# Towards Deductive Verification of Concurrent Linux Kernel Code with Jessie

Mikhail, Mandrykin
ISP RAS
Moscow, Russia
Email: mandrykin@ispras.ru

Alexey, Khoroshilov
ISP RAS
Moscow, Russia
Email: khoroshilov@ispras.ru

***Abstract*—The paper considers the challenge of deductively verifying Linux kernel code written in C programming language with extensive use of low-level memory operations and interactions with the highly concurrent environment. The paper presents an initial approach to specification and verification of concurrent code working with shared data by proving the code's compliance with specified synchronization discipline. The proposal is illustrated with an example specifying a user-side simplified model of the read-copy-update synchronization mechanism widely used within the Linux kernel.**

***Keywords*—*Verification, concurrency, ownership, invariants, C semantics.***

## I. Introduction

Deductive verification is an area of static formal software verification which studies various approaches for expressing the correctness of a program with respect to some specification in terms of multiple mathematical propositions called *verification conditions* or *VCs* and analyzing those resulting VCs by means of partially or fully automated logical reasoning. One of the key characteristics of deductive verification techniques is their *soundness* meaning that they allow to ensure the conformance of a program with its specification under certain assumptions by relying on the correctness of the involved deductive reasoning tools. The reasoning tools typically employed in the area include interactive and automated theorem provers and satisfiability modulo theories (SMT) solvers. Apart from the various provers and solvers, deductive verification techniques are usually supplied by the corresponding support tool implementations which automate the steps required for translating the original program into the resulting set of VCs.

With regard to the challenge of the most sound and complete verification of Linux kernel code fragments applying deductive verification tools has both its advantages and drawbacks. On the one hand, deductive verification tools can potentially provide a way for modular sound verification of a wide range of properties on existing low-level system code requiring nearly no modifications to it, which is especially important for Linux kernel code which was not initially developed with intent for formal verification. This is made possible by the compelling generality of the underlying approach, which relies on significant expressiveness of the underlying logical systems. On the other hand, the corresponding reasoning tools always have restricted capabilities due to the general undecideability of the resulting decision problems and very high algorithmic complexity of the decision procedures being involved. Practical deductive verification tools usually address these issues by means of involving human interaction and applying various intricate techniques for reduction, decomposition and simplification of the resulting VCs as well as by providing supplementary tools to facilitate the management of the involved tools and human interaction artifacts obtained during the verification process. Reduction usually results from modularity of the applied verification frameworks, decomposition is achieved by separation applied to the resulting VCs on multiple levels e.g., separating source program paths, memory regions, various properties of the source code behavior (its safety, explicitly specified behaviors, frame conditions for reasoning about effects of imperative code, etc.) and propositional structure of the resulting logical formula, simplification also comes from several prospects, the most important ones being the assumed typing disciplines and memory models which allow some highly efficient static analysis algorithms to be applied for preliminary resolving some necessary conditions and providing the back-end decision procedures with helpful additional assumed or inferred facts related to the decision problems being resolved.

For Linux kernel code the major challenges for deductive verification thus lie in achieving efficiency (simplicity) of encoding in context of weak typing discipline, highly diverse pragmatics and complicated memory model of the C programming language with extensive involvement of various language extensions and low-level memory operations, and, even more so, in maintaining modularity while reasoning about highly concurrent environment of the Linux kernel.

The ASTRAVER project [1] aims at establishing a toolchain for highly automated deductive verification of Linux kernel modules. The toolchain is developed on top of two platforms for source-code analysis and deductive verification. The first, front-end platform is FRAMA-C [2], a suite of tools dedicated to source code analysis of software written in C. FRAMA-C is extensible and collaborative platform with plugin-based architecture, which includes its own WP [3] plugin for deductive verification and allows dynamic linkage of third-party plugins. In particular, the JESSIE plugin, which was originally part of the WHY deductive verification platform, was forked and is

being further developed as part of the ASTRAVER project. As a capable C source-code analysis platform, FRAMA-C provides good compatibility with GNU C including its various specific features and extensions used in the Linux kernel code. The platform also provides its own adaptation of the CIL infrastructure for source code transformation, which greatly facilitates rewriting and normalization of the original source code performed by the JESSIE plugin in order to deal with the complicated memory model and semantics of C. The JESSIE plugin provides support for deductive verification by means of translating the initial C program along with the corresponding functional specifications written in a dedicated specification language (FRAMA-C uses ACSL [4]) into a number of modules and theories in the WHY3ML [5] programming and specification language, an input language of the WHY3 [6] deductive verification platform. Thus, the WHY3 platform serves as the second, back-end deductive verification platform, providing an expressive input language and implementing verification condition (VC) management: generation, transformation, discharging through external theorem provers, and proof management. WHY3 has extensible mechanism for external prover support with pre-implemented drivers for several SMT-solvers [7], [8], saturation-based provers [9], [10] and proof assistants [11], [12]. The efficiency of VC encoding is accomplished by the WP and JESSIE plugins though different means. While WP plugin achieves flexibility and performance by generating VCs directly and thus supporting high-level VC rewriting (through QED plugin [3]) and several memory models (`Hoare`, `typed` and `bit-wise`) configurable separately for each VC, the JESSIE plugin implements one optimized hybrid memory model based on region separation [13] and effect inference [14] with some support for explicit low-level pointer type reinterpretations [15] while relying on the WHY3 platform for the VC management. In our experience during the initial stage of the ASTRAVER project the JESSIE memory model (with some significant modifications [15]) was identified expressive enough to represent the sequential part of Linux kernel code semantics (including various kinds of pointer casts and bit-wise reasoning). Meanwhile there is currently no support for concurrent semantics neither by the Frama-C nor by the Why3 platform.

As most modern system software, Linux kernel modules are concurrent. Moreover, the Linux kernel code makes heavy use of lock-free synchronization mechanisms such as atomic operations, memory barriers and (largely) RCU [16].

The most notable challenge in verification of concurrent programs is, unsurprisingly, the absence of a single continuous control flow. While sequential deductive verification approaches for VC generation such as weakest precondition calculus rely on the user-provided contracts expressed as pre-conditions, postconditions and invariants attached to program control points, a concurrent environment seemingly undermines the whole matter by allowing non-deterministic interference with other concurrent executions on the paths between the control points. Such capability of control flow interruption breaks locality and composability of reasoning and leads to a combinatorial explosion in the number of possible execution paths. Even introduction of synchronization primitives and atomic operations by itself doesn't significantly help with reducing the arising vastly grown search space of possible interleavings. Luckily, there are techniques for avoidance of explicit exhaustive search in the resulting space by transferring the invariants previously attached to program control points to the manipulated data. After such methodological shift, the introduction of a synchronization discipline allows reduction to coarse-grained concurrency within which the interferences of other executions can only occur in a relatively small number of explicitly delimited program locations and specified data invariants need only be checked locally. This allows to fully recover the locality of reasoning at cost of some unsoundness regarding program termination. The ownership methodology with *locally checked two-state invariants*(LCI) [20] is one of such techniques. The key concepts introduced by the extended ownership methodology are the notions of *two-state invariants*, *admissibility*, and *claims*. Together they provide a foundation for local deductive reasoning by means of assigning invariants to types of objects manipulated by the concurrent program. The ownership methodology is adapted for classical first-order logic supported by most modern automated reasoning tools and was previously successfully applied in the VCC [18] deductive verification tool.

While many modular thread-local approaches to verifying concurrent programs only support locking synchronization primitives ([17]), VCC implements a concurrency model capable to express many cases of both locking and lock-free synchronization. VCC concurrency model extends the ownership methodology with support for atomic updates on volatile data. The tool is conceptually similar to the FRAMA-C-JESSIE-WHY3 toolchain, in particular it accepts annotated subset of C as the source language, establishes certain object model on top of C, implements typed memory model with reinterpretation [19] and relies on an SMT-solver backend [7].

In this paper we suggest an initial proposal for extension of the ACSL specification language supported by FRAMA-C with ownership methodology primitives similar to the ones available in VCC and provide several examples usages of the proposed extensions to formalize some aspects of correct usage of basic RCU synchronization primitives in the code of Linux kernel modules.

## II. OWNERSHIP METHODOLOGY AND LCI

Ownership methodology is an object-oriented discipline imposing an invariant that any thread can perform sequential (non-atomic) updates only on the objects that it *owns* and can perform reads only on objects that it either owns or can prove to remain unchanged (*closed*). Objects are organized into an ownership forest where each object always has exactly one *owner* distinct form that object, except form the threads that are also regarded as objects and always own themselves. Objects are typed and each object type is ascribed with a number of *two-state invariants* that constrain possible update operations on the object and can refer to both the state before

and after an update. Multiple sequential updates on objects are performed in groups so that each such group of updates can be seen as a single update by the concurrent system as a whole (this represents so-called coarse-grained concurrency which is guaranteed to soundly approximate fined-grained concurrency under the assumptions of the ownership methodology[21]). To allow grouped sequential updates and object initialization in presence of two-state invariants, objects are extended with binary ghost state indicating whether an object is currently being updated. Objects that are being updated are called *open*, while objects that aren't are called *closed*. From the mentioned methodology restrictions it directly follows that only objects owned by current thread can be opened. Meanwhile if an object is guaranteed to remain closed, it's guaranteed to remain unchanged so it can be safely read. Two-state invariants are supposed to necessarily hold only for transitions between states in which the corresponding objects are closed. There is, though, one exception for the transition corresponding to the opening operation itself which requires invariants to be checked in order to allow *claiming* (section IV-B). The most important property achievable by the ownership methodology with two-state invariants is that local and independent check of the invariants separately for each object update is sufficient to prove preservation of the invariants in the whole concurrent system. In order to achieve this property the invariants are required to be *admissible*. An invariant is admissible if it is preserved by any transition that preserves invariants of all modified objects. Admissibility is a non-local property that can generally depend on any object type invariant, but it is monotonic in a sense that once proven, it cannot be broken by adding new object type definitions. Checking that all provided invariants are admissible allows to localize the invariant preservation checks by regarding only the invariants of directly updated objects.

In VCC this ownership methodology is extended with a notion of volatile fields that can be both written or read atomically independently from the ownership on the object, but provided that object's non-volatile fields are not being updated, i.e. the object remains closed. All atomic updates on volatile fields must preserve all the invariants of the updated object. Extension of the methodology with volatile fields doesn't break locality of invariant checks while providing a way to express certain extra synchronization mechanisms.

## III. Ownership methodology and ACSL

ACSL [4] was not originally developed with ownership methodology in mind, hence, expectedly, there are a number of issues either with direct integration of the ownership methodology into the language or with applying the methodology on top of it. The most obvious and significant issues include the following ones:

- Unlike VCC, ACSL does not establish or assume any object-oriented paradigm on top of C, neither it even has typed semantics. The Jessie plugin, though, already does transform input C/ACSL functionally specified programs into object-oriented intermediate language (also called Jessie [22]). So the major task for supporting ownership methodology would be exposing some parts of the Jessie intermediate language capabilities to ACSL, in particular additional pre-defined ghost fields required by the ownership methodology e.g., `\closed` and `\owns`.
- In VCC there is no direct analogue of the ACSL notion of validity exposed through built-in constructions `\valid` and `\valid_read`. The most close counterparts for them would be notions of opened objects for non-volatile fields, closed objects within atomic blocks for volatile ones (a union of these two closely corresponds to `\valid`), and notions of objects owned by current thread or closed objects, jointly closely corresponding to `\valid_read`. For the purpose of the most straightforward integration of already verified purely sequential code fragments into the context of the adopted concurrent paradigm we suggest unifying the mentioned corresponding notions by expressing them through one another in the final translated WHY3ML programs. Here the ownership methodology concepts are considered more primitive while `\valid` and `\valid_read` are supposed to be expressed through them. In most cases this would allow using parts of previously verified sequential code in concurrent context directly without any changes in the corresponding annotations (one example is calling functions requiring validity of some objects in contexts where the objects are open).
- ACSL approach to specifying function contracts, particularly **assigns** and **allocates/frees** clauses, doesn't obviously correspond to VCC object writeability, let alone there is no corresponding constructions for specifying memory allocation footprint in VCC. Similar to the suggested solution to the validity issue those clauses can be relatively easily mapped to writes on open objects and ownership summaries correspondingly. So whenever an object opened throughout the whole function (in both the pre- and post- states while never being closed in between) is modified, it should be mentioned in the **assigns** clause, and whenever a function obtains or gives up ownership of an object, this should be mentioned in the corresponding **allocates/frees** clause. The letter correspondence can be somewhat surprising to the user, so the **allocates/frees** clauses can be supplied with more intuitive syntactic equivalents e.g., **takes/drops**, **acquires/releases** or some other pair.
- ACSL currently (as of version Sodium-20150201 [4]) defines two kinds of type invariants, namely strong and weak ones. Two-state invariants are neither of these, so they should be added to the ACSL specification and exposed to the user. In the following we are using the suggested keyword **2state** to distinguish two-state type invariants.
- One another non-obvious, but highly desirable addition to ACSL in context of ownership methodology is capability to define ghost structures implicitly from any location in the executable code (rather than in the global scope). This is mostly intended to significantly simplify the usage of *claims* (section IV-B).

## IV. Ownership methodology by example

### A. Spinlock

Let's now demonstrate an example of quite a simple synchronization mechanism formalized in terms of the ownership methodology. The model intentionally simplified specification of `spinlock` listed in figure 1 exemplifies integration of the ownership methodology into ACSL. The example basically repeats the corresponding example from the VCC tutorial [23] reproduced in the proposed extension of ACSL, but it also highlights some ACSL-specific characteristics. The most noticeable of them is using a special predicate `\new` to distinguish closing a freshly allocated object from closing a previously opened object, since the first case generally requires a different check for object invariants (consider an invariant `counter = \old(counter) + 1`, it's not possible to prove it for object initialization). ACSL also requires explicit specification of allocation/owning footprint. In the example we don't use sugared predicate `\wrapped` equivalent to the conjunction of `\owned` (by current thread) and `\closed`. We also suggest pervasive usage of explicit built-in `\acquire` and `\release` constructions to operate on objects' `\owns` sets; the sets are supposed to be always empty right after the objects' allocations.

### B. Claims

Similar to the example in the VCC tutorial our ACSL version initially makes unrealistic assumptions by requiring `\closed(l)`. The problem with such precondition is that it's not evident how it could be directly proven for a thread which doesn't own the lock. For these purposes VCC uses special notion of a *claim* object, whose sole purpose is to be owned by some thread while being occasionally opened to serve as a proof of other objects' closedness. To simplify usage of claims VCC introduces special *claim counter* fields that are checked on object opens and are allowed to be modified only through special claim creation/destruction operations. Sample definition of a claim structure is presented in figure 2. This definition requires using a special `claim` attribute to avoid structure type invariant admissibility check since the invariant is not directly admissible. Though it's possible to define claims with admissible invariants this would require using claim sets instead of simpler claim counters and therefore would be less efficient. The admissibility of claim invariants is guaranteed by special preconditions on object opens and special treatment of claim counters. In VCC methodology the role of claims is not limited to asserting closedness of objects in their `claimed` sets, but is extended to serve as a mere substitute for assertions in a concurrent environment. Therefore claim creation/claim counter increase and claim destruction/claim counter decease operations are also extended appropriately to allow for claiming arbitrary assertions on closed objects. The corresponding suggested syntax for these operations in ACSL is the following: `\claim(claim_struct_name, object, predicate)` and `\unclaim(claim, object)`. As these operations are performed on closed objects, the corresponding claim counter fields are always volatile. They can be added to a structure type by supplying it with ghost `claimable` attribute. The `\claim`/`\unclaim` operations generally require implicit declaration of claim structure types along with the invariants provided at claim creation locations, here comes the need for the corresponding capability in ACSL. The operations can only be preformed inside atomic blocks, which also allows to synchronize ghost claim counters with real reference counters by supplying the claimable objects with the corresponding invariants and putting corresponding atomic reference counter updates into the same atomic blocks.

### C. Read-copy update mechanism

Read-copy update (RCU) [16] is a synchronization mechanism with support for lock-free readers and partially lock-free writers. Within the RCU mechanism writing is called *updating* and is performed in two separate phases. The first, *removal* phase amounts to a memory barrier-protected atomic update of a pointer with either a pointer generally known to be invalid (a NULL or a poisoned pointer) or a pointer to an object totally prepared for any arbitrary read operations. The second, *reclamation* phase comprises an exclusive (usually lock-protected) deallocation of the obsolete object remained after the update performed on the corresponding preceding removal phase. The two phases of updating are separated by a blocking synchronization operation which ensures that the corresponding obsolete (already removed, but not yet deallocated) object remains valid for readers until they completely give up accessing it. To ensure this no-more-reads condition, the readers must always delimit their accesses within a pair of explicit calls to special read-section opening/closing marker functions and also ensure freshness of the objects they access strictly before accessing them by calling a special dereference-protecting construction at least once per each read-section. Thus, the blocking synchronization operation performed by the writers can simply wait till the end of all the currently active read-sections which would ensure that no more readers remained accessing the obsolete objects and they can be then safely deallocated. Thus, essentially, an RCU mechanism implementation exposes the following five core basic primitives: 1) the memory barrier-protected atomic update primitive to perform the removal phase of an update operation (let's call this primitive `rcu_assign_pointer`), 2) the blocking writer-side synchronization operation to demarcate the start of the reclamation phase of an update (or simply `synchronize_rcu`), 3) the read-section opening marker (`rcu_read_lock`), 4) the read-section closing marker (`rcu_read_unlock`), 5) the dereference-protecting construction (`rcu_dereference`).

We suggest an example (shown in figure 3) of a simplified functional specification for these primitives intended to be used in RCU user-side code to ensure correct usage of the exposed RCU interface. The example serves a preliminary proof-of-concept for the applicability of the extended ownership

```
1 /*@ volatile_owns */ struct spinlock {
2   volatile unsigned int slock;
3   //@ ghost void *resource;
4 };
5
6 /*@ 2state type invariant
7   @    same_resource(struct spinlock l) =
8   @      \old(l.resource) == l.resource;
9   @*/
10
11 /*@ 2state type invariant
12   @    ownership(struct spinlock l) =
13   @      !l.slock ==>
14   @        \subset(resource, \owns(&l));
15   @
16   @*/
17
18 /*@ requires \owned(obj) && \closed(obj);
19   @ requires \valid(l) && \new(l);
20   @ requires \owns(l) == \empty;
21   @ frees obj;
22   @ ensures \closed(l);
23   @ ensures l->resource == obj;
24   @*/
25 void spin_lock_init(struct spinlock *l)
26 /*@ (ghost void *obj) */
27 {
28   l->slock = 0;
29   /*@ ghost {
30     @    l->resource = obj;
31     @    \release(obj, l);
32     @    \close(l);
33     @ }
34     @*/
35 }
36
37 /*@ requires \closed(l);
38   @ allocates l->resource;
39   @ ensures \owned(l->resource) && \closed(l->resource);
40   @*/
41 void spin_lock(struct spinlock *l)
42 {
43   int stop = 0;
44   do {
45     /*@ atomic (l)*/ {
46       stop = !cmpxchg(&l->slock, 1, 0);
47       /*@ ghost
48         @    if (stop)
49         @      \acquire(l->resource, l);
50         @*/
51     }
52   } while (!stop);
53 }
54
55 /*@ requires \closed(l);
56   @ requires \owned(l->resource) && \closed(l->resource);
57   @ frees l->resource;
58   @*/
59 void spin_unlock(struct lock *l)
60 {
61   /*@ atomic (l) */ {
62     l->slock = 0;
63     //@ ghost \release(l->resource, l);
64   }
65 }
```

Fig. 1. Spinlock example

methodology to verification of Linux kernel modules that widely [24] use the RCU interfaces provided by the Linux kernel core. The most subtle aspect of the RCU mechanism is the notion of protected pointer that should only be dereferenced within the corresponding read-section. Another noticeable complication is ensuring that both `rcu_assign_pointer` and `rcu_dereference` are used on the same memory location (address) and that the deallocation of each obsolete object is appropriately protected by a separate lock.

```
1 /*@ ghost claim struct claim {
2   @  set<void *> claimed;
3   @ };
4   @
5   @ // 2state type invariant
6   @ //   claim(struct claim c) =
7   @ //     \forall void *o;
8   @ //       \subset(o, c.claimed) ==>
9   @ //       \closed(o);
10  @*/
```

Fig. 2. Sample claim structure definition.

## V. Conclusion

We suggested an extension of the ACSL specification language supported in particular by the Frama-C-Jessie-Why3 toolchain with support for concurrent code by using the ownership methodology previously implemented in the VCC deductive verification tool and applied for large critical concurrent systems such as Microsoft Hyper-V. We also suggested an initial functional specification for the exposed interface of the RCU synchronization mechanism to show how the VCC ownership methodology (with a minor extension) can be applied for verification of Linux kernel modules.

The suggested formalization, though, has not been formally verified using model-checking or some other suitable technique. Another remaining task is mitigating one inherent possible source of unsoundness of the ownership methodology arising from possible unrestricted usage of atomic blocks.

## Acknowledgment

The research was supported by the Ministry of Education and Science of the Russian Federation (unique project identifier RFMEFI60414X0051)

## References

[1] [Online]. Available: http://linuxtesting.org/astraver
[2] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, "FRAMA-C, A Software Analysis Perspective", *Proceedings of International Conference on Software Engineering and Formal Methods 2012 (SEFM12)*, October 2012.
[3] L. Correnson, Z. Dargaye, A. Pacalet, "WP (Draft) Manual", [Online]. Available: http://frama-c.com/download/frama-c-wp-manual.pdf.
[4] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, "ACSL: ANSI/ISO C Specification Language. Version 1.7", `http://frama-c.com/download/acsl.pdf`.
[5] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, "Why3: Shepherd your herd of provers", *Boogie 2011: First International Workshop on Intermediate Verification Languages*, 2011.
[6] J.-C. Filliâtre, A. Paskevich, "Why3 — where programs meet provers", *In Programming Languages and Systems*, pp. 125–128, Springer Berlin Heidelberg, 2013.

```
/*@ axiomatic rcu {
  @   type rcu_section = integer;
  @
  @   2state type invariant
  @     rcu_secion(rcu_section s) = \owned(&s);
  @
  @   logic struct lock *rcu_lock(void **loc, void *obj);
  @   predicate rcu_reclaimable(void *obj);
  @ }
  @
  @ ghost rcu_section current_section = 0;
  @ ghost set<void *> rcu_dereferenced = \empty;
  @*/

/*@ requires current_section == 0;
  @ assigns current_section;
  @ ensures current_section != 0;
  @*/
void rcu_read_lock(void);

/*@ requires current_section != 0;
  @ assigns current_section;
  @ assigns rcu_dereferenced;
  @ frees rcu_dereferenced;
  @ ensures current_section == 0;
  @ ensures rcu_dereferenced == \empty;
  @*/
void rcu_read_unlock(void);

/*@ requires rcu_lock(loc, obj) == l;
  @ requires rcu_reclaimable(obj);
  @ allocates l;
  @ ensures \closed(l);
  @*/
void synchronize_rcu(void)
  /*@ (ghost void **loc, void *obj, struct lock *l) */;

/*@ requires \owned(l) && \closed(l);
  @ requires l->resource == obj;
  @ requires \claim_count(l) == 0;
  @ frees l;
  @ ensures rcu_lock(loc, obj) == l;
  @ ensures rcu_reclaimable( *loc);
  @*/
void rcu_protect(void **loc, void *obj,
                 struct lock *l);

#define rcu_assign_pointer(p, v, l) \
  {\
    rcu_protect(&p, v, l);\
    p = v;\
    v;\
  }

/*@ requires current_section != 0;
  @ requires rcu_lock(loc, obj) != NULL;
  @ assigns rcu_dereferenced;
  @ allocates obj;
  @ ensures obj != NULL ⟹ \closed(obj);
  @ ensures rcu_dereferenced =
  @       \union(\old(rcu_dereferenced), obj);
  @*/
void *rcu_deref(void **loc, void *obj)

#define rcu_dereference(p, c) \
  ((typeof(p)) rcu_deref(&p, p));
```

Fig. 3. Simplified RCU model formalization.


[7] L. De Moura, N. Bjørner, "Z3: An efficient SMT solver", *In Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer Berlin Heidelberg, 2008.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, "CVC4", *In Computer aided verification*, pp. 171–177, Springer Berlin Heidelberg, January, 2011.

[9] A. Riazanov, A. Voronkov, "The design and implementation of Vampire", *In AI communications*, vol. 15(2, 3), pp. 91–110, 2002.

[10] S. Schulz. "System Description: E 1.8", *In Proceedings of the 19th LPAR*, Stellenbosch, pp. 477–483, LNCS 8312, Springer Verlag, 2013.

[11] Y. Bertot, P. Castéran, "Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions", *Springer Science & Business Media*, 2013.

[12] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, M. Srivas, "PVS: Combining Specification, Proof Checking, and Model Checking", *In Proceedings of Computer-Aided Verification '96*, pp. 411–414, 1996.

[13] T. Hubert, C. Marché, "Separation analysis for deductive verification", *In Heap Analysis and Verification*, Braga, Portugal, March, 2007.

[14] J.-P. Talpin, P. Jouvelot, "Polymorphic type region and effect inference", Technical Report EMP-CRI E/150, 1991.

[15] M. Mandrykin, A. Khoroshilov, "High level memory model with low level pointer cast support for Jessie intermediate language", *In Programming and Computer Software*, Vol. 41, No. 4, pp. 197–208, 2015.

[16] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni, "Read-copy update", *In AUUG Conference Proceedings*, p. 175, 2001.

[17] C. Flanagan, S. N. Freund, S. Qadeer, "Thread-modular verification for shared-memory programs", *In ESOP 2002*, Number 2305 in LNCS, Springer, pp. 262–277, 2002.

[18] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, "VCC: A practical system for verifying concurrent C", *In Theorem Proving in Higher Order Logics*, pp. 23–42, Springer Berlin Heidelberg, 2009.

[19] E. Cohen, M. Moskal,S. Tobies, W. Schulte, "A precise yet efficient memory model for C", *In Electronic Notes in Theoretical Computer Science*, vol. 254, pp. 85–103. 2009.

[20] E. Cohen, M. Moskal, W. Schulte, S. Tobies, "Local Verification of Global Invariants in Concurrent Programs", *In Computer Aided Verification*, Springer Berlin Heidelberg, pp. 480–494, January, 2010.

[21] E. Cohen, M. Moskal, W. Schulte, S. Tobies. "A practical verification methodology for concurrent programs", Tech. Rep. MSR-TR-2009-15, Microsoft Research, 2009. (`http://research.microsoft.com/pub`)

[22] J.-C. Filliâtre, C. Marché, "The Why/Krakatoa/Caduceus platform for deductive program verification", *In Proceedings of the 19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Springer, 2007.

[23] M. Moskal, W. Schulte, E. Cohen, M. A. Hillebrand, S. Tobies, "Verifying C programs: a VCC tutorial", MSR Redmond, EMIC Aachen, 2012.

[24] P.E. McKenney, S. Boyd-Wickizer, J. Walpole, "RCU usage in the Linux kernel: one decade later", Technical report, 2013.