The Journal of Systems and Software 159 (2020) 110450

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A survey on the use of access permission-based specifications for program verification[☆]

Ayesha Sadiq[*], Yuan-Fang Li, Sea Ling

Faculty of Information Technology, Monash University, Clayton, Australia

ARTICLE INFO

Article history:
Received 26 July 2018
Revised 15 August 2019
Accepted 19 October 2019
Available online 21 October 2019

Keywords:
Access permissions
Program verification
Concurrency
Protocol verification
Permission inference
Survey

ABSTRACT

Verifying the correctness and reliability of imperative and object-oriented programs is one of the grand challenges in computer science. In imperative programming models, programmers introduce concurrency manually by using explicit concurrency constructs such as multi-threading. Multi-threaded programs are prone to synchronization problems such as data races and dead-locks, and verifying API protocols in object-oriented programs is a non-trivial task due to improper and unexpected state transition at run time. This is in part due to the unexpected sharing of program states in such programs. With these considerations in mind, access permissions have been investigated as a means to reasoning about the correctness of such programs. Access permissions are abstract capabilities that characterize the way a shared resource can be accessed by multiple references.

This paper provides a comprehensive survey of existing access permission-based verification approaches. We describe different categories of permissions and permission-based contracts. We elaborate how permission-based specifications have been used to ensure compliance of API protocols and to avoid synchronization problems in concurrent programs. We compare existing approaches based on permission usage, analysis performed, language and/or tool supported, and properties being verified. Finally, we provide insight into the research challenges posed by existing approaches and suggest future directions.

1. Introduction

Correctness and reliability of software programs written in imperative and object-oriented languages such as Java and C++ have always been a major challenge for the IT industry. This is because of the implicit dependencies that exist between the code and program states. In imperative programs different program parts may access the same mutable state, without exposing this information to each other and, consequently, may cause unwanted interference or inconstant states. Preventing such errors is important to ensure the compliance of API (Application Programming Interface) protocols in object-oriented programs and to verify the correctness of increasingly ubiquitous multi-threaded applications.

Modern object-oriented programs are highly reliant on reusable APIs that often define usage protocols i.e., the desired sequence of method calls that API clients must follow for underlying objects to work properly. Typestates (Strom and Yemini, 1986) have been designed to specify usage protocols and verify their behavior. A typestate abstractly defines an object's state at the execution time. However, statically tracking object state is a non-trivial task because of unexpected transitions between states during program execution. In multi-threaded programs, managing synchronization between threads is a complicated and challenging task for programmers due to thread interleaving and heap interference, which can lead to concurrency problems such as deadlocks, data races. The situation becomes worse in the presence of unrestricted aliasing, the hallmark feature of imperative and object-oriented languages (Bierhoff et al., 2009b).

Earlier work on verifying correctness of sequential programs dates back to Hoare's logic (Hoare, 1969) that defines a set of axioms (calculus) and formal inference rules to specify and verify desired properties and static analysis techniques such as Floyd (1967). Handling thread non-interference for Java-like concurrent programs dates back to Owicki-Gries' axiomatic method, (Owicki and Gries, 1976), and Jones' rely-guarantee principle (Jones, 1983). These approaches are considered the traditional ways of performing shared-memory program verification. Since the seminal work of HoareOwicki-Gries (Owicki and Gries, 1976) and Jones (Jones, 1983), many logic-based verification approaches and type-effect systems have been developed to avoid problems such as data races, deadlocks, atomicity violations in shared-memory programs.

[☆] Editor: Dr Earl Barr

[*] Corresponding author.

E-mail addresses: ayesha.sadiq@monash.edu, ayeshasadiq.gull@gmail.com (A. Sadiq), yuanfang.li@monash.edu (Y.-F. Li), chris.ling@monash.edu (S. Ling).

https://doi.org/10.1016/j.jss.2019.110450
0164-1212/© 2019 Elsevier Inc. All rights reserved.

The commonly used verification approaches either perform deductive verification or employ theorem proving techniques where formal correctness proofs are used to verify program properties based on the input specifications (Huisman, 2001; Flanagan et al., 2003; Abadi et al., 2006; OHearn, 2007; Villard et al., 2010; Caires and Seco, 2013). Others conduct a dynamic analysis of the input program through model checking (Visser et al., 2003; Chaki et al., 2004; Chaki and Gurfinkel, 2018) or perform static analysis (Boyapati et al., 2002; Engler and Ashcraft, 2003; Voung et al., 2007; Naik et al., 2009; Dias et al., 2013) to approximate runtime behavior of the program to verify its correctness.

However, in the last decades, static contract checking based on Hoare logic and the development of advanced, simplified, automated program verifiers (Fähndrich and Logozzo, 2011; Pradel et al., 2012; Filliâtre and Paskevich, 2013; Carr et al., 2017) had been an active area of research for the verification of program behavior. Although these approaches are usable and quite promising, the support for concurrency and aliasing is limited. As most of the real-world applications are inherently multi-threaded, the next step was to develop tools and techniques that can reason about shared-memory programs and control aliasing in a sound and efficient way. With these considerations in mind, permission-based program logics and tools became influential because of their practicality, expressiveness and strong reasoning power to handle both aliasing and concurrency.

Access permission, formally called Boyland's fractional permission (Boyland, 2003), is a formalism inspired by Linear Logic (Girard, 1987) and Separation Logic (O'Hearn et al., 2001; Reynolds, 2002). The former treats permissions as linear resources and the latter simplifies the specifications and verification of shared-memory programs efficiently. Fractional permissions were originally proposed to verify non-interference of the program states in parallel programs, using either read or write accesses on the referenced objects. The formalism was later extended by Bornat et al. (2005) to allow read sharing of the shared program states. Bierhoff and Aldrich (2007) extended fractional permissions as symbolic permissions to model both the read/write operations and aliasing information of a program state at one place.

A study of the literature shows that access permissions have been investigated to address different concerns related to security, concurrency, and protocol verification. Notable threads of research include Plural (Bierhoff and Aldrich, 2007; 2008; Beckman, 2009), Chalice (Leino et al., 2009; Leino and Müller, 2009), VeriFast (Jacobs et al., 2011), VPerm (Le et al., 2012), Pulse (Siminiceanu et al., 2012), Sample (Ferrara and Müller, 2012), Plaid (Aldrich et al., 2012), VerCors (Amighi et al., 2012; Blom et al., 2014; Amighi et al., 2014; Huisman and Mostowski, 2015; Amighi et al., 2015; Blom et al., 2017; Joosten et al., 2018) and Viper (Müller et al., 2016; Müller et al., 2017), to name a few.

This survey summarizes the pragmatics of Boyland's permission sharing model (Boyland, 2003) and its variants (accounting and symbolic permissions), discussed in detail in Sections 3 and 5, that have been used in the literature to verify the correctness and behavior of sequential and concurrent programs, based on access permissions. To the best of our knowledge, this survey is the first attempt to provide readers a comprehensive overview of existing access permission-based program verification approaches and tools since the introduction of Boyland's fractional permissions and using fractional, accounting, and symbolic permissions.

We categorize the research work covered in this survey along the following three dimensions:

1. Verification of API protocols in typestate-based sequential and concurrent programs.
2. Verification of common concurrency problems such as race conditions and deadlock etc., in concurrent (multi-threaded) programs.
3. Automatic inference of access permissions for sequential and concurrent programs.

Within each of the above categories, the existing approaches are compared and contrasted based on the following criteria:

- The type of underlying program (`Prog`) such as sequential or concurrent program.
- The programming language (`Lang`) used or developed as a specification language.
- The realization of the proposed technique in the form of prototype tool (`Tool`).
- The type of analysis (`Analy`) i.e. static or dynamic performed.
- The kind of permission abstraction (`Perm-Kind`) such as fractional, counting or symbolic permissions, supported as a part of specifications.
- The access notations or contracts (`Perm-Specs`) specified as program annotations.
- The permissions or access notations (`Perm-Infer`) inferred.
- The annotation overhead (`Anno`) (if any) posed by the approach.
- The functional or behavioral properties (`Properties`) verified, based on the permission-based specifications.

With some of the discussed approaches, we elaborate on the usage of permission-based specifications to show how existing approaches annotate programs with different types of access permissions to verify program behavior against specifications. Further, we provide the detailed statistics in Tables 4, 5, 6, 8, 9, and 11 (taken from the surveyed papers) to quantify the manual effort, in terms of program size and annotation overhead posed by the existing approaches. Furthermore, in every section, a summary of the studied work in chronological order (where possible), following the above mentioned criteria, is given in the tabular form.

Section 2 briefly discusses the related formal theories and type systems in the literature as seminal and background work to program verification. Section 3 provides an overview of access permissions. Sections 4, 5, 6 covers the three categories of the state of access permission-based program verification, mentioned previously. Finally, Section 7 provides an insight into the use of access permission-based specifications, the research challenges posed by the existing approaches and suggest future research directions.

2. Related formalisms to program verification and parallelization

This section briefly presents other formal type theories for program verification and parallelization. A type system can verify the desired interaction between system components as types can classify program entities and the permissible results of the computations. The beauty of type systems is the assurance that if a program is type-checked then it is guaranteed to be free from certain classes of errors.

Earlier work on type systems mainly focused on the results of a computation in a program in terms of its correctness. Then the study of type discipline and concurrency theory inspired the development of formal type systems that can statically formulate and verify the intended properties of a program behavior, along with the permissible results of the computations.

Behavioral type

Behavioral type theory is one such formalism that was originally proposed to verify concurrent programs based on process algebra (Nielson and Nielson, 1993; 1996). A behavioural type system uses behavioral types, a type-based abstraction, to formally describe the software entities such as communication protocols, interfaces, web services and contracts, as a sequence of operations in a concurrent and distributed environment. Formally speaking, a behavioral type system is a compositional type system that can directly model the interaction between system components as a notion of choice, causality and resource usage. Session types and behavioural contracts are two notions related to behavioural types.

Since the introduction of behavioral types, many type-based effect and proof systems, using the concepts of behavioural types, session types, spatial logic and processes as types, have been developed to study various correctness and behavioral properties such as unique receptiveness, race freedom, deadlocks, and livelocks in large-scale concurrent and distributed systems (Sangiorgi, 1999; Chaki et al., 2002; Igarashi and Kobayashi, 2005; Dezani-Ciancaglini et al., 2005; Kobayashi and Sangiorgi, 2010). However, in these approaches, type-based specifications are explicitly added to the model and verify the usage patterns of resources and communication objects. The behavioral type theory then subsequently integrated with session types (Igarashi and Kobayashi, 2001; Chaki et al., 2002; Igarashi and Kobayashi, 2005; Dezani-Ciancaglini et al., 2005) where type-based specifications are explicitly added to model and verify the usage patterns of communication objects in concurrent and distributed environment.

Recent work on behavioral separation in a distributed environment can be attributed to Caires (2008) who developed a spatial-behavioural typing system, to model the resource independence and synchronization in a distributed (concurrent) environment, based on the Spatial logic (Caires and Cardelli, 2003; 2002). The type system using parallel and sequential composition operators and resource ownership are handled using the type modality. Later, Caires and Seco (2013) developed a behavioral separation programming language based on λ-calculus to ensure the disciplined interference between resources in higher-order concurrent programs having fork/join parallelism. The language is based on the behavioral type systems (Honda et al., 1998; Chaki et al., 2002) that incorporates a behavioral view of the program properties and employs Concurrent Separation Logic (Reynolds, 2002; OHearn, 2007), to separate the dynamic behavior of run-time values rather than separating program states itself. However, program effects are computed based on the explicitly specified assertions to ensure the safety of concurrent programs. A detailed study of behavioral type systems and related methodologies can be found in Ancona et al. (2016).

Session type

Session types, a notion of behavioral types, was originally introduced by Honda et al. (1998) to ensure the disciplined interaction between two partners in a distributed environment and later extended in the work of Honda (Honda et al., 2008) to incorporate an arbitrary number of participants in the same environment.

Recently, session types were extended with formal type system to control aliasing and to enforce usage protocols in a concurrent environment for Java and Java-like languages, such as SessionJ (Hu et al., 2008; 2010), Yak (Militão and Caires, 2009) and Mungo (Gay et al., 2015), to name a few. Further, the linearity of resources is an important and recurring theme in concurrency that studies behavioral type systems, for process calculi and as mentioned by Honda (1993), Linear Logic can be considered as a source of inspiration for some aspects of session types.

Recent work on Linear Logic-based session types includes the work of Caires and Pfenning (2010), who proposed a Curry-Howard style interpretation of binary session types in intuitionistic Linear Logic, to expose a deep relationship between both concepts. Recently, with this line of work, Gay and Vasconcelos (2010) and Wadler (2014) developed a new calculus CP and a linear functional language GV, to establish a connection between session types and the Linear Logic, that can yield a process calculus, free from data races and deadlocks. A comprehensive study of program verification approaches based on the session and behavioral types can be found in (Hüttel et al., 2016).

Typestate

Yet another important formalism, a notion of behavioural types, is typestate. Typestate was first defined by Strom and Yemini (1986) as a new programming language concept that determines the operations permitted on objects in a given context. According to Garcia et al. (2014) “typestate reflects how legal operations on objects can change at run-time as their internal state changes”. Typestate associates state information with a variable of a given type, which is then subsequently used to decide the valid operations to be called on an instance of that type. Typestate is suitable to represent resources that follow state transition systems that follow the ‘open then close’ semantics. For example, a database connection can only execute a database command if it is in the open state.

Typestates were originally developed for imperative languages without the notion of objects, but later extended with the behavioural-type discipline, to support verification of object-oriented languages such as Vault (DeLine and Fähndrich, 2002), Fugue (DeLine and Fähndrich, 2004). Furthermore, typestates were integrated as a first-class language construct in typestate-oriented language (Aldrich et al., 2009; Sunshine et al., 2011) to verify the correctness of the usage protocols in state-based sequential programs. In the new language, objects are being modeled not only as classes, but also the changing abstract states, where the correctness of the program is determined by tracking state transitions between different objects at execution time, thereby ensuring the correct usage of protocols. Later, the typestate-oriented programming led the development of a gradual typestate system (Garcia et al., 2014) that integrates access permissions with gradual types (Siek and Taha, 2007) to control aliasing in a more robust way.

Recent developments in state-based protocol checking and verification includes approaches (Caires and Seco, 2013; Garcia et al., 2014; Militão et al., 2014b; Gay et al., 2015), that handle aliasing in a more robust way and verify communication protocols in distributed and concurrent object-oriented languages. These approaches ensure the basic memory safety conservatively, by associating typestate invariants with each referenced location and by ensuring that the type invariants hold for every store of this location. However, all the approaches discussed above focus on identifying violations of protocols but none of them check the higher-level (behavioral) characteristics of the protocols themselves, for example, their usage in practice and the complexity associated with their definition, that are vital in verifying correctness of many program properties (Beckman et al., 2011).

Ownership type

Another stream of type-based systems is ownership type, a mechanism to express the sharing of program references (Noble et al., 1998; Clarke et al., 2013), in the way that allows controlled aliasing between objects by mitigating the undesirable effects to other objects.

Since its introduction, ownership types have been used in many formal approaches to provide safe aliasing control mechanisms (Boyapati and Rinard, 2001; Clarke and Wrigstad, 2003; Müller and Rudich, 2007; Cohen et al., 2009), and to control object deallocation explicitly (Matsakis et al., 2014). However, in ownership-based verification approaches, programmers explicitly define ownership invariants as locking or access information to specify dependencies at the object level and use this information to avoid data races. A complete stream of research on ownership types can be found in the work of Clarke et al. (2013).

Atomic set

Atomic sets have been used to detect atomicity violation and to avoid data races in concurrent programs. An atomic set defines a set of memory locations that share some consistency property and needs to be updated atomically. The atomic set can be viewed as a generalization of Hoare's monitors (Hoare, 1974) to multiple objects. They can be better integrated into the Java language.

Earlier work using atomic sets dates back to Vaziri et al. (2006)'s data centric programming model that defines `atomic set serializability`. Atomic set serializability is a disciplined interference criterion to avoid the problematic interleaving scenarios in the shared-memory programs based on atomic sets. Vaziri extended her previous work to support multiple object interference (Vaziri et al., 2010). Subsequent work used atomic sets to detect atomicity violation statically (Kidd et al., 2011) and dynamically (Xu et al., 2005; Hammer et al., 2008; Lai et al., 2010). The trend is followed by many local data-centric concurrency control mechanism and type systems (Dolby et al., 2012; Marino et al., 2013), to verify program behavior based on atomic sets.

Data-centric concurrency control is one alternative to the explicit locking mechanism. In contrast to the control-centric synchronization approaches, (Artho et al., 2003; Lu et al., 2008), the local data-centric approaches combine all fields of an object that require consistency, for all the control flow paths of program execution, into an atomic set and updates them atomically to avoid data races.

Contrary to data-centric concurrency control and the use of atomic sets, the development of type systems (Flanagan et al., 2003; Abadi et al., 2006; Flanagan et al., 2008) has been influential to ensure the atomicity and data-race freedom in concurrent programs, and to reduce the annotation overhead associated with manually adding synchronization primitives at code level. However, unlike atomic sets, in type systems, programmers provide explicit synchronization primitives, as locking and guarded specifications at field or class level, required by the code.

Recently, the notion of atomic sets was replaced by atomic variables by Paulino et al. (2016). The objective was to handle the complexity associated with the use of memory structures in atomic sets. The proposed approach applies a resource-centered view of the data-centric concurrency control. However, in the proposed type system, programmers explicitly define the synchronization primitives on the individual data items that require atomic updates, to guarantee the progress of synchronization for all program execution scenarios.

Uniqueness and immutability

Another area of interest has been the controlled sharing and interference of object references in imperative object-oriented programs. Sharing is a situation when a piece of memory is accessed by more than one reference, say `x` and `y`, so that a change to `x` affects `y` as well. Therefore, changes to one object may leave other objects in an inconsistent state, causing unwanted interference and subsequently, data races.

Work has been done to restrict the usage of references notably using access-based type annotations such as `uniqueness`, `immutability` and `read-only` (Clarke and Wrigstad, 2003; Boyland, 2006; 2010; Gordon et al., 2012; Clebsch et al., 2015). The objective was to identify isolated states that can be safely handled by one or more threads, thereby avoiding the unwanted interference and alternatively data races. However, the type system infers sharing effects such as `uniqueness` and `immutability` for the object references by computing an equivalence relationship for a set of free variables by evaluating the input expression. The inferred effects are then used to determine which part of the code can be safely shared between multiple threads to maintain the integrity of the data.

Recent development in this area is Giannini's type and effect system (Giannini et al., 2018a; 2018b) that expresses sharing in imperative programs based on the `pure` calculus (Capriccioli et al., 2016), where memory stores are modeled by rewriting the source code terms rather than by modifying the auxiliary storage.

Rely-guarantee protocols

The logic-based program verification that employs rely-guarantee reasoning has been another active area of research that verifies the correctness of usage protocols and avoids inter-thread interference in concurrent programs (Parkinson and Bierman, 2005; Vafeiadis and Parkinson, 2007; Dinsdale-Young et al., 2010). However, in these approaches, rely-guarantee specifications are explicitly added at the state or thread level to model and control the concurrent interactions safely.

The most recent is a sub-structural type system (Militão et al., 2014a) that uses `rely-gurantee` protocol abstraction to model the interfering interaction of aliases to the shared states. The type system then ensures that the aliases always get a determined value regardless of the potential changes made by the program context during interleaving. However, the system explicitly assigns a separate role to each alias with a "rely $\Rightarrow$ guarantee" relation between aliases. Later, Militão et al. (2016) extended the approach and developed a composition procedure based on linear capabilities (Morrisett et al., 2005), to address the decidability of protocol composition and its integration with the protocol abstraction.

Separation logic

Among other logic-based approaches for data race freedom, Separation Logic that is based on Hoare logic, attained much attention for controlling aliasing and verifying program behavior.

Hoare's logic for conditional critical regions (Hoare, 1972) and monitors (Hoare, 1974) was widely adopted because of the simplicity and practicality of their use in Java-like programs. Hoare's logic limits thread interference to a few synchronization points. However, it cannot syntactically enforce a safe monitor synchronization. This is because of the potential risk of aliasing in a Java program where multiple threads can manipulate shared-memory data in an unsafe manner. O'Hearn (O'Hearn et al., 2001) and Reynolds (Reynolds, 2002) extended Hoare's logic to Classic Separation Logic, a new program logic with new connectives and separation conjunction ($*$), to reason about sequential programs that manipulate pointer data structures.

Hoare logic uses triples $\{P\}S\{Q\}$ where P and Q are predicates over program states that define the `required` and `ensured` properties of an expression statement S. However, in Separation Logic, the idea is to explicitly divide each program state, related to a current method call, into a heap and a store part, to allow explicit local reasoning about the heap memory. In this approach, the heap `h` is divide into two disjoint parts say h_1 and h_2 using separation formula of the form $\phi_1 * \phi_2$ where ϕ_1 is a pointer valid for the part h_1 and ϕ_2 is a pointer valid for the part h_2. The conjunction $*$ operator combines two disjoint parts of the same heap. The separation formula ensures that two threads accessing the same location do not interfere to verify program behavior. The idea of using separation formula to verify heap structure, dates back to the seminal work of Reynolds (1978) who proposed a syntactic interference control mechanism to constrain the effects of interference in Algol-like languages.

Eventually, Separation Logic was realized as a new program logic called Concurrent Separation Logic (CSL) (Brookes, 2004; OHearn, 2007) to reason about multi-threaded programs, with an assumption that if two threads can operate on disjoint parts of the same heap location without interfering with each other, they can be verified in a safe and isolated way. CSL enforces correct synchronization of the shared-memory data logically, rather than syntactically. The idea of CSL was then extended in several substructural type systems and concurrent approaches (Gotsman et al., 2007; Appel and Blazy, 2007; Hobor et al., 2008) to guarantee data race freedom in the shared-memory concurrent programs, and has been applied in high-order imperative concurrent languages and type systems (Schwinghammer et al., 2011; Jensen and Birkedal, 2012). However, in Separation Logics-based approaches, the separation predicates are explicitly specified in the program to define the access rights on the memory locations.

A move to permission-based specifications

Among all formal approaches to the verification of shared-memory programs such as atomic set, behavioral type, session-type, rely-guarantee reasoning and Separation Logic, is access permission. Access permission is an abstract capability that combines type (effect) systems and, provides more advanced support for reasoning about heap resources (Boyland, 2003).

The notion of access permissions is built on Linear Logic (Girard, 1987), that treats permissions as linear resources, and the Classic Separation Logic (O'Hearn et al., 2001) that performs local reasoning of program behavior against specifications. However, Classic Separation Logic does not support the concurrent read access of a memory location by multiple references or threads. Therefore, Boyland (2003) and Bornat et al. (2005) combined Separation Logic with abstract capabilities, called access permissions, to allow concurrent reading of a program state.

Compared to classic verification methods such as Owicki-Gries (Owicki and Gries, 1976) for concurrent programs, the permission-based Separation Logic ensures that: (a) only one reference (thread) can write on a particular location at any given time, thereby mutating data in a safe way; (b) if a location is read by a thread, all other threads can only have read permission for that location, thereby implying data race freedom without the need to explicitly check for the interference between threads in concurrent programs.

Plural (Bierhoff and Aldrich, 2007; Beckman et al., 2008), a permission-based program verifier, was the next development phase where access permissions were combined with typestate abstractions to statically ensure the mutability of object's states, following Separation Logic, and to verify protocol compliance in Java-like sequential and concurrent programs.

Access permissions were then subsequently used in many formal approaches (Huisman and Mostowski, 2015; Müller et al., 2017) to identify and ensure the mutability of object's states and to verify program behavior in shared-memory programs.

3. Access permissions: An overview

Access permissions are abstract capabilities that can encode effects (read and write) and aliasing information of a referenced object at one place. There are three main categories of access permissions:

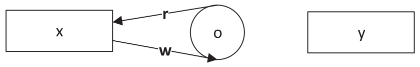
Fractional permissions (Boyland, 2003). A fractional (share) permission say s is a concrete mathematical value that defines a shared ownership (access) of referenced objects `o` in a concurrent setting where s is a fraction between 0 and 1 inclusive. A value 0 represents the absence of permission, while a value 1 represents full permission and any value greater than zero represents shared read-only access on the referenced object. Fractional permissions can be used to split a `full` permission (with value 1) into a number of fractions and then to distribute these fractions among multiple references, and so on. The splitting function for a `full` permission, say `s`, when divided between two references, say `s1` and `s2`, can be written as $s_1 + s_2 = s$ with each reference having a fraction of `s` in the range (0, 1).

Counting permissions (Bornat et al., 2005). A counting permission is a special fractional permission where `s` is an integer value between 0 and a maximum constant value, where zero represents the absence of permission and the maximum value represents `full` permission on the referenced object `o`. The read-only access on the referenced object `o` is represented by a non-zero integer value such that $0 < s \leq$ `max`.

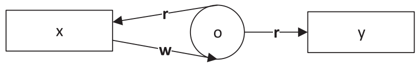
Symbolic permissions (Bierhoff and Aldrich, 2007). Symbolic permissions, simply called access permissions, are extension of Boyland's fractional permission sharing model but, instead of using concrete fractional value to represent and split permissions among multiple references, symbolic permissions represent and track permission flow through the system using permission types such as `unique` or `immutable` etc. Access permissions splitting and joining rules for symbolic permissions are given in Section 3.2.

There are five types of symbolic permissions that can be assigned to a reference `x`, for a referenced object `o`, in the presence of its alias `y`.

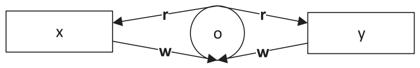
`unique(x)`: This permission provides reference `x` an exclusive read and modify access on the referenced object `o` at any given time. No other reference (e.g. `y`) to the same object can co-exist while `x` has unique permission on an object.



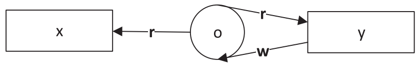
`full(x)`: This permission grants reference `x` with read and write access to the referenced object `o`, and at the same time `o` may also be read, but not written, by other references such as `y`.



`share(x)`: This permission is the same as `full(x)`, except that now other references such as `y` can also write on the referenced object `o`.



`pure(x)`: This permission gives reference `x` a read but not a write access on a referenced object `o`. Moreover, other references such as `y` may have both read and write access on the same object.



`immutable(x)`: This permission grants a non-modifying access on the referenced object `o` to both the current reference `x` and any other reference such as `y`.

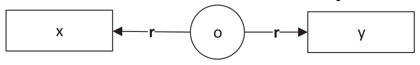


Table 1 summarises how access permissions can co-exist on a referenced object `o` by the current reference (*This Reference* `x`), and by another reference (*Other Reference* `y`).

Table 1
Co-existing access permissions (Bierhoff and Aldrich, 2007).

This Reference x	Access Rights	Other Reference y
unique	read/write	none
full	read/write	pure
share	read/write	share, pure
pure	read	full, pure, immutable
immutable	read	immutable, pure

Table 2
Access permissions splitting and joining rules (Bierhoff and Aldrich, 2007).

Splitting and joining Rules	Rule #
unique(x;o;k) $\Leftrightarrow$ full(x_1;o;k_1) $\otimes$ pure(x_2;o;k_2)	Rule I
unique(x;o;k) $\Leftrightarrow$ immutable(x_1;o;k_1) $\otimes$ immutable(x_2;o;k_2)	Rule II
full(x;o;k) $\Leftrightarrow$ share(x_1;o;k_1) $\otimes$ pure(x_2;o;k_2)	Rule III
share(x;o;k) $\Leftrightarrow$ full(x_1;o;k_1) $\otimes$ pure(x_2;o;k_2)	Rule IV
immutable(x;o;k) $\Leftrightarrow$ pure(x_1;o;k_1) $\otimes$ immutable(x_2;o;k_2)	Rule V
unique(x;o;k) $\Leftrightarrow$ share(x_1;o;k_1) $\otimes$ share(x_2;o;k_2)	Rule VI
immutable(x;o;k) $\Leftrightarrow$ immutable(x_1;o;k_1) $\otimes$ immutable(x_2;o;k_2)	Rule VII
share(x;o;k) $\Leftrightarrow$ share(x_1;o;k_1) $\otimes$ pure(x_2;o;k_2)	Rule VIII
share(x;o;k) $\Leftrightarrow$ share(x_1;o;k_1) $\otimes$ share(x_2;o;k_2)	Rule X
full(x;o;k) $\Leftrightarrow$ full(x_1;o;k_1) $\otimes$ pure(x_2;o;k_2)	Rule XI

3.1. Access permission contracts in the spirit of Linear Logic and the Design by Contract Principle

Linear logic (Girard, 1987) traditionally treats access permissions as resources that cannot be duplicated (discarded). Access permission contracts in Linear Logic are specified using the Linear Logic implication connective ($\multimap$). The connective ($\multimap$) operator is used to specify a method's pre- and post-conditions in Linear Logic. As indicated by P $\multimap$ Q, permissions in the pre-conditions P are consumed before a method runs, and it produces Q as post-conditions when the method completes its execution. Once a method consumes permissions on the referenced objects they are no longer available to other methods until the method returns the consumed permissions on the same objects.

In Design by Contract Principle (Meyer, 1988), contracts are obligations and rights of the client and the implementing class itself. Contracts are specified using the `requires` and the `ensures` clauses that represent a method's pre and post-conditions respectively (Meyer, 1992; Leavens et al., 2006).

In the spirit of the Design by Contract Principle, permission-based specifications at method level represent `contracts` where permission-based obligations are defined as pre-conditions P that client of a class must guarantee before calling methods of the class, and permission-based rights represent post-conditions Q that must hold for both the client and the implementing class after executing the specified method. The idea of specifying pre- and post-conditions as contracts date back to Hoare's work (Hoare, 1969) on formal verification of software applications and has recently been applied to permission-based verification and parallelization approaches (Cataño et al., 2014; Militao et al., 2010; Huisman and Mostowski, 2015; Müller et al., 2017).

3.2. Access Permission splitting and joining rules

Access permission can be split into one or more relaxed permissions (fractions of original permission, using fractional values in the range (0,1)) and then merged back into more restrictive or original permission. This phenomenon is known as fractional permission analysis where fractions keep tracks of the way the permissions were split and joined back. This information can be used to verify system properties based on certain specific criteria and to parallelize execution of a program by tracking permission-based dependencies in the program. Table 2 shows access permissions splitting and joining rules.

In Table 2, let x represent current reference, o represent the referenced object and k represent the fraction of permission assigned to a particular reference, where at least one of x_1 and x_2 is x, and $k_1 + k_2 = k$. The operator multiplicative conjunction (A $\otimes$ B) denotes the simultaneous occurrence of permissions by multiple references, say x_1, x_2, on the same referenced object o. The symbol $\Leftrightarrow$ represents the two way operation of splitting and joining permissions. For example, `unique` access permission (Rule-I) having k fractions can be divided into k_1 fraction of `full` and k_2 fraction of `pure` permission and then joined back to make the `unique` permission. Likewise, a `unique` access permission (Rule VI) can be split into two `share` permissions but cannot be split into a `share` and `immutable` permission as `immutable` cannot co-exist with the `share` permission. The linearity of resources forces the `unique` permission to be replaced by two `share` permissions which can be further split according to splitting rules and then combined to recover the original permission.

4. Permission-based verification of API protocols

This section introduces the first dimension of the state of the art permission-based verification approaches in detail. The focus is on the verification of API protocols for single- and multi-threaded programs. Table 3 provides a summary of the permission-based protocol verification approaches studied in this research.

4.1. Verification of API protocols in single-threaded programs

In object-oriented programs, objects define usage protocols. Usage protocols are constraints on the order of method invocations that the client of the protocol must follow for the underlying objects to work properly. Typestates have been used to specify usage protocols in many formal approaches where state information is associated with a variable of a given type, which is subsequently tracked to decide the valid operation sequence to be called on an instance of that type. It is generally acknowledged that statically tracking object's state is a challenging task in the presence of unrestricted aliasing. This can be attributed to the improper and unexpected state transition during program execution, and that can happen in situations such as a) when a method uses a data structure having aliases deeply nested in the hierarchy and causes side effects, or b) when aliased parameters are passed to a method expecting non-aliased parameters. Access permissions have been used, as part of formal specifications, to specify the intended design and to verify the correctness of usage protocols in API protocols in many formal approaches such as Plural (Bierhoff and Aldrich, 2007; 2008; Bierhoff, 2009; Bierhoff et al., 2009b), JavaSyp (Bierhoff, 2011) and Plaid (Aldrich et al., 2011; 2012) and other related techniques.

Typestate verification using Linear Logic

Bierhoff (2009) presented a formal specification language and a type system to soundly and modularly verify API protocols in sequential programs based on access permissions. The aim was twofold, firstly to verify protocol conformance with actual program implementation in the presence of aliasing, and secondly to check whether the client of the program obeys the specified protocol.

With this stream of work, Bierhoff presented a permission-based modular protocol checking approach and a tool (Bierhoff and Aldrich, 2007) for a Java-like object-oriented language. In this approach, programmers express their design intents, as a valid sequence of events associated with a particular object, and the aliasing information using permission-based typestate contracts, written in Linear logic-based specifications at the method level. The

Table 3
Access permission-based protocol verification.

Ref.	Prog	Lang	Tool	Analy	Perm-Kind	Perm-Specs	Perm-Infer	Anno	Properties
Bierhoff and Aldrich (2007)	Seq	Plural(NSL)	Plural	(St,D)	Sym	(U,S,F,P,I)	N	Y	VoUP
Beckman et al. (2008)	Con	Plural(NSL)	Plural	St	Sym	(U,S,F,P,I)	N	Y	VoUP, RCwAB
Beckman (2009)	Con	Plural(NSL)	Sync-or-Swim	St	Sym	(U,I,S,F,P)	N	Y	VoUP, RCwSB
Militao et al. (2010)	Seq	–	Plural	St	Sym	(U,I,S,F,P)	N	Y	RCs
Bierhoff (2011)	Seq	Java-like(NSL)	JavaSyp	St	Sym	(U,I)	N	Y	CME
Aldrich et al. (2011)Aldrich et al. (2012)	Seq	Plaid(NSL)	Plaid	(St, D)	Sym	(U,S,I)	N	Y	VoUP
Naden et al. (2012)	Seq&Con	Plaid(NSL)	Plaid	(St,D)	Sym	(U,S,I)	N	Y	RCs

Keys to the table: Seq = sequential, Con = concurrent, St = static, D = dynamic, *Sym* = symbolic, Fract = fractional, U = unique, I = immutable, S = share, F = full, P = pure, NSL = new specification language, RCs = race conditions, VoUP = verification of usage protcols, RCwAB = race conditions with atomic blocks, RCwSB = race conditions with synchronized blocks, CME = concurrent modification exceptions, $\mathbb{Z}$ set of Integers, $\mathbb{R}$ set of Real numbers, $\mathbb{N}$ set of positive Integers.

```
 1 interface Iterator<c : Collection, k : Fract> {
 2 states available, end alive
 3 boolean hasNext():
 4 pure(this) ⊸ (result = true ⊗ pure(this) in available)
 5            ⊕ (result = false ⊗ pure(this) in end)
 6 Object next(): full(this) in available ⊸ full(this)
 7 // other methods such as finalize() etc. can be written similarly
 8 interface Collection {
 9  //  methods such as  add(), size(), remove(), contains() etc. comes here
10  Iterator<this, k > iterator(): immutable(this, k)⊸ unique(result)}
11 }
```

Listing 1. Permission-based typestate specifications of a read-only Iterator (Bierhoff and Aldrich, 2007).

system supports five kinds of symbolic permissions i.e., unique, immutable, full, share and pure. The Linear logic operators are already explained in Section 3.2, except the additive disjunction (⊕) that represents an alternative occurrence of multiple tasks.

Listing 1 shows a sample permission-based typestate contract for a read-only iterator. The aim is to avoid the concurrent modification of `Collection` object when the `iterator` is in progress.

According to the usage protocol, an `iterator` can be in one of the two states at any moment i.e. `available` or `end`. The state `alive` is a state inherited from the root `Object` type (Line 2). In Line 11, an `iterator` object is created with unique permission in `Collection` class. Importantly, it can be observed that when an iterator is created, it stores a reference to a collection object being iterated in one of its fields. This reference should be associated with the appropriate permission i.e., immutable to guarantee immutability of the `Collection` object while iteration is in progress.

The pre-condition (pure(this)) in Line 4 specifies that method `hasNext()` requires pure permission on the referenced object as it just tests or reads iterator's state. As method `next()` can change `iterator`'s state, it needs full (this) (Line 6). The method `hasNext()` determines whether another object is already present in the Collection object with `available` state, or if the iteration has reached its `end`. The post-condition (Line 4 and 5) specifies if the result is true. It is legal to call the `next()` method on the same object in the `available` state, Otherwise, it is illegal. The post-conditions of both methods further show that they return the consumed permission back on the referenced object when they exit. The system leverages this information through method implementations to track state transition in the presence of aliasing, to guarantee the absence of concurrent modifications, and to verify whether program implementation follows the design intents.

This approach is realised in Plural (Bierhoff and Aldrich, 2008), a permission-based automated protocol checking and conformance tool, implemented as a Java Eclipse plugin. In Plural, a program is specified with permission-based pre and post-conditions at method (parameter) level using JSR-175[1] annotations to check whether the client of the APIs follows the specified protocol.

[1] https://jcp.org/en/jsr/detail?id=175.

Plural performs intra-procedural static analysis, called Data Flow Analysis (DFA) of the annotated code to identify and track specified pre- and post-permissions across method calls for every program variable (parameter, receiver object, and local variable) and issues warning for protocol violations in the program. Plural is built on top of a Crystal analyzer[2] that performs dynamic state tests (branch-sensitive flow analysis) of a program to track and report exceptions to the underlying objects. Plural also checks the structure of the provided specifications by implementing an Annotation analyzer. The Effect checker in Plural identifies whether a method is immutable or whether it produces side effects. The Fraction analyser tracks the flow of permissions through the system to split and join permissions associated with a referenced object.

Later, Bierhoff et al. (2009b) extended the modular protocol checking approach to check the soundness and effectiveness of Plural in specifying large case studies for real APIs and large third party open-source code bases, for example, Database Connectivity (JDBC) API in Apache Beehive project[3] and PMD, a static code analyzer from DaCapo benchmark[4] that implements Java `iterator` API. The objective is to measure the precision in terms of false positives, the computational cost and the annotation overhead associated in manually specifying and verifying these APIs with Plural annotations.

The approach follows the Design by Contract Principle to explicitly specify state invariants at the method level. State invariants are permission-based typestate assertions with a valid typestate that should hold when an object is in a specific state. The approach uses the concepts of 'capture' and 'release' permissions to avoid inter-object dependencies at the method level. Listing 2 shows a sample code snippet in Plural for `Connection` class in Java JDBC API.

In Listing 2, `@States` annotation (Line 1) specifies concurrent typestates for a `connection` object. The method `createStatement()` (Line 5) creates statements, in the `Connection` interface, with unique permission in 'open' state.

[2] https://code.google.com/archive/p/crystalsaf/.
[3] http://beehive.apache.org/.
[4] http://dacapobench.org/.

```
1 @States({"open", "closed"})
2 public interface Connection {
3  @Capture(param = "conn")
4  @Perm(requires = "share(this, open)", ensures = "unique(result) in open")
5  Statement createStatement() throws SQLException;
6  @Full(ensures = "closed")
7  void close() throws SQLException;
8 }
```

Listing 2. A Java JDBC connection interface (fragment) with Plural specifications (Bierhoff et al., 2009b).

Table 4
Annotation overhead for sample APIs verified in (Bierhoff, 2009; Bierhoff et al., 2009b).

Program Statistics and Annotation Overhead			
Program	SLOC	Methods	#Annotations
JDBC	9,866	440	838
Beehive	2,158	65	66
PMD	39,400	-	617

Keys to the table: SLOC = source line of code.

Table 5
Summary of annotation overhead for sample programs in Plaid (Stork et al., 2014).

Program Statistics and Annotation Overhead			
Program	SLOC	#AnnoLOC	#Annotations
webserver	227	47 (20.7%)	59
dic/global	169	41 (24.2%)	65
dic/fine	251	71 (28.3%)	109

Keys to the table: SLOC = source lines of code, AnnoLOC = annotated lines of code.

When the connection object is closed it invalidates all the statements created with it, leading to runtime errors if a programmer uses invalidated statements. To avoid this error, the `createStatement()` method captures the `this` Connection and treats it as the `conn` parameter of the returned `Statement` object, using `@Capture` annotation in Line 3. The annotation @Perm in Line 4 with `requires`, clause specifies a share permission on the captured object as the pre-permission with open state guaranteed, and the `ensures` clause specifies that method returns a new statement object in open state with a unique permission on it.

The permissions on the `conn` object are explicitly released (using `@Release`) when method `close()` is called on the `Statement` object or when the `Statement` object is no longer in use. The method `close()` calls method `isClosed()` to check status of the `conn` object before closing it, (not included in the sample program due to brevity). It ensures the current state of the connection object to be `closed` (using annotation @TrueIndicates ("closed")). The `@Full` annotation on method `close()` in Line 6 indicates that `Full` permission is both passed into and returned from the method. The `ensures` clause in Line 6 specifies the state at the return point; either open or closed states are permissible when it is called, so there is no `requires` clause, but a `Full` permission should still be passed to the `close()` method.

The analysis then tracks the input specifications to verify protocol compliance with program implementation and to verify correct usage of the protocol by the client program. However, the verification is performed at the cost of manually annotating program with permission based specifications. Table 4 shows the annotation overhead reported by the authors in the paper.

Typestate verification using Plaid

Plaid (Aldrich et al., 2011; 2012) is a new typestate-oriented programming language that verifies the correctness of programs based on access permissions. The aim was to extend the previous typestate-oriented language (Aldrich et al., 2009) with first class-states and access permissions. Further, having access permission support in the Plaid infrastructure, Stork et al., (Stork et al., 2014) developed a by-default concurrent programming language and a runtime system, to parallelise execution of sequential programs based on access permissions.

Every type in Plaid is represented as tuples having a type structure and associated permissions that express the aliasing and the mutability of the corresponding object's typestate. Plaid borrows its grammar and lexical structure from the Java Specification Language (JSL) and provides interoperability with Java programs. Classes in Plaid are represented using the keyword `state` and transitions between states are represented by the state transition symbol '$\gg$' that distinguishes pre-state from post-state. Plaid supports three types of symbolic access permissions i.e., unique, immutable and share in the specifications. The keyword 'none' is used when no permissions are required to access an object.

Listing 3 shows an annotated code fragment for a buffer implementation in Plaid. A buffer can be in one of the two states i.e. `EmptyBuffer` and `FullBuffer` (Line 1). The method `put()` is associated with an empty buffer. The signature of the method `put()` (Line 4) specifies that the state of the receiver object should change from `EmptyBuffer` to `FullBuffer` when the buffer receives some element. The state `FullBuffer` requires a field element `elem` that is passed as method parameter `e`. The permission-based contract (Line 4) specifies that the passed element has `unique` permission in the `Element` state and the method does not return any permission (`none`) to the caller of the method. This is because a field reference with exclusive rights (`unique`) has been created for that element in `FullBuffer` state. Otherwise returning permission back to a caller would cause a violation of the uniqueness property. Likewise, the `FullBuffer` state has a single operation `get()` (not included here due to brevity), that returns the current state of the object in reference `elem` and ensures that the receiver object will go back to an `EmptyBuffer` state.

Plaid runtime leverages permission flow through the system along with associated typestate information to ensure protocol compliance at runtime. The type system allows permission splitting, joining and type casting automatically (when and where required) using the permission splitting and joining rules given in Table 2. However, the verification comes at the cost of manually adding permissions-based typestate information in the program. Table 5 shows the annotation overhead reported by authors themselves in the paper.

Naden et al. (2012) presented a type system and a flexible permission borrowing mechanism for `unique`, `shared`, and `immutable` permissions without using explicit fractions of permissions. Permission borrowing is an extraction of permission from a source field, temporarily using the borrowed permission, and returning part or all of it to the source field. The aim was to prevent the concurrent modifications of the shared objects based on symbolic permissions.

```
1 state Buffer comprises EmptyBuffer, FullBuffer {}
2 ...
3  state EmptyBuffer caseof Buffer {
4  method void put(unique Element ≫ none e) [EmptyBuffer ≫ FullBuffer] {
5    this ← FullBuffer {elem = e};}
6 }
```

Listing 3. A buffer implementation (fragment) in Plaid (Aldrich et al., 2011).

```
1 public class ArrayList<T> {
2  @Excl private T[] a;
3  private int size ;
4  //@invariant 0 <= size & a != null & size <= a.length;
5  //@requires  0 <= index & index < size ;
6  @Imm public T getElem(int index) {
7    imm: return a[index];}
8 }
```

Listing 4. A Java array list (fragment) with permission-based JML contract in JavaSyp (Bierhoff, 2011).

```
1 class Connection {
2 boolean isConnected():share(this, ?) ⊸
3  (result == true ⊗ share(this, CONNECTED)) ⊕
4  (result == false ⊗ share(this, IDLE)) {
5  atomic:{
6   return (this.socket != null);}
7  }
8 }
```

Listing 5. Permission-based typestate specifications for method isConnected() in a Connection class (Beckman et al., 2008).

The type system is based on the Plaid language. Like Plaid, it supports three types of symbolic permissions i.e unique, immutable and share but unlike Plaid, where a field is reassigned with a new value to recover permission on the reference, in the Naden's type system the caller function itself returns the original permission consumed on the reference.

Unlike other techniques (Boyland, 2003; Bierhoff and Aldrich, 2007; Jacobs et al., 2011; Heule et al., 2011) that support permission borrowing, this approach provides a more intuitive and natural abstraction to model and to reason about the permission flow through the system, making permission tracking flexible and much easier for programmers. However, like Plaid, it expects programmers to explicitly specify permissions-based state information as a part of method specifications.

Typestate verification using JML

Bierhoff (2011) combines symbolic permissions (Bierhoff and Aldrich, 2007), with JML contracts (Leavens et al., 2006) to reason about aliasing and to detect the absence of Concurrent Modification Exceptions (CMEs) and other recurrent programming errors, such as `IndexOutofBoundsExceptions` exceptions in realistic data structures such as Java ArrayList.

Although JML specifications have been used, in may formal approaches (Rodríguez et al., 2005; Araujo et al., 2008; Kim et al., 2009; Cok, 2011), to verify functional correctness and domain-specific properties of sequential and concurrent programs, the support for concurrency and aliasing is rather limited. On the contrary, access permissions provide flexible aliasing control mechanism to track all the references of a particular object and update state changes to all such references. The presented approach defines permission-based class invariants as JML contracts.

The technique implements a permission tracking algorithm as a prototype tool called JavaSyp [5] (Symbolic Permissions for efficient static program verification). In JavaSyp, permission-based invariants are specified using Java annotations and tracked as part of the type checking procedure, to ensure that the specified invariants hold as long as a the client has permission to the referenced object and to control aliasing. In this approach, permission tracking is straightforward as tracking symbolic values is much easier than tracking fractional permissions. However, it only supports two kinds of permissions, i.e., unique and immutable using annotations `@Excl` and `@Imm` respectively with the referenced object.

Listing 4 shows an annotated version of conventional Java ArrayList object a declared with unique permission in Line 2. The list object maintains a list of elements in the order placed originally. The method `getElem()` method returns the element on the given location (`index`) with immutable permission on it (Line 6). The invariants (Line 4) for method `getElem()` specifies that object a should be a non-null reference having at least one element in it and the total number of elements in a should not exceed the declared `size`. The annotation `@requires` in Line 5 specifies a pre-condition for method `getElem()` that, before calling this method, the `index` parameter must be between 0 and `size-1`. JavaSyp performs static analysis of the annotated code to generate verification conditions (VCs) based on the specifications. The program is then verified against inferred conditions using the SMT solver (Barrett et al., 2010).

4.2. Verification of API protocols in multi-threaded programs

Beckman (2010) extended the Bierhoff's modular automatic protocol checking approach to verify "if the object protocols work correctly even in the presence of concurrent modifications by multiple threads". In this stream of work, this section discusses the use of permission-based specifications to verify usage protocols in concurrent programs (Beckman et al., 2008; Beckman, 2009; Militao et al., 2010) using different mechanisms.

Typestate verification with atomic blocks

Beckman et al. (2008) extended the permission-based modular protocol checking approach (Bierhoff and Aldrich, 2007) to verify the correctness of usage protocols for a set of concurrent programs such as JChannel[6] and Reservation Manager that use atomic blocks as synchronization primitives. The objective was to enforce the correct use of typestates at runtime and to verify API protocol compliance with its specifications. The approach uses five kinds of symbolic permissions to identify aliasing and to approximate whether a referenced object can be thread-shared or not.

Listing 5 shows a sample method `isConnected()` in a `Connection` class with permission-based typestate specifications.

The pre-condition "`share(this, ?)`" in Line 2 asserts that the method needs share permission on the receiver object, which needs to be in the unknown (`?`) state. Likewise, the post-condition in Line 3 specifies that if the method returns `true`, the receiver object should get the original permission back while in the `CONNECTED` state Otherwise, it would get the same permission

[5] http://code.google.com/p/syper.

[6] http://www.cs.cmu.edu/~nbeckman/research/atomicver/.

Table 6
Annotation overhead for sample programs verified in (Beckman, 2010).

Program Statistics and Annotation Overhead					
Program	Classes	APIs	Methods	SLOC	#Annot.
JabRef	813	7	4,072	74,217	268
JSpider	187	1	951	8,955	30

Keys to the table: SLOC = source lines of code, Anno. = individual annotations.

back but in the IDLE state in Line 4. Exclusive access to full, share and pure references are maintained using atomic blocks in Line 5.

The approach is realised as part of the Plural tool. The analysis identifies the abstract state of a referenced object before calling a method, and discovers the way it would be shared with other objects. If the permission on a particular reference (thread) indicates that the referenced object can be accessed simultaneously by other references (threads), as is the case with full, share and pure permissions, it assumes that the object is thread-shared.

The approach discards state information of the local variables, this happens only when objects are not accessed inside an atomic block, having pure and share permissions as the objects with these permissions can be modified by other threads and it is difficult to track them statically through atomic blocks.

The limitation of this work is the use of atomic blocks as mutual exclusion primitives. Atomic blocks are generally associated with transactional memory systems and have limited usage in today's applications. The current analysis generates false positives for programs having synchronization primitives other than atomic blocks.

Typestate verification with synchronized blocks

Beckman (2009) extended the previous type system to perform typestate verification of concurrent programs having synchronized blocks as mutual exclusion primitives.

In this approach, every program reference is associated with a permission kind (having a permission type and an abstract state that is part of the reference type). Like Beckman et al. (2008), the system distinguishes between thread-local and thread-shared objects based on permission contracts. The approach is implemented in a tool called Sync-or-Swim for Java that performs static analysis of the program (within a method). It identifies the references on which the current thread is known to have synchronized and tracks the permissions associated with references as they flow with method's pre- and post-conditions, to ensure that an object can be modified concurrently. The analysis discards state information for thread-shared (modified by other references) objects unless it is statically known that the same references have previously been synchronized.

Unlike other behavioral checkers for concurrent programs (Jacobs et al., 2005) that require lock-based specifications to identify the part of heap to be protected, the proposed approach can verify program behavior without requiring lock-based specifications. Like other single-threaded typestate verification approaches, it requires programmers to explicitly specify the aliasing (permission) and typestate information at the code level. Table 6 shows the manual effort (annotation overhead), reported by authors, to verify program behavior for the sample programs in Sync-or-Swim tool.

Typestate verification with views

Militao et al. (2010) expands on Bierhoff's permission type system for Plural (Bierhoff and Aldrich, 2007), by going beyond the five traditional types of permissions. The approach introduces a new abstraction called View that is a projection of an object with a small set of access permissions associated with individual components (fields and/or methods) of an object.

The type system combines view-based controlled aliasing with typestates and Boyland's fractional permissions to manage safe initialization of different sections of an object reference, to track state information and to ensure safe access of the referenced objects in a unique-writer and multiple-readers scenario. An immutable view can be shared with an inbound number of copies and a write request merges all the readers back to a single writer using fractional permissions. However, it does not support aliasing of the form where an object can be shared between multiple writers with a state guarantee. In this approach, a view behaves as a state except that it can be split, merged (recombined) using fractional permissions. Therefore, it resembles the permission accounting model (Bornat et al., 2005) where views are treated as accountable parts of a typestate thereby, allowing local reasoning of the shared-memory programs.

The analysis of all the Plural-based verification approaches in Sections 4.1 and 4.2 shows that Plural can identify common challenges for specifying and implementing usage protocols in real-world case studies. It helps programmers to statically follow usage protocols without actually executing the program. However, Plural analysis is limited as it cannot identify errors in the specifications and might use non-consistent specifications. Moreover, there is no reachability analysis support in Plural, which means that a programmer may write inconsistent specifications at the method level and consequently, methods with these specifications will never be called by any client code, resulting in unreachable code.

Complementing the Plural tool, research (Siminiceanu et al., 2012), has been done to verify the correctness of manually added Plural specifications as well as verifying the program behavior. Further, it provides limited support to the verification of typestate invariants. It can only check invariants on boolean properties, e.g. checking for non-nullness, However, it cannot verify invariants that involve arithmetic predicates, e.g. $x > 0$ (where x is an integer field). Moreover, it requires programmers to explicitly specify the design intents of the API protocols as permission-based typestate specifications in the program which results in annotation overhead for the programmers.

In the same fashion, Plaid and related approaches forces a client program to follow the desired sequence of method calls to verify the correctness of typestate-based programs, but at the cost of adding permission-based annotations as a part of type declarations at the code level. Moreover, Plaid does not support full and pure permission as for its analysis. In Plaid, permission-based typestate specifications are added as a part of the program to verify proper state transition between multiple objects during the execution of a method.

5. Verification of race conditions and deadlocks in multi-threaded programs

In imperative and object-oriented programming languages, the biggest challenge has been the correctness of concurrent multi-threaded programs in the presence of aliasing and to avoid domain-specific problems such as deadlocks and race conditions. Access permissions have been used to characterize the way a shared resource can be accessed by multiple threads and to handle aliasing in many verification approaches. The general idea is to assign permission to program references to access memory locations and track the permission flow through the system to enforce mutual exclusion mechanisms in shared-memory concurrent programs.

This section discusses the second dimension of the survey i.e., the use of permission-based specifications for verification of domain specific problems in concurrent programs.

5.1. Permission sharing and accounting models

Permission sharing and accounting models (Boyland, 2003; Bornat et al., 2005; Parkinson and Bierman, 2005; Appel and Blazy, 2007; Hobor et al., 2008; Dockins et al., 2009) facilitate thread-local reasoning for shared-memory concurrent programs and to ensure race-free sharing of heap locations.

As discussed previously in Section 3, the Boyland's permission sharing model (Boyland, 2003) also called fractional permission, defines shared ownership of resources in a concurrent environment. The sharing policy maps a permission fraction (share) as a rational number $\mathbb{R}$, in the range (0, 1], to allow read or write operation on a particular memory location. The fractional model has been used to handle problems that follow concurrent divide-and-conquer algorithms where a shared (read) permission can be divided into multiple shared permissions, to an unbounded depth, for any possible pattern of divide-and-conquer.

Although fractional permissions are infinitely splittable, this permission sharing model does not satisfy the disjointness property because rational numbers are not ideal for resource sharing, as shown by Parkinson and Bierman (2005), who proposed a permission sharing model, that allows both read sharing and disjointness of resources, to formalize and verify a subset of single-threaded Java programs with Separation Logic. In this model, resource invariants are defined using permission-based abstract predicates defined at the `Object` class level with an empty footprint, (permissions associated with a memory location), that each subclass extends to hold additional fields.

Bornat et al. (2005) proposed a permission accounting model and a light-weight verification approach to handle the accounting problem associated with reader-writer locks in concurrent programs. The approach extends separation relation $\mapsto$ in classic Separation Logic (Reynolds, 2002) and associates fractional permissions with each heap location to allow read sharing of heap locations. In this approach, each heap location x is treated as a map having E addresses with a permission value z, where z represents the level of permission carried by a heap location, as shown in Formula (1).

$$x \underset{z}{\mapsto} E \Rightarrow 0 \le z \le 1 \tag{1}$$

The idea is to count the number of shared tokens using an integer counter say s. It is incremented or decremented when a reader locks (receives a read token) or unlocks (returns the share token) respectively, $s > 0$ means there are outstanding read tokens but $s = 0$ implies the absence of outstanding readers, which means that a writer may acquire, hence ensuring a race-free sharing of heap locations.

Appel and Blazy (2007) presented the operational semantics and developed a Sequential Separation Logic to extend the C Minor language, a mid-level imperative programming language as a machine independent-intermediate language. The approach evaluates expressions as functions in Coq[7], a formal language that combines mathematical functions, axioms and theorems together with a semi-interactive environment to develop machine-checked proof assistants. The approach provides an end-to-end machine-checked correctness proof of the proposed logic in Coq. Unlike the classical Separation Logic (O'Hearn et al., 2001) where expressions are evaluated independent of heaps, the approach associates each expression evaluation with a footprint. "A footprint is a mapping from memory addresses (ν) to permissions" (Appel and Blazy, 2007).

In the proposed semantics, footprint (ϕ) is considered as a set of fractional permissions (Bornat et al., 2005) to verify non-interference of load and store operations in memory. A memory store yields result only if reading or writing a chunk of memory type, say *ch* at location ν is legal according to its footprint. For example, the semantics $\phi \vdash load_{ch}\nu$ (or $\phi \vdash store_{ch}\nu$) depicts that all the addresses from location ν to $\nu + |ch| - 1$ can be read (or write). Loading memory outside the footprint yields exceptions and causes expression evaluation to stop. The disjoint sum of two footprints $\phi_0 \oplus \phi_1 = \phi$ ensures the exclusive read/write or read-only ownership of the underlying memory.

Hobor et al. (2008) extended the Appel and Blazy's machine-checked soundness proof and Leroy's compiler-correctness proof in a concurrent setting, and developed a concurrent C Minor language having shared-memory and first-class locks and thread. He proposed a modular concurrent operation semantics as a generalization of Concurrent Separation Logic (CSL) (OHearn, 2007) but it goes beyond CSL as it allows dynamic lock and thread creation.

In the semantics, a world w corresponds to a footprint ϕ as in the work of Appel and Blazy (Appel and Blazy, 2007). A world specifies permissions for the current thread but this semantic deals with load (store) operations for multiple threads. The need was to evaluate an expression with a guarantee that footprints (ϕ) of different threads are disjoint. For this purpose, the approach defines permission-based lock invariants to grant or restrict ownership for the accessed memory by extending the classic separation relation $\mapsto$ as follows:

$$e \underset{\pi}{\mapsto} R \tag{2}$$

The relation shows an expression e maps to a memory address with resource invariant R. Every expression acquires a lock before its evaluation. Each lock is associated with a resource invariant R, where each invariant is supported by a unique set of memory addresses and worlds that inform the lock ownership π acquired or lost by each thread. The approach implements Parkinson's (Parkinson and Bierman, 2005) permission sharing model to define ownership. A 100% share represents full ownership and a non-empty ownership ($0 < \pi < 100\%$) represents read-only access. Any access without ownership means the program has no semantics and the evaluation stops.

Dockins et al. (2009) proposed a tree share permission accounting model that is more powerful than Bornat's token accounting model. It rectifies the problems with Parkinson's permission model. Tree share is a boolean-labelled binary tree that supports both splitting and token accounting for the shared reading of resources in concurrent settings. Although Boyland's fractional share model is infinitely splittable, it does not satisfy the disjointness property and may pose read/write and write/write race conditions. Similarly, in Bornat's token accounting model, a central authority lends out the total permission into shares in the form of permission tokens when and where required. It counts the outstanding tokens to verify permission accounting. These models satisfy the positivity of resources but not the disjointedness.

Unlike previous share accounting models, Dockins et al. (2009) defines heaps as partial functions from memory locations (L) to values (V) with pairs of non-unit shares (S). The token factories are represented using non-negative integers and the tokens themselves are represented using negative integers. When a token is pushed back into the factory, the integers are added. The token factory's share becomes zero when it gets all its tokens back. The extended points-to operator relation is given below:

$$l \underset{s,\ n}{\mapsto} \nu \tag{3}$$

The relation specifies that memory location l contains a value ν with a non-unit share s that is indexed by an integer n. If n is zero, the share s is full. If n is positive, it means that n tokens are missing over s in the token factory, and the negative value for n depicts that token factory has $size - n$ shares. The extended model

[7] http://coq.inria.fr.

```
1 class Account{
2  read (this.x)
3  void getBalance(){ x; }
4  writes (this.x)
5  void setBalance(int newX) { x = newX; }
6  void deposit(int x){
7  requires (this) { balance = balance + x; }
8 }
```

Listing 6. Code segments showing read, write and lock usage annotations in Fluid.

supports and satisfies all the required properties such as disjointedness, cross, and infinite splitting in permission sharing models.

5.2. Permission-based verification techniques and tools

This section describes and discusses the permission-based verification approaches and tools such as Fluid (Zhao, 2007), Chalice (Leino et al., 2009; Leino and Müller, 2009), Verifast(Jacobs et al., 2010; 2011), Pulse(Siminiceanu et al., 2012; Cataño et al., 2014; Ahmed and Cataño, 2018), Heap-Hop (Villard et al., 2010), HIP/SLEEK (Hobor and Gherghina, 2012; Jacobs and Piessens, 2011), HJp (Westbrook et al., 2012), VerCors (Amighi et al., 2012; Blom et al., 2014; Amighi et al., 2014; Huisman and Mostowski, 2015; Amighi et al., 2015; Blom et al., 2017; Joosten et al., 2018) and Viper (Müller et al., 2016; Müller et al., 2017) that have been developed to resolve concurrency problems in concurrent and parallel programs.

Table 7 provides a summary of the existing permission-based verifications tools and related approaches to verify common concurrency problems.

Fluid

Zhao (2007) developed a permission-based language and a type system to enforce a fixed locking order mechanism in Java-style multi-threaded programs having unstructured parallelism and synchronization.

The technique was realised as a prototype tool in the Fluid project (Greenhouse and Scherlis, 2002; Greenhouse, 2003). In their approach, a program is explicitly annotated with a method's effects and lock invariants. The method's effect specifies the read or write operation on the current object `this` or any of its field e.g., the annotations `reads(this.x)` and `writes(this.x)` in Listing 6, in Line 2 and 4. The lock invariant specifies the synchronized access of a referenced object, inside the method body, using the `requires (this)` clause in Line 7, that a method call for the `deposit()` method should be inside a synchronized block to acquire a `lock` on the receiver object `this`.

The system then translates high-level access annotations into low-level (fractional) permissions to distinguish the read and write effects of a method on the referenced objects. The system assigns `unique` permission with a referenced object if it is being written in the method body and a value less than 1 is assigned for the read operation on the same object. The type system uses this information to ensure that a given expression can be executed with assigned permissions but it does not verify program behavior based on input specifications.

Further, Zhao et al. (2008) proposed a synchronization policy to avoid the unnecessary synchronization effects in the previous approach. The system uses "permission nesting" to interpret the safe and correct usage of lock-based specifications associated with a field.

Chalice

Leino and Müller (2009) presented a permission-based verification method to prevent problems such as deadlocks and race

```
1 class Cell {
2 int val ;
3 Cell Clone()
4 requires rd(this.val);
5 ensures acc(result.val) ∧ rd(this.val);{
6  Cell tmp := new Cell ;
7  tmp.val := this.val ;
8  return tmp ;}
9 }
```

Listing 7. A sample program with accessibility predicates in Chalice (Leino and Müller, 2009).

conditions, that arise due to dynamic locking orders in multi-threaded programs. The approach ensures concurrent sharing and un-sharing of objects among multiple threads based on Boyland's fractional permissions (Boyland, 2003). The system uses permission percentages (between 0 and 100) instead of permission fractions. A permission percentage is a fractional permission with a definite size that splits a field permission among several monitors or threads. A thread can access a shared object, (heap location), if it has permission to do so. The approach defines three types of permissions based on their percentages: 'Full', 'Some' and 'No'.

The technique was realized in Chalice (Leino et al., 2009), a concurrent program verifier that supports programs with fork/join parallelism, monitors invariants and automatically verifies the absence of deadlocks and data races. In Chalice, programmers annotate programs with permission-based contracts using access predicates for each heap location. The annotation `acc(o.f)` represents 'Full' permission (100%) on a field of object `o` that shows that a thread has exclusive access on `o.f`. The fractional permission having `n` percentage of the actual permission is represented as `acc(o.f, n)`. A non-zero ('Some') permission depicts read-only access to location `o.f`, denoted as `rd(o.f)`.

Listing 7 shows a sample method specifications in Chalice. The pre-condition of the method `Clone()` in Line 4 specifies that the caller of the method must possess non-zero (read) permission on location `this.val` before calling this method. Following the Design by Contract Principle, the post-condition in Line 5 specifies that the callee should generate `Full` permission on `result.val` field and return the input (read) permission to location `this.val`. Otherwise, the system will not be able to recover `Full` permission on it and consequently, the location would remain immutable forever.

The annotated program is analyzed to verify whether the code respects the permission contract for every thread scheduling, as permissions flow between threads and monitors or between multiple threads. The analysis verifies that the sum of permissions for all threads remains less than or equal to 100% to ensure thread non-interference.

VeriFast

Jacobs et al. (2010, 2011) developed a sound, modular automatic program verification tool VeriFast to verify single- and multi-threaded programs written in C and Java. To enable verification, the programmer defines lemma functions in the program. Lemma functions are like ordinary C functions, except that lemma functions and calls of lemma functions are written within annotations. In VeriFast, lemma functions are interactively specified in the program following the Separation Logic-style of specifications. They serve as proofs to ensure that a method terminates without producing any side effects in the system.

The approach simulates shared variables as heap locations and associates a permission coefficient using Boyland's fractional permission to each heap location to represent its access rights. The coefficient lies within (0,1] where 1 represents exclusive rights to manipulate a particular heap location and any value

Table 7
Permission-based verification of race conditions and deadlocks.

Reference	Prog	Lang	Tool	Analy	Perm-Kind	Specs	Perm-Infer	Anno	Properties
Boyland (2003)	Con	PSM	-	-	Frac	$[0, 1]\cap\mathbb{R}$	N	-	RCs
Bornat et al. (2005)	Con	PSAM	-	-	Count	$[0, 1]\cap\mathbb{Z}$	N	-	RCs
Parkinson and Bierman (2005)	Seq	Java-like (NSL)	-	-	Counting	total read*	N	-	RCs, VoADT
Appel and Blazy (2007)	Seq	CMinor (PSAM)	Coq	-	Counting	$[0, 1]\cap\mathbb{N}$	N	-	RCs, MCCP
Zhao (2007) Zhao et al. (2008)	Con	Java	Fluid†	St	Frac	read write*	N	Y	RCs
Hobor et al. (2008)	Con	CMinor (PSAM)	Coq	-	Counting	[0, 100]%	N	-	RCs, MCCP.
Dockins et al. (2009)	Con	CMinor	Coq	Frac	-	(s, n), $n \in \mathbb{Z}$	-	Y	MCP
Leino and Müller (2009)	Con	Chalice	Chalice	D	Frac	full some no	N	Y	RCs, DLcks
Leino et al. (2009)	Con	Chalice	Chalice	D	Frac	acc(x) rd(x)‡	access pure	Y	RCs, DLcks
Villard et al. (2010)	Con	NSL	Heap-Hop	SFEA	-	$x \mapsto C$	N	Y	RCs, DLcks MLeaks, VoUP
Jacobs et al. (2010, 2011)	Seq& Con	C and Java (NSL)	VeriFast	D	Frac Count	read write*	N	Y	RCs, VoNullP, AIOBEx
Jacobs and Piessens (2011)	Con	NSL	VeriFast	D	Frac Count	$[0, 1]\cap\mathbb{R}$	N	Y	FGSM, RCs
Hobor and Gherghina (2012)	Con	NSL	HIP SLEEK	D	Frac	$(0, 1]\cap\mathbb{R}$	N	Y	RCs
Westbrook et al. (2012)	Con	HJ	HJp	St	Frac	Shared read (ωR) private write (ωW)$\omega \in \{0, 1, \epsilon\}$	N	Y	RCs
Siminiceanu et al. (2012) Cataño et al. (2014)	Seq&Con	Plural	Pulse	St RGA	Sym	(U,I,S,F,P)	N	Y	RCs, DLcks VoAPs, VoNullP
Amighi et al. (2012); Blom et al. (2017)	Con	OpenCL PVL Java OpenMP	VerCors	D	Frac	Perm(x, π), $\pi \in (0, 1]\cap\mathbb{R}$	N	Y	RCs,VoFP,MLeaks
Blom et al. (2014)	Con	OpenCL	VerCors	D	Frac	Perm(x,π)$\pi \in$ {rd, rw}	N	Y	RCs, VoGPGPU
Amighi et al. (2014)	Con	Java	VerCors	D	Frac	Perm(x, π)$\pi \in (0, 1]\cap\mathbb{R}$	N	Y	RCs, FCoJP
Huisman and Mostowski (2015)	Con	Java(PSM)	KeY and PVS	D	-	readPerm(x, $Perm$)writePerm(x, $Perm$)$^{\alpha}$	N	Y	RCs, AVROfrac
Amighi et al. (2015)	Con	Java	Vercors	St	Frac Count	$(0, 1]\cap\mathbb{R}$	N	Y	RCs.
Joosten et al. (2018)	Seq	OpenCL PVL Java OpenMP	VerCors	D	Frac	Perm(x, π), $\pi \in (0, 1]\cap\mathbb{R}$	N	Y	
Müller et al. (2016); Müller et al. (2017)	Seq	Java Chalice OpenCL Scala	Viper	St	Frac	acc(x) acc(x, rd)	N	Y	RCs
Ahmed and Cataño (2018)	Seq& Con	JML	Pulse	St	Sym	(U,I,S,F,P)	N	Y	VoJMLC, RCs

Keys to the table: Seq = sequential, Con = concurrent, St = static, D = dynamic, *Sym* = symbolic, Fract = fractional, U = unique, I = immutable, S = share, F = full, P = pure, $\mathbb{Z}$ = set of Integers, $\mathbb{R}$ set of Real numbers, $\mathbb{N}$ set of positive Integers, NSL = new specification language, PSM = permission sharing model, PSAM = permission sharing and accounting model, RCs = race conditions, DLcks = deadlocks, VoNullP = verification of null pointers, VoUP = verification of usage protcols, VoAPs = verification of access permission-based specifications, VoJMLC = verification of JML contracts, AROfrac = Avoiding reasoning overhead associated with fractional permissions, FCoJP = functional correctness of Java programs, VoGPGPU = verification of GPGPU programs, FGSM = fine-grained synchronization mechanism, AIOBEx, = ArrayIndexOutofBoundsExceptions, MLeaks = memory leaks, MCCP = machine-checked correctness proofs, VoADT = verification of abstract data types, VoFP = verification of functional properties, x heap location, rd read access, C = session type contract, α a permission slice. * permissions are read and write accesses, † implemented in the Fluid project, ‡ shows accessibility predicates.

Table 8
Summary of annotation overhead for sample programs in VeriFast (Jacobs et al., 2010).

Program Statistics and Annotation Overhead		
Program	SLOC	#AnnoLOC
chat server	242	114
linked list and iterator	332	194
composite	345	263
JavaCard applet	340	95
GameServer	383	148

Keys to the table: SLOC = source lines of code, AnnoLOC = annotated lines of code.

smaller than 1 represents a shared (read) access by multiple threads. The analysis works in a way that each method is symbolically executed based on other method's contracts to verify its calls. The logic-based specifications are tracked through the system to detect exceptions such as `NullPointer` and `ArrayIndexOutOfBoundsExceptions` exceptions in the program, and to verify the domain-specific problems in a program such as race conditions. However, the approach poses annotation overhead to programmers for explicitly adding the permission-based specifications in the program. Table 8 shows the annotation overhead, reported by authors, to verify program behavior in VeriFast.

Heap-Hop

Villard et al. (2010) developed a program prover Heap-Hop[8] based on Hoare's monitors and copyless message-passing mechanism (Villard et al., 2009), an alternative to lock-based parallelism where only pointers to a message content in memory are transferred. The objective was to verify deadlocks, data races and to ensure the absence of memory leaks in heap manipulating concurrent programs, particularly those that involve communication protocols with list and tree structures. The approach uses `channels` as synchronization mechanisms where each channel consists of two endpoints, say e and f, dynamically allocated on the heap.

Heap-Hop requires programmers to specify pre- and post-conditions, as ownership information for the heap locations, and loop invariants defined as Separation Logic (Reynolds, 2002) formulae. The communications between endpoints are governed using a contract C, a form of session types (Takeuchi et al., 1994), which specifies a valid sequence of message m passing on a channel. In Heap-Hop, ownership of cell to a heap location is represented using the notation $x\mapsto$ and the point-to relation $e\mapsto C\{a\}$ specifies a contract C in the state a with respect to a particular endpoint e.

The approach generates verification conditions based on the input specifications. It performs symbolic (forward) execution analysis of the generated conditions to determine what input (conditions) will cause each part of a program to execute and verifies the intended behavior of a program. The approach ensures that message sending never fails, and message reception should be blocked until the right message is received.

HIP/SLEEK

Hobor and Gherghina (2012) developed a Hoare-style concurrent Separation Logic that verifies Pthreads-style synchronization mechanism called barriers. Pthreads (POSIX Threads) is an API for threaded programming that manages various procedure calls for thread creation, destruction, and synchronization (Butenhof, 1997). A common use of barrier calls is to manage a pool of threads in a pipeline. In Pthreads, barriers are used to redistribute ownership (as read and write access) of resources (memory cells) simultaneously between multiple threads. At barrier calls, every thread gives up its write access to the portion of memory allocated to it and gets back the read-only access to the entire memory.

The approach extends the concept of permission shares in DSA (Dockins et al., 2009) and assigns positive share to each thread to access a particular location. A `full` share is required to modify a particular location. A `full` share can be split into multiple `partial` shares that are merged back to get back the `full` share. The idea is to ensure that if a thread has a `partial` share for a particular location, no other thread has `full` share (permission) for that location.

Unlike previous Concurrent Separation Logics (OHearn, 2007; Hobor et al., 2008) that focuses on programs with critical sections, locks, and channels respectively, the approach uses barriers to model resource redistribution, and verifies if barriers are accessed safely in a concurrent environment. The idea is to associate some positive (fractional) share of the barrier itself as a precondition and to ensure that the sum of all preconditions entails `full` share of the barrier.

For example, the assertion barrier (bn, π, cs) defines a precondition that specifies a barrier bn with a positive share π having a state cs that holds before entering a barrier. The state of barrier changes as threads are released from the barrier, and the next stage will follow based on the post-condition barrier (bn, π, ns) when the state transitions to a new state ns. A `full` permission ensures that no thread has a 'stale' view of the barrier state to ensure thread non-interference. The approach extends the HIP/SLEEK tool set (Gherghina et al., 2011; Nguyen and Chin, 2008) to verify concurrent programs with barrier calls. SLEEK is based on Separation Logic and HIP employs Hoare's rules to program verification.

Later, Jacobs and Piessens (2011) in his work on fine-grained concurrency verified some of the program examples using the VeriFast tool. The sample programs were taken from the HIP/SLEEK project and experiments were performed by implementing the barrier calls as locks. The experiments revealed that VeriFast poses more annotation overhead compared to HIP/SLEEK tool. For example, authors reported in the paper that programmers need to add approximately 30 lines of annotation for a program with 30 lines of source code whereas VeriFast requires more 600 annotated lines as a user provided input to verify the same program.

Pulse

Pulse (Siminiceanu et al., 2012) is an automatic formal verification approach and a tool that verifies the correctness of Plural (permission-based typestate contracts) specification itself, rather than the program implementation and its behavior. The goal was to write semantically correct specifications to verify program behavior based on these specifications. Like Plural, Pulse follows the Design by Contract Principle to specify permission-based contracts at the method level. It supports five kinds of symbolic permissions: `unique`, `full`, `share`, `pure` and `immutable` in method specifications.

In Pulse, programmers specify design intents as permission-based typestate invariants and lock-based specifications at the code level to avoid deadlocks and data races. State invariants are used to enforce the properties that should hold during program execution and to handle the design level inconsistencies in a program such as `Null` pointer references.

In an extended work, Cataño et al. (2014) evaluated the efficacy and the expressiveness of Plural specifications on a multi-threaded application called Multi-threaded Task Server (MTTS), to evaluate its design and verify its behavior using Pulse.

Listing 9 shows lock-based typestate contracts in MTTS using Plural specifications.

[8] http://www.lsv.fr/Software/heap-hop/.

```
contract C {//session type contract
initial state a { !m ↦ b; !m ↦ c; }
state b {}
final state c {}
}
foo() { (e,f) = open(C); send(m,e); receive(m,f); close(e,f); }
```

Listing 8. A sample method with permission-based contracts in Heap-Hop

```
1 @Perm(requires="Full(this) in NotAcq", ensures="Full(this) in Acq")
2 public abstract void acquire( ) { }
3 @Perm( requires="Full(this) in Acq", ensures="Full(this) in NotAcq"
4 public abstract void release( ) { }
5 }
```

Listing 9. Lock-based typestate contracts using access permissions in MTTS (Cataño et al., 2014).

Pulse defines lock-based specifications to ensure mutual exclusion to a critical section. The annotations 'Acq' and 'NotAcq' are used to represent the state of a lock i.e., to be acquired or not-acquired respectively. The locks are acquired using method `acquire()` in Line 2 and released using method `release()` in Line 3. The permission contract in Line 1, dictates that the `acquire()` method needs `Full` permissions, as pre-permission on the mutex (lock) object while acquiring a lock, and transitions it from 'NotAcq' into 'Acq' typestate. Similarly, the specification in Line 3 shows that the `release()` method, before releasing the lock, needs `Full` permissions on the lock object that is in 'Acq' state and transitions it from 'Acq' into 'NotAcq' typestate.

The code of the critical section is then enclosed between a call to method `acquire()` and a call to method `release()` to ensure mutual exclusion. The typestate transition in the given specification ensures that non-nested calls to method `acquire()` will always happen after a call to `release()` method. The permission contract ensures that if a thread has acquired a lock, it needs to be released before being used by other threads. However, as discussed previously, Plural does not support the reachability analysis of input specifications and cannot verify the absence of deadlocks caused by the input specifications. Pulse avoids deadlocks by using `try-catch-finally` statement in the code and enclosing call to method `release()` in a `finally` block to ensure that method `release()` is always called regardless of the termination status of the method.

Additionally Pulse detects violations of intended semantics using the model checking power of evmdd-smc (Roux and Siminiceanu, 2010) symbolic model checker. It helps programmers write semantically correct specifications and find possible concurrency at the method level, but to exploit the full potential of the Pulse tool, the programmers need to manually add permission-based typestate specifications in the source program, resulting in annotation overhead for the programmers. The authors reported that "it took six months on manually adding the specifications for 49 Java classes with 14,451 lines of source lines of Java code and 546 annotated lines of permission-based specifications in MTTS". Moreover, in Pulse, the use of model checker can create state-space explosion problems even for a program of average size.

Ahmed and Cataño (2018) proposed an automatic translation technique that encodes JML-encoded Finite State Machine (FSM) specification of a Java program into Plural specifications (permission-based typestate contracts). The encoded specification was fed into Pulse to find problems such as unreachable states, unreachable methods and sink states (deadlocks) in the input specifications, and to reason about the correctness of the underlying program before it is implemented.

HJp

Westbrook et al. (2012) proposed a permission-based type system that supports task parallelism, array parallelism, and object isolation. The system called 'Habanero Java with permissions (HJp)' is an extension of their previous work on the Habanero Java (HJ) language. "HJ itself is a task-parallel extension of Java language" (Cav et al., 2011). The main idea of HJp is that each object can be in any of the two permission modes, i.e., `shared read` or `private read-write`, at any moment. The `shared read` model specifies that any task (thread) can read from the object but none of them is allowed to write on it and `private read-write` mode shows that only one task is permitted to read from or write to the object. The system provides a practical solution to prevent data races for non-trivial parallel programs implementing multiple synchronization primitives, and parallel patters instead of just one.

The type system extend Boyland's fractional permissions with two new permission types: `aliased write` and `storable` permission. Unlike previous approaches (Bierhoff and Aldrich, 2007) where write (unique) permission is only supported for non-aliased objects at any one time, the `aliased write` permission supports write operations on aliased objects. In the system, multiple threads can write on multiple objects without actually having `unique` permissions on them, as long as the permissions are not passed to other threads. The `storable` permission provides a new and simple way for expressing transitive permission in complex objects such as a `linked list`. Storable permission associates permission to the 'whole tree of objects' instead of associating it to a single object. The permissions transitivity between objects is managed by defining `exclusive fields`, using the keyword `exclusive`, at the class level. This feature makes the technique different from existing approaches that require more technical machinery, and sound approximations, to manage permission transitivity in complex objects.

However, it requires programmers to adds access keywords at the method level such as reading, shared reading, writing and exclusive, to indicate permissions held by the method arguments on entry to and exit from the method. Moreover, the keywords `acquire` and `release` are explicitly added in the code to acquire and release (`aliased-write` and `storable`) permissions on the referenced objects.

Table 9 shows the annotation overhead, reported in the paper, to verify program behavior for the sample programs.

VerCors

Blom et al. (2014) proposed a simplified version of Kernel Programming Language (Betts et al., 2012) and a permission-based

Table 9
Summary of the annotation overhead for the sample programs in HJp (Westbrook et al., 2012).

Program Statistics and Annotation Overhead				
Program	SLOC(Methods)	#AnnLOC-MK	#AnnLOC-SP	#AnnLOC
NPB.CG	1070 (61)	25	7	32
JGF.Series	225 (15)	6	3	9
JGF.LUFact	467 (20)	16	11	27
GF.SOR	175 (12)	6	4	10
JGF.Moldyn	741 (57)	9	29	38
JGF.RayTracer	810 (67)	57	22	79
BOTS.NQueens	95 (3)	3	0	3
BOTS.Fibonacci	70 (3)	0)	0	0
BOTS.FFT	4480 (46)	33	0	0
PDFS	537 (26)	10	8	0
DPJ.BarnesHut	682 (56)	18	10	28
DPJ.MonteCarlo	2877 (287)	151	22	173
DPJ.IDEA	228 (18)	9	8	17
DPJ.CollisonTree	1032 (69)	108	24	132
DPJ.K-Means	501 (38)	25	6	31

Keys to the table: SLOC = source lines of code, AnnLOC-MK= annotated lines of code for adding method keywords, AnnLOC-SP = annotated lines of code for adding storable permissions, AnnLOC = total annotated lines of code.

Separation Logic to reason about the correctness of the GPU kernel written in OpenCl[9]. As the GPU kernel extensively uses threads to support parallelism, the objective was to verify the functional correctness of GPU programs and to ensure data race freedom in the underlying architecture.

The work follows an earlier work of Haack and Hurlin (2008) on verification of the muti-threaded programs in which a thread can only access or update a particular memory location if it has permission to read or write. The permissions in a program are provided, in the form of barrier specifications, using the read-write (`rw`) or read-only (`rd`) annotations. Multiple threads with read permissions can access the same location but only one thread can hold write permission at a time to change its content. The specifications combine first-order logic formulas with permissions-based accessibility predicates and the separating conjunction operator (*). The same idea was applied to delegate permissions across work groups and then to distribute permissions over threads.

The approach was validated using the VerCors project set.[10] VerCors (Amighi et al., 2012; Amighi et al., 2016; Blom et al., 2017) is a program verifier (tool chain) that verifies parallel and concurrent software. It supports programs written in OpenCL, PVL,[11] (a subset of) Java and OpenMP. The specification language and program logic in VerCors is based on earlier work on permission-based Concurrent Separation Logic (Haack et al., 2008) that supports Java. The verification technology in VerCors is built on top of the Viper (Müller et al., 2016) framework. Therefore, it uses Silicon and Silver, as intermediate verification languages, which natively support an expressive permission model. Moreover, it exploits verification power of Chalice (Leino et al., 2009) and Boogie (Barnett et al., 2006) as program verifiers.

VerCors takes an input program annotated with Separation Logic-style specifications. It encodes the input program into a series of intermediate representations and generates verification problems in the intermediate language, understandable by Viper, Chalice or Boogie, to be verified by the SMT solvers. However, the approach creates annotation overhead for programmers to add permission-based specifications in the program, as authors reported themselves "writing annotations can be very tedious. Not only is it necessary to write the contract for every method, it is also necessary to include many hints to the prover inside the code".

Further, VerCors in (Amighi et al., 2014) extends JML specifications with fractional permissions to reason about the functional correctness of Java programs. The approach supports multiple synchronization primitives in a Java program. The permission-based contracts and class invariants are defined in the input program, using the conjunction operator * in the Separation Logic, as JML comments. Access permissions are specified using a propositional formula of the form `Perm(e.f, π)` where π represents fractional permission in the range (0, 1] assigned to an individual field f of object e. The permissions are then transferred between threads at synchronization points and analyzed with the execution of program. Although, VerCors tool can generate many of the specifications itself, the annotation overhead in specifying permission-based contract in the input program is still a concern in this work, also evident from the authors perspective "our specification method in principle is very verbose, specifications at many different levels are required".

Listing 10 shows a sample Java class `point` in VerCors with permission-based access predicates. In line 2, a state predicate `state(frac p)` specifies that `p` permission is required on disjoint locations, `this.x` and `this.y` in memory. These predicates are then used to specify permission contract for the same locations at the method level. For example, the pre-condition `state(1)` of method `set()` in Line 5 specifies that the method requires `full` (write) permission on locations `x` and `y`, and the post-condition `ensures state(1)` ensures that the method returns the same permission on the corresponding locations when it exits. The invariant clause, in Line 4 specifies a functional property that both points should be in the first or third quarter of its cartesian space.

The contracts of methods `plot()` and `getQuarter()` in Line 8 and Line 10 respectively, specify that both methods require read permission `p` on locations `x` and `y`, which means that they can be executed simultaneously by multiple threads without the fear of data races. This is because the pre-conditions of both are non-interfering with respect to memory. However, this is not true for the method `set()` as it requires `full` permission on the same memory locations.

Instead of simply defining the amount in fractions of permission transferred, Huisman and Mostowski (2015) extended the previous fractional permission model in VerCors by having symbolic expressions which include the kind of *transfer* applied to permission, and the owner of the transferred permission. The approach facilitates high-precision, complex synchronization scenarios in concurrent data structures, and supports permission tracking at a high level of abstraction as compared to the previously mentioned approaches such as Veri-Fast (Jacobs et al., 2011) and Chalice (Leino et al., 2009).

In this approach, the program is annotated with symbolic (permission) expressions using JML annotations in the functional style. The analysis then tracks the owners using permission expressions and checks their permission return paths to reason about their behavior. The system identifies permission owners using object references and manages a list of owners. Whenever permissions are assigned to some owner, it is being added in the list and when an owner returns permissions, it is removed from the list. Each owner is considered as a permission slice. If all slices refer to the same owner, it means that the owner would have `full` permission. Otherwise, access is partial (read).

Listing 11 shows a sample read and write resource locking mechanism in Java in fractional permissions style. The Line 2 specifies that acquiring a lock (`cl`) transfers full (1) permission for location `o.x` to the locking thread and a read permission (1/2) for location `o.y` to access these locations. When the lock is released,

[9] Khronos OpenCL Working Group, The OpenCL specification, http://www.khronos.org/opencl/.
[10] https://fmttools.ewi.utwente.nl/redmine//projects/vercors-verifier/wiki/Puptol/.
[11] Prototypal Verification Language.

```
1  public class Point{
2  //@ resource state(frac p) = Perm(this.x, p) * Perm(this.y, p);
3    private int x, y;
4  //@  invariant (x >= 0 && y >= 0)   (x <= 0 && y <= 0);
5  //@  requires state(1); ensures state(1);
6   public void set(int xv, int yv){ this.x = xv; this.y = yv; }
7  //@ given frac p; requires state(p); ensures state(p);
8  public void plot(){}
9  //@ given frac p; requires state(p); ensures state(p);
10 public int getQuarter(){}
11 }
```

Listing 10. A Java class Point example in (Amighi et al., 2014).

```
1  class Client {
2  cl.lock(); // produces Perm(o.x, 1) and Perm(o.y, 1/2)
3  o.x = o.y; // write o.x, read o.y
4  cl.unlock(); // consumes Perm(o.x, 1) and Perm(o.y, 1/2)
5  . . . }
6  class Lock {
7  //@ requires !locked; ensures locked;
8  //@ ensures Perm(o.x,1) ** Perm(o.y,1/2);
9   void lock();
10 //@ requires Perm(o.x,1) ** Perm(o.y,12);
11 //@ requires locked; ensures !locked;
12  void unlock();
13 }
```

Listing 11. A simple lock and its fractional permission style specifications (Huisman and Mostowski, 2015).

in Line 4, the current thread transfers the same permission back to the lock object.

Listing 12 shows a code segment of a simple resource locking mechanism. Before acquiring the lock, permissions for location `o.x` and `o.y` are mapped to list `cl` (Line 2), as both of them belong to `cl`. In line 3, acquiring a lock assigns `full` permission on `o.x` to the locking thread represented as `ct` while it gets partial permission (one slice only) for location `o.y` and the list becomes [ct, cl] in Line 3. which means that lock still owns the remaining 1 slice on `y` that can be made available when required. The post-condition of method `lock()` specifies how permission to object `o.x` and `o.y` changes when function `lock` and `unlock` are called. It shows that the transfer function `transPerm()` in Line 8, forces its owner thread `ct` to give up all the rights completely on `o.x` while function `transPermSplit()`, in Line 9 transfers the old permission in slices to the specified thread `ct`.

When function `unlock()` is called, (Line 5) permissions are returned to lock by replacing current thread `ct` with `cl` permission, on `o.x` and `[cl, cl]` on `o.y` that can be merged again into object lock permission `[cl]`. The method `unlock` is not specified here due to brevity.

This information is then used to manage permissions at synchronization points while threads are being forked or joined and to reason about their behavior.

The permission theory was formalized in the KeY tool (Beckert et al., 2007), an interactive verifier for Java, that is based on dynamic logic. The system extends the KeY tool with permission accounting to verify program properties that are based on purely first-order reasoning. The general program properties that require structural induction proofs are validated using the PVS tool (Owre et al., 1992), because of its automated deduction and theorem proving capability.

Amighi et al. (2015) proposed a variant of OHearn's Concurrent Separation Logic (OHearn, 2007) to perform practical reasoning of Java-like concurrent programs having main concurrency primitives such as dynamic thread creation, thread joining, wait-notify scenarios and lock reentrant mechanism.

The system combines Parkinson's share model with Boyland's fractional permissions to support inheritance of resource invariants and class parameters and to avoid data races in realistic applications.

In Parkinson share model, resource invariants are defined using abstract predicates at the class `Object` level, with an empty footprint (permissions associated with a memory location) that each subclass extends to hold additional fields. Like Parkinson's share model, access for a particular heap location is maintained using a resource's invariant property, where 1 represents full (exclusive) permission to a heap location, and a fractional value in the interval (0, 1) defines the concurrent read access of a particular location. The idea is that a thread having partial permission is not allowed to write on a heap location and the total permission to access a heap location cannot exceed 1.

Like the OHearn's approach, when a thread acquires a lock, it gets access to part of a heap location specified as a resource's invariant property. Upon unlocking, it transfers access of the same resource back to lock, to re-establish the resource's invariant property. The permissions are transferred between threads at the time of thread creation, thread joining and at lock entrances and reentrance points. However, the verification is performed at the cost of manually writing specification, as a part of the input program, that creates annotation overhead for programmers.

Recently, Joosten et al. (2018) performed experiments to verify the correctness of sequential programs with VerCors (Amighi et al., 2016). The data set used for the experiment consists of gap buffer data, a data-structure commonly used in text editors and a combinatorial problem based on Project Euler problem #114 structures.

To perform verification, the approach requires programmers to explicitly specify the intended functional behaviour of the underlying data structure, as a sequence of characters, along with the permission-based ownership (fractional) annotations defined at the class field level. The permission annotations are specified using predicates of the form `Perm(`x`, `π`)`, with x representing the shared field and π is a fractional permission in the range (0...1]. The value 1 represents the `full` access on x and any value $pi < 1$ represents read access to x. The approach ensures that the sum of permissions for the shared location x is less than 1 to ensure the safe memory accesses and to verify the absence of race conditions.

The experiments revealed that VerCors verification infrastructure is capable of reasoning about sequential programs. However, the verification in VerCors comes at the cost of increasing programmers burden for manually annotating programs with ownership information.

Viper

Müller et al. (2016); Müller et al. (2017) developed a verification infrastructure called Viper. It targets a sequential, object-based intermediate language `Silver` that encodes a flexible permission model and supports user-defined predicates and functions. The infrastructure includes two back-end verifiers and four front-end tools for Chalice, Java, Scala, and OpenCL that was developed

```
class Client {
// Perm(o.x), Perm(o.y) are [cl]
cl.lock(); // Perm(o.x) becomes [ct], Perm(o.y) becomes [ct, cl]
o.x = o.y; // [ct] → write access, [ct, cl] → read access
cl.unlock(); // Perm(o.x) becomes [cl], Perm(o.y) becomes [cl, cl]
. . . }
class Lock {
//@ ensures Perm(o.x) == transPerm(this, ct, \old(Perm(o.x)));
//@ ensures Perm(o.y) == transPermSplit(this, ct, \old(Perm(o.y)));
void lock(); . . .}
```

Listing 12. A simple lock specification in (Huisman and Mostowski, 2015).

```
 1 field data: Seq[Int]
 2 define sorted(s) forall i: Int, j: Int :: 0 <= i && i < j && j < s
 3                                         ==> s[i] <= s[j]
 4  method insert(this: Ref, elem: Int) returns (idx: Int)
 5  requires acc(this.data) && sorted(this.data)
 6  ensures acc(this.data) && sorted(this.data)
 7  ensures 0 <= idx && idx <= old(this.data)
 8  ensures this.data == old(this.data)[0..idx] ++
 9              Seq(elem) ++ old(this.data)[idx..]{
10   idx := 0
11   while(idx < this.data && this.data[idx] < elem)
12   invariant acc(this.data, 1/2)
13   ...
14   { idx := idx + 1 }
15   ...
16  }
```

Listing 13. A sorted integer list and its specifications in Viper (Müller et al., 2017).

as a part of VerCors project (Blom and Huisman, 2014). A Viper program does not have classes and an object can access every field declared in a program. Moreover, there is no implicit receiver object for methods and functions.

In a Viper program, a programmer defines accessibility predicates (Parkinson and Bierman, 2005), as permission-based pre- and post-conditions and loop invariants for heap structures to verify its behavior. A method can access a particular heap location if the appropriate permissions are held by that location. The permissions are then transferred between method execution and the loop body to verify program behavior based on the input specifications rather than using its implementation.

Listing 13 shows a sample sorted integer list `data` with access predicates in a Viper program. Line 1 declares an integer list as a data field of sequence data type. The macro `sorted(s)` in Line 2 sorts input list `s` in ascending order. The `insert` method adds a new element `elem` in the `Ref` list and returns the index `idx` where the new element was inserted.

The pre-condition of method `insert()` in Line 5 specifies that the method requires `full` permissions on the object list `this.data` and it should be sorted. The post-condition in Line 6 guarantees that when the method exits it returns the sorted list to the caller with the consumed permission. The second post-condition in Line 7 constrains and thus validates the index, while the third post-condition in Line 8 relates the current state of the list with the method's pre-state, using an `old` expression.

The `insert()` method iterates over `data` list to determine where to insert the new element `elem` in Line 11. The loop invariant (Line 12) specifies that loop body needs a half (read) permission on the list, while the second half permission would be held by method execution to ensure that the loop body does not modify the list. The Viper's front-end tools then encode the annotated program into an intermediate language acceptable by the back-ends tools, to verify its behavior.

6. Automatic inference of access permissions

Permission-based access notations have been generated as means for program verification in many approaches (Bierhoff et al., 2009a; Leino et al., 2009; Ferrara and Müller, 2012; Le et al., 2012; Heule et al., 2011; 2013; Sadiq et al., 2016; Dohrau et al., 2018; Sadiq et al., 2019a; 2019b). The generated specifications are either in the form of read/write accesses, fractional or symbolic permissions. The overall goal of these approaches was to relieve programmers from specification overhead resulting from manually adding permission-based annotations in a source program for verification purpose. Table 10 shows a summary of the work done to infer permission-based specification in sequential and concurrent programs.

6.1. Inference of read & write accesses

As discussed previously, in Section 5.2, Chalice (Leino et al., 2009) is a verification framework that verifies the correctness of multi-threaded programs written in the Chalice language. Chalice uses autoMagic, a command-line option, to infer the read and write accesses for the heap locations specified with accessibility predicates. The inferred notations are in the form of `pure` and `access` notations that represent `read-only` and `full` permission respectively for the specified heap locations.

Le et al. (2012) proposed a new permission system to avoid data races in multi-threaded applications having fork/join parallelism. The objective was to ensure the absence of data races for program variables that are not heap variables but can be accessed by multiple threads.

The scheme infers variable permissions at the method level using `procedure specifications`. However, in the procedure specification, a programmer explicitly specifies state changes (if any) for the referenced variable accessed by the current thread,

Table 10
Access permission inference for sequential and concurrent programs.

Reference	Prog	Lang	Tool	Analy	Perm-Kind	Perm-Specs	Perm-Infer	Anno	Properties
Bierhoff et al. (2009a)	Seq	Plural (NSL)	Plural	(St,D)	Sym	(U,I,S,F,P)	Frac	Y	VoUP
Leino et al. (2009)	Con	NSL	Chalice	D	Frac	acc(x) rd(x)‡	access pure	Y	RCs DLcks
Heule et al. (2011) Heule et al. (2013)	Con	-	Chalice	St	Frac	acc(x, 1) acc(x, rd)	full read	Y	RCs
Le et al. (2012)	Con	NSL	VPerm	D	Frac	@full[ν] @value[ν]$^{\phi}$	full zero	Y	RCs
Ferrara and Müller (2012)	Con	Scala	Sample	St	-	acc(x, p) p $\in$ (0, 1] $\cap\mathbb{R}$ p $\in$(0, 1] $\cap\mathbb{Z}$, p $\in$ (0, 100]%	Frac Count Chalice	Y	RCs
Dohrau et al. (2018)	Con	Viper	Scala	St	Frac	[0,1]$\cap\mathbb{R}$	read & write	Y	RCs
Sadiq et al. (2016, 2019a, 2019b)	Seq	Java	Sip4J[a]	St	Sym	-	(U,I,S,F,P)	N	EIC,CRA,INullP

Keys to the table: Seq = sequential, Con = concurrent, St = static, D = dynamic, *Sym* = symbolic, Fract = fractional, U = unique, I = immutable, S = share, *F* = full, *P* = pure, $\mathbb{Z}$ = set of Integers, $\mathbb{R}$ set of Real numbers, NSL new specification language, RCs = race conditions, EIC = Enabling implicit concurrency, DLcks = deadlocks, INullP = identifying null pointers, CRA = code reachability analysis, VoUP = verification of usage protocols, *x* heap location, ν represents a non-heap location. *rd* for the read access. ‡ accessibility predicates.

[a] https://github.com/Sip4J/Sip4J.

```
1 int creator(ref int x, ref int y)
2 requires @full [x, y]
3 ensures  @full [y] ∧ y′ = y + 2 ∧ res = tid and @full[x] ∧ x′ = x + 1 ∧
      thread = tid;{
4  int tid = fork(inc, x, 1);
5  inc(y, 2);
6  return tid;
7 }
```

Listing 14. A fork/join program fragment with procedure specifications (Le et al., 2012).

using permission-based state invariants without actual variable permission. The generated permissions are in the form of notations such as `full` or `zero` where `full` represents exclusive rights on the referenced variable and the notation `zero` represents the absence of permission. The proposed technique then tracks permission flow between threads to ensure safe access to the shared variables.

Listing 14 shows an example procedure specification example in a sample fork-join program. The method `creator()` takes two variables `x` and `y` as reference parameters. The `requires` clause in Line 2 specifies that the method needs `full` permission on the referenced variables `x` and `y` as pre-permissions when the method is called. The `ensures` clause specifies that the method should generate the same permissions as post-permissions on the same referenced variables when it exits (Line 3). The state changes are represented using prime ′ notation. For example, the specification "y′ = y + 2" in Line 3 specifies state changes for the referenced variable `y` that should hold after the method completes its execution. These specifications are then tracked in the system to generate actual variable permissions for the referenced variables.

The proposed scheme was realized in a concurrent program verifier called Vperm[12] that verifies the correctness of concurrent applications written in C/C++ language. The approach does not handle phased access to a shared variable by multiple threads, in which case a translation algorithm is used to simulate the affected variables as pseudo-heap locations.

6.2. Inference of fractional permissions

Bierhoff et al. (2009a) proposed a deterministic algorithm to infer permission flow through the program while verifying usage protocols. The objective was to avoid the permission tracking overhead associated with splitting and joining the fractional permission during verification.

[12] http://loris-7.ddns.comp.nus.edu.sg/project/vperm/.

The algorithm is implemented in the Plural tool (Bierhoff and Aldrich, 2008) that performs dataflow analysis of the program with `in` and `out` permissions as developer-provided annotations. The system collects linear constraints over fractional variables by tracking the flow of permissions in the program. The analysis then ensures the satisfiability of constraints in a modular fashion. The approach supports polymorphism over fractions that not only facilitates modular reasoning of the program but also avoids imprecision in loops by allowing permission consumption inside loops. Furthermore, the technique automatically infers loop invariants in a program.

Ferrara and Müller (2012) proposed a permission inference technique to infer fractional and counting permissions for heap locations in a class-based language having threads and monitors. The technique performs static analysis of the source program and inference is based on abstract interpretations (Cousot and Cousot, 1977), a theory for defining and soundly approximating the semantics of a program. The approach firstly computes symbolic values (approximations) for each heap location using the pre- and post-conditions and lock invariants defined at the method and class level respectively. It then infers constraints over these symbolic values to reflect permission-based intermediate representation for the heap locations. Finally, it generates specifications in the form of fractional (value between 0 and 1) and counting (value between 0 and Integer:MAX_VALUE) permissions for each heap location in the program.

The symbolic permissions ($\overline{AV}$) for each heap location are calculated as "the summation of symbolic values s_i multiplied by integer coefficients a_i (to represent how many times the permission is consumed or returned) and an integer constant c" (see Formula 4). The integer constant `c` represents `full` permission that is inhaled when an object is created.

$$\overline{AV} = \sum_i a_i * s_i + c, \text{where } a_i, c \in \mathbb{R}, s_i \in SV \tag{4}$$

```
1 class W1 {
2 var c : Cell;
3 method Inc()
4  ensures c.c1 == old(c.c1) + 1{
5  acquire c;
6  c.c1 := c.c1 + 1;
7  release c;
8  }
9 }
```

Listing 15. The OwickiGries program (fragment) with method-level specifications in (Ferrara and Müller, 2012)

For example, the expression 1 * `Pre(C,m,c:f)` + 1 * `MI(C,c:f)` + 0 represents symbolic permissions computed for each heap location (`c:f`) in method `m()` of class `C` where `Pre(C,m,c:f)` represents the symbolic value (s_i) assigned to location (`c:f`), as pre-condition before acquiring a lock, and the notation `MI(C;c:f)` represents monitor (MI) acquired on location `c:f`. Further, the notation `Post(C,m,c:f)` represents a symbolic value assigned to a heap location, as post-condition, to get the original permission back on it when the monitor is released. The technique then infers constraints over these symbolic values to generate actual permissions as fractional permissions.

The inference technique is implemented in `Sample` (Static Analyzer of Multiple Programming LanguagEs)[13] that supports programs written in Scala (Odersky et al., 2004). Listing 15 shows the OwickiGries (Owicki and Gries, 1976) program fragment as an input program. In the example, all expressions are self-explanatory except the old expression `old (c.c1)`, in Line 4, that allows post-conditions to refer back to the pre-state of a referenced variable and its associated predicates.

In method `Inc()`, between `acquire c` and `release c` clauses (Line 5 and 7), the current thread is assigned with a symbolic permission 1 * Pre(W1, Inc, c:c1) + 1 * MI(Cell, c:c1) for location *c*: *c*1. When method exits (Line 9), is gets the symbolic permission 1 * Pre(W1, Inc, c:c1) = 1 * Post(W1, Inc, c:c1) as post-condition since the monitor of `c` is released. Solving constraints over these symbolic values, the system generates `full` (1) as fractional permission for location `c:c1` when the method completes its execution and control is passed to Line 8.

The system works very well for Chalice lattice domain. However, the analysis based on fractional permissions is challenging and sometimes, the system converges the generated permissions back to zero to explicitly terminate analysis. Moreover, it provides limited support to infer permission contracts for programs having recursive data structures. The rate of inferring permission contracts for such programs is at minimum 36% and 68% at maximum. Moreover, like Chalice, manually annotating code with pre and post-conditions and monitor invariants create significant annotation overhead for programmers.

In an extended work of Ferrara and Müller's permission inference, Dohrau et al. (2018) proposed a static analysis to infer permission-based contracts for array manipulating concurrent programs. The technique is based on Separation and related logics (Reynolds, 2002; Smans et al., 2009). The idea is to explicitly associate a separate (fractional) permission for each array element to specify its accessibility by parts of the program. The value 1 represents `full` access while `rd`, a positive fraction of permission, represents the concurrent (read) access to a memory location. The analysis then infers read and write accesses for the specified memory locations to generate permission contracts at the method-level and within the loop.

[13] http://www.pm.inf.ethz.ch/research/sample.html.

For example, the approach associates each array element say $q_a[q_i]$ with a fraction of permission using conditional expression of the form $q_a = array \wedge q_i = index$? 1 : 0 that specifies `full` permission (1) for element *array*[*index*] and no permission for all other elements. The permission required for each loop iteration is computed using a `maximum expression` that calculates the maximum of permission required by each referenced variable changed in a particular loop iteration. The whole (complete) loop execution depends on the maximum of all the fractions over all loop iterations. It is used to infer read and write specifications for all indices of an array, for the whole loop.

However, it is generally acknowledged that tracking concrete fractional values is a cumbersome task for programmers especially when fractions continue to decrease indefinitely for a particular scenario (Heule et al., 2013). Moreover, the use of fractional permissions makes the specifications too low-level which can be tedious to add manually and harder to reuse and adapt for programmers.

6.3. Inference of symbolic permissions

Heule et al. (2011, 2013) proposed a technique to automatically convert fractional permissions into abstract read/write permissions for shared-memory concurrent programs. The objective was to specify concurrent constructs such as fork/join threads, locks/monitors with abstract permissions to avoid the complex reasoning overhead associated with fractional shares.

The abstract read permissions allow programmers to reason at a high-level of abstraction than using the fractional values for reading. The objective was to avoid the complex reasoning overhead associated with handling concrete values in fractional permissions during verification. The proposed methodology is implemented in Chalice. The system generates two kinds of permissions i.e., `full` and `read`. Like Chalice, it takes a program annotated with accessibility predicates such as `acc(x.f,1)` and `acc(x.f, rd)` at the method level. The value 1 is mapped to represent the `full` (read and write) permission and `rd` represents the shared `read` permission (a part of permission that is not full) for the referenced object `x.f`. Moreover, the system automatically computes read (`rd`) permission instead of programmers having to compute this value explicitly.

Sadiq et al. (2016, 2019a, 2019b) developed a permission inference approach and a tool, GAP, to automatically infer access permission contracts for sequential Java programs. The aim was to free programmers from specification overhead to manually annotate program with permission-based contracts to help enable implicit concurrency present in a sequential program.

The permission inference approach performs modular static analysis (data-flow and alias-flow analysis) of the source code. It automatically extracts the implicit dependencies present between the code (methods) and shared states in the source program and maps them in the form of five types of symbolic permissions such as `unique`, `full`, `immutable`, `pure`, and `share`, without using any method level specifications. Further, it automatically generates Plural specifications i.e., access permission contracts, using a single typestate ‘alive’, for the objects accessed at the method level, to verify the correctness of the inferred specifications by the existing model-checking tool i.e., Pulse and to reason about their concurrent behavior.

Listing 16 shows sample methods `getColl()` and `initColl()` with the inferred access permission contracts in Plural format. The proposed technique generates `pure` permission as pre-permission (Line 3) on the receiver object (`this`) as method `getColl()` read the referenced field (`coll`). Further, it generates `full` permission on `this` in method `initColl()` (Line 5) as method writes on `coll`. Following the Design by

```
class ArrayColl{
 Integer[] coll;
 @Perm(requires="pure(this) in alive", ensures="pure(this) in alive")
 public static void getColl(){ return this.coll;}
 @Perm(requires="full(this) in alive", ensures="full(this) in alive")
 public static void initColl(){
  for (int i = 0; i < 10; i++){this.coll[i] = i*2;}
}
```

Listing 16. The inferred permission contracts in Sip4J (Sadiq et al., 2019a)

Table 11
Annotation Overhead computed for the benchmark programs in (Sadiq et al., 2019a).

Program Statistics and Annotation Overhead					
Benchmark	Program	SLOC	Methods	#AnnLOC	#Annot.
Plural	Crystal	17,512	2,188	2,500	6,691
-	Pulse	7,671	461	513	4,850
JGB	montecarlo	1,370	196	204	1,975
	euler	1,080	51	52	1,073
	search	666	50	60	691
	moldyn	608	43	52	901
	lufact	549	42	50	437
	crypt	488	40	46	385
	series	359	37	43	207
	sor	354	34	42	267
	sparsemat	327	33	36	316
Æminium	blackscholes	437	21	56	694
	gaknapsack	232	50	38	250
	health	232	18	25	334
Plaid	webserver	143	12	11	11
	fft	91	11	16	44
	quicksort	66	9	13	17
	shellsort	58	7	10	44
	integral	40	5	7	17
	fibonacci	22	4	8	9
-	Example	71	12	15	78
-	**Total**	**32,376**	**3,111**	**3,485**	**18,647**

Keys to the table: SLOC = source lines of code, AnnLOC = annotated lines of code), Annot. = individual annotations at the field level.

Contract Principle, the consumed permission are generated as post-permissions for the referenced objects when a method completes its execution.

The GAP framework computes the minimal annotation overhead to quantify the manual effort that existing permission-based verification and parallelization approaches can pose to programmers to perform their intended task. Table 11 shows the program statistics and the annotation overhead for the benchmark and realistic Java applications used in the paper.

7. An insight into research challenges and future directions

Access permissions (Boyland, 2003; Bornat et al., 2005; Bierhoff and Aldrich, 2007) are a novel abstraction that can model read and write effects of a referenced object as well as its aliasing information. Access permission sharing and accounting models attained considerable attention in the research community in the last decade because of their rich expressiveness and sound reasoning capabilities to specify and verify the correctness of shared-memory programs.

Permission-based specifications have been used to verify intended design, functional and behavior properties of sequential and concurrent programs and to infer the permission-based specifications itself in many formal approaches. This survey organizes the access permission-based verification of single- and multi-threaded programs into three dimensions according to the use of access permissions and their aims: verifying the correctness of API protocols (Section 4), avoiding synchronization problems in multi-threaded programs (Section 5) and inferring automatically permission-based specifications (Section 6). Tables 3, 7 and 10 summarizes and contrasts the surveyed works based on the analysis criteria defined in Section 1.

The analysis shows that some of the verification tools such as Viper, VerCors, VeriFast to name a few, are leading in the research community as research prototypes and provide sound reasoning mechanisms to verify program behavior written in high-level programming language such as Java. Moreover, the underlying approaches support rich synchronization constructs such as fork-join parallelism and atomic blocks, while others are still limited in their support for verification due to the following reasons.

Permission annotation overhead. The common problem with all the permission-based verification approaches is the annotation overhead associated with the need to manually add permission-based dependences (invariants, contracts, assertions, etc.) or other access notations, to explicitly specify state changes and grant or restrict access to multiple references (threads), on the shared memory locations. It is generally acknowledged that manually annotating programs is laborious, challenging and time-consuming.

Permission verification overhead. Given the intricacies in creating permission-based specifications, it is very likely for a programmer to omit important dependencies or extract the wrong dependencies (read instead of write). Moreover, there is no guarantee that the manually added specifications are spelled correctly (free of typo errors) and follows their intended semantics (defined in Section 3) that in turn pose verification overhead for verifying the correctness of the input specifications itself.

Although some of the existing approaches (Siminiceanu et al., 2012) present solution to this problem, by identifying the missing specifications and by verifying that the specifications follow their intended semantics, the techniques themselves are limited in ensuring whether the program implementation complies with the input specifications and vice versa. This is because the analysis is based on input specifications rather than program implementation. Furthermore, although, access permissions support modular reasoning of a program behavior without analyzing the entire program analysis, certain program properties such as global invariants and liveness, are hard to verify.

Permission tracking overhead. Existing verification and parallelization approaches use different forms of permission types such as fractional permissions, based on their expressiveness and to facilitate the ease of analysis. The runtime systems then analyze the permission flow through the system to verify program behavior against the specification and compute the data dependencies in the system based on the specification.

In particular, fractional permissions use fractional style to express the access rights for a reference in the range (0,1] but its analysis is challenging for programmers, due to the overhead associated with tracking the splitting and the joining of concrete values. It is generally acknowledged that tracking fractional values in a program is a cumbersome task for programmers. The situation becomes more serious when fractions are split into multiple levels, which may create concerns relating to the proper termination of the analysis and affects the soundness of the technique itself.

Counting permissions are complementary to fractional permissions but they do not support all types of synchronization primitives. Symbolic permissions combine the access rights and aliasing information of a reference and have been used to allow programmers to reason about the program correctness, against specification at a higher level of abstraction than fractional or counting permissions. Therefore, automatic inference of permission-based specifications in the form of symbolic values is desirable to free the programmers from the low-level analysis overhead associated with adding and tracking concrete values in the program.

In addition to the specification overheads and related problems discussed above, the following factors may also hinder the wider adoption of existing permission-based verification approaches.

Programming paradigms Existing permission-based verification approaches are mostly based on new programming paradigms and runtime systems to support access permission as part of the language. It can be challenging for most programmers who may not be adept at the new syntax and semantics in order to exploit their functionalities. These factors could limit the adoption of the existing approaches to verify programs written in mainstream programming languages such as Java.

Program constructs. Existing verification approaches have limited support for synchronization constructs such as fork-join parallelism, atomic blocks or semaphores. There is also limited support for the recursive data structures. These limitations restrict their ability to verify realistic applications.

Program analysis. Existing approaches either perform static and dynamic program analysis or employ model-checking techniques to verify program properties. All have their pros and cons that affect the scalability of these approaches when analyzing program constructs and verifying the program behavior. For example, the techniques employing model checkers may face state-space explosion problems even for a program of average size.

Despite this, existing permission-based verification approaches are quite promising in verifying program behavior. The common problem in all the existing approaches is the annotation overhead associated with the need to manually add the permission-based specifications in the source program. Therefore, the automatic inference of permission-based specifications from the source program can be the first step to exploit the verification power of these approaches, without posing any extra burden on programmers, to enhance their applicability in the IT industry.

Overall, given the number of different permission-based formal type theories and programming models that received remarkable attention over the last decade in the research community, there is an impending need to make these approaches adoptable and adaptable for general-purpose program development, and verification for mainstream programming languages such as Java.

The first step toward this goal can be an integration of the most widely used permission-based verification tools that at least support a common programming models such as Java. For example, Plural, Pulse, Verifast, VerCors and Sip4J fall in this category.

The analysis of existing approaches reveals that the inference of access permission contracts can be used to automatically compute the dependencies between methods while making the side effects explicit. As access permissions pose their own ordering constraints, the computed dependencies can be used to automatically infer the synchronization primitives (locking and ordering constraints) from the source code of a sequential program. The generated specifications can then be used to enforce the locking policy to different program parts at different levels of granularity (method, instruction or task).

This information can be used to automatically parallelize program execution for mainstream programming languages such as Java, to the extent permitted by the computed dependencies, without using any new programming language and runtime system to support access permissions. The permission-based parallelization can free the programmers from the low-level synchronization and reasoning overhead associated with handling multi-threading in sequential programming paradigms. Further, the inference of permission-based synchronization constructs (such as acquire and release locks with permission invariants) can be used to verify the behavior of concurrent programs, that have already been written using multi-threading, without imposing extra work on the programmer side.

The ideal solution to all the above challenges can be the integration of the commonly used abstractions such as typestates and access permissions, as first-class language constructs in the mainstream programming models, to develop a complete, sound, modular, automated and economically feasible framework for everyday program development and verification. The new technique could be realized to provide modelling, verification, and parallelization support without posing any extra burden on programmers.

Acknowledgments

This research was conducted under Endeavour Leadership Award, for P.hD., funded by the Department of Education and Training, Government of Australia and Faculty of Information Technology, Monash University, under Faculty Postgraduate Publication Award.

Appendix A. Vitae

- **Miss. Ayesha Sadiq** completed her PhD in the Faculty of Information Technology at Monash University in 2019. Her research interest primarily lies in software engineering with a focus on programming languages, program analysis, software security analysis and formal modelling and verification of real time applications.
- **Dr. Yuan-Fang Li** is a Senior Lecturer at Faculty of Information Technology, Monash University, Australia. He received his PhD in computer science from National University of Singapore in 2006. His research interests include knowledge graphs, knowledge representation and reasoning, ontology languages, and software engineering.
- **Dr. Sea Ling** works in the Faculty of Information Technology at Monash University as a senior lecturer. His fundamental research interest lies in software engineering techniques with specific focus on formal methods and programming languages. Currently, his research is on extending and applying these techniques to the areas of ubiquitous and pervasive computing.

References

Abadi, M., Flanagan, C., Freund, S.N., 2006. Types for safe locking: static race detection for Java. ACM Trans. Program. Lang. Syst. 28 (2), 207–255.

Ahmed, I., Cataño, N., 2018. Checking JML-encoded finite state machine properties. In: 2018 International Conference on Advancements in Computational Sciences (ICACS), pp. 1–9.

Aldrich, J., Beckman, N.E., Bocchino, R., Naden, K., Saini, D., Stork, S., Sunshine, J., 2012. The Plaid language: Typed core specification. Technical Report. DTIC Document.

Aldrich, J., Bocchino, R., Garcia, R., Hahnenberg, M., Mohr, M., Naden, K., Saini, D., Stork, S., Sunshine, J., Tanter, others, 2011. Plaid: a permission-based programming language. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, pp. 183–184.

Aldrich, J., Sunshine, J., Saini, D., Sparks, Z., 2009. Typestate-oriented Programming. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. ACM, pp. 1015–1022.

Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M., 2014. Verification of Concurrent Systems with VerCors. In: Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures. Springer International Publishing, pp. 172–216.

Amighi, A., Blom, S., Huisman, M., 2016. Vercors: A layered approach to practical verification of concurrent software. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 495–503. doi:10.1109/PDP.2016.107.

Amighi, A., Blom, S., Huisman, M., Zaharieva-Stojanovski, M., 2012. The {VerCors} Project: Setting Up Basecamp. In: Programming Languages meets Program Verification (PLPV 2012).

Amighi, A., Haack, C., Huisman, M., Hurlin, C., 2015. Permission-based separation logic for multithreaded java programs. Logical Methods in Computer Science.

Ancona, D., Bono, V., Bravetti, M., 2016. Behavioral types in programming languages. Now Publishers Inc.

Appel, A.W., Blazy, S., 2007. Separation Logic for small-step cminor. In: Theorem Proving in Higher Order Logics. Springer Berlin Heidelberg, pp. 5–21.

Araujo, W., Briand, L., Labiche, Y., 2008. Concurrent contracts for Java in JML. In: Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on. IEEE, pp. 37–46.

Artho, C., Havelund, K., Biere, A., 2003. High-level data races. In: Software Testing Verification and Reliability.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., Leino, K.R.M., 2006. Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects. Springer Berlin Heidelberg, pp. 364–387.

Barrett, C., Stump, A., Tinelli, C., 2010. The SMTLIB Standard Version 2.0, 1–85. online at http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf.

Beckert, B., Hhnle, R., Schmitt, P.H.P.H., 2007. Verification of object-oriented software: the KeY approach. Springer.

Beckman, N.E., 2009. Modular typestate checking in concurrent Java programs. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. ACM, pp. 737–738.

Beckman, N.E., 2010. Types for Correct Concurrent API Usage, PhD thesis, technical report CMU-ISR-10-131.

Beckman, N.E., Bierhoff, K., Aldrich, J., 2008. Verifying Correct Usage of Atomic Blocks and Typestate. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications. ACM, pp. 227–244.

Beckman, N.E., Kim, D., Aldrich, J., 2011. An Empirical Study of Object Protocols in the Wild. In: Proceedings of the 25th European Conference on Object-oriented Programming. Springer-Verlag, pp. 2–26.

Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P., Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P., 2012. GPUVerify. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12. ACM Press, p. 113.

Bierhoff, K., 2009. Api Protocol Compliance in Object-oriented Software. Ph.D. Dissertation. Carnegie Mellon Univ., Pittsburgh, PA, USA. ISBN: 978-1-109-31660-5. AAI3370353.

Bierhoff, K., 2011. Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning. In: Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software. ACM, pp. 19–32.

Bierhoff, K., Aldrich, J., 2007. Modular typestate checking of aliased objects. OOPSLA '07, 42. ACM.

Bierhoff, K., Aldrich, J., 2008. PLURAL: Checking Protocol Compliance Under Aliasing. In: Companion of the 30th International Conference on Software Engineering. ACM, pp. 971–972.

Bierhoff, K., Beckman, N.E., Aldrich, J., 2009a. Polymorphic fractional permission inference. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Software Testing and Analysis.

Bierhoff, K., Beckman, N.E., Aldrich, J., 2009b. Practical API Protocol Checking with Access Permissions. In: ECOOP, pp. 195–219.

Blom, S., Darabi, S., Huisman, M., Oortwijn, W., 2017. The vercors tool set: Verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (Eds.), Integrated Formal Methods. Springer International Publishing, Cham, pp. 102–110.

Blom, S., Huisman, M., 2014. The vercors tool for verification of concurrent programs. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Blom, S., Huisman, M., Miheli, M., 2014. Specification and verification of GPGPU programs. Sci. Comput. Program 95, 376–388.

Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M., 2005. Permission accounting in Separation Logic. SIGPLAN Not. 40 (1), 259–270.

Boyapati, C., Lee, R., Rinard, M., 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. ACM, pp. 211–230.

Boyapati, C., Rinard, M., 2001. A Parameterized Type System for Race-free Java Programs. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. ACM, pp. 56–69.

Boyland, J., 2003. Checking Interference with Fractional Permissions. In: Proceedings of the 10th International Conference on Static Analysis. Springer-Verlag, pp. 55–72.

Boyland, J., 2006. Why we should not add readonly to Java (yet).. The Journal of Object Technology 5 (5), 5.

Boyland, J.T., 2010. Semantics of fractional permissions with nesting. ACM Trans. Program. Lang. Syst. 32 (6), 22:1–22:33.

Brookes, S., 2004. A Semantics for Concurrent Separation Logic. Springer, Berlin, Heidelberg, pp. 16–34.

Butenhof, D.R., 1997. Programming with POSIX threads. Addison-Wesley Professional.

Caires, L., 2008. Spatial-behavioral types for concurrency and resource control in distributed systems. Theor. Comput. Sci. 402 (2–3), 120–141.

Caires, L., Cardelli, L., 2002. A Spatial Logic for Concurrency (Part II). In: Proceedings of the 13th International Conference on Concurrency Theory. Springer-Verlag, pp. 209–225.

Caires, L., Cardelli, L., 2003. A Spatial Logic for concurrency (Part i). Inf. Comput. 186 (2), 194–235.

Caires, L., Pfenning, F., 2010. Session types as intuitionistic linear propositions. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Caires, L., Seco, J.C., 2013. The Type Discipline of Behavioral Separation. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 275–286.

Capriccioli, A., Servetto, M., Zucca, E., 2016. An imperative pure calculus. Electron. Notes Theor. Comput. Sci. 322 (C), 87–102.

Carr, S.A., Logozzo, F., Payer, M., 2017. Automatic contract insertion with ccbot. IEEE Trans. Software Eng. 43 (8), 701–714.

Cataño, N., Ahmed, I., Siminiceanu, R.I., Aldrich, J., 2014. A case study on the lightweight verification of a multi-threaded task server. Sci. Comput. Program. 80, 169–187.

Cav, V., Zhao, J., Shirako, J., Sarkar, V., 2011. Habanero-Java: The New Adventures of Old X10. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java. ACM, pp. 51–61.

Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H., 2004. Modular verification of software components in C. IEEE Trans. Software Eng. 30 (6), 388–402.

Chaki, S., Gurfinkel, A., 2018. BDD-Based Symbolic Model Checking. In: Handbook of Model Checking. Springer International Publishing, pp. 219–245.

Chaki, S., Rajamani, S.K., Rehof, J., 2002. Types As Models: Model Checking Message-passing Programs. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 45–57.

Clarke, D., Wrigstad, T., 2003. External Uniqueness Is Unique Enough. Springer, Berlin, Heidelberg, pp. 176–200.

Clarke, D., stlund, J., Sergey, I., Wrigstad, T., 2013. Ownership Types: A Survey. Springer, Berlin, Heidelberg, pp. 15–58.

Clebsch, S., Drossopoulou, S., Blessing, S., Mcneil, A., 2015. Deny Capabilities for Safe, Fast Actors. In: Proceedings of the 5th....

Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S., 2009. VCC: A Practical System for Verifying Concurrent C. Springer, Berlin, Heidelberg, pp. 23–42.

Cok, D.R., 2011. OpenJML: JML for Java 7 by extending OpenJDK. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, pp. 238–252.

DeLine, R., Fähndrich, M., 2002. Adoption and focus: practical linear types for imperative programming. Programming language design and implementation, ACM SIGPLAN.

DeLine, R., Fähndrich, M., 2004. Typestates for Objects. In: Odersky, M. (Ed.), ECOOP 2004 – Object-Oriented Programming. Springer Berlin Heidelberg, pp. 465–490.

Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S., 2005. A distributed object-oriented language with session types. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Dias, R.J., Pessanha, V., Loureno, J.M., 2013. Precise Detection of Atomicity Violations. Springer, Berlin, Heidelberg, pp. 8–23.

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V., 2010. Concurrent Abstract Predicates. In: Proceedings of the 24th European Conference on Object-oriented Programming. Springer-Verlag, pp. 504–528.

Dockins, R., Hobor, A., Appel, A.W., 2009. A fresh look at separation algebras and share accounting. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Dohrau, J., Summers, A.J., Urban, C., Münger, S., Müller, P., 2018. Permission inference for array programs. In: Chockler, H., Weissenbacher, G. (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 55–74.

Dolby, J., Hammer, C., Marino, D., Tip, F., Vaziri, M., Vitek, J., 2012. A data-centric approach to synchronization. ACM Trans. Program. Lang. Syst. 34 (1), 4:1–4:48.

Engler, D., Ashcraft, K., 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. ACM, pp. 237–252.

Fähndrich, M., Logozzo, F., 2011. Static contract checking with abstract interpretation. In: Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software. Springer-Verlag, pp. 10–30.

Ferrara, P., Müller, P., 2012. Automatic Inference of Access Permissions. In: Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings. Springer Berlin Heidelberg, pp. 202–218.

Filliâtre, J.C., Paskevich, A., 2013. Why3 - Where programs meet provers. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Flanagan, C., Freund, S.N., Lifshin, M., Qadeer, S., 2008. Types for atomicity: static checking and inference for Java. ACM Trans. Program. Lang. Syst. 30 (4), 20:1–20:53.

Flanagan, C., Qadeer, S., Flanagan, C., Qadeer, S., 2003. A type and effect system for atomicity. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation - PLDI '03. ACM Press, pp. 338–349.

Floyd, R.W., 1967. Assigning meanings to programs. Proceedings of Symposium on Applied Mathematics 19, 19–32 http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf.

Garcia, R., Tanter, Wolff, R., Aldrich, J., 2014. Foundations of typestate-oriented programming. ACM Trans. Program. Lang. Syst. 36 (4), 12:1–12:44.

Gay, S.J., Gesbert, N., Ravara, A., Vasconcelos, V.T., 2015. Modular session types for objects. Logical Methods in Computer Science 11 (4). doi:10.2168/LMCS-11(4:12)2015.

Gay, S.J., Vasconcelos, V.T., 2010. Linear type theory for asynchronous session types. J. Funct. Program.

Gherghina, C., David, C., Qin, S., Chin, W.-N., 2011. Structured specifications for better verification of heap-manipulating programs. In: FM 2011: Formal Methods. Springer Berlin Heidelberg, pp. 386–401.

Giannini, P., Richter, T., Servetto, M., Zucca, E., 2018. Tracing sharing in an imperative pure calculus. CoRR abs/1803.0.

Giannini, P., Servetto, M., Zucca, E., 2018. A Type and Effect System for Uniqueness and Immutability. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. ACM, pp. 1038–1045.

Girard, J.-Y., 1987. Linear logic. Theor. Comput. Sci. 50 (1), 1–101.

Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J., 2012. Uniqueness and reference immutability for safe parallelism. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12.

Gotsman, A., Berdine, J., Cook, B., Sagiv, M., 2007. Thread-modular shape analysis. SIGPLAN Not. 42 (6), 266–277 http://reports-archive.adm.cs.cmu.edu/anon/2003/CMU-CS-03-135.pdf.

Greenhouse, A., 2003. A Programmer-Oriented Approach to Safe Concurrency. Technical Report.

Greenhouse, A., Scherlis, W.L., 2002. Assuring and evolving concurrent programs: annotations and policy. In: Proceedings of the 24th International Conference on Software Engineering.

Haack, C., Huisman, M., Hurlin, C., 2008. Reasoning about java's reentrant locks. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Haack, C., Hurlin, C., 2008. Separation Logic Contracts for a Java-Like Language with Fork/Join. In: Algebraic Methodology and Software Technology. Springer Berlin Heidelberg, pp. 199–215.

Hammer, C., Dolby, J., Vaziri, M., Tip, F., 2008. Dynamic Detection of Atomic-Set-Serializability Violations. In: Proceedigns of the 30th International Conference on Software Engineering (ICSE' 08).

Heule, S., Leino, K.R.M., Müller, P., Summers, A.J., 2011. Fractional permissions without the fractions. In: Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs. ACM, p. 1.

Heule, S., Leino, K.R.M., Müller, P., Summers, A.J., 2013. Abstract read permissions: Fractional permissions without the fractions. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. Springer, pp. 315–334.

Hoare, C.A.R., 1969. An axiomatic basis for computer programming. Commun. ACM 12 (10), 576–580.

Hoare, C.A.R., 1972. Towards a Theory of Parallel Programming. In: The Origin of Concurrent Programming. Springer New York, pp. 231–244.

Hoare, C.A.R., 1974. Monitors: an operating system structuring concept. Commun. ACM 17, 549–557.

Hobor, A., Appel, A.W., Nardelli, F.Z., 2008. Oracle semantics for concurrent separation logic. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Hobor, A., Gherghina, C., 2012. Barriers in concurrent separation logic: now with tool support!. Logical Methods in Computer Science.

Honda, K., 1993. Types for Dyadic Interaction. Springer, Berlin, Heidelberg, pp. 509–523.

Honda, K., Vasconcelos, V.T., Kubo, M., 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In: Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems. Springer-Verlag, pp. 122–138.

Honda, K., Yoshida, N., Carbone, M., 2008. Multiparty Asynchronous Session Types. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 273–284.

Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K., 2010. Type-safe eventful sessions in Java. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Hu, R., Yoshida, N., Honda, K., 2008. Session-based distributed programming in Java. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Huisman, M., 2001. Reasoning about Java programs in higher order logic using PVS and Isabelle. PhD Thesis. Computing Science Institute, University of Nijmegen.

Huisman, M., Mostowski, W., 2015. A symbolic approach to permission accounting for concurrent reasoning. In: Proceedings - IEEE 14th International Symposium on Parallel and Distributed Computing, ISPDC 2015.

Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Denilou, P.-M., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G., 2016. Foundations of session types and behavioural contracts. ACM Comput. Surv. 49 (1), 3:1–3:36.

Igarashi, A., Kobayashi, N., 2001. Resource usage analysis. In: POPL.

Igarashi, A., Kobayashi, N., 2005. Resource usage analysis. ACM Trans. Program. Lang. Syst. 27 (2), 264–313.

Jacobs, B., Leino, K., Piessens, F., Schulte, W., 2005. Safe concurrency for aggregate objects with invariants. In: Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05). IEEE, pp. 137–146.

Jacobs, B., Piessens, F., 2011. Expressive modular fine-grained concurrency specification. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 271–282.

Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F., 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and java. In: NASA Formal Methods Symposium. Springer, pp. 41–55.

Jacobs, B., Smans, J., Piessens, F., 2010. A Quick Tour of the VeriFast Program Verifier. Springer, Berlin, Heidelberg, pp. 304–311. doi:10.1007/978-3-642-17164-2_21.

Jensen, J.B., Birkedal, L., 2012. Fictional Separation Logic. In: Seidl, H. (Ed.), Programming Languages and Systems. Springer Berlin Heidelberg, pp. 377–396.

Jones, C.B., 1983. Specification and Design of (Parallel) Programs. In: IFIP Congress.

Joosten, S.J.C., Oortwijn, W., Safari, M., Huisman, M., 2018. An exercise in verifying sequential programs with vercors. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops. ACM, pp. 40–45 http://doi.acm.org/10.1145/3236454.3236479.

Kidd, N., Reps, T., Dolby, J., Vaziri, M., 2011. Finding concurrency-related bugs using random isolation. Int. J. Software Tools Technol. Trans.

Kim, T., Bierhoff, K., Aldrich, J., Kang, S., 2009. Typestate protocol specification in JML. In: Proceedings of the 8th international workshop on Specification and verification of component-based systems. ACM, pp. 11–18.

Kobayashi, N., Sangiorgi, D., 2010. A hybrid type system for lock-freedom of mobile processes. ACM Transactions on Programming Languages and Systems 32 (5), 1–49.

Lai, Z., Cheung, S.C., Chan, W.K., 2010. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10.

Le, D.-K., Chin, W.-N., Teo, Y.-M., 2012. Variable permissions for concurrency verification. In: Formal Methods and Software Engineering: 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12–16, 2012. Proceedings. Springer Berlin Heidelberg, pp. 5–21.

Leavens, G.T., Baker, A.L., Ruby, C., 2006. Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31 (3), 1–38.

Leino, K.R.M., Müller, P., 2009. A basis for verifying multi-threaded programs. In: European Symposium on Programming. Springer, pp. 378–393.

Leino, K.R.M., Müller, P., Smans, J., 2009. Verification of Concurrent Programs with Chalice. In: Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures. Springer Berlin Heidelberg, pp. 195–222.

Lu, S., Park, S., Seo, E., Zhou, Y., Lu, S., Park, S., Seo, E., Zhou, Y., Lu, S., Park, S., Seo, E., Zhou, Y., Lu, S., Park, S., Seo, E., Zhou, Y., 2008. Learning from mistakes. ACM SIGOPS Operating Systems Review 42 (2), 329.

Marino, D., Hammer, C., Dolby, J., Vaziri, M., Tip, F., Vitek, J., 2013. Detecting deadlock in programs with data-centric synchronization. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 322–331.

Matsakis, N.D., Klock, F.S., Matsakis, N.D., Klock II, F.S., 2014. The rust language. In: Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology - HILT '14. ACM Press, pp. 103–104.

Meyer, B., 1988. Object-Oriented Software Construction, 1st Prentice-Hall, Inc.

Meyer, B., 1992. Applying 'design by contract'. Computer 25 (10), 40–51.

Militao, F., Aldrich, J., Caires, L., 2010. Aliasing Control with View-based Typestate. In: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs. ACM, pp. 7:1–7:7.

Militão, F., Aldrich, J., Caires, L., 2014. Rely-Guarantee Protocols. In: ECOOP 2014 – Object-Oriented Programming. Springer Berlin Heidelberg, pp. 334–359.

Militão, F., Aldrich, J., Caires, L., 2014. Substructural Typestates. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages Meets Program Verification. ACM, pp. 15–26.

Militão, F., Aldrich, J., Caires, L., 2016. Composing interfering abstract: Protocols. In: 30th European Conference on Object-Oriented Programming, ECOOP 2016, 56, pp. 161–1626.

Militão, F., Caires, L., 2009. An Exception Aware Behavioral Type System for Object-Oriented Programs. In: INFORUM 2009 - Simpsio de Informtica. Faculdade de Cincias - Universidade de Lisboa, pp. 1–12.
Morrisett, G., Ahmed, A., Fluet, M., 2005. L3: A Linear Language with Locations. In: Typed Lambda Calculi and Applications. Springer Berlin Heidelberg, pp. 293–307.
Müller, P., Rudich, A., 2007. Ownership transfer in universe types. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07.
Müller, P., Schwerhoff, M., Summers, A.J., 2016. Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (Eds.), Verification, Model Checking, and Abstract Interpretation. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 41–62.
Müller, P., Schwerhoff, M., Summers, A.J., 2017. Viper: A Verification Infrastructure for Permission-based Reasoning. In: Dependable Software Systems Engineering.
Naden, K., Bocchino, R., Aldrich, J., Bierhoff, K., 2012. A Type System for Borrowing Permissions. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 557–570.
Naik, M., Park, C.S., Sen, K., Gay, D., 2009. Effective static deadlock detection. In: Proceedings - International Conference on Software Engineering.
Nguyen, H.H., Chin, W.-N., 2008. Enhancing program verification with lemmas. In: Proceedings of the 20th International Conference on Computer Aided Verification. Springer-Verlag, pp. 355–369.
Nielson, F., Nielson, H.R., 1993. From CML to Process Algebras. Springer, Berlin, Heidelberg, pp. 493–508.
Nielson, F., Nielson, H.R., 1996. From CML to its Process Algebra. Theor. Comput. Sci.
Noble, J., Vitek, J., Potter, J., 1998. Flexible alias protection. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).
Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M., 2004. An overview of the Scala programming language. Technical Report.
O'Hearn, P., Reynolds, J., Yang, H., 2001. Local Reasoning about Programs that Alter Data Structures. In: Computer Science Logic. Springer Berlin Heidelberg, pp. 1–19.
OHearn, P.W., 2007. Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375 (1-3), 271–307.
Owicki, S., Gries, D., 1976. Verifying properties of parallel programs: An axiomatic approach. Commun. ACM 19 (5), 279–285.
Owre, S., Rushby, J.M., Shankar, N., 1992. PVS: a prototype verification system. 11th International Conference on Automated Deduction.
Parkinson, M., Bierman, G., 2005. Separation logic and abstraction. ACM SIGPLAN Notices 40 (1), 247–258.
Paulino, H., Parreira, D., Delgado, N., Ravara, A., Matos, A., 2016. From Atomic Variables to Data-centric Concurrency Control. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM, pp. 1806–1811.
Pradel, M., Jaspan, C., Aldrich, J., Gross, T.R., 2012. Statically checking API protocol conformance with mined multi-object specifications. In: Proceedings - International Conference on Software Engineering.
Reynolds, J.C., 1978. Syntactic Control of Interference. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, pp. 39–46.
Reynolds, J.C., 2002. Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. IEEE, pp. 55–74.
Rodríguez, E., Dwyer, M., Flanagan, C., Hatcliff, J., Leavens, G., Robby, 2005. Extending JML for Modular Specification and Verification of Multi-threaded Programs. In: ECOOP 2005 - Object-Oriented Programming.
Roux, P., Siminiceanu, R., 2010. Model Checking with Edge-valued Decision Diagrams. In: Second {NASA} Formal Methods Symposium - {NFM} 2010, Washington D.C., USA, April 13-15, 2010. Proceedings, pp. 222–226.
Sadiq, A., Li, Y., Ling, S., Li, L., Ahmed, I., 2019. Automatic inference of symbolic permissions for sequential java programs. CoRR abs/1902.05311.
Sadiq, A., Li, Y.-F., Ling, S., Ahmed, I., 2016. Extracting Permission-Based Specifications from a Sequential Java Program. In: 21st International Conference on Engineering of Complex Computer Systems, United Arab Emirates, November 6-8, 2016, pp. 215–218.
Sadiq, A., Li, Y.F., Ling, S., Ahmed, I., 2019. Sip4j: Statically inferring access permission contracts for parallelising sequential java programs. In: Proceedings of the 34rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2019, 2019.
Sangiorgi, D., 1999. The name discipline of uniform receptiveness. Theor. Comput. Sci 221, 457–493.
Schwinghammer, J., Birkedal, L., Reus, B., Yang, H., 2011. Nested hoare triples and frame rules for higher-order store. Logical Methods in Computer Science 7 (3).
Siek, J., Taha, W., 2007. Gradual typing for objects. In: Ernst, E. (Ed.), ECOOP 2007 – Object-Oriented Programming. Springer Berlin Heidelberg, pp. 2–27.
Siminiceanu, R.I., Ahmed, I., Cataño, N., 2012. Automated verification of specifications with typestates and access Permissions. ECEASST 53.
Smans, J., Jacobs, B., Piessens, F., 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. Springer, Berlin, Heidelberg, pp. 148–172.
Stork, S., Naden, K., Sunshine, J., Mohr, M., Fonseca, A., Marques, P., Aldrich, J., 2014. aem: A permission-based concurrent-by-default programming language approach. ACM Trans. Program. Lang. Syst. 36 (1), 1–42.
Strom, R.E., Yemini, S., 1986. Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. (1) 157–171.
Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, 2011. First-class state change in plaid. ACM SIGPLAN Notices.
Takeuchi, K., Honda, K., Kubo, M., 1994. An interaction-based language and its typing system. In: In PARLE94, volume 817 of LNCS, pp. 398–413.
Vafeiadis, V., Parkinson, M., 2007. A Marriage of Rely/Guarantee and Separation Logic. In: Caires, L., Vasconcelos, V.T. (Eds.), CONCUR 2007 – Concurrency Theory. Springer Berlin Heidelberg, pp. 256–271.
Vaziri, M., Tip, F., Dolby, J., 2006. Associating Synchronization Constraints with Data in an Object-oriented Language. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 334–345.
Vaziri, M., Tip, F., Dolby, J., Hammer, C., Vitek, J., 2010. A Type System for Data-Centric Synchronization. In: ECOOP 2010 – Object-Oriented Programming. Springer Berlin Heidelberg, pp. 304–328.
Villard, J., Lozes, É., Calcagno, C., 2009. Proving copyless message passing. In: Programming Languages and Systems. Springer Berlin Heidelberg, pp. 194–209.
Villard, J., Lozes, É., Calcagno, C., 2010. Tracking Heaps That Hop with Heap-Hop. Springer, Berlin, Heidelberg, pp. 275–279.
Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F., 2003. Model checking programs. Automated Software Engineering 10 (2), 203–232.
Voung, J.W., Jhala, R., Lerner, S., 2007. RELAY. In: Proceedings of 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07. ACM Press, p. 205.
Wadler, P., 2014. Propositions as sessions. In: J. Funct. Program.
Westbrook, E., Zhao, J., Budimli, Z., Sarkar, V., 2012. Practical Permissions for Race-free Parallelism. In: Proceedings of the 26th European Conference on Object-Oriented Programming. Springer-Verlag, pp. 614–639.
Xu, M., Bodk, R., Hill, M.D., Xu, M., Bodk, R., Hill, M.D., 2005. A serializability violation detector for shared-memory server programs. ACM SIGPLAN Notices 40 (6), 1.
Zhao, Y., 2007. Concurrency analysis based on fractional permissions. University of Wisconsin-Milwaukee.
Zhao, Y., Yu, L., Bei, J., 2008. The Permission Approach to Comprehend Lock-Based Synchronization Policy. In: 2008 International Conference on Advanced Computer Theory and Engineering. IEEE, pp. 709–713.