2018 IEEE Third International Conference on Data Science in Cyberspace

Reasoning about Cloud Storage Systems

Hanpin Wang, Zhao Jin, Lei Zhang, Yuxin Jing, Jiang Xu and Yongzhi Cao
Key Laboratory of High Confidence Software Technologies (MOE),
School of Electronics Engineering and Computer Science,
Peking University, Beijing 100871, China
Email: {whpxhy, jinzhao, zhangleijuly, jyx, jiang.xu, caoyz}@pku.edu.cn

***Abstract*—The rapid growth of data has restricted the capability of traditional storage technologies to store and manage data. Massive growth in big data generated through cloud storage systems (CSSs) has been observed. An important feature of CSSs is that data are stored in blocks, and each block is considered as a storage unit. Hence, CSSs usually have two kinds of storage units: ordinary locations and block locations. It makes CSSs very different from ordinary storage systems. Then how do we appeal formal methods to model, describe and reason about CSSs? In this paper, based on Separation Logic, we propose a systematic method to verify the correctness of management programs in CSSs.**

The main contributions are as follows. (1) A language is introduced to describe the cloud storage management. (2) Assertions in Separation Logic are extended to describe the properties of blocks in CSSs. (3) Hoare-style rules are proposed to reason about the CSSs. Pre- and post-conditions are pairs of assertions. Using these methods, the partial correctness of cloud storage management can be verified.

***Index Terms*—cloud storage systems, Separation Logic, modeling language, formal verification**

1. Introduction

Data creation is occurring at a record rate, referred to big data, has emerged as a widely recognized trend. Cloud computing is a fast-growing technology to perform massive-scale and complex computing. Big data and cloud computing are conjoined [10]. The former provides users the ability to process queries across multiple datasets, and the latter provides the underlying engine. As an elementary part of cloud computing, cloud storage plays an important role in underlying data collection, storage, maintenance and output. Nowadays, many clouding computing systems have already been put into use, with their own cloud storage systems (CSSs). One typical example is Google File System (GFS) [8], which is a scalable distributed file system for large data-intensive applications and takes Google cloud computing into practice.

Like GFS, CSSs have more characteristics than traditional storage systems. An important one is that data files are stored in a way of block. That is, the storage resources of CSSs consist of small block spaces with a fixed size. When storing a data file, the system may first cut the file into some segments, and then put these segments into proper blocks, taking every segment as a whole. Take GFS as an example. Roughly, GFS works as follows. When a costumer makes a request to store a file in a GFS, the system cuts the file into several fragments, each with an equal size (usually 64MB) except for the last one. Then the system assigns a sequence of blocks (called "chunks" [8] in GFS) to deposit fragments of the file, one by one. Accordingly, each chunk is of size 64M. Finally it returns all the block addresses so that it can build a file table to record the relation between the file fragments and the blocks. Other CSSs, like HDFS [6], have a similar procedure.

At first glance, the above procedure of saving a file into blocks in a CSS looks similar to that of saving values into memory cells in a memory management system. But there are still some substantial differences. For example, an array can be put into a series of continuous memory cells, whereas there are no "continuous" block addresses in most CSSs. Instead, a file may be put into several discrete blocks, even spreading in different block servers. These differences make the properties of cloud storage management very different from those of traditional memory management. Although the concept of block storage has been proposed for many years, it becomes more complicated in CSSs. Therefore it brings the problem how to ensure the reliability of such CSSs.

The reliability of CSSs may include many issues, such as retrievability, search efficiency, correctness of management programs, etc. Here we focus on the correctness of management programs, which is the basis of the reliability of CSSs. Formal methods are mature enough for developing the correctness of most computer programs. Furthermore, formal method for traditional file system is a hot topic. P. Gardner and G. Nizik proposed their work about local reasoning for the POSIX file system [7, 16]. W.H Hesselink and M.I Lal provided abstract definitions for file systems which are defined as a partial function from paths to data [11]. In addition, some efforts to formalize CSSs have been made. V. Serbanescu et al. proposed a modeling method for distributed data aggregation service relied on a storage system [19, 20]. Stephen et al. used formal methods to analyze data flow and proposed an execution model for

978-1-5386-4210-8/18/$31.00 ©2018 IEEE
DOI 10.1109/DSC.2018.00024

executing Pig Latin scripts in cloud systems without sacrificing confidentiality of data [21]. Ateniese et al. used so-called third party auditability in their "provable data possession" model to ensure the file integrity on untrusted storages [1, 2]. Wang et al. achieved a privacy-preserving method in public cloud data auditing systems, and proposed an auditing mechanism with lightweight communication and computation cost [23, 24]. I. Pereverzeva et al. found a formal solution for CSSs development [17]. It had the capability of modeling large and elastic data storage system.

All the above works cannot reason about the existed CSSs. Neither it has a unified modeling language and specifications. Separation Logic [18], which is a Hoare-style logic [12], has a strong theoretical and practical significance. By using separation logic, some verification systems have been implemented [3–5, 14, 15, 22].

In this paper, based on Separation Logic, we propose a systematic method to verify the correctness of management programs in CSSs. The main contributions are as follows.

- A language is defined to describe the cloud storage management. The new commands mainly focus on block operations. Accordingly, its operational semantics is given by using the concepts of heap and store.
- Assertions in Separation Logic are extended to describe the properties of blocks in CSSs. Quantifiers over block variables and file variables are incorporated with ordinary Separation Logic assertions. A tautology inference system is built up and is proved to be sound.
- Hoare-style rules are proposed to reason about the CSSs. Pre- and post-conditions are pairs of assertions. One component of every pair is used to describe the properties of ordinary locations (as in memory), while the other component is used to describe the properties of block locations.

We end up our paper with an example of appending file to another to demonstrate the feasibility of our method.

2. The Modeling Language for CSSs

The Modeling language is an extension of **WHILE**h [9] programming language with new ingredients to reflect the new characteristics in CSSs, which is, data files are stored as block. When storing a data file, the system may first cut a file into some segments, and then put these segments into proper blocks with taking every segment as a whole. Naturally, we introduce the modeling language for CSSs by adding new constructs for files and blocks. For space reasons, the content of a block in our language will be considered as a single integer value.

2.1. Syntax

In our language there are four kinds of expressions: (arithmetic) location expressions, file expressions, block expressions, and Boolean expressions. The full syntax for expressions and commands in our language is as follows:

$$
\begin{array}{rcl}
e & ::= & n \mid x, y, ... \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid \#f \\
fe & ::= & \mathbf{nil} \mid f \mid fe \cdot bk \mid fe_1 \cdot fe_2 \\
bk & ::= & n \mid b \mid f(e) \\
be & ::= & e_1 = e_2 \mid e_1 \leq e_2 \mid bk_1 == bk_2 \mid bk_1 <== bk_2 \mid \\
 & & \mathbf{true} \mid \mathbf{false} \mid \neg be \mid be_1 \wedge be_2 \mid be_1 \vee be_2 \\
C & ::= & x := e \mid x := \mathbf{cons}(e_1, ..., e_n) \mid x := [e] \mid [e] := e' \\
 & & \mid \mathbf{dispose}(e) \mid f := \mathbf{create}(bk^*) \mid \\
 & & f := \mathbf{append}(bk^*) \mid f_1 := f_2 \centerdot bk^* \mid \mathbf{delete}\ f \mid \\
 & & b := bk \mid b := \{bk\} \mid \{bk\} := bk' \mid \mathbf{delete}\ bk \mid \\
 & & C; C' \mid \mathbf{if}\ be\ \mathbf{then}\ C\ \mathbf{else}\ C' \mid \mathbf{while}\ be\ \mathbf{do}\ C'
\end{array}
$$

where e is written for arithmetic location expressions, fe for file expressions, bk for block expressions, be for Boolean expressions, and C for commands.

Intuitively, $\#f$ means the block numbers the file f occupies, and $f(e)$ points out the address that the i-th block of the file f corresponds to, where i is the value of the expression e. As usual we write $bk_1 << bk_2$ to express the Boolean expression $(bk_1 <== bk_2) \wedge \neg(bk_1 == bk_2)$, and $e_1 < e_2$ to express $(e_1 <= e_2) \wedge \neg(e_1 = e_2)$.

Besides the commands in **WHILE**h , which contains all commands of IMP [25], we introduce some new commands to describe the special operations about files and blocks in cloud storage management program.

File commands contain four core operations, which seem to be enough to describe the majority of daily-file operations in CSSs.

- $f := \mathbf{create}(bk^*)$: creation;
- $f := \mathbf{append}(bk^*)$: content appending;
- $f_1 := f_2 \centerdot bk^*$: address appending;
- $\mathbf{delete}\ f$: file deletion.

Block commands express block operations. They are as follows.

- $b := \{bk\}$: block lookup;
- $b := bk$: block assignment;
- $\{bk\} := bk'$: block mutation;
- $\mathbf{delete}\ bk$: block deletion.

2.2. Domains

The model has five components: Stores_V, Stores_B, Stores_F, Heaps_V, and Heaps_B. The Stores_V is a total function mapping from location variables to addresses. The Stores_B is a total function mapping from block variables to block addresses. Strictly speaking, values, addresses, and block addresses are different in type. But in our language, to permit unrestricted address arithmetic, we assume that all the values, addresses, and block addresses are integers. The Stores_F is a total function mapping from file variables into a sequence of block addresses. The Heaps_V is indexed by a subset Loc of the integers, and is accessed using indirect addressing $[e]$, where e is an (arithmetic) location expression. The Heaps_B is indexed by a subset BlockLoc of

the integers, and is accessed using indirect addressing $\{bk\}$, where bk is a block expression.

$$\text{Values} \triangleq \{..., -1, 0, 1, ...\} = \mathbb{Z}$$
$$\text{Loc}, \text{BLoc}, \text{Atoms} \subseteq \text{Values}$$
$$\text{Var} \triangleq \{x, y, ...\} \quad \text{BKVar} \triangleq \{b_1, b_2, ...\} \quad \text{FVar} \triangleq \{f_1, f_2, ...\}$$
$$\text{Stores}_V \triangleq \text{Var} \to \text{Values} \quad \text{Stores}_B \triangleq \text{BKVar} \to \text{Values}$$
$$\text{Stores}_F \triangleq \text{Var} \to \text{BLoc}^*$$
$$\text{Heaps}_V \triangleq \text{Loc} \rightharpoonup_{\text{fin}} \text{Values}$$
$$\text{Heaps}_B \triangleq \text{BLoc} \rightharpoonup_{\text{fin}} \text{Values}$$

where $\rightharpoonup$ and $\rightharpoonup_{\text{fin}}$ can be found in [9]. Loc, BLoc and Atoms are disjoint, and $\text{BLoc}^* = \{(bloc_1, ..., bloc_n) \mid bloc_i \in \text{BLoc}, n \in \mathbb{N}\}$, and for any element $(bloc_1, ..., bloc_n)$ of BLoc^*, $|(bloc_1, ..., bloc_n)|$ means its length, that is, $|(bloc_1, ..., bloc_n)| = n$.

In order for allocation to always succeed, we place a requirement on the sets Loc and BLoc. For any positive integer m, there are infinitely many sequences of length m of consecutive integers in Loc. For any positive integer n, there are infinitely many sequences of length m of discrete integers in BLoc. This requirement is satisfied if we take Loc and BLoc as the non-negative integers. Then we could take Atoms as the negative integers, and **nil** as -1.

The states of our language is defined as follows:

$$\text{States} \triangleq \text{Stores}_V \times \text{Stores}_B \times \text{Stores}_F \times \text{Heaps}_V \times \text{Heaps}_B$$

A state $\sigma \in$ States is a 5-tuple: $(s_V, s_B, s_F, h_V, h_B)$.

2.3. Semantics of the modeling language

Once the states are defined, we can specify the evaluation rules of our new expressions. Notice that when we try to give out the semantic of a expression, some stores and heaps may not be used. For example, expression $\#f$ only needs Stores_F and Stores_V. So we will only list the necessary stores and heaps for each expression. Note that in our language, expressions are heap-independent.

$$[\![\#f]\!](s_V, s_F) = |s_F(f)|; \qquad [\![b]\!](s_V, s_B, s_F) = s_B(b);$$
$$[\![f(e)]\!](s_V, s_B, s_F) = bloc_i, \text{ if } s_F(f) = (bloc_1, ..., bloc_n), [\![e]\!](s_V, s_F) = i \text{ and } 1 \le i \le n;$$
$$[\![\mathbf{nil}]\!](s_V, s_F) = (); \qquad [\![f]\!](s_V, s_B, s_F) = s_F(f);$$
$$[\![fe \cdot bk]\!](s_V, s_B, s_F) = (bloc_1, ..., bloc_n, bloc'), \text{ if } [\![fe]\!](s_V, s_B, s_F) = (bloc_1, ..., bloc_n) \text{ and } [\![bk]\!](s_V, s_B, s_F) = bloc';$$
$$[\![fe_1 \cdot fe_2]\!](s_V, s_B, s_F) = (bloc_1, ..., bloc_n, bloc_1', ..., bloc_n'), \text{ if } [\![fe_1]\!](s_V, s_B, s_F) = (bloc_1, ..., bloc_n), \text{ and } [\![fe_2]\!](s_V, s_B, s_F) = (bloc_1', ..., bloc_n');$$
$$[\![bk_1 == bk_2]\!](s_V, s_B, s_F) = \begin{cases} T & \text{if } [\![bk_1]\!](s_V, s_B, s_F) = [\![bk_2]\!](s_V, s_B, s_F) \\ F & \text{otherwise;} \end{cases}$$
$$[\![bk_1 <== bk_2]\!](s_V, s_B, s_F) = \begin{cases} T & \text{if } [\![bk_1]\!](s_V, s_B, s_F) <= [\![bk_2]\!](s_V, s_B, s_F) \\ F & \text{otherwise.} \end{cases}$$

To state the semantics formally, following [9], we use the crucial operations on the heaps:

- $\text{dom}(h_H)$ denotes the domain of a heap $h_H \in \text{Heaps}_H$, where h_H range over h_V and h_B, Heaps_H range over Heaps_V and Heaps_B, and $\text{dom}(s_S)$ is the domain of a store $s_S \in \text{Stores}_S$, where s_S range over s_V, s_B and s_F, Stores_S range over Stores_V, Stores_B and Stores_F;
- $h_H \# h_H{}'$ indicates that the domains of h_H and $h_H{}'$ are disjoint;
- $h_H * h_H{}'$ is defined when $h_H \# h_H{}'$ holds and is a finite function obtained by taking the union of h_H and $h_H{}'$;
- $i \mapsto j$ is a singleton partial function which maps i to j;
- for a partial function f from U to W and f' from V to W, the partial function $f[f']$ from U to W is defined by:

$$f[f'](i) \triangleq \begin{cases} f'(i) & \text{if } i \in \text{dom}(f'), \\ f(i) & \text{if } i \in \text{dom}(f), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here we give out the operational semantics of our new commands, which are specified by the following inference rules. Each command is interpreted using a relation "$\rightsquigarrow$" between configurations. Each configuration includes a state σ, a command-state pair $\langle C, \sigma \rangle$, or a special configuration *fault* indicating a memory fault.

$$\frac{(bk_1, \sigma) \longrightarrow m_1, ..., (bk_n, \sigma) \longrightarrow m_n}{\langle f := \mathbf{create}(bk_1, ..., bk_n), \sigma \rangle \rightsquigarrow (s_V, s_B, s_F[(m_1{}', ..., m_n{}')/f], h_V, [h_B | m_1{}' : m_1 | ... | m_n{}' : m_n])}$$
where $m_1{}', ..., m_n{}' \in \text{BLoc} - \text{dom}(h_B)$.

$$\frac{(bk_1, \sigma) \longrightarrow m_1, ..., (bk_n, \sigma) \longrightarrow m_n}{\langle f := \mathbf{append}(bk_1, ..., bk_n), \sigma \rangle \rightsquigarrow (s_V, s_B, s_F[(s_F(f) \centerdot (m_1{}', ..., m_n{}'))/f], h_V, [h_B | m_1{}' : m_1 | ... | m_n{}' : m_n])}$$
where $m_1{}', ..., m_n{}' \in \text{BLoc} - \text{dom}(h_B)$.

$$\frac{(bk_1, \sigma) \longrightarrow m_1, ..., (bk_n, \sigma) \longrightarrow m_n}{\langle f_1 := f_2 \centerdot (bk_1, ..., bk_n), \sigma \rangle \rightsquigarrow (s_V, s_B, s_F[(s_F(f_2) \centerdot (m_1, ..., m_n))/f_1], h_V, h_B)}$$

$$\langle \mathbf{delete}\ f, \sigma \rangle \rightsquigarrow (s_V, s_B, s_F \rceil (\text{dom}(s_F) \setminus \{f\}), h_V, h_B)$$

$$\frac{(bk, \sigma) \longrightarrow m}{\langle b := \{bk\}, \sigma \rangle \rightsquigarrow (s_V, s_B[h_B(m)/b], s_F, h_V, h_B)}$$

$$\frac{(bk, \sigma) \longrightarrow m}{\langle b := bk, \sigma \rangle \rightsquigarrow (s_V, s_B[m/b], s_F, h_V, h_B)}$$

$$\frac{(bk, \sigma) \longrightarrow m_1, (bk', \sigma) \longrightarrow m_2}{\langle \{bk\} := bk', \sigma \rangle \rightsquigarrow (s_V, s_B, s_F, h_V, h_B[m_2/m_1])}$$

$$\frac{(bk, \sigma) \longrightarrow m}{\langle \mathbf{delete}\ bk, \sigma \rangle \rightsquigarrow (s_V, s_B \rceil (\text{dom}(s_B) \setminus \{m\}), s_F, h_V, h_B)}$$

$[f|x:a]$ denotes the function that maps variable x into value a and all other variables y in $\mathrm{dom}(f)$ into $f(y)$. $[f\rceil s]$ denotes the restriction of the function f to domain s. They both can be found in [18]. $(m_1,...,m_n)$. $(m_1{}',...,m_n{}')$ means the conjunction of two sequences. Since $s_F(f)$ is a sequence of block addresses, $s_F(f)$.$(m_1,...,m_n)$ denotes a sequence of block addresses which is a composition of $s_F(f)$ followed by $(m_1,...,m_n)$.

3. The Assertion Language for CSSs

To describe the properties of CSSs, we construct a logic to deal with both locations and blocks. **BI** Pointer Logic [13] provides an elegant and powerful formalism for ordinary locations. We extend **BI** pointer logic with the file and block expressions to describe files and blocks. Following [9], the formal semantics of an assertion is defined by a satisfactory relation " $\models$ " between a state and an assertion. $(s_V, s_B, s_F, h_V, h_B) \models p$ means that the assertion p holds in the state $(s_V, s_B, s_F, h_V, h_B)$.

We rename the assertions of **BI** pointer logic as *location assertions*, meanwhile call the block formulas as *block assertions*. Both location assertions and block assertions are built on expressions.

3.1. Location Assertion

3.1.1. Syntax. Location assertions describe the properties about locations. However, there are some differences between the **BI** assertions and location assertions. Quantifiers over file variables are allowed in location assertions. This makes the location assertions much more complicated, since they are not first-order quantifiers.

$$\begin{aligned}\alpha \quad ::= \quad & \mathbf{true}_V \mid \mathbf{false}_V \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid \\ & \alpha_1 \to \alpha_2 \mid e_1 = e_2 \mid e_1 \leq e_2 \mid \forall x.\alpha \mid \exists x.\alpha \mid \\ & \mid \forall f.\alpha \mid \exists f.\alpha \mid \mathbf{emp}_V \mid e \mapsto e' \mid \alpha_1 * \alpha_2 \mid \\ & \alpha_1 \mathrel{-\!\!*} \alpha_2\end{aligned}$$

3.1.2. Semantics. Intuitively, the truth value of a location assertion depends only on the stores and heaps for location variables. Note that a location assertion may contain such kind of expressions as $\#f$. So the satisfactory relation depends on the stores for files as well. Let s_V be a store for variables, s_F be a store for files, and h_V be a heap for variables. Given a location assertion α, we define $s_V, s_F, h_V \models \alpha$ by induction on α in the following table 1.

Definition of other location assertions are similar to these in **BI** pointer logic.

TABLE 1. SEMANTICS OF LOCATION ASSERTION

$s_V, s_F, h_V \models \mathbf{true}_V$;
$s_V, s_F, h_V \models \alpha_1 \to \alpha_2$ iff if $s_V, s_F, h_V \models \alpha_1$
then $s_V, s_F, h_V \models \alpha_2$;
$s_V, s_F, h_V \models (\forall x.\alpha)$ iff $s_V[n/x], s_F, h_V \models \alpha$ for any $n \in$ Loc;
$s_V, s_F, h_V \models (\forall f.\alpha)$ iff $s_V, s_F[\bar{n}/f], h_V \models \alpha$
for any $\bar{n} = (n_1, n_2, \ldots, n_k) \in$ BLoc*;
$s_V, s_F, h_V \models (\exists f.\alpha)$ iff $s_V, s_F[\bar{n}/f], h_V \models \alpha$
for some $\bar{n} = (n_1, n_2, \ldots, n_k) \in$ BLoc*;
$s_V, s_F, h_V \models \mathbf{emp}_V$ iff $\mathrm{dom}(h_V) = \emptyset$;
$s_V, s_F, h_V \models \alpha_1 * \alpha_2$ iff there exist h_V^1, h_V^2 with
$h_V^1 \# h_V^2$ and $h_V = h_V^1 * h_V^2$ such that
$s_V, s_F, h_V^1 \models \alpha_1$ and $s_V, s_F, h_V^2 \models \alpha_2$.

3.2. Block Assertion

3.2.1. Syntax. Block assertions describe the properties about blocks. So they would include the logic operations and heap operations on block expressions besides ordinary logical connectives and quantifies.

$$\begin{aligned}\beta \quad ::= \quad & \mathbf{true}_B \mid \mathbf{false}_B \mid \neg\beta \mid \beta_1 \wedge \beta_2 \mid \beta_1 \vee \beta_2 \mid \\ & \beta_1 \to \beta_2 \mid bk_1 == bk_2 \mid bk_1 <== bk_2 \mid \forall x.\beta \mid \\ & \exists x.\beta \mid \forall b.\beta \mid \exists b.\beta \mid \forall f.\beta \mid \exists f.\beta \mid \mathbf{emp}_B \mid \\ & bk \mapsto bk' \mid \beta_1 * \beta_2 \mid \beta_1 \mathrel{-\!\!*} \beta_2 \mid f \looparrowright fe \mid fe = fe'\end{aligned}$$

3.2.2. Semantics. Obviously, the truth value of a block assertion depends on the stores and heaps for blocks. As a block expression may contains the form $f(e)$, the truth value of a block assertion also depends on the stores for variables and files. Let s_V be a store for variables, s_B be a store for blocks, s_F be a store for files, and h_B be a heap for blocks. Given a block assertion β, we define $s_V, s_B, s_F, h_B \models \beta$ by induction on β in the following table 2.

TABLE 2. SEMANTICS OF BLOCK ASSERTION

$s_V, s_B, s_F, h_B \models \mathbf{true}_B$;
$s_V, s_B, s_F, h_B \models bk_1 <== bk_2$ iff $[\![bk_1]\!]\sigma \leq [\![bk_2]\!]\sigma$;
$s_V, s_B, s_F, h_B \models (\forall x.\beta)$ iff $s_V[n/x], s_B, s_F, h_B \models \beta$
for any $n \in$ Loc;
$s_V, s_B, s_F, h_B \models (\exists b.\beta)$ iff $s_V, s_B[n/b], s_F, h_B \models \beta$
for some $n \in$ BLoc;
$s_V, s_B, s_F, h_B \models (\forall f.\beta)$ iff $s_V, s_B, s_F[\bar{n}/f], h_B \models \beta$
for any $\bar{n} = (n_1, n_2, \ldots, n_k) \in$ BLoc*;
$s_V, s_B, s_F, h_B \models (\beta_1 \mathrel{-\!\!*} \beta_2)$ iff for any block heap $h_B{}'$,
if $h_B{}' \# h_B$ and $s_V, s_B, s_F, h_B{}' \models \beta_1$,
then $s_V, s_B, s_F, h_B * h_B{}' \models \beta_2$;
$s_V, s_B, s_F, h_B \models f \looparrowright fe$ iff $[\![f]\!]\sigma = \bar{m}$ and $[\![fe]\!]\sigma = \bar{n}$,
and $h_B(m_i) = n_i$ for $1 \leq i \leq k$,
where $\bar{m} = (m_1, m_2, ..., m_k), \bar{n} = (n_1, n_2, ..., n_k) \in$ BLoc*
such that $|\bar{m}| = |\bar{n}| = k$.

Definition of other block assertions are similar to these in location assertions.

3.3. Global Assertion

3.3.1. Syntax. Roughly speaking, (location assertion,block assertion) pairs are called *global assertions*, and there are several operations between pairs. More precisely, we use

the symbol p to stand for the global assertions, with the following BNF equation:

$$p ::= \langle \alpha, \beta \rangle \mid \mathbf{true} \mid \mathbf{false} \mid \neg p \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid p_1 \to p_2 \mid \forall x.p \mid \exists x.p \mid \forall b.p \mid \exists b.p \mid \forall f.p \mid \exists f.p \mid \mathbf{emp} \mid p_1 * p_2 \mid p_1 \mathrel{-\!\!*} p_2$$

3.3.2. Semantics. The truth value of a global assertion depends on all kinds of stores and heaps. Let s_V be a store for variables, s_B be a store for blocks, s_F be a store for files, h_V be a heap for variables and h_B be a heap for blocks. Given a global assertion p, we define $s_V, s_B, s_F, h_V, h_B \models p$ by induction on p in the following table 3.

TABLE 3. SEMANTICS OF GLOBAL ASSERTION

$\mathbf{true} \triangleq \langle \mathbf{true}_V, \mathbf{true}_B \rangle$; $\mathbf{emp} \triangleq \langle \mathbf{emp}_V, \mathbf{emp}_B \rangle$;
$s_V, s_B, s_F, h_V, h_B \models \langle \alpha, \beta \rangle$ iff $s_V, s_F, h_V \models \alpha$ and $s_V, s_B, s_F, h_B \models \beta$;
$s_V, s_B, s_F, h_V, h_B \models \neg p$ iff $s_V, s_B, s_F, h_V, h_B \not\models p$;
$s_V, s_B, s_F, h_V, h_B \models p_1 \vee p_2$ iff $s_V, s_B, s_F, h_V, h_B \models p_1$ or $s_V, s_B, s_F, h_V, h_B \models p_2$;
$s_V, s_B, s_F, h_V, h_B \models p_1 \to p_2$ iff if $s_V, s_B, s_F, h_V, h_B \models p_1$ then $s_V, s_B, s_F, h_V, h_B \models p_2$;
$s_V, s_B, s_F, h_V, h_B \models \forall x.p$ iff $s_V[n/x], s_B, s_F, h_V, h_B \models p$ for any $n \in$ Loc;
$s_V, s_B, s_F, h_V, h_B \models \forall b.p$ iff $s_V, s_B[n/b], s_F, h_V, h_B \models p$ for any $n \in$ BLoc;
$s_V, s_B, s_F, h_V, h_B \models \forall f.p$ iff $s_V, s_B, s_F[\bar{n}/f], h_V, h_B \models p$ for any $\bar{n} = (n_1, n_2, \ldots, n_k) \in$ BLoc*;
$s_V, s_B, s_F, h_V, h_B \models \mathbf{emp}$ iff $\mathrm{dom}(h_V) = \emptyset$ and $\mathrm{dom}(h_B) = \emptyset$;
$s_V, s_B, s_F, h_V, h_B \models p_1 * p_2$ iff there exists h_H^1, h_H^2 with $h_H^1 \# h_H^2$ and $h_H = h_H^1 * h_H^2$ such that $s_V, s_B, s_F, h_V^1, h_B^1 \models p_1$ and $s_V, s_B, s_F, h_V^2, h_B^2 \models p_2$,
where h_H^i range over h_V^i and h_B^i and $i = 1, 2$;
$s_V, s_B, s_F, h_V, h_B \models (p_1 \mathrel{-\!\!*} p_2)$ iff for any block heap $h_H{}'$, if $h_H{}' \# h_H$ and $s_V, s_B, s_F, h_V{}', h_B{}' \models p_1$, then $s_V, s_B, s_F, h_V * h_V{}', h_B * h_B{}' \models p_2$,
where h_H range over h_V and h_B.

There are some other laws about global assertions which will be stated in another paper.

4. The Specification Language for CSSs

Specifications for CSSs can be similarly built up with those of Separation Logic. But we restrict the pre- and post-conditions to be global assertions only. Also we will only discuss partial correctness here. Formally, a specification is of the form $\{p\}\ c\ \{q\}$, where p and q are global assertion. The semantics of specifications for CSSs is similar to that for Hoare Logic or Separation Logic.

4.1. Transfer Function

According to syntax, any Boolean expression can not be any kind of assertion. Hence it cannot appear in any specification. Thus Boolean expressions in **if** or **while** commands cannot be used directly in pre- or post-conditions. To fix this problem, we define the transfer function T that maps Boolean expressions to global assertions.

$T \in$ Transfer functions = Boolean expressions $\rightharpoonup$ Global assertions

$$T(e_1 = e_2) = \langle e_1 = e_2, \mathbf{true}_B \rangle$$
$$T(e_1 \leq e_2) = \langle e_1 \leq e_2, \mathbf{true}_B \rangle$$
$$T(bk_1 == bk_2) = \langle \mathbf{true}_V, bk_1 == bk_2 \rangle$$
$$T(bk_1 <== bk_2) = \langle \mathbf{true}_V, bk_1 <== bk_2 \rangle$$
$$T(\mathbf{true}) = \mathbf{true} \quad T(\mathbf{false}) = \mathbf{false} \quad T(\neg be) = \neg T(be)$$
$$T(be_1 \wedge be_2) = T(be_1) \wedge T(be_2)$$
$$T(be_1 \vee be_2) = T(be_1) \vee T(be_2)$$

Now we are able to propose specification rules. Generally, a rule consists of some premises and a conclusion. As in most kinds of logic, we distinguish such rules that have no premises and call them axioms.

4.2. Axioms

We propose one or more axioms for each basic command:

Axioms of skip and variable assignment are similar to corresponding axioms in Hoare Logic.

$$\{p\}\ \mathbf{skip}\ \{p\} \tag{A1}$$

$$\{\langle x = x' \wedge \mathbf{emp}_V, \beta \rangle\}\ x := e\ \{\langle x = e[x'/x] \wedge \mathbf{emp}_V, \beta \rangle\} \tag{A2}$$

Variable allocation and lookup will change Stores_V, thus may change the truth value of the variable assertion. Here we confine that the variable changed does not appear in the block assertion.

$$\{\langle x = x' \wedge \mathbf{emp}_V, \beta \rangle\}\ x := \mathbf{cons}(e_1, ..., e_n)\ \{\langle x \mapsto e_1[x'/x], e_2[x'/x], ..., e_n[x'/x], \beta \rangle\} \tag{A3}$$

where x does not appear in β.

$$\{\langle x = x' \wedge e \mapsto x'', \beta \rangle\}\ x := [e]\ \{\langle x = x'' \wedge e[x'/x] \mapsto x'', \beta \rangle\} \tag{A4}$$

where x does not appear in β.

Variable mutation and deallocation only have impact on heap_V, so the block assertion part will not change.

$$\{\langle e \mapsto -, \beta \rangle\}\ [e] := e'\ \{\langle e \mapsto e', \beta \rangle\} \tag{A5}$$

$$\{\langle e \mapsto -, \beta \rangle\}\ \mathbf{dispose}(e)\ \{\langle \mathbf{emp}_V, \beta \rangle\} \tag{A6}$$

For the block commands, as block expressions will never appear in variable assertions, so only the block assertion part will change.

$$\{\langle \alpha, b == b' \wedge \mathbf{emp}_B \rangle\}\ b := bk\ \{\langle \alpha, b == bk[b'/b] \wedge \mathbf{emp}_B \rangle\} \tag{A7}$$

$$\{\langle \alpha, bk \mapsto - \rangle\}\ \{bk\} := bk'\ \{\langle \alpha, bk \mapsto bk' \rangle\} \tag{A8}$$

$$\{\langle \alpha, bk \mapsto - \rangle\}\ \mathbf{delete}\ bk\ \{\langle \alpha, \mathbf{emp}_B \rangle\} \tag{A9}$$

$$\{\langle \alpha, b == b' \wedge bk \mapsto b'' \rangle\}\ b := \{bk\} \quad \{\langle \alpha, b == b'' \wedge bk[b'/b] \mapsto b'' \rangle\} \tag{A10}$$

In most time, we use block lookup command to find the content of a file block, so we add this axiom of block lookup in special case to make the verification more powerful.

$$\{\langle \#f_2 = i-1, f_1 \rightsquigarrow f_2 \cdot bk \cdot f_3 \rangle\}\ b := \{f_1(i)\} \quad \{\langle \#f_2 = i-1, f_1 \rightsquigarrow f_2 \cdot bk \cdot f_3 \wedge b == bk \rangle\} \tag{A11}$$

where b does not appear in bk.

Commands of file will change Stores_F, thus may have impact on both variable assertions and block assertions. To make axioms simple, we only discuss the situation when the file changed in commands does not appear in the variable assertion.

$$\{\langle \alpha, f = f' \wedge \mathbf{emp}_B \rangle\}\ f := \mathbf{create}(bk_1, ..., bk_n) \quad \{\langle \alpha[f'/f], f \rightsquigarrow bk_1[f'/f] \cdot ... \cdot bk_n[f'/f] \rangle\} \tag{A12}$$

$$\{\langle \alpha, f = f' \wedge f \rightsquigarrow fe \rangle\}\ f := \mathbf{append}(bk_1, ..., bk_n) \quad \{\langle \alpha[f'/f], f \rightsquigarrow fe \cdot bk_1[f'/f] \cdot ... \cdot bk_n[f'/f] \rangle\} \tag{A13}$$

$$\{\langle \alpha, f \rightsquigarrow - \rangle\}\ \mathbf{delete}\ f\ \{\langle \alpha, \mathbf{emp}_B \rangle\} \tag{A14}$$

where f does not appear in α.

$$\{\langle \alpha, f_1 \rightsquigarrow - \wedge f_2 \centerdot bk^* \rightsquigarrow fe \cdot (bk_1, ..., bk_n) \rangle\}\ f_1 := f_2 \centerdot bk^* \quad \{\langle \alpha, f_1 \rightsquigarrow fe \cdot (bk_1, ..., bk_n) \rangle\} \tag{A15}$$

where f does not appear in α.

4.3. Rules

Rule of composition applies to sequentially executed programs.

$$\frac{\{p\}\ C\ \{q\} \quad \{q\}\ C'\ \{r\}}{\{p\}\ C; C'\ \{r\}} \tag{R1}$$

Conditional rule states that a postcondition common to **then** and **else** part is also a postcondition of the whole **if** statement.

$$\frac{\{p \wedge T(be)\}C\{q\} \quad \{p \wedge \neg T(be)\}C'\{q\}}{\{p\}\ \mathbf{if}\ be\ \mathbf{then}\ C\ \mathbf{else}\ C'\ \{q\}} \tag{R2}$$

While rule states that the loop invariant is preserved by the loop body.

$$\frac{\{p \wedge T(be)\}C'\{p\}}{\{p\}\ \mathbf{while}\ be\ \mathbf{do}\ C'\ \{p \wedge \neg T(be)\}} \tag{R3}$$

Consequence rule means we can strengthen the precondition or weaken the postcondition.

$$\frac{\models p' \rightarrow p \quad \{p\}C\{q\} \quad \models q \rightarrow q'}{\{p'\}C\{q'\}} \tag{R4}$$

Similar to Auxiliary Variable Elimination rule of Separation Logic, we have the following Auxiliary Variable Elimination rules.

$$\frac{\{p\}C\{q\}}{\{\exists x_S.p\}C\{\exists x_S.q\}}\ x_S \notin Y_S \tag{R5}$$

where $x_S \in Var_S$ and x_S range over x_V, x_B, and x_F.

Frame rule is also extended to Global Assertions.

$$\frac{\{p\}C\{q\}}{\{p * r\}C\{q * r\}}\ \text{Modify}_S(C) \cap \text{FV}(r) = \emptyset \tag{R6}$$

where $\text{Modify}_S(C)$ range over $\text{Modify}_V(C)$, $\text{Modify}_B(C)$, and $\text{Modify}_F(C)$.

4.4. Example: A Proof of APPEND Function

Now, we will demonstrate how to use these axioms and rules by a simple example that proves the correctness of append function.

```
function Append (f_1, f_2)
i := 1;
while i <= #f_1 do
bc := {f_1(i)};
f_2 := append(bc);
i := i + 1;
end while
end function
```

After the executing of function Append, the content of f_2 should be it's initial content followed by the content of f_1, or formally:

$$\{\exists f_3 \langle \#f_3 = \#f_1, f_1 \rightsquigarrow f_3 * f_2 \rightsquigarrow f_4 \rangle\}$$
$$Append();$$
$$\{\exists f_3 \langle \#f_3 = \#f_1, f_1 \rightsquigarrow f_3 * f_2 \rightsquigarrow f_4 \cdot f_3 \rangle\}$$

where f_1 is the file appended to f_2, f_3 is the content of f_1, f_4 is the initial content of f_2.

Here is the proof.

①$\{\exists f_3 \langle \#f_3 = \#f_1, f_1 \rightsquigarrow f_3 * f_2 \rightsquigarrow f_4 \rangle\}$ (Given)
②$\{\exists f_3 \langle 1 \leq \#f_1 + 1 \wedge \#\mathbf{nil} = 0 \wedge \#f_3 = \#f_1, f_1 \rightsquigarrow \mathbf{nil} \cdot f_3 * f_2 \rightsquigarrow f_4 \cdot \mathbf{nil} \rangle\}$ (①)
③$\{\exists f_5 \exists f_6 \langle 1 \leq \#f_1 + 1 \wedge \#f_5 = 0 \wedge \#f_6 = \#f_1, f_1 \rightsquigarrow f_5 \cdot f_6 * f_2 \rightsquigarrow f_4 \cdot f_5 \rangle\}$ (②)
④$\{\exists f_5 \exists f_6 \langle 1 \leq \#f_1 + 1 \wedge \#f_5 = 0 \wedge \#f_6 = \#f_1, f_1 \rightsquigarrow f_5 \cdot f_6 * f_2 \rightsquigarrow f_4 \cdot f_5 \rangle\}$
$i := 1;$
$\{\exists f_5 \exists f_6 \langle i \leq \#f_1 + 1 \wedge \#f_5 = i - 1 \wedge \#f_6 = \#f_1 - i + 1, f_1 \rightsquigarrow f_5 \cdot f_6 * f_2 \rightsquigarrow f_4 \cdot f_5 \rangle\}$ (③ A2)

We are proving the correctness of the function step by step. So after the first command $i := 1;$, we get a complicated condition from the initial condition. The next

command is a while-loop. To use the rule of while-loop(R3), we must find the loop invariant first.

Let $A \equiv \exists f_5 \exists f_6 \langle i \leq \#f_1 + 1 \wedge \#f_5 = i - 1 \wedge \#f_6 = \#f_1 - i + 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 \cdot f_5 \rangle$,
$C \equiv bc := \{f_1(i)\}; f_2 := \mathbf{append}(bc); i := i + 1$,
$be \equiv i \leq \#f_1$,

Here f_5 means the content copied, f_6 means the rest content. We will prove A is the loop invariant, i.e. $\{A \wedge T(be)\}\ C\ \{A\}$.

⑤$A \wedge T(be) = \{\exists f_5 \exists f_6 \langle i \leq \#f_1 \wedge \#f_5 = i - 1 \wedge \#f_6 = \#f_1 - i + 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 \cdot f_5 \rangle\}$

⑥$\{\exists f_5 \exists f_7 \exists bk \langle i \leq \#f_1 \wedge \#f_5 = i - 1 \wedge \#f_7 = \#f_1 - i + 1 - 1, f_1 \leadsto f_5 \cdot bk \cdot f_7 * f_2 \leadsto f_4 \cdot f_5 \rangle\}$ (⑤)

⑦$\{\exists f_5 \exists f_7 \exists bk \langle i \leq \#f_1 \wedge \#f_5 = i - 1 \wedge \#f_7 = \#f_1 - i + 1 - 1, f_1 \leadsto f_5 \cdot bk \cdot f_7 * f_2 \leadsto f_4 \cdot f_5 \rangle\}$
$bc := \{f_1(i)\}$;
$\{\exists f_5 \exists f_7 \exists bk \langle i \leq \#f_1 \wedge \#f_5 = i - 1 \wedge \#f_7 = \#f_1 - i + 1 - 1, f_1 \leadsto f_5 \cdot bk \cdot f_7 * f_2 \leadsto f_4 \cdot f_5 \wedge bc = bk \rangle\}$ (⑥ A11)

⑧$\{\exists f_5 \exists f_7 \exists bk \langle i \leq \#f_1 \wedge \#f_5 = i - 1 \wedge \#f_7 = \#f_1 - i + 1 - 1, f_1 \leadsto f_5 \cdot bk \cdot f_7 * f_2 \leadsto f_4 \cdot f_5 \wedge bc = bk \rangle\}$
$f_2 = append(bc)$;
$\{\exists f_5 \exists f_7 \exists bk \langle i \leq \#f_1 \wedge \#f_5 = i - 1 \wedge \#f_7 = \#f_1 - i + 1 - 1, f_1 \leadsto f_5 \cdot bk \cdot f_7 * f_2 \leadsto f_4 \cdot f_5 \cdot bk \rangle\}$ (⑦ A13)

⑨$\{\exists f_8 \exists f_7 \langle i \leq \#f_1 \wedge \#f_8 = i - 1 + 1 \wedge \#f_7 = \#f_1 - i + 1 - 1, f_1 \leadsto f_8 \cdot f_7 * f_2 \leadsto f_4 \cdot f_8 \rangle\}$ (⑧)

⑩$\{\exists f_5 \exists f_6 \langle i \leq \#f_1 \wedge \#f_5 = i - 1 + 1 \wedge \#f_6 = \#f_1 - i + 1 - 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 \cdot f_5 \rangle\}$ (⑨)

⑪$\{\exists f_5 \exists f_6 \langle i \leq \#f_1 \wedge \#f_5 = i - 1 + 1 \wedge \#f_6 = \#f_1 - i + 1 - 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 \cdot f_5 \rangle\}$
$i := i + 1$;
$\{\exists f_5 \exists f_6 \langle i \leq \#f_1 + 1 \wedge \#f_5 = i - 1 \wedge \#f_6 = \#f_1 - i + 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 \cdot f_5 \rangle\}$ (⑩ A2)

Let $f_6 = bk \cdot f_7$, we can get ⑥ from ⑤. And let $f_8 = f_5 \cdot bk$, we can get ⑨ from ⑧. Exchange f_7 and f_8 in ⑨ to f_6 and f_5, we can get ⑩.

And we notice that $\{\exists f_5 \exists f_6 \langle i \leq \#f_1 + 1 \wedge \#f_5 = i - 1 \wedge \#f_6 = \#f_1 - i + 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 \cdot f_5 \rangle\}$ in ⑪ is A itself. So we prove that $\{A \wedge T(be)\}\ C\ \{A\}$. By the rule of while-loop(R3), we get $\{A\}$ **while** be **do** C $\{A \wedge \neg T(be)\}$.

⑫$A \wedge \neg T(be) = \exists f_5 \exists f_6 \langle i = \#f_1 + 1 \wedge \#f_5 = i - 1 \wedge \#f_6 = \#f_1 - i + 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 * f_5 \rangle$

⑬$\{\exists f_5 \exists f_6 \langle i \leq \#f_1 + 1 \wedge \#f_5 = i - 1 \wedge \#f_6 = \#f_1 - i + 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 * f_5 \rangle\}$ (⑫)
while be do C;
$\{\exists f_5 \exists f_6 \langle i = \#f_1 + 1 \wedge \#f_5 = i - 1 \wedge \#f_6 = \#f_1 - i + 1, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 * f_5 \rangle\}$ (⑤-⑪ R3)

From the while-loop we know when it finishes, $i = \#f_1 + 1$. We can get following conclusions from ⑬.

⑭$\{\exists f_5 \exists f_6 \langle i = \#f_1 + 1 \wedge \#f_5 = \#f_1 \wedge \#f_6 = 0, f_1 \leadsto f_5 \cdot f_6 * f_2 \leadsto f_4 * f_5 \rangle\}$ (⑬)

⑮$\{\exists f_5 \langle \#f_5 = \#f_1, f_1 \leadsto f_5 * f_2 \leadsto f_4 * f_5 \rangle\}$ (⑭)

⑯$\{\exists f_3 \langle \#f_3 = \#f_1, f_1 \leadsto f_3 * f_2 \leadsto f_4 * f_3 \rangle\}$ (⑮)

With all ①-⑯ and the rule of composition(R1), we proved the correctness of append function.

5. Conclusion

In the paper, we have extended Separation Logic to verify the correctness of management programs in CSSs. The imperative language of Separation Logic is incorporated with block operations. Assertions in Separation Logic are extended to contain quantifiers over block variables and file variables. Hoare-style rules are proposed to reason about the CSSs. Using these methods, the partial correctness of cloud storage management can be verified.

Future work will be done alone the following directions: (1) Consider more characteristics of CSSs, such as parallelism, (key,value) pairs and the relations between blocks and locations. (2) Investigate the expressiveness, decidability and model checking algorithms of assertions. (3) Work out the expressiveness and completeness of the specification rules above.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grants 61170299, 61370053 and 61572003.

References

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, *Provable data possession at untrusted stores*, Proceedings of the 14th acm conference on computer and communications security, 2007, pp. 598–609.

[2] G. Ateniese, R.D Pietro, L.V Mancini, and G. Tsudik, *Scalable and efficient provable data possession*, Proceedings of the 4th international conference on security and privacy in communication netowrks, 2008, pp. 9:1–9:10.

[3] Josh Berdine, Cristiano Calcagno, and Peter W O'Hearn, *Symbolic execution with separation logic*, Aplas, 2005, pp. 52–68.

[4] Dino Distefano and Matthew J Parkinson J, *jstar: Towards practical verification for java*, Acm sigplan notices, 2008, pp. 213–226.

[5] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis, *Deny-guarantee reasoning*, In esop09: European symposium on programming, volume 5502 of lncs, 2009.

[6] Apache Software Foundation, *Apache hadoop distributed file system*, 2016. (accessed 16.05.10).

[7] P. Gardner, G. Ntzik, and A. Wright, *Local reasoning for the posix file system*, Proceedings of the 23rd european symposium on programming, 2014, pp. 169–188.

[8] S. Ghemawat, H. Gobioff, and S. Leung, *The google file system*, Proceedings of the 19th acm symposium on operating systems principles, 2003, pp. 29–43.

[9] Yang H, *Local reasoning for stateful programs*, Ph.D. Thesis, 2001.

[10] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan, *The rise of big data on cloud computing: Review and open research issues*, Information Systems **47** (2015), 98–115.

[11] W.H Hesselink and M.I Lal, *Formalizing a hierarchical file system*, Formal Aspects of Computing **24** (2012), no. 1, 27–44.

[12] C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **12** (October 1969), no. 10, 576–580.

[13] Samin S Ishtiaq and Peter W O'hearn, *Bi as an assertion language for mutable data structures*, ACM SIGPLAN Notices **36** (2001), no. 3, 14–26.

[14] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer, *Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning*, ACM SIGPLAN Notices **50** (2015), no. 1, 637–650.

[15] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal, *A relational model of types-and-effects in higher-order concurrent separation logic.*, Popl, 2017, pp. 218–231.

[16] G. Ntzik and P. Gardner, *Reasoning about the posix file system: Local update and global pathnames*, SIGPLAN Not. **50** (October 2015), no. 10, 201–220.

[17] I. Pereverzeva, L Laibinis, E. Troubitsyna, M. Holmberg, and M. Pöri, *Formal modelling of resilient data storage in cloud*, Proceedings of the 15th international conference on formal engineering methods, 2013, pp. 363–379.

[18] J.C Reynolds, *Separation logic: a logic for shared mutable data structures*, Proceedings of the 17th annual ieee symposium on logic in computer science, 2002, pp. 55–74.

[19] V. Serbanescu, F. Pop, V. Cristea, and G. Antoniu, *A formal method for rule analysis and validation in distributed data aggregation service*, World Wide Web **18** (2015), no. 6, 1717–1736.

[20] Vlad Serbanescu, Florin Pop, Valentin Cristea, and Gabriel Antoniu, *Architecture of distributed data aggregation service*, Advanced information networking and applications (aina), 2014 ieee 28th international conference on, 2014, pp. 727–734.

[21] J.J Stephen, S. Savvides, R. Seidel, and P. Eugster, *Program analysis for secure big data processing*, Proceedings of the 29th acm/ieee international conference on automated software engineering, 2014, pp. 277–288.

[22] Kasper Svendsen and Lars Birkedal, *Impredicative concurrent abstract predicates.*, Esop, 2014, pp. 149–168.

[23] C. Wang, Q. Wang, K. Ren, N. Cao, and W. Lou, *Toward secure and dependable storage services in cloud computing*, IEEE Transactions on Services Computing **5** (2012April), no. 2, 220–232.

[24] C. Wang, Q. Wang, K. Ren, and W. Lou, *Privacy-preserving public auditing for data storage security in cloud computing*, Proceedings of the 30th ieee international conference on computer communications, 2010March, pp. 1–9.

[25] G. Winskel, *The formal semantics of programming languages: an introduction*, MIT press, Cambridge, 1993.