2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Revisiting Concurrent Separation Logic and Operational Semantics

Pedro Soares
LIACC
Universidade do Porto
Porto, PT
ptcsoares@fc.up.pt

António Ravara
CITI & DI-FCT
Universidade NOVA de Lisboa
Lisboa, PT
aravara@fct.unl.pt

Simão Melo de Sousa
LIACC & DI-FE
Universidade da Beira Interior
Covilhã, PT
desousa@di.ubi.pt

***Abstract*—We present a new soundness proof of Concurrent Separation Logic (CSL) based on a structural operational semantics (SOS). We build on two previous proofs and develop new auxiliary notions to achieve the goal. One uses a denotational semantics (based on traces). The other is based on SOS, but was obtained only for a fragment of the logic — the Disjoint CSL — which disallows modifying shared variables between concurrent threads. In this work, we lift such restriction, proving the soundness of full CSL with respect to a SOS. Thus contributing to the development of tools able of ensuring the correctness of realistic concurrent programs. Moreover, given that we used SOS, such tools can be well-integrated in programming environments and even incorporated in compilers.**

***Keywords*-operational semantics; concurrency; soundness**

I. Introduction

The aim of this work is to present a new soundness proof for Concurrent Separation Logic [5], with respect to a structural operational semantics [9]. This work adapts and extends the results presented by Brookes [3] and by Vafeiadis [13].

The axiomatic verification of programs goes back to Hoare Logic [4]. This seminal work introduces two key ideas, i) the specification of programs by means of what is known by a Hoare triple: $\{P\}C\{Q\}$, where P and Q are first order formulae, called the precondition and postcondition respectively, and C is an imperative program; ii) a deductive proof system to ensure the partial correctness of programs. A program is partially correct, if every execution of C from a state respecting the precondition does not abort and when it terminates the postcondition holds for its final state. The state for this logic is formed only by the store, i.e. a function that records the value of each variable. Hoare's work gave rise to numerous deductive systems, for instance the Owicki-Gries method ([7], [8]) and Separation Logic ([6], [11]).

The Owicki-Gries method is one of the first attempts to give a resource sensitive proof system for concurrent programs. To do this, Owicki and Gries augmented the programming language with i) parallel composition, $C \parallel C$; ii) local resources, $\mathsf{resource}\ r\ \mathsf{in}\ C$; and iii) a critical region, $\mathsf{with}\ r\ \mathsf{when}\ B\ \mathsf{do}\ C$, where r denotes a resource. Each resource has a mutual exclusion lock, an invariant condition, and a set of protected variables.

The execution of parallel composition non-deterministically chooses one of the commands to execute. As usual, the execution is assumed to be weakly fair, i.e. if a command is continually available, then it will be eventually selected for execution. The resource command declares a local variable r to be used during the execution of C. The critical region command waits for the availability of the resource r, and when B holds, it acquires r and starts the execution of C, after the execution of C terminates the resource r is made available.

The programs derivable by the Owicki-Gries method have to preserve the resource invariants when the resource is available, and respect the protection of variables by resources, i.e. a program needs to acquire all resources protecting a variable, before the program can change that variable. The parallel rule proposed by Owicki [7] requires that every variable occurring in the derivation proof of one command is not be changed by another command, except for variables protected by a resource such that the variable only appear inside the critical region's proof. Thus, the Owicki-Gries method is not compositional.

Separation Logic (SL) supports reasoning about imperative programs with shared mutable data and consequently about dynamical data structures, such as lists and trees. In order to do this, the assertion and program languages used by Hoare had to be augmented. The assertions are extended with the constructs $\mathtt{emp}$, the empty memory; $e \mapsto e'$, a single memory cell e with the value e'; and $P * Q$, two disjoint memory's parts such that one satisfies P and the other satisfies Q. In this settings, the memory is represented by the heap — a partial function from the set of locations to the set of values. The store and the heap together define the state of a program.

The programing language is augmented with commands for memory manipulation. Naturally, the proof system is also extended with a rule for each new command and with a frame rule, used to enlarge the portion of memory considered in the conditions of a specification. This rule is crucial to achieve local reasoning: program specifications only need to consider the relevant memory for their execution. Therefore, this local reasoning mechanism can be used to establish the partial correctness of disjoint concurrent programs, i.e. concurrent programs which do not change shared variables.

In order to prove the soundness of the frame rule, and thus of local reasoning, it is sufficient to ensure the validity of two key properties: safety monotonicity and frame property. Safety monotonicity states that if a program does not abort for

1066-6192/15 $31.00 © 2015 IEEE
DOI 10.1109/PDP.2015.85

a given memory portion, then the program does not abort for any memory portion that contains the initial one. The frame property says that if a program does not abort for a given memory portion, then every execution on a larger memory corresponds to an execution on the initial memory.

Since the introducing of SL, different authors adapted it to the verification of concurrent programs. Vafeiadis and Parkinson introduced RGSep, combining SL with Rely/Guarantee reasoning [14]. Reddy and Reynolds introduced a syntactic control of interference in SL [10], borrowing ideas from works on fractional permissions [1]. O'Hearn proposed Concurrent Separation Logic (CSL), combining SL with the Owicki-Gries method [5]. Brookes formalized CSL, extending the traditional Hoare triples with a resource context Γ and a rely-set A, what leads to specifications of the form $\Gamma \models_A \{P\}C\{Q\}$. A resource context records the invariant and the protected variables of each resource. A rely-set consists of all variables relevant for its derivation tree. This set ensures that CSL is a compositional proof method, proved sound with respect to a denotational semantics based on traces, where a program state is represented by a store, a heap and sets of resources, expressing resource ownership [3]. Actually, the rely-set was introduced after Wehrman and Berdine discovered a counter-example to the initial version of CSL [2], and it is analogous to the set of variables used by Owicki and Gries to check non-interference in their parallel rule.

Alternatively, Vafeiadis proposed a structural operational semantics (SOS) for concurrent programs synchronizing via resources, and proved the soundness of a part of CSL, the Disjoint CSL (DCSL) [13]. DCSL differs from CSL in the side condition of the parallel rule. The side condition in DCSL forbids the modification of variables that appears in other specifications, however the side condition in CSL, only, forbids the modification of variables in the other rely-sets.

Our motivation for this work was to remove the disjointness condition and obtain a soundness proof using a SOS for the full CSL (Section VI). The goal is relevant because CSL has been adopted as the basis for most modern program logic, and it is a step in the development of more expressive provers well integrated in software development environments and compilers. Not only it allows proving correct concurrent programs handling shared resources, but also provides techniques to equip compilers with mechanisms of detecting data-races.

Concretely, the contributions of this work are the following:

1) A novel notion of environment transition that simulates actions made by other threads. We define it taking into account the rely-set, available resources and their invariants (Section V-A). This relation is crucial to study the soundness of the parallel rule.
2) The resource configuration that expresses ownership. It is defined by three sets: owned resources, locked resources, and available resources (Section IV-A). In our work, a program state is formed by a store, a heap and a resource configuration.
3) An illustrative example that we prove correct in CSL, showing the proof system's expressiveness (Section II).

II. Motivating example

To clarify that DCSL is not as expressive as CSL, we present below an example of parallel operations over a stack that cannot be proved correct in former but can be in the latter.

Let us specify the operation of *push* and *pop* in a stack. The stack is represented by the resource st in the following way

$$st(\{\mathsf{z},\mathsf{y}\}) : stack(\mathsf{z}),$$

where $\{\mathsf{z},\mathsf{y}\}$ is the protected variables, and $stack(\mathsf{z})$ is
$(\mathsf{z} = null \wedge \mathtt{emp}) \vee (\exists_{a,b} \mathsf{z} \mapsto a, b * stack(b))$.

The operations *pop* and *push* over a stack are defined below.

$$pop() := \mathsf{with}\ st\ \mathsf{when}\ \neg(\mathsf{z} = null)\ \mathsf{do}\ (\ \mathsf{y{:=}z};\ \mathsf{x1{:=}y};\ \mathsf{z{:=}[y+1]};\ \mathsf{disp(y+1)})$$
$$push(\mathsf{x}) := \mathsf{with}\ st\ \mathsf{do}\ (\mathsf{y{:=}cons(x,z)}\ ;\ \mathsf{z{:=}y})$$

The operation *pop* picks the first node of a non-empty stack and passes it to the variable $x1$. In the following specification, the program performs a pop over a shared stack and it disposes the memory space retrieved by the stack.

$$st(\mathsf{z},\mathsf{y}) : stack(z) \vdash \{\mathtt{emp}\}pop()\ ;\ \mathsf{disp(x1)}\{\mathtt{emp}\}. \quad (1)$$

To prove this result in DCSL, we use the rules of SL and the critical region rule, that is presented in the next section for CSL by omitting the rely-set. Consider the following derivation, that proves the validity of the program inside the critical region.

$$\begin{array}{ll} \vdash & \{\mathtt{emp} * \exists_{a,b}\ \mathsf{z} \mapsto a, b * stack(b)\} \\ & \mathsf{y{:=}z}; \\ & \{\mathtt{emp} * \exists_{a,b}\ \mathsf{y} \mapsto a, b * stack(b)\} \\ & \mathsf{x1{:=}y}; \\ & \{\mathtt{emp} * \exists_{a,b}\ \mathsf{x1} \mapsto a * \mathsf{y{+}1} \mapsto b * stack(b))\} \\ & \mathsf{z{:=}[y+1]}; \\ & \{\mathtt{emp} * \exists_{a}\ \mathsf{x1} \mapsto a * \mathsf{y{+}1} \mapsto \mathsf{z} * stack(\mathsf{z}))\} \\ & \mathsf{disp(y+1)} \\ & \{(\exists_a\ \mathsf{x1} \mapsto a) * (stack(\mathsf{z}))\}. \end{array}$$

Applying critical region rule, we obtain,

$$st(\mathsf{z},\mathsf{y}) : stack(\mathsf{z}) \vdash \{\mathtt{emp}\}pop()\{\ \exists_a\ \mathsf{x1} \mapsto a\}.$$

By the SL's rules, sequential and deallocation, we have (1).

Now, we turn our attention to the *push* operator over a stack, showing that the following specification is valid in the context of DCSL. Let *push* insert an element $x2$ in the top of a stack.

$$st(\mathsf{z},\mathsf{y}) : stack(\mathsf{z}) \vdash \{\mathtt{emp}\}push(\mathsf{x2})\{\mathtt{emp}\}.$$

As before, from SL inference rules, we obtain the specification below. Then we can apply the critical region rule to obtain the specification above.

$$\begin{array}{ll} \vdash & \{\mathtt{emp} * stack(\mathsf{z})\} \\ & \mathsf{y{:=}cons(x2,z)} \\ & \{\mathsf{y} \mapsto \mathsf{x2}, \mathsf{z} * stack(\mathsf{z})\} \\ & \{\mathtt{emp} * \exists_{a,b}\ \mathsf{y} \mapsto a, b * stack(b)\} \\ & \mathsf{z{:=}y} \\ & \{\mathtt{emp} * \exists_{a,b}\ \mathsf{z} \mapsto a, b * stack(b)\} \end{array}$$

$$
\begin{array}{rcl}
e & := & x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \\
B & := & \texttt{true} \mid \texttt{false} \mid e_1 = e_2 \mid e_1 < e_2 \mid B_1 \wedge B_2 \mid \neg B \\
P & := & B \mid \neg P \mid P_1 \wedge P_2 \mid \forall x P \mid \texttt{emp} \mid e \mapsto e' \mid P_1 * P_2
\end{array}
$$

Fig. 1. Syntax of the Assertion Language

Until now we shown that each specification is derivable in DCSL; now we want to study their parallel composition. To apply the parallel rule we need that the variables modified by one program cannot occur free in the other specification. $mod(pop()\,;\mathsf{disp(x1)}) = \{\mathsf{x1}, \mathsf{y}\}, mod(push(\mathsf{x2})) = \{\mathsf{x2}, \mathsf{z}\}$.

The variables z and y are used in both specifications. Hence it is not possible to apply the DCSL parallel rule and obtain a specification for the parallel execution of *pop* and *push*.

In order to express the specification above in the context of CSL it is necessary to define the rely-set for the operation of *pop* and *push*, that are $\{\mathsf{x1}\}$ and $\{\mathsf{x2}\}$, respectively.

It is straightforward, using the derivations above, to infer, in CSL, the following specifications:

$$st(\mathsf{z},\mathsf{y}) : stack(\mathsf{z}) \vdash_{\{\mathsf{x1}\}} \{\texttt{emp}\} pop()\,;\mathsf{disp(x1)}\{\texttt{emp}\}$$

$$st(\mathsf{z},\mathsf{y}) : stack(\mathsf{z}) \vdash_{\{\mathsf{x2}\}} \{\texttt{emp}\} push(\mathsf{x2})\{\texttt{emp}\}.$$

To apply the CSL parallel rule, we need to check that there is no interference between rely-sets and modified variables. Since $mod(push(\mathsf{x2})) \cap \{\mathsf{x1}\} = \emptyset$ and $mod(pop()\,;\mathsf{disp(x1)}) \cap \{\mathsf{x2}\} = \emptyset$, by parallel rule we infer:

$$st(\mathsf{z},\mathsf{y}) : stack(\mathsf{z}) \vdash_{\{\mathsf{x1},\mathsf{x2}\}}$$
$$\{\texttt{emp}\}(pop()\,;\mathsf{disp(x1)}) \parallel push(\mathsf{x2})\{\texttt{emp}\}.$$

As this example shows we can prove correctness of more programs using CSL than DCSL. In the next section, we shall overview CSL and present some of its inference rules (a complete presentation is in the work of Brookes [3]).

III. Concurrent Separation Logic

We revisit Concurrent Separation Logic (CSL), as presented by Brookes [3]. First, we define the assertion language, then the syntax of commands for concurrent programs, and finally the inference rules for CSL.

A. Assertion Language

Consider a set **Var** of variables, ranged over by $\mathsf{x}, \mathsf{y}, \ldots$, and a set **Val** of values, that includes the integers and the value $null$. These meta-variables may be indexed or primed.

The grammar in Fig. 1 defines the syntax of the assertion language. We assume the usual definitions of free variables of an assertion (FV), and of (partial) substitution of a (free) variable in an assertion.

We use the definition of SL for the *validity* of an assertion with respect to the pair (s, h), where s and h are denoted by *storage* and *heap*, respectively, and given by the functions:

$$s : \mathbf{Var} \to \mathbf{Val}, \quad h : \mathbf{Loc} \rightharpoonup \mathbf{Val},$$

where $\mathbf{Loc} \subset \mathbb{N}$ is the set of current locations. For an assertion P, we write $s, h \models P$ if the assertion is valid for the pair (s, h), and we write $\models P$ if $s, h \models P$ for every pair (s, h).

For a given heap and storage, the precise assertions uniquely determine the subheap that verifies it.

Definition 1. *We say that an assertion P is* precise *if for every pair (s, h) there is at most one subheap h' of h such that $s, h' \models P$.*

The *resource context* Γ is used to represent a shared state. The resource context Γ has the form

$$r_1(X_1) : R_1, r_2(X_2) : R_2, \ldots, r_n(X_n) : R_n, \qquad (2)$$

where r_i are distinct resources names, R_i are assertions and $X_i \subseteq \mathbf{Var}$ such that $FV(R_i) \subseteq X_i$, for each $i = 1, 2, \ldots, n$. The set of variables is called protected variables and each assertion represents a resource invariant. As usual, every assertion used in a resource context must be precise. The restriction is necessary as showed by Reynolds in an illuminating example [5, Section 11], because CSL has the conjunction rule.

Let $Res(\Gamma)$ denote the set of resources names appearing in Γ, ranged over by r_i. Furthermore, let $PV(\Gamma)$ denote the set of all variables protected by resources in Γ, and $PV(r_i)$ denote the set of variables protected by r_i.

B. Programming Language

The language includes the basic commands to manipulate storage and heap:

$$c := \mathsf{x{:=}e} \mid \mathsf{x{:=}[e]} \mid \mathsf{[e]{:=}e'} \mid \mathsf{x{:=}cons}(\overline{\mathsf{e}}) \mid \mathsf{disp(e)}.$$

The basic commands use the notation of SL. The bracket parenthesis denotes an accesses to a heap location. There are also commands to allocate new locations of memory and to free a location, respectively, $\mathsf{x{:=}cons}(\overline{\mathsf{e}})$ and $\mathsf{disp(e)}$.

The following grammar defines the syntax of the programming language. In the introduction, we gave a brief explanation for the last three commands.

$$
\begin{array}{rcl}
C & := & \mathsf{skip} \mid c \mid C_1\,;C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\
 & & \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{resource}\ r\ \mathsf{in}\ C \\
 & & \mathsf{with}\ r\ \mathsf{when}\ B\ \mathsf{do}\ C \mid C_1 \parallel C_2.
\end{array}
$$

The set of variables modified by a program C, $mod(C)$, consists of all variables x such that the program C has one of the following commands: $\mathsf{x{:=}e}$, $\mathsf{x{:=}[e]}$ or $\mathsf{x{:=}cons(e)}$.

C. Inference rules

In this section, we present the most relevant inference rules for CSL as stated by Brookes [3]. First, we define what is a well-formed specification in CSL.

Definition 2. *Let Γ be a resource context, A a set of variables, P, Q assertions and C a command. The* specification of a program *has the form*

$$\Gamma \vdash_A \{P\}C\{Q\}.$$

Moreover we say that the specification of the program is well-formed*, if $FV(P, Q) \subseteq A$ and $FV(C) \subseteq A \cup PV(\Gamma)$.*

$$\frac{}{\Gamma \vdash_A \{P\}\mathsf{skip}\{P\}}\ (\text{SKP}) \quad \frac{mod(c) \cap PV(\Gamma) = \emptyset \quad \vdash^{SL} \{P\}c\{Q\}}{\Gamma \vdash_A \{P\}c\{Q\}}\ (\text{BC}) \quad \frac{\Gamma \vdash_{A_1} \{P_1\}C_1\{P_2\} \quad \Gamma \vdash_{A_2} \{P_2\}C_2\{P_3\}}{\Gamma \vdash_{A_1 \cup A_2} \{P_1\}C_1\,;C_2\{P_3\}}\ (\text{SEQ})$$

$$\frac{\Gamma \vdash_A \{P\}C\{Q\} \quad mod(C) \cap FV(R) = \emptyset}{\Gamma \vdash_{A \cup FV(R)} \{P * R\}C\{Q * R\}}\ (\text{FRA}) \quad \frac{\Gamma \vdash_{A_1} \{P_1\}C_1\{Q_1\} \quad \Gamma \vdash_{A_2} \{P_2\}C_2\{Q_2\} \quad (*)}{\Gamma \vdash_{A_1 \cup A_2} \{P_1 * P_2\}C_1 \parallel C_2\{Q_1 * Q_2\}}\ (\text{PAR})$$

$$\frac{\Gamma \vdash_{A \cup X} \{(P \wedge B) * R\}C\{Q * R\}}{\Gamma, r(X) : R \vdash_A \{P\}\mathsf{with}\ r\ \mathsf{when}\ B\ \mathsf{do}\ C\{Q\}}\ (\text{CR}) \quad \frac{\Gamma, r(X) : R \vdash_A \{P\}C\{Q\}}{\Gamma \vdash_{A \cup X} \{P * R\}\mathsf{resource}\ r\ \mathsf{in}\ C\{Q * R\}}\ (\text{RES})$$

Fig. 2. Rules of the Inference System

In Fig. 2, we present some inference rules of CSL. The inference rules are only applied for well-formed specifications. The specifications derivable by SL are denoted by $\vdash^{SL}$.

The rule for basic commands are inherited from SL by adding the rely-set and imposing that protected variables are not modified. The sequential and frame rules are very similar to the respective rules of SL, but the rely-set needs to take into account the rely-set of both program or the variables of the framed assertion.

In the critical region rule, if the command inside the critical region preserves the invariant, when B is initial respected, then the resource context can be expanded by r. Note that the rely-set does not need to include all protected variables, however the well-formedness of the specification must be preserved. In the local resource rule, we are able to take out a resource from the assumption's resource context to the conclusion's local condition. The parallel rule has the side condition below that restricts the interference between programs.

$$mod(C_1) \cap A_2 = mod(C_2) \cap A_1 = \emptyset. \qquad (*)$$

In order to obtain the inference rules of DCSL we erase the rely-set from the CSL inference rules and change the side condition in the parallel rule to the following condition:

$$mod(C_1) \cap FV(C_2, P_2, Q_2) = mod(C_2) \cap FV(C_1, P_1, Q_1) = \emptyset.$$

Note that it is obvious that every valid specification in DCSL is also valid in CSL, by taking $A_i = FV(P_i, C_i, Q_i)$, $i = 1, 2$.

IV. Operational Semantics

In this section, we describe a structural operational semantics (SOS) that we use to prove the soundness of CSL. We mostly follow the approach of Vafeiadis [13].

A. Program transition

We start by extending the programming language with a command for executions inside a critical region. We denote this command by $\mathsf{within}\ r\ \mathsf{do}\ C$, where r is an acquired resource and C is the command in execution. In the extended programming language, we can associate to each command a set of locked resources, $Locked(C)$. This set is composed by all resource r such that there is a subcommand $\mathsf{within}\ r\ \mathsf{do}\ C$ which is not a subcommand of $\mathsf{resource}\ r\ \mathsf{in}\ C$.

Let O, L, D be disjoint pairwise subsets of resources names, we say that $\rho = (O, L, D)$ is a *resource configuration*. The set O represents the resources owned by the running program, the set L represents the resources locked by others programs and the set D represents the resources available. We write $r \in \rho$ ($r \notin \rho$) if $r \in (O \cup L \cup D)$ ($r \notin (O \cup L \cup D)$, respectively).

Usually the state of a machine in SL consists of a storage, s, and a heap, h. However, we define a program's state by a triple (s, h, ρ). The program transitions, that define the SOS, are represented by the relation $\rightarrow_p$ defined from the tuple $(C, (s, h, \rho))$ to $(C', (s', h', \rho'))$ or the abort state (abort), where C, C' are commands and $(s, h, \rho), (s', h', \rho')$ are states.

For a basic command c we denote by $[c](s, h)$ the result of executing c for the pair (s, h), in the context of SL. The result of execution c to a pair (s, h) can be a pair (s', h') or abort. An excerpt of the program transitions are in Fig. 3. The full set of rules for SOS is in the technical report [12].

Since most of the program transition are standard, we only emphasize how we manage the resource configuration. First note that it is not changed by any transition of basic commands (BCT). The acquisition of a resource by the transition $(W0)$ requires that the resource is available and transfers it to the set of owned resources; the release of a resource made by $(W2)$ returns the resource to the set of available resources. The local resource command does not add the resource to the resource configuration, since that would break locality, i.e., the local resource should only be visible to who created it. For the local resource we use the set of Locked resources, $Locked(C)$, to determine if a resource should be in the set of owned or available resources. In the next section, we prove that $Locked(C)$ is equal to the set of owned resources along an execution starting in a non-extended command.

In Fig. 4, we include transitions that abort. As in SL, a memory fault cause the program to abort. The parallel command goes to abort if one of its commands aborts. The local resource command leads to a abort situation if the command tries to create a pre-existing resource. The critical region command goes to abort if it tries to acquire an undeclared resource, if the execution inside the critical region aborts, or if an acquired resource is not in the set of owned resources.

B. Properties of program transitions

We state now the main properties of the program transitions (the proofs can be found in the technical report [12]). We start with the safety monotonicity and the frame properties that are essential to show the soundness of the frame rule, as well as

$$\frac{[c](s,h) = (s'h')}{c,(s,h,\rho) \to_p \mathsf{skip},(s',h',\rho)}\ (BCT) \qquad \frac{}{\mathsf{skip};C_2,(s,h,\rho) \to_p C_2,(s,h,\rho)}\ (S1)$$

$$\frac{C_1,(s,h,\rho) \to_p C_1',(s',h',\rho')}{C_1\,;C_2,(s,h,\rho) \to_p C_1'\,;C_2,(s',h',\rho')}\ (S2) \qquad \frac{C_1,(s,h,\rho) \to_p C_1',(s',h',\rho')}{C_1 \parallel C_2,(s,h,\rho) \to_p C_1' \parallel C_2,(s',h',\rho')}\ (P1)$$

$$\frac{C_2,(s,h,\rho) \to_p C_2',(s',h',\rho')}{C_1 \parallel C_2,(s,h,\rho) \to_p C_1 \parallel C_2',(s',h',\rho')}\ (P2) \qquad \frac{}{\mathsf{skip}\|\mathsf{skip},(s,h,\rho) \to_p \mathsf{skip},(s,h,\rho)}\ (P3)$$

$$\frac{C,(s,h,(O\cup\{r\},L,D)) \to_p C',(s',h',\rho') \quad r \notin \rho = (O,L,D) \quad r \in Locked(C)}{\mathsf{resource}\ r\ \mathsf{in}\ C,(s,h,\rho) \to_p \mathsf{resource}\ r\ \mathsf{in}\ C',(s',h',\rho' \setminus \{r\})}\ (R1)$$

$$\frac{C,(s,h,(O,L,D\cup\{r\})) \to_p C',(s',h',\rho') \quad r \notin \rho = (O,L,D) \quad r \notin Locked(C)}{\mathsf{resource}\ r\ \mathsf{in}\ C,(s,h,\rho) \to_p \mathsf{resource}\ r\ \mathsf{in}\ C',(s',h',\rho' \setminus \{r\})}\ (R2)$$

$$\frac{r \notin \rho}{\mathsf{resource}\ r\ \mathsf{in}\ \mathsf{skip},(s,h,\rho) \to_p \mathsf{skip},(s,h,\rho)}\ (R0) \qquad \frac{\rho = (O\cup\{r\},L,D) \quad \rho' = (O,L,D\cup\{r\})}{\mathsf{within}\ r\ \mathsf{do}\ \mathsf{skip},(s,h,\rho) \to_p \mathsf{skip},(s,h,\rho')}\ (W2)$$

$$\frac{r \in O \quad C,(s,h,(O\setminus\{r\},L,D)) \to_p C',(s',h',(O',L',D'))}{\mathsf{within}\ r\ \mathsf{do}\ C,(s,h,(O,L,D)) \to_p \mathsf{within}\ r\ \mathsf{do}\ C',(s',h',(O'\cup\{r\},L',D'))}\ (W1)$$

$$\frac{\rho = (O,L,D\cup\{r\}) \quad \rho' = (O\cup\{r\},L,D) \quad s(B) = \mathtt{true}}{\mathsf{with}\ r\ \mathsf{when}\ B\ \mathsf{do}\ C,(s,h,\rho) \to_p \mathsf{within}\ r\ \mathsf{do}\ C,(s,h,\rho')}\ (W0)$$

Fig. 3. Program Transitions

$$\frac{[c](s,h) = \mathsf{abort}}{c,(s,h,\rho) \to_p \mathsf{abort}}\ (BCA) \qquad \frac{C_1,(s,h,\rho) \to_p \mathsf{abort}}{C_1 \parallel C_2,(s,h,\rho) \to_p \mathsf{abort}}\ (PA1) \qquad \frac{C_2,(s,h,\rho) \to_p \mathsf{abort}}{C_1 \parallel C_2,(s,h,\rho) \to_p \mathsf{abort}}\ (PA2)$$

$$\frac{r \in \rho}{\mathsf{resource}\ r\ \mathsf{in}\ C,(s,h,\rho) \to_p \mathsf{abort}}\ (RA) \qquad \frac{r \notin \rho}{\mathsf{with}\ r\ \mathsf{when}\ B\ \mathsf{do}\ C,(s,h,\rho) \to_p \mathsf{abort}}\ (WA)$$

$$\frac{r \notin O}{\mathsf{within}\ r\ \mathsf{do}\ C,(s,h,(O,L,D)) \to_p \mathsf{abort}}\ (WA2) \qquad \frac{C,(s,h,\rho\setminus\{r\}) \to_p \mathsf{abort}}{\mathsf{within}\ r\ \mathsf{do}\ C,(s,h,\rho) \to_p \mathsf{abort}}\ (WA1)$$

Fig. 4. Abort Transitions

of the parallel rule.

Let h, g be heaps. If they have disjoint domains we write $h \perp g$, and we denote by $h \uplus g$ the union of disjoint heaps.

Proposition 1. *Let C be a command and let (s,h,ρ) be a state. Suppose h_F is a heap such that $h \perp h_F$.*

If $C,(s,h,\rho) \not\to_p$ abort then $C,(s,h \uplus h_F,\rho) \not\to_p$ abort.

Proposition 2. *Let $(s,h,\rho),(s',h',\rho')$ be states and let C, C' be commands. Suppose h_F is a heap such that $h \perp h_F$. If $C,(s,h,\rho') \not\to_p$ abort and $C,(s,h \uplus h_F,\rho) \to_p C',(s',h',\rho')$, then h_F is a subheap of h' and*

$$C,(s,h,\rho) \to_p C',(s',h' \setminus h_F,\rho').$$

By safety monotonicity and frame property we know that the execution of parallel commands only affects his own heap. We state dual properties for the resource configuration.

Proposition 3. *Let C be a command, (s,h) be a SL's state and $(O_1 \cup O_2, L, D),(O_1, L \cup O_2, D)$ be resource configurations. If $C,(s,h,(O_1 \cup O_2, L, D)) \not\to_p$ abort, then*

$$C,(s,h,(O_1 \cup O_2, L, D)) \not\to_p \mathsf{abort}.$$

Proposition 4. *Let C, C' be commands, let $(s,h),(s',h')$ be SL's states and let ρ_1, ρ_2, ρ' be resource configurations.*

Suppose that $\rho' = (O', L, D')$, $\rho_1 = (O_1 \cup O_2, L, D)$ and $\rho_2 = (O_1, L \cup O_2, D)$. If $C,(s,h,\rho_2) \not\to_p$ abort and $C,(s,h,\rho_1) \to_p C',(s',h',\rho')$, then $O_2 \subseteq O'$ and

$$C,(s,h,\rho_2) \to_p C',(s',h',(O' \setminus O_2, L \cup O_2, D')).$$

The previous propositions allow us to make a correspondence between the transitions in a parallel execution to transitions of its commands executed independently.

We say that a command C is *reachable* from a CSL's command C' if there are $(s,h,\rho),(s',h',\rho')$ and k such that

$$C',(s,h,\rho') \to_p^k C,(s',h',\rho')$$

and $C',(s,h,\rho') \not\to_p^j \mathsf{abort}$ for every $j \leq k$, where $\to_p^i$ denotes the composition of i transitions. In the next proposition, we see that owned resources are equal to locked resources, along an execution starting from a non-extended command.

Proposition 5. *Let C, C' be commands, let Γ be a resource context, let $(s,h,(\emptyset,\emptyset,Res(\Gamma)),(s',h',\rho')$ be states and let $k \geq 0$ such that C' is reachable from C.*

If $C, (s, h, (\emptyset, \emptyset, Res(\Gamma)) \rightarrow_p^k C', (s', h', \rho')$, *then*

$$\rho' = (Locked(C'), \emptyset, Res(\Gamma) \setminus Locked(C')).$$

The proposition above reinforces the idea that the transitions $(R1)$ and $(R2)$ are well defined. Furthermore, it completely describes the resource configuration along an execution.

V. Validity

In this section, we start by defining the validity of specifications in the SOS presented before.

Let Γ be a resource context with the form (2). For any $D = \{r_{i_1}, r_{i_2}, \ldots, r_{i_k}\} \subseteq Res(\Gamma)$, we define

$$\circledast_{r \in D} \Gamma(r) = R_{i_1} * R_{i_2} * \ldots * R_{i_k},$$

$$inv(\Gamma) = \circledast_{r \in Res(\Gamma)} \Gamma(r).$$

Definition 3. *We write* $\Gamma \models \{P\}C\{Q\}$, *if for every state* (s, h) *such that* $s, h \models P * inv(\Gamma)$, *we have that*

- $C, (s, h, (\emptyset, \emptyset, res(\Gamma)) \not\rightarrow_p^k$ **abort**, *for every* $k \geq 0$. *And*
- *If there exist a state* (s', h') *and* $k \geq 0$ *such that*

$$C, (s, h, (\emptyset, \emptyset, res(\Gamma))) \rightarrow_p^k \textbf{skip}, (s', h', (\emptyset, \emptyset, res(\Gamma))),$$

then $s', h' \models Q * inv(\Gamma)$.

We were not able to inductively prove the soundness of CSL using this notion, because we could not emulate the modification of parallel execution in all parts. The next subsections are devoted to how we overcome this difficulty. Thus we introduce the environment transition, that will be essential to spread changes made in the state by one program to other parallel programs. And we give a refined notion of validity for the SOS extended with the environment transition, which we call safety. We finish this section by seeing that safety implies validity.

A. Environment transition

In order to define the environment transition, we define the environment transformation respecting a set of variables. This transformation modifies the storage and the resource configuration, afterwards the environment transition combines this transformations with modification in the shared heap.

Definition 4. *Let* $(s, h, (O, L, D)), (s', h', (O', L', D'))$ *be states and* $A \subseteq$ ***Var***. *We say that* $(s, h, (O, L, D))$ *is transformed by the environment to* $(s', h, (O, L', D'))$ *respecting* A *and we write* $(s, h, (O, L, D)) \overset{A}{\rightsquigarrow} (s', h, (O, L', D'))$ *if* $s(x) = s'(x)$, *for every* $x \in A$, *and* $L' \cup D' = L \cup D$.

Note that the environment transformation preserves the local heap and the owned resources, since other programs cannot change them. Furthermore, the relation defined by the environment transformation, $\overset{A}{\rightsquigarrow}$, is an equivalence relation and it is order reversing with respect to A. In the next proposition, we state this properties.

Proposition 6. *Let* $A', A \subseteq$ ***Var***. *The relation* $\overset{A}{\rightsquigarrow}$ *is an equivalence relation. If* $A' \subseteq A$ *and* $(s, h, \rho) \overset{A}{\rightsquigarrow} (s', h', \rho')$, *then* $(s, h, \rho) \overset{A'}{\rightsquigarrow} (s', h', \rho')$.

We denote the environment transition by $\xrightarrow{A,\Gamma}_e$, it is a relation between $(C, (s, h \uplus h_G, \rho))$ and $(C, (s', h \uplus h'_G, \rho'))$, where C is a command and $(s, h \uplus h_G, \rho)$, $(s', h \uplus h'_G, \rho')$ are states, and it is defined by the rule below. Consider the set $A' = A \cup \bigcup_{r \in Locked(C)} PV(r)$. If $(s, h, \rho) \overset{A'}{\rightsquigarrow} (s', h, \rho')$, $s, h_G \models \circledast_{r \in D} \Gamma(r)$ and $s', h'_G \models \circledast_{r \in D'} \Gamma(r)$, then

$$\frac{}{C, (s, h \uplus h_G, \rho) \xrightarrow{A,\Gamma}_e C, (s', h \uplus h'_G, \rho')} \ (E).$$

As noted before the environment transition is used to simulate modification done by other program. The environment transition can be used to: change the storage, except for variables in the rely-set A or variables protected by an acquired resources; interchange locked resources and available resource; and modify the available shared heap.

We extend the transitions on the SOS with the environment transition, and we define the relation $\xrightarrow{A,\Gamma}$ from $(C, (s, h, \rho))$ to $(C', (s', h', \rho'))$ or to **abort**, where C, C' are commands and $(s, h, \rho), (s', h', \rho')$ are states. This relation is given by

$$\xrightarrow{A,\Gamma} = \rightarrow_p \cup \xrightarrow{A,\Gamma}_e .$$

B. Safety

For a command C, we associate the set of variables passive to be *changed* by C in the next transition, and we denote it by $chng(C)$. This set consists of all variables x such that C can perform a transition using x:=e; x:=[e] or x:=cons(e).

The next definition expresses the safety of program with respect to a state in the extended operational semantics, for the next n transitions. We include some additional properties that will be useful to prove the soundness of CSL.

Definition 5. *Let* C *be a command, let* (s, h, ρ) *be a state, let* Γ *be a resource context, let* Q *be an assertion and* $A \subseteq$ ***Var***. *We say that* $Safe_0(C, s, h, \rho, \Gamma, Q, A)$ *is always valid, and* $Safe_{n+1}(C, s, h, \rho, \Gamma, Q, A)$ *is valid if:*

(i) *If* $C =$ **skip**, *then* $s, h \models Q$;

(ii) $C, (s, h, \rho) \not\rightarrow_p$ **abort**;

(iii) $chng(C) \cap \bigcup_{r \in L \cup D} PV(r) = \emptyset$;

(iv) *For every* h_G *such that* $h \perp h_G$, $s, h_G \models \circledast_{r \in D} \Gamma(r)$ *and*

$$C, (s, h \uplus h_G, \rho) \xrightarrow{A,\Gamma} C', (s', \hat{h}, \rho'),$$

then there exist h' *and* h'_G *such that* $\hat{h} = h' \uplus h'_G$, $Safe_n(C', s', h', \rho', \Gamma, Q, A)$ *is valid and*

$$s', h'_G \models \circledast_{r \in D'} \Gamma(r).$$

The property (i) states that if the execution terminates, then Q is respected. In the property (ii), we ensure that the next transition of C does not abort for the state (s, h, ρ). The property (iii) guarantees that the next transition of C does not change variables protected by resources not owned. In

the final condition (iv), we require that the available shared state is preserved after every transition and that the posterior transitions respects this conditions.

In the next theorem, we see that if a program is safe for every number of transitions and for every state that respects the pre-condition, then the correspondent specification is valid with respect to the SOS. The theorem is proved by induction on the number of program's transitions.

Theorem 1. *Let C be a command, let P, Q be assertions, let Γ be a resource context and $A \subseteq$ **Var**. If for every state (s, h) and $n \geq 0$ such that $s, h \models P$, we have that $Safe_n(C, s, h, (\emptyset, \emptyset, Res(\Gamma)), \Gamma, Q, A)$ is valid, then*

$$\Gamma \models \{P\}C\{Q\}.$$

In order to prove the soundness of CSL, by the theorem above, is sufficient to show that every derivable specification on CSL implies safety, a result we prove in the next section.

VI. Soundness

We sketch here the soundness of CSL with respect to the SOS. First, we state the main result of this work, the soundness of CSL. Next we present an intermediate theorem that, together with the Theorem 1, proves the main result. The intermediate theorem says that every derivable specification in CSL is safety in the extended operational semantics.

Theorem 2. *If $\Gamma \vdash_A \{P\}C\{Q\}$, then $\Gamma \models \{P\}C\{Q\}$.*

Theorem 2 is an immediate consequence of the next theorem and Theorem 1.

Theorem 3. *Let C be a command, let P, Q be assertions, let Γ be a resource context and $A \subseteq$ **Var**. If $\Gamma \vdash_A \{P\}C\{Q\}$, then for every state (s, h) and $n \geq 0$ such that $s, h \models P$, we have that $Safe_n(C, s, h, (\emptyset, L, D), \Gamma, Q, A)$ is valid, where $L \cup D = Res(\Gamma)$.*

In the next lines, we carry out an informal proof of this theorem by studying the inference rules of CSL. The proof is carried by induction on the inference rules and we give an auxiliary result for some of the inference rules. The skip rule (SKP) has the following auxiliary proposition.

Proposition 7. *Let (s, h, ρ) be a state, let Γ be a resource context, let Q be an assertion and $A \subseteq$ **Var** such that $FV(Q) \subseteq A$. If $s, h \models Q$, then $Safe_n(\mathsf{skip}, s, h, \rho, \Gamma, Q, A)$ is valid for every $n \geq 0$.*

We prove this result by induction on n. The first properties of safety are immediate, because the state respects Q, the command skip does not abort and it does not modify protected variables. For the last property, we note that the execution of skip is composed only by environment transition. And the environment transition does not modify the variables in the rely-set neither the local heap. Therefore Q remains valid after every transition and we can apply the induction hypothesis.

In order to check the safety of basic commands rules (BC), we argue mostly as in the context of SL. For any state that respects the precondition, the execution does not abort and after the program transition (BCT) the post condition is established. We remark that (BCT) does not change the validity of shared properties, because it does not change protected variables neither the shared heap. The environment transition respects the local state, because every free variable in the conditions are in the rely-set, and it respects the shared state, by definition. Moreover if an environment transition is done before (BCT), then the precondition is preserved. If the environment transition is done after (BCT) the post condition is preserved, by the Proposition 7. In the technical report [12], we formalize this argument for each basic commands.

The soundness of the frame rule (FRA) is supported by the following proposition.

Proposition 8. *Let C be a reachable command, let Γ be a resource context, let $(s, h \uplus h_R, \rho)$ be a state, let Q, R be assertions and $A \subseteq$ **Var** such that $s, h_R \models R$. If $Safe_n(C, s, h, \rho, \Gamma, Q, A)$ is valid and $mod(C) \cap FV(R) = \emptyset$, then $Safe_n(C, s, h \uplus h_R, \rho, \Gamma, Q * R, A \cup FV(R))$ is valid.*

The proposition follows from the safety monotonicity and frame property (Propositions 1 and 2). We note that R is valid after every transition, because $FV(R)$ is not modified by the command and the rely-set includes it.

Next we study the parallel rule (PAR).

Proposition 9. *Let s be a storage, let $C_1 \parallel C_2$ be a reachable command, let h, h_1, h_2 be heaps, let ρ, ρ_1, ρ_2 be resource contexts, let Q_1, Q_2 be assertions and $A_1, A_2 \subseteq$ **Var** such that $h = h_1 \uplus h_2$, $\rho = (O_1 \cup O_2, L, D)$, $\rho_1 = (O_1, L \cup O_2, D)$, $\rho_2 = (O_2, L \cup O_1, D)$ and $FV(Q_i) \subseteq A_i$, for $i = 1, 2$.*

Suppose that $A_1 \cap mod(C_2) = A_2 \cap mod(C_1) = \emptyset$.

If $Safe_n(C_i, s, h_i, \rho_i, \Gamma, Q_i, A_i)$ is valid for $i = 1, 2$, then $Safe_n(C_1 \parallel C_2, s, h, \rho, \Gamma, Q_1 * Q_2, A_1 \cup A_2)$ is valid.

As before we prove this result by induction on n. The firsts three properties of safety are immediate from the safety of C_1 and C_2, and Propositions 1 and 3.

In order to apply the induction step we use the environment transition. If the parallel execution transits by a program transition, then there are three cases. First case, a transition is done by C_1. We perform the same transition on C_1 (by Propositions 2 and 4) and an environment transition on C_2, that replicates the changes performed by the program transition. This environment transition exists because the variables modified by the program C_1 are different from the rely set A_2. In the second case, a transition is done by C_2, and we do analogous transitions. The third case is the joint of parallel commands. In this case, we do a reflexive environment transition on C_1 and C_2. If the program transits by an environment transition, then we perform the same environment transition on C_1 and C_2. This environment transition can be used because the rely-set of $C_1 \parallel C_2$ includes the rely-set of each command. Therefore we can apply the inductive hypothesis and obtain the proposition.

The safety of the critical region rule follows from the safety inside the critical region. Because any environment transition performed before the critical region does not break

the precondition's validity and when a program enters a critical region its invariant is valid and B is respected. Therefore the next result establishes safety for the critical region rule.

Proposition 10. *Let C be a reachable command, let (s, h, ρ) be a state, let Γ be a resource context, let Q be an assertion and $A \subseteq$ **Var**. Suppose that $\Gamma' = \Gamma, r(X) : R$ is a resource context, $\rho = (O, L, D)$, $r \in O$ and $FV(Q) \subseteq A$.*

*If $Safe_n(C, s, h, \rho \setminus \{r\}, \Gamma, Q * R, A \cup X)$ is valid, then $Safe_n($**within** r **do** $C, s, h, \rho, \Gamma', Q, A)$ is valid.*

The safety inside the critical region is a consequence of the safety of C with the rely-set extended by X and without the resource r, because their execution are equivalents. Note that when the execution inside the critical region terminates the resource invariant is valid. Hence the shared state respects the global properties, when the resource is made available.

In the proposition below, we give properties for the local resource when the resource is available or locked. In the work of Vafeiadis [13], there is a similar proposition in the context of DCSL. Using its second property, we can prove the soundness of the local resource rule.

Proposition 11. *Let C be a reachable command, let (s, h, ρ) be a state, let Γ be a resource context, let Q be an assertion and $A \subseteq$ **Var**. Suppose that $\Gamma' = \Gamma, r(X) : R$ is a resource context, $\rho = (O, L, D)$, $r \notin \rho$ and $FV(Q) \subseteq A$. We have the following statements:*

- *Suppose that $r \in Locked(C)$.*
  *If $Safe_n(C, s, h, (O \cup \{r\}, L, D), \Gamma', Q, A)$ is valid, then $Safe_n($**resource** r **in** $C, s, h, \rho, \Gamma, Q * R, A \cup X)$ is valid.*
- *Assume that exists h_R such that $h_R \perp h$ and $s, h_R \models R$. Suppose that $r \notin Locked(C)$.*
  *If $Safe_n(C, s, h, (O, L, D \cup \{r\}), \Gamma', Q, A)$ is valid, then $Safe_n($**resource** r **in** $C, s, h \uplus h_R, \rho, \Gamma, Q * R, A \cup X)$ is valid.*

This proposition is proved by induction on both properties in the following way: first we prove that both properties are true when $n = 0$; then we assume that both properties are true for $n \geq 0$ and prove that each property is true for $n + 1$.

The program transitions inside the local resource have an equivalent program transition for the command C, except for the transition $(R0)$. In those cases we apply one of the inductive step depending on resource's ownership. For the case $(R0)$, we note that the execution inside the local resource had terminated and the invariant R is respected. If the local resource transits by an environment transition, then there is an equivalent environment transition in C.

In the technical report [12], we formalize the arguments presented before and we prove the soundness of all CSL [3].

VII. Conclusion

This work presents a proof of correctness of CSL based on SOS, the first we are aware of. We build on two previous proofs, one for the full logic, using a denotational semantics based on traces, and another for a fragment of CSL, the DCSL. The formalization of the results presented in a theorem prover is a naturally extension to the present work.

A proof based on SOS is important, as this form of semantics closer mimics the execution of an imperative program. Therefore, it paves the way to the development of more expressive proving tools that are able to deal with truly concurrent programs manipulating shared resources. Our work may also provide insight on how to develop provably correct compilers able of detecting data-races.

Our aim was lifting the (severe) restriction of forcing concurrent threads to manipulate only disjoint sets of variables, since it does not allow proving correct many interesting and useful programs. To attain this goal, we re-used the notion of "rely-set" , a notion crucial to obtain the soundness result of CSL with respect to the denotational semantics. The adaptation was not trivial and required developing several auxiliary notions, but established a proof technique that may now be used in other contexts.

Acknowledgment

This work was partially funded by Fundação para a Ciência e Tecnologia through AVIACC project, grant PTDC/EIA-CCO/117590, and CITI/FCT/UNL, grant Pest-OE/EEI/UI0527/2014.

References

[1] J. Boyland (2003): *Checking Interference with Fractional Permissions*. In: *SAS, Lecture Notes in Computer Science* 2694, Springer, pp. 55–72.
[2] S. Brookes (2007): *A semantics for concurrent separation logic*. *Theoretical Computer Science* 375(1-3), pp. 227–270.
[3] S. Brookes (2011): *A Revisionist History of Concurrent Separation Logic*. *ENTCS* 276, pp. 5–28.
[4] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Communications of the ACM* 12(10), pp. 576–580.
[5] P. W. O'Hearn (2007): *Resources, concurrency, and local reasoning*. *Theoretical Computer Science* 375(1-3), pp. 271–307.
[6] P. W. O'Hearn, J. C. Reynolds & H. Yang (2001): *Local Reasoning about Programs that Alter Data Structures*. In: *CSL, Lecture Notes in Computer Science* 2142, Springer, pp. 1–19.
[7] S. S. Owicki (1976): *A Consistent and Complete Deductive System for the Verification of Parallel Programs*. In: *STOC*, ACM, pp. 73–86.
[8] S. S. Owicki & D. Gries (1976): *Verifying Properties of Parallel Programs: An Axiomatic Approach*. *Communications of the ACM* 19(5), pp. 279–285.
[9] G. D. Plotkin (2004): *A structural approach to operational semantics*. *Journal of Logic and Algebraic Programming* 60–61, pp. 17–139.
[10] U. S. Reddy & J. C. Reynolds (2012): *Syntactic control of interference for separation logic*. In: *POPL*, ACM, pp. 323–336.
[11] J. C. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures*. In: *LICS*, IEEE Computer Society, pp. 55–74.
[12] P. Soares, A. Ravara & S. Melo de Sousa (2014): *An Operational Semantics for Concurrent Separation Logic*. Technical Report RR-DCC-2014-11, Department of Computer Science, Faculty of Science, University of Porto. Available at http://www.dcc.fc.up.pt/dcc/Pubs/TReports/TR14/dcc-2014-11.pdf.
[13] V. Vafeiadis (2011): *Concurrent Separation Logic and Operational Semantics*. *ENTCS* 276, pp. 335–351.
[14] V. Vafeiadis & M. J. Parkinson (2007): *A Marriage of Rely/Guarantee and Separation Logic*. In: *CONCUR, Lecture Notes in Computer Science* 4703, Springer, pp. 256–271.