_r}$	common (relative) locks held
$\varphi : \mathscr{D}$	main memory	$\Psi : \mathbb{D}$	global invariant

Table 2. Domain table of the model as in [10] (not a 1:1 mapping of the domains).

Naively trying to use the formal model is computationally infeasible. Even for two threads there is an exponential growth of interleavings by the number of statements. Therefore, we abstract the model according to Section 2.3 and over-approximate the semantic space of the program soundly using a lifting $[\![s]\!]^\sharp : \mathbb{D} \longrightarrow \mathbb{D}$ of the semantic meaning of the commands to abstract domains that is monotonic and allows us to make statements about the effects the commands have on the properties we care about. $\mathbb{D}$ is the lattice of abstract states. This corresponds to the abstraction function α discussed in Section 2.3. It helps us deal with complex control flow, path-sensitivity, resolving questions about the ownership of memory locations and respecting arbitrary calling contexts.

The Goblint analyzer is built upon a number of foundational concepts [3, 6, 7, 9]. They provide the algorithmic framework and analyses that are used together to check for race freedom. All threads contribute to a single global constraint system whose fixpoint simultaneously over-approximates the behavior of the entire program. There is some room for thread-modularity; we can identify the side-effect that each thread has on the rest of the program independently [9].

In order to know how to alter the global invariants, we need to compute a local invariant for each program point in the abstract transition system. Program points in the abstract CFG theoretically carry a calling context that is relevant to the performed transitions. Function calls create new such contexts but also potentially alter the caller's context. We do not have to concern ourselves with this, because we assume $\mathbf{entry}_e$ and $\mathbf{combine}_e$ operators as defined in [7]. They fork local states and merge the effect of the called functions with the state of the caller. Conceptually, this is similar to semantically inlining the functions. In C, function calls can also happen through pointers; multiple functions might have to be over-approximately analyzed and have the results joined at the call site.

If V is the set of maintained constraint variables representing the program properties (locksets and state invariants), the effects of taking abstract transitions in the control flow graph can be modeled as constraints in the form of $x \leftarrow f$ that operate on the state via functions $f : (V \longrightarrow \mathbb{D}) \longrightarrow \mathbb{D}$ that modify $x \in V$ via join-operations. A constraint variable assignment $\sigma : V \longrightarrow \mathbb{D}$ is a solution

of the system for all constraints if $\sigma(x) \sqsupseteq f(\sigma)$ [3]. A trivial solution is $\sigma(x) := \top$, but remember that we want to compute the *least fixpoint*. Still, transitions also influence the global state and can be described by a function $\mathbf{trans}_e : \mathbb{D}_{\text{local}} \times \mathbb{D}_{\text{global}} \longrightarrow \mathbb{D}_{\text{local}} \times \mathbb{D}_{\text{global}}$. They can not be expressed as a regular constraint system with finite variable dependencies [3]. The constrains have to account for that [7] and are described as functions in the form of:

$$f : \left((V \longrightarrow \mathbb{D}_{\text{local}}) \times \mathbb{D}_{\text{global}}\right) \longrightarrow \left(\mathbb{D}_{\text{local}} \times \mathbb{D}_{\text{global}} \times 2^V\right).$$

They are *side-effecting* in that they influence subsequent runs by mutating state relevant to other invariants. Any property one might care about can be solved for by expressing it in this way [3]. They take in the previous local invariant and a global state and return the updated local and global state. Affected constraints, the third return value, have to be reevaluated. Intra-procedural constraint systems for program points are turned into side-effecting inter-procedural systems to achieve context-sensitivity [3]. This leads to a worklist-based algorithm design; the constraints that require reevaluation are continuously re-enqueued until a fixpoint is reached.

The Goblint analyzer [9] uses a general constraint solver like this to compute a fixpoint on Ψ and the lockset information Λ. It is general in that it can be redefined what constitutes a domain. The abstract domain can be arbitrarily complex. Goblint allows for the users to extend the framework by custom analyses provided that they supply the required lattice and operators. This means that the system is also parametric, e.g., in the kind of heap abstraction used (how memory regions are defined, what the values are, etc.). One has to define how an edge transition transforms the state of a node. Based on that the constraint system is generated [9]. In our case, we concern ourselves with value invariants and lockset invariants. Note that the order of the lockset lattice is reversed in a must-analysis. There are two functions that we will use to operate on the global invariants. We also need the auxiliary function prot_Λ to find the globals are protected by a lockset λ according to Λ.

$$\text{prot}_\Lambda(\lambda) = \{x \in \mathbb{G} \mid \lambda \cap \Lambda(x) \neq \emptyset\} \qquad \text{mf}_s(\psi, \lambda)(x) = \begin{cases} \lambda & \text{if } x \in [\![s]\!]^\sharp_{rw}(\psi) \\ \bot & \text{otherwise} \end{cases}$$

$$\text{sync}_s(\lambda, \Lambda, \psi, \psi')(x) = \begin{cases} \psi'(x) & \text{if } x \in [\![s]\!]^\sharp_{rw}(\psi) \wedge x \notin \text{prot}_\Lambda(\lambda) \\ \bot & \text{otherwise} \end{cases}$$

#	Constraint	Description
(1)	$(\lambda_v, \psi_v) \sqsupseteq [\![s]\!]^\sharp(\lambda_u, \psi_u)$	Lockset and state lattices respect the abstract semantics of commands.
(2)	$\left(\lambda_{n_t}, \psi_{n_t}\right) \sqsupseteq (\top, \psi_0)$	The initial state is properly composed and the lockset at that point is $\top = \emptyset$.
(3)	$\Lambda \sqsupseteq \text{mf}_s(\lambda_u, \psi_u)$	The lockset always held when accessing some data is a subset of the lockset held at during an individual access.
(4)	$\Psi \sqsupseteq \text{sync}_s(\lambda_u, \Lambda, \psi_u, \psi_u)$	When flushing local changes to global state, the global invariant only ascends in the lattice.
(5)	$\psi_u \sqsupseteq \text{sync}_s(\lambda_u, \Lambda, \psi_u, \Psi)$	When invalidating the local cache with updated global data, the local invariant only ascends in the lattice.

Table 3. Effects of the constraints induced by the consistency model [10].

mf_s updates the global lockset map Λ based on the accesses of a global. The abstraction of the memory operations is given by the function sync_s. It is used to modify the state invariants – as pre-

viously mentioned, only synchronization operations are required to change the global invariants outside of protected sections as we do not care about the details of intra-thread-semantics. That is why configurations might occur that are practically unreachable [10]. These definitions of our execution environment induce a set of constraints. They specify how the domains interact with the operators we just defined. Table 3 contains an overview of them with their respective description.

Note that these constraints map onto the definition of constraint functions introduced earlier of the form $x \leftarrow f$. If the result of the evaluation of a constraint is $\sqcup$-ed with the invariants, $\leftarrow$ is effectively a $\sqsupseteq$. The functions $[\![s]\!]^\sharp$, mf_s and sync_s, depending on what input parameters are given, transform the (global) invariants in different ways and potentially require reevaluation of each other. Also note that some of the functions that are used in the constraint definitions are higher-order functions, because they specify the constraints for each program point.

3.2 Analysis

```
typedef struct {
  Head *next;
} Head;

typedef struct {
  Head a, b;
  int32_t data[2];
  mutex_t ms[2];
} Node;

Head l_a, l_b;
mutex_t m_a, m_b;
bool cond = true;
```

```
void violate_invariant(void) {
  lock(&m_a);
  Head *h = l_a.next;
  Node *n = container_of(h,Node,"a");
  lock(&n->ms[0]);
  lock(&n->ms[1]);
  n->data[0] = 6; // violation!
  n->data[1] = 7; // violation!
  n->data = {0, 0};
  unlock(&n->ms[1]);
  unlock(&n->ms[0]);
  unlock(&m_a);
} // This is thread template 1.
```

```
void verify_invariant(int i) {
  lock(&m_b);
  Head *h = l_b.next;
  Node *n = container_of(h,Node,"b");
  if (cond) lock(&n->ms[i]);
  if (n->data[i] == 0) {
    if (cond) unlock(&n->ms[i]);
  } else {
    unlock(&n->ms[i]);
    n->data[i] = -1; // race!
  }
  unlock(&m_b);
} // This is thread template 2.
```

Example 1. Complex locking scheme with interlinked value and lock invariants.

Now we want to find value invariants to prove the lockset analysis to be conclusive and find lockset invariants to prove the value invariants to be conclusive. We narrow down what locks are always held when accessing relevant data and how the data behaves. Consider Example 1. Let's assume static knowledge about at least one element being in each list, the first one in each corresponding to the same node container and the input index `i` being valid. Still, there is a lot of convolution when only concerning ourselves with individual variables and locks as a result of different heap regions, field accesses and array indices. One can imagine that this is still a minimal example and that the complexity can rise indefinitely when dealing with arbitrary lists, because of a combinatorial explosion of values to consider.

We broaden our concept of variable names that map to memory locations to a set $\mathbb{G}$ of static identifiers. These *owners* represent disjoint regions of shared memory. Both owner expressions and lock expressions in the program code are considered in a symbolic representation, that encodes their universal semantic meaning independently of concrete syntax. Owner expressions are representatives of their may-alias equivalence class and lock expressions are representatives of their must-alias equivalence class. That means that on a `lock` we can add the lock expressions based on must-equality to the current lockset and on an `unlock` we can remove the locks of the entire may-alias equivalence class of owners using a single abstract value. [10]

Somehow, we have to derive these expression representatives from the source program. Goblint employs a wide variety of different analyses here. They include heap-region analysis [6], value abstraction, pointer/integer must-alias analysis, type information and constant propagation. Abstract

values could, e.g., be constants, (partial) ranges or sets. Two regions are part of the same equivalence class, if they can reach each other through iterated pointer and field access [6]. We can combine these different types of information and their resulting memory partitions in a product lattice that preserves soundness. That way we can differentiate between accesses to, e.g., struct fields or array indices. That means the domain of abstract owners looks a lot more like $\mathbb{G} = R \times \mathrm{T} \times F \times I \times \cdots$ (regions, types, fields, symbolic indices, ...). Individual analyses may indicate a race at a struct field, but considering multiple domains shows that concurrent accesses operate on different locations.

If we want to match arbitrary expressions containing information of all of these domains to equivalence classes efficiently, we have to be able to express relative dependencies. By establishing this form of "communication" between the domains, we get very close to a *sum* of domains instead of a product. We will use the grammars $\mathrm{adr}_{\mathrm{abs}}$ and $\mathrm{adr}_{\mathrm{rel}}$ for syntactically defining absolute and relative address expressions that are translations of source code snippets [10].

$$\mathrm{adr}_{\mathrm{abs}} ::= \&A \mid p \mid \mathrm{adr}_{\mathrm{abs}}.f \mid \mathrm{adr}_{\mathrm{abs}}.[e] \qquad \mathrm{adr}_{\mathrm{rel}} ::= \&A \mid \star \mid \mathrm{adr}_{\mathrm{rel}}.f \mid \mathrm{adr}_{\mathrm{rel}}.[e_\star]$$

In Example 1, an access to the expression `n->data[0]` while holding the locks `n->ms[0]` and `m_a` would convert to an absolute symbolic representation of $\Lambda(R(\mathrm{n}), \tau_{\mathrm{Node}}.\mathrm{data}.[0]) = \{\&\mathrm{m\_a}, \mathrm{n.ms}.[0]\}$ in thread 1. Note how the combination of memory regions, types, fields and indices now defines the arguments of Λ. However, these symbolic expressions are not very helpful for tracking global invariants as individual expressions can vary a lot and carry meaning specific to the local context. We want to extract the *relative* meaning of them and pool their locksets. We can utilize the results of different analyses to determine their interdependence and decouple the information. When accessing a struct field, the base pointer allows us to determine if the lockset can be converted from symbolic to relative when altering the global invariants. When dealing with arrays, an integer must-equality analysis on the indices can correspond an element access with a specific lock. Expressed relatively, the previous expression becomes $\Lambda(R(\mathrm{n}), \tau_{\mathrm{Node}}.\mathrm{data}.[\top]) = \{\&\mathrm{m\_a}, \star.\mathrm{ms}.[\star_0]\}$. The index and field are expressed relative to the owner's index and type.

These relative expressions require a slight modification of the constraint functions previously introduced. We need a function $\mathrm{rel} : \left(\mathbb{D} \times E^* \times 2^{\mathscr{M}_s}\right) \longrightarrow 2^{\mathscr{M}_r}$ that translates symbolic locksets to relative locksets [10] based on concrete expressions $e \in E$ that correspond to the owner: $\mathrm{rel}(\psi, \mathrm{es}, M) := \left\{m' \in \mathscr{M}_r \mid \exists m \in M : \left[\mathrm{es}, \vec{\star}\right]_\psi(m) = m'\right\}$. We now require functions that use memory locations to use tuples $(x, \mathrm{es}) \mid \mathrm{es} \in E^*$. The semantic substitution operator $[x, r]_\psi(e)$ replaces every sub-expression e' that corresponds to an owner x in the state ψ with e in r [10].

$$\mathrm{mf}_s(\psi, \lambda)(x) = \begin{cases} \mathrm{rel}(\psi, \mathrm{es}, \lambda) \text{ if } (x, \mathrm{es}) \in [\![s]\!]^\sharp_{rw}(\psi) \\ \bot \qquad\qquad \text{otherwise} \end{cases}$$

$$\mathrm{sync}_s(\lambda, \Lambda, \psi, \psi')(g) = \begin{cases} \psi'(x) \text{ if } (x, \mathrm{es}) \in [\![s]\!]^\sharp_{rw}(\psi) \wedge g \notin \mathrm{prot}_\Lambda(\mathrm{rel}(\psi_i, \mathrm{es}, \lambda)) \\ \bot \qquad \text{otherwise} \end{cases}$$

Table 4 contains an example run for our lockset analysis. The program location u will just be the line number. The table entries contain the state *after* their execution. Λ will only contain the relevant specified values for reducing complexity. For notation purposes the privatized `n->data` array will be all that $\psi(u)$ contains. We will also skip instructions irrelevant to the analyzed invariants.

First, thread 1 is chosen arbitrarily. The initial array values are 0. No locks are held and global common locks are not specified, therefore $\Lambda(x) := \bot = \mathscr{M}_r$. Then, a global list lock and the array slot locks are acquired. In lines 6 and 7 the array is modified while being protected. The common locks are updated with a $\sqcup$ operation. Type information and constant propagation produce a relative lockset. In line 8 the locally violated value invariant is restored. The last lines of the template release the locks and the global invariant Ψ is updated due to constraint (4).

t	u	$\psi(u)$	$\lambda(u)$	Λ
1	0	$[0,0]$	$\emptyset$	$x \mapsto \mathscr{M}_r$
	1-5	$[0,0]$	{&m_a, n.ms.[0], n.ms.[1]}	$x \mapsto \mathscr{M}_r$
	6-7	$[6,7]$	{&m_a, n.ms.[0], n.ms.[1]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\&\mathrm{m\_a}, \star.\mathrm{ms}.[\star_0]\}$
	8	$[0,0]$	{&m_a, n.ms.[0], n.ms.[1]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\&\mathrm{m\_a}, \star.\mathrm{ms}.[\star_0]\}$
	9-11	$[0,0]$	$\emptyset$	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\&\mathrm{m\_a}, \star.\mathrm{ms}.[\star_0]\}$
2	1-4	$[0,0]$	{&m_b, n.ms.[i]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
	5	$[0,0]$	{&m_b, n.ms.[i]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
	6	$[0,0]$	{&m_b}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
	8-9	$\bot$	$\bot = \mathscr{M}_s$	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
	11	$[0,0]$	$\emptyset$	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
1	1-5	$[0,0]$	{&m_a, n.ms.[0], n.ms.[1]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
	6-7	$[6,7]$	{&m_a, n.ms.[0], n.ms.[1]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
	8	$[0,0]$	{&m_a, n.ms.[0], n.ms.[1]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
	9-11	$[0,0]$	$\emptyset$	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
2	1-4	$[0,0]$	{&m_b, n.ms.[i]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$
	5	$[0,0]$	{&m_b, n.ms.[i]}	$\Lambda(R(\mathrm{n}), \tau_{\mathsf{Node}}.\mathrm{data}.[\top]) = \{\star.\mathrm{ms}.[\star_0]\}$

Table 4. Schematic run of the fixpoint algorithm on Example 1.

In the run of thread 2, no locks are held initially and the local invariant is updated according to constraint (5). This is a $\sqcup$ operation. In the first few lines, the lock corresponding to the input index is acquired. In line 5, the array is accessed and the list-specific locks are removed from Λ with a $\sqcup$ operation. Integer-must-equality analysis finds the global invariant to be unchanged. The conditional branch evaluates to `true` due to the value invariant and the lock is unlocked in line 6. The lines 8 and 9 are considered dead code as a result of the value invariant. Locals are set to $\bot$ and do not influence global invariants. After the branching, we have no definite information about the array values – we over-approximate. Finally, the list lock is released in line 11.

The global invariants have changed since the run of thread 1. It has to be reevaluated to ensure context-sensitivity. The lock acquires and value accesses are the same. The last iteration does not change anything either and the algorithm terminates with a fixpoint in the second run of thread 2!

We see how this algorithm is both path- and context-sensitive. Execution of the unreachable branch in the example would be a race. Different execution contexts of the same thread are taken into account. In Example 1, the correctness of thread 2 depends on the value of `cond`. If it is not statically known, the problem is that the value abstraction possibly creates an infinite domain and the number of paths to consider grows exponentially with the number of conditional guards. *Conditional constant propagation* [9] solves this. The domain $\mathbb{D}_{\mathrm{lock}} \longrightarrow \mathbb{D}_{\mathrm{value}}$ is introduced to map lock expressions to conditional assumptions. When encountering a new conditional, we split the state and the `true`-branch is analyzed. New states $[\{\mathrm{n.mtxs}.[\star_0]\} \mapsto [\mathrm{cond} \mapsto \texttt{true}]]$ and $[\emptyset \mapsto [\mathrm{cond} \mapsto \texttt{false}]]$ are created, subsequent conditional guards can be treated as constants and `false`-paths are considered to be dead code.

3.3 Concretization

The requirements for γ are relatively light in our case and guaranteed by the constraints. The following conditions hold iff $(\vec{u}, \vec{\mu}, \vec{\sigma}, \vec{w}, \varphi) \in \gamma(S)$ for a solution $S = (\lambda, \Lambda, \psi, \Psi)$ [10]:

(1) The muteces held are soundly approximated and the locksets do not overlap.
(2) The global invariant contains the values for all unprotected globals.
(3) A thread's memory view is sound for all globals it may access.
(4) The set of pending writes w_i must include all updated protected globals.

4 Implementation

Every domain added to the analyzer must provide a partial order $\sqsubseteq$, a join $\sqcup$, and (optional) widening operator ∇. The domains discussed in [10] and used in Example 1 are just instantiations of these lattice requirements. Provided constraints are inserted into a worklist and evaluated in the current environment. Updates to constraint variables trigger the recomputation of constraints that depend on the variable if the new value is strictly greater than the old one according to the lattice. For a fixpoint to be reached, a lattice has to satisfy ACC or has to enable convergent over-approximation under widening. Sometimes, a lattice contains infinite ascending chains. The widening operator loosens the conditions of the previous state and the result of the transformation to accelerate the chain traversal by finding an element where the $\sqcup$ escapes the infinite chain.

The Goblint documentation contains an introductory example that shows how one can implement a custom analysis and use it to prove a simple invariant. Goblint also uses other techniques to exclude potential races that are more closely related to the execution environment. *Thread uniqueness* exists when only one thread ever accesses a variable. In this case empty common locksets do not indicate races. This can be modeled by an artificial unique mutex for that thread. *Happens-before* analysis concerns itself with specific program patterns, where certain operations always happen in a well-defined order, e.g. subsequent file operations. The introduction of custom domain elements that extend the owners and distinguish between these different categories of operations solves this. Overall, Goblint significantly improves the scalability of sound static race detection.

5 Related Work

Seidl et al. [7] presented the fundamentals of the fixpoint computation model of mutually recursive constraints used in our algorithm. They model a program as a system of abstract transfer equations and provide a general algorithm to compute the least fixpoint of the arising constraint system over the invariant lattice. This work does not yet handle inter-procedural properties.

Apinis et al. [3] introduced the side-effecting constraint systems that Goblint is built upon and extends the described fixpoint framework from [7] to combine local contextual properties with global invariants, while explicitly tracking dependencies between constraint variables. This formulation is particularly relevant for concurrent analyses, where local reasoning about control flow must be combined with global information.

Vojdani et al. [9] presented the foundational concepts of the Goblint analyzer. Goblint models thread interactions as a constraint system and computes least fixpoints while employing various supporting analyses. Concurrency analysis becomes scalable while remaining sound. The system is parametric in its abstract domains and the analyses used.

Seidl et al. [6] presented techniques to partition memory into canonical abstract owners that represent disjoint sets of concrete locations closed under reachability. This region analysis improves reasoning about dynamically allocated locks and data without tracking individual concrete allocations. Our framework relies on this notion of canonical representatives for address expressions.

References

[1] Linux Device Drivers, Third Edition. Retrieved December 30, 2025 from https://lwn.net/Kernel/LDD3/

[2] Linux Kernel Documentation. Retrieved January 1, 2026 from https://www.kernel.org/doc/Documentation/memory-barriers.txt

[3] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. 2012. Side-effecting constraint systems: a swiss army knife for program analysis. In *Asian Symposium on Programming Languages and Systems*, 2012. 157–172.

[4] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002. 57–68.

[5] Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. 2010. Shape analysis of low-level C with overlapping structures. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2010. 214–230.

[6] Helmut Seidl and Vesal Vojdani. 2009. Region analysis for race detection. In *International Static Analysis Symposium*, 2009. 171–187.

[7] Helmut Seidl, Varmo Vene, and Markus Muller-Olm. 2003. Global invariants for analysing multi-threaded applications. In *Proceedings-Estonian Academy Of Sciences Physics Mathematics*, 2003. 413–436.

[8] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. (1955).

[9] Vesal Vojdani and Varmo Vene. 2009. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, 2009. 141–155.

[10] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016. 391–402.