TLP **18** (2): 252–295, 2018. © Cambridge University Press 2018
doi:10.1017/S1471068418000017 First published online 10 April 2018

*A concurrent constraint programming interpretation of access permissions**

CARLOS OLARTE
ECT, Universidade Federal do Rio Grande do Norte, Natal-RN, Brazil
(*e-mail:* `carlos.olarte@gmail.com`)

ELAINE PIMENTEL
DMAT, Universidade Federal do Rio Grande do Norte, Natal-RN, Brazil
(*e-mail:* `elaine.pimentel@gmail.com`)

CAMILO RUEDA
DECC, Pontificia Universidad Javeriana Cali, Valle del Cauca, Colombia
(*e-mail:* `camilo.rueda@gmail.com`)

submitted 1 December 2016; revised 14 February 2018; accepted 21 February 2018

Abstract

A recent trend in object-oriented programming languages is the use of access permissions (APs) as an abstraction for controlling concurrent executions of programs. The use of AP source code annotations defines a protocol specifying how object references can access the mutable state of objects. Although the use of APs simplifies the task of writing concurrent code, an unsystematic use of them can lead to subtle problems. This paper presents a declarative interpretation of APs as linear concurrent constraint programs (`lcc`). We represent APs as constraints (i.e., formulas in logic) in an underlying constraint system whose entailment relation models the transformation rules of APs. Moreover, we use processes in `lcc` to model the dependencies imposed by APs, thus allowing the faithful representation of their flow in the program. We verify relevant properties about AP programs by taking advantage of the interpretation of `lcc` processes as formulas in Girard's intuitionistic linear logic (ILL). Properties include deadlock detection, program correctness (whether programs adhere to their AP specifications or not), and the ability of methods to run concurrently. By relying on a focusing discipline for ILL, we provide a complexity measure for proofs of the above-mentioned properties. The effectiveness of our verification techniques is demonstrated by implementing the Alcove tool that includes an animator and a verifier. The former executes the `lcc` model, observing the flow of APs, and quickly finding inconsistencies of the APs vis-à-vis the implementation. The latter is an automatic theorem prover based on ILL.

KEYWORDS: access permissions, concurrent constraint programming, linear logic, focusing

* This work has been partially supported by CNPq, CAPES/Colciencias/INRIA's project STIC AmSud and the project FWF START Y544-N23.

1 Introduction

Reasoning about concurrent programs is much harder than reasoning about sequential ones. Programmers often find themselves overwhelmed by the many subtle cases of thread interactions they must be aware of to decide whether a concurrent program is correct or not. In order to ensure program reliability, the programmer needs also to figure out the right level of thread atomicity to avoid race conditions, cope with mutual exclusion requirements, and guarantee deadlock freeness.

All these problems are aggravated when software designers write programs using an object-oriented (OO) language and use OO strategies to design their programs. In an OO program, objects can have multiple references (called aliases) that can modify local content concurrently. This significantly increases the complexity of the design of sound concurrent programs. For instance, data race conditions arise when two object references read and write concurrently from/to an object memory location. To cope with data races, one could simply place each object access within an atomic block, but this would affect negatively program performance. A better strategy could be to lock just the objects that are shared among threads. However, it then becomes hard to estimate which objects should be shared and which locations should be protected by locks just by looking at the program text.

Languages like Æminium (Stork *et al.* 2009), Plaid (Sunshine *et al.* 2011), and Mezzo (Pottier and Protzenko 2013) propose a strategy to design sound and reliable concurrent programs based on the concept of *access permissions* (APs) (Boyland *et al.* 2001). APs are abstractions about the aliased access to an object content and they are annotated in the source code. They permit a direct control of the access to the mutable state of an object. Making explicit the access to a shared mutable state facilitates verification and enables parallelization of code. For instance, a *unique* AP, describing the case when only one reference to a given object exists, enforces absence of interference and simplifies verification; a *shared* AP, describing the case when an object may be accessed and modified by multiple references, allows for concurrent executions but makes verification trickier.

Although APs greatly help to devise static strategies for correct concurrent sharing of objects, the interactions resulting from dynamic bindings (e.g., aliasing of variables) might still lead to subtle difficulties. Indeed, it may happen that apparently correct permission assignments in simple programs lead to deadlocks.

We propose a linear concurrent constraint programming (`lcc`) (Fages *et al.* 2001) approach for the verification of AP annotated programs. Concurrent constraint programming (`ccp`) (Saraswat *et al.* 1991; Saraswat 1993) is a simple model for concurrency that extends and subsumes both concurrent logic programming and constraint logic programming. Agents in `ccp` interact by *telling* constraints (i.e., formulas in logic) into a shared *store* of partial information and synchronize by *asking* if a given information can be deduced from the store. In `lcc`, constraints are formulas in Girard's intuitionistic linear logic (ILL) (Girard 1987) and *ask* agents are allowed to *consume* tokens of information from the store.

We interpret AP programs as `lcc` agents that *produce* and *consume* APs when evolving. We use constraints to keep information about APs, object references, object

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

fields, and method calls. Moreover, the constraint entailment relation allows us to verify compliance of methods and arguments to their AP signatures. The constraint system specifies also how the APs can be transformed during the execution of the program.

We are able to verify AP programs by exploiting the declarative view of `lcc` agents as formulas in ILL. The proposed program verification includes (1) deadlock detection; (2) whether it is possible for methods to be executed concurrently or not; and (3) whether annotations adhere to the intended semantics associated with the flow of APs or not.

The key for this successful specification and analysis of AP annotations as ILL formulas is the use of a linear logic's *focusing* discipline (Andreoli 1992). In fact, by using focusing, we can identify which actions need to interact with the environment or not, either for choosing a path to follow, or for waiting for a guard to be available (e.g., the possession of an AP on a given object). This gives a method for measuring the complexity of focused ILL (ILLF) proofs in terms of actions, hence establishing an upper bound of the complexity for verifying the above-mentioned properties. Moreover, as shown in (Olarte and Pimentel 2017), focusing guarantees that the interpretation of `lcc` processes as ILL formulas is *adequate* at the highest level (full completeness of derivations): one step of computation (in `lcc`) corresponds to one step of logical reasoning. Hence, our encodings of AP annotations as ILL formulas is faithful w.r.t. the proposed `lcc` model.

The contributions of this work are three-fold: (1) the definition of a logical semantics for APs, (2) provision of a procedure for the verification of the above-mentioned properties as well as a complexity analysis for it, and (3) the implementation of the verification approach as the Alcove tool (`http://subsell.logic.at/alcove2/`). The logical structure we impose on APs thus allows us to formally reason about the behavior of AP-based programs and give a declarative account of the meaning of these annotations. It is worth noticing that we are not considering a specific AP-based language. Instead, we give a logical meaning to the machinery of APs and type states (see Section 7) present in different languages. This allows us to provide static analyses independent of the runtime system at hand. For concreteness, we borrow the AP model of Æminium (Stork *et al.* 2009), a concurrent OO programming language based on the idea of APs.

The paper is organized as follows. Section 2 presents the syntax of the AP-based language used here and Section 3 recalls `lcc`. Section 4 presents the interpretation of AP programs as `lcc` agents. We also show how the proposed model is a runnable specification that allows observing the flow of a program's permissions. We implemented this model as the Alcove LCC Animator explained in Section 4.3. Section 5 describes our approach to program verification and its implementation as the Alcove LL prover. It also presents a complexity analysis of the proposed verification. Two compelling examples of our framework are described in Section 6: the verification of a critical zone management system and a concurrent producer–consumer system. Section 7 concludes the paper.

A preliminary short version of this paper was published in (Olarte *et al.* 2012). The present paper gives many more examples and explanations and provides precise

```
1  class stats {...} // Definition of statistics
2  class collection { // Collection of elements
3   collection() none(this) => unq(this) {...}//constructor
4   sort() unq(this) => unq(this) {...}
5   print() imm(this) => imm(this) {...}
6   compStats(stats s) imm(this),unq(s) => imm(this),unq(s)
        {...}
7   removeDuplicates() unq(this) => unq(this){...}}
8  main() {
9    let collection c, stats s in
10     c := new collection()
11     s := new stats()
12     c.sort()
13     c.print()
14     c.compStats(s)
15     c.removeDuplicates()
16   end}
```

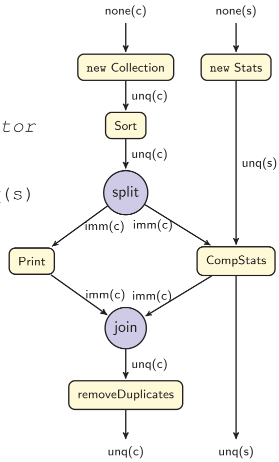

Fig. 1. Example of an AP annotated program and its permission flow graph.

technical details. In particular, in this paper, we identify the fragment of `lcc` (and ILL) required for the specification of AP programs and we show that this fragment allows for efficient verification techniques. Moreover, the language (and analyses) considered here take into account *Data Group (DG) Permissions* (Leino 1998; Stork *et al.* 2009), a powerful abstraction that adds application-level dependencies without sacrificing concurrency (see Section 2.1).

2 Access permissions in object-oriented programs

We start with an intuitive description of APs and data group access permissions (DGAPs). In Section 2.2, we give a formal account of them.

APs are abstractions describing how objects are accessed. Assume a variable x that points to the object o. A *unique* permission to the reference x guarantees that x is the sole reference to object o. A *shared* permission provides x with reading and modifying access to o, which allows other references to o (called *aliases*) to exist and to read from it or to modify it. The *immutable* permission provides x with read-only access to o, and allows any other reference to o to exist and to read from it. Let us use a simple example to explain APs and the *concurrency-by-default* behavior (Stork *et al.* 2009) they offer. Figure 1 shows a program, taken and slightly modified from (Stork *et al.* 2009), that operates over a collection of elements. Starting at line 8, the program creates an object of type *collection* at line 10 and an object of type *stats* at line 11. The program sorts the collection c at line 12, and prints it at line 13. It computes some statistics at line 14, and removes duplicates from the collection at line 15. Lines 3–7 declare the signatures for the methods. The constructor builds a unique reference to a new collection at line 3. Methods *sort* and *removeDuplicates* modify the content of the collection and they require a unique reference to it. Method *compStats* requires and returns an immutable (read-only) AP to the collection c and a unique AP to the parameter s.

Given the AP signature of these methods, the AP dataflow is computed where (1) conflicting accesses are ordered according to the lexical order of the program; and (2) non-conflicting instructions can be executed concurrently. For instance, methods in lines 12 and 13 cannot be executed concurrently (since *sort* requires a unique permission) and methods in lines 13 and 14 can be executed concurrently (since both methods require an immutable permission on c). Finally, the method in line 15 cannot be executed concurrently with *compStats* since *removeDuplicates* requires a unique AP. Hence, what we observe is that the unique permission returned by the constructor is consumed by the call of method *sort*. Once this method terminates, the unique permission can be split into two immutable permissions, and methods *print* and *compStats* can be executed concurrently. Once both methods have finished, the immutable APs are joined back into a unique AP, and the method *removeDuplicates* can be executed.

2.1 Share permissions and data groups

As we just showed, unique permissions can be split into several immutable APs to allow multiple references to *read*, simultaneously, the state of an object. Therefore, from the AP annotations, the programming language can determine, automatically, the instructions that can be executed concurrently and those that need to be executed sequentially (Figure 1).

In the case of *share* APs, several references can *modify* concurrently the state of the same object. Hence, the programmer needs additional control structures to make explicit the parts of the code that can be executed concurrently. Consider for instance the following excerpt of code:

```
1 let Subject s, Observer o1, Observer o2 in
2   s := new Subject()
3   o1 := new Observer(s) // Requires a share permission on s
4   o2 := new Observer(s)
5   s.update() // Requires a share permission on s
6   s.update()
```

where the constructor *Observer* as well as the method *update* require a share permission on s. Assume also that the intended behavior of the program is that the method *update* should be executed only after the instantiation of the *Observer* objects.

If we were to handle share permissions as we did with immutable permissions in the previous section, once the new instance of *Subject* is created in line 2, the unique permission s has on it can be split into several share permissions. Hence, statements in lines 3 to 6 could be executed concurrently. This means that a possible run of the program may execute the method *update* in lines 5 and 6 before instantiating the *Observers* in lines 3 and 4, which does not comply with the intended behavior of the program.

Higher level dependencies in AP programs can be defined by means of DGs (Leino 1998) as in (Stork *et al.* 2009; Stork *et al.* 2014). Intuitively, a DG represents a collection of objects and it controls the flow of share permissions on them. For that, two kinds of DGAPs are defined: an *atomic* permission

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

```
1  class Subject <dg>{
2     Subject() none(this) => unq(this)
3     update() shr : dg(this) => shr : dg(this)}
4  class Observer<dg>{
5     Observer(Subject<dg> s) none(this),shr:dg(s)
          => unq(this),shr:dg(s)}
6  main(){
7    group<g>
8    let Subject s, Observer o1, Observer o2 in
9     split(g){
10       s := new Subject<g>()
11       o1 := new Observer<g>(s)
12       o2 := new Observer<g>(s) }
13    split(g){
14       s.update()
15       s.update()}
16   }
```

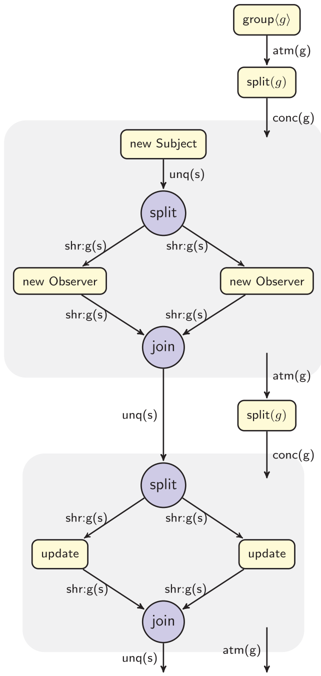

Fig. 2. Subjects and observers with data groups.

provides exclusive access to the DG, much like a unique AP for objects. Then, working on an atomic DGAP leads to the sequentialization of the code; on the contrary, a *concurrent* DGAP allows other DGAPs to coexist on the same DG. Therefore, a concurrent DGAP allows for the parallel execution of the code.

Consider the code in Figure 2, taken and slightly modified (syntactically) from (Stork *et al.* 2009). In line 1, the class *Subject* is declared with a DG parameter *dg*. This parameter is similar to a type parameter (template) in modern OO programming languages. The method *update* (line 3) requires a share permission on the DG *dg* to be invoked. The class *Observer* (line 4) is also defined with a DG parameter *dg* and its constructor requires an object of type $Subject\langle dg\rangle$.

The statement `group`$\langle g\rangle$ (line 7) creates a DG and assigns to it an atomic DGAP. The `split`(g) instruction *consumes* such atomic permission on g and splits it into three concurrent DGAP, one per each statement in the block (lines 10, 11, and 12). According to the data dependencies, statements in lines 11 and 12 can be executed concurrently once s is instantiated in line 10. When the instructions in lines 11 and 12 have terminated, the concurrent DGAPs on g are joined back to an atomic permission on g. When this happens, the `split` block in line 13 consumes such permission and splits it into two concurrent DGAPs so that statements in lines 14 and 15 can be executed concurrently. Hence, DGAPs enforce a *DG dependency* between the two blocks (in gray in the figure) of code.

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

(group decl.)	G	$::=$	$\mathtt{group}\langle g_1, \cdots, g_n\rangle$
(permissions)	p	$::=$	$\mathtt{unq} \mid \mathtt{shr} : g \mid \mathtt{imm} \mid \mathtt{none}$
(programs)	P	$::=$	$\langle \widetilde{CL}\, main\rangle$
(class decl.)	CL	$::=$	$\mathbf{class}\ c\ \langle\widetilde{g}\rangle\ \{\ F\ \widetilde{CTR}\ \widetilde{M}\ \}$
(types)	T	$::=$	$c\langle\widetilde{g}\rangle$
(field decl.)	F	$::=$	$\cdot \mid \mathbf{attr}\ \widetilde{T\,a}$
(method decl.)	CTR	$::=$	$c(\widetilde{T\,x})\ \mathtt{none}(\mathtt{this}), \widetilde{p(x)}\ \Rightarrow\ p(\mathtt{this}), \widetilde{p(x)}\ \{s\}$
	M	$::=$	$m(\widetilde{T\,x})\ p(\mathtt{this}), \widetilde{p(x)}\ \Rightarrow\ p(\mathtt{this}), \widetilde{p(x)}\ \{s\}$
(main)	$main$	$::=$	$\mathbf{main}\ \{G\ s\}$
(references)	r	$::=$	$x \mid x.a \mid \mathtt{this} \mid \mathtt{this}.a$
(righthand side)	rhs	$::=$	$r \mid null$
(statements)	s	$::=$	$\mathtt{let}\ \widetilde{T\,x}\ \mathtt{in}\ s\ \mathtt{end} \mid r\langle g\rangle := rhs \mid r.m(\widetilde{r}) \mid$ $r := \mathbf{new}\ c\langle\widetilde{g}\rangle(\widetilde{r}) \mid \mathtt{split}\langle\widetilde{g}\rangle\{\widetilde{s}\} \mid \{\widetilde{s}\}$

Fig. 3. Syntax of AP annotated programs. *c*, *m*, *a*, *x*, *g* ranges, respectively, over name of classes, methods, fields, variables, and data groups. $\tilde{x}$ denotes a possibly empty sequence of variables $x_1, \ldots, x_n$. This notation is similarly used for other syntactic categories.

2.2 An access permission-based language

This section formalizes the syntax we have used in the previous examples. Our language is based on the core calculus μÆminium (Stork *et al.* 2014) for Æminium (Stork *et al.* 2009), an OO programming language where concurrent behavior arises when methods require non-conflicting APs, as exemplified in the last section. As a core calculus, it focuses on the mechanisms to control the flow of permissions and it abstracts away from other implementation details of the language (e.g., control structures). Unlike μÆminium, our core calculus abstracts away also from (1) specific implementation details to guarantee atomicity (e.g., the `inatomic` statement used in the intermediate code of Æminium to keep track of entered atomic blocks), (2) mechanisms to define internal DGs, i.e., DGs in our language can only be declared in the main function, and (3) we do not consider class inheritance.

Programs are built from the syntax in Figure 3. DGs are declared with $\mathtt{group}\langle g_1, \cdots, g_n\rangle$. APs on objects can be unique, immutable, or none (for null references), and share on the DG g ($\mathtt{shr} : g$). A program consists of a series of class definitions ($\widetilde{CL}$) and a *main* body. A class can be parameterized in zero or more DG names ($\langle\widetilde{g}\rangle$) and it contains zero or several fields (F), constructors ($\widetilde{CTR}$), and methods ($\tilde{M}$).

A class field (attribute) is declared by using a valid type and a field name. Note that a type is simply an identifier of a class with its respective DG parameters.

Constructors and methods specify the required permissions for the caller ($p(\mathtt{this})$) and the arguments ($\widetilde{p(x)}$) as well as the permissions restored to the environment.

A reference r can be a variable, the self-reference `this` or a field selection as in $x.a$. As for the statements, we have the following:

- The `let` constructor allows us to create local variables.
- After the assignment $r\langle g\rangle := rhs$, both r and rhs point to the same object (or null) as follows: (1) if rhs has a unique permission on an object o, then after the assignment, r and rhs have a share permission $\mathtt{shr} : g$ on o. This explains the need of "$\langle g\rangle$" in the syntax. As syntactic sugar one could decree that $r_l := r_r$ means $r_l\langle\mathtt{default}\rangle := r_r$ where `default` is a predefined DG; (2) if rhs has a $\mathtt{shr} : g$ (resp

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

imm) permission on o, then r and rhs end with a shr : g (resp imm) permission on o. Finally, (4) if $rhs = null$ or it is a null reference, then r and rhs end with a none permission. We note that in assignments, the right- and left-hand sides are references. We do not lose generality since it is possible to unfold more general expressions by using local variables.

– An object's method can be invoked by using a reference to it with the right number of parameters as in $r.m(\tilde{r})$. We assume that in a call to a method (or constructor), the actual parameters are *references* (i.e., variables, including this, and attributes) and not arbitrary expressions. Since we have parameters by reference, we assume that the returned type is void and we omit it in the signature.
– A new instance of a given class is created by $r :=$ new $c\langle\tilde{g}\rangle(\tilde{r})$ where we specify the required DG parameters $\tilde{g}$ and actual parameters $(\tilde{r})$ of the constructor.
– For each g_i, split$\langle g_1, \ldots, g_n\rangle\{s_1 \cdots s_m\}$ consumes an atomic or a concurrent DGAP on g_i. Then, it splits each of such DGAPs into m concurrent DGAPs (one per each statement in the block). Once the statements in the block have finished their execution, the concurrent DGAPs created are consumed and the original DGAPs are restored.
– Finally, we can compose several statements in a block $\{s_1, \ldots, s_n\}$ where the concurrent execution of statements is allowed according to the data dependencies imposed by the APs and the DGAPs: once s_i has successfully consumed its needed permissions, the execution of s_{i+1} may start concurrently. Moreover, if s_i cannot acquire the needed permissions, it must wait until such permissions are released by the preceding statements.

Remark 2.1 (*Circular recursive definitions*)
The AP language in Figure 3 allows us to write recursive methods. However, the language lacks control structures (e.g., *if-then-else* statements) to specify base cases in recursive definitions. This language must be understood as a core language to specify the AP mechanisms and not as a complete OO programming language implementing the usual data and control structures. Hence, we shall assume that there are no circular recursive definitions in the source program. Due to the lack of control structures, this is an unavoidable restriction to guarantee termination of the analyzes in Sections 4 and 5.

Dependencies and execution. Recall that blocks of sentences are enclosed by curly brackets. Hence, we say that a sentence s occurs in a block if s is inside the braces of that block.

Definition 1 (*Conflicts and dependencies*)
Let s_i and s_j be statements that occur in the same block. We say that s_i and s_j are in *conflict* if both statements use an object o in conflicting modes, i.e., either (1) s_i or s_j require a unique permission on o, (2) s_i requires a share permission on o and s_j requires an immutable permission on o, or (3) s_i and s_j require a share permission on different DGs. Two blocks split$\langle\tilde{g}\rangle\{s_1, \ldots, s_n\}$ and split$\langle\tilde{g'}\rangle\{s'_1, \ldots, s'_m\}$ occurring in the same block are in conflict if $\tilde{g} \cap \tilde{g'} \neq \emptyset$.

The semantics of AP programs share with other semantics for OO languages (see, e.g., Igarashi *et al.* 2001) the rules and contexts to keep track of references,

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

objects, as well as lookup tables to identify class names, fields, and methods with their respective definitions. Additionally, in the case of AP programs, the semantics relies on an evaluation context to keep track of the DGs created as well as the available APs in the system. This context plays an important role in the semantic rules: a statement s is executed only if the evaluation context possesses all the permissions required by s. Moreover, in order to allow parallel executions, once s consumes the needed permissions, the next statement (in the lexical order of the program) is enabled for execution. For instance, in a block of statements $s_1, \ldots, s_n$, the execution starts by enabling s_1. Each enabled statement s_i checks whether the required permissions are available. If this is the case, such permissions are consumed, s_i starts its execution and the next statement s_{i+1} is enabled. When s_i terminates its execution, the consumed permissions are restored to the environment. On the other side, if the permissions required by s_i are not available, s_i must wait until the needed permissions are released/produced by the preceding statements. Hence, non-conflicting blocks lead to concurrent executions and conflicting blocks are sequentialized according to the lexical order of the program (as in Figures 1 and 2).

The reader may refer to Stork *et al.* 2014, Sections 3.2 and 3.3 for the semantic rules of μÆminium that can be easily adapted to the sublanguage in Figure 3. In Section 4, we give a precise definition of the needed evaluation contexts by using constraints (i.e., formulas in logic) and the state transformation by means of concurrent processes consuming and producing those constraints.

3 Linear concurrent constraint programming

ccp (Saraswat and Rinard 1990; Saraswat *et al.* 1991; Saraswat 1993) (see a survey in (Olarte *et al.* 2013)) is a model for concurrency that combines the traditional operational view of process calculi with a declarative view based on logic. This allows ccp to benefit from the large set of reasoning techniques of both process calculi and logic.

Agents in ccp *interact* with each other by *telling* and *asking* information represented as *constraints* to a global store. The type of constraints is parametric in a *constraint system* (Saraswat *et al.* 1991) that specifies the basic constraints that agents can tell and ask during execution. Such systems can be specified as Scott information systems as in (Saraswat and Rinard 1990), (Saraswat *et al.* 1991), or they can be specified in a suitable fragment of logic (see, e.g., (Fages *et al.* 2001; Nielsen *et al.* 2002)).

The basic constructs (processes) in ccp are (1) the *tell* agent c, which adds the constraint c to the store, thus making it available to the other processes. Once a constraint is added, it cannot be removed from the store (i.e., the store grows monotonically). And (2), the *ask* process $c \rightarrow P$, that queries if c can be deduced from the information in the current store; if so, the agent behaves like P, otherwise, it remains blocked until more information is added to the store. In this way, ask processes define a simple and powerful synchronization mechanism based on entailment of constraints.

$$\frac{}{c \longrightarrow c}\ \mathsf{init} \qquad \frac{}{\Gamma \longrightarrow \top}\ \top_R \qquad \frac{\Gamma \longrightarrow c}{\Gamma, 1 \longrightarrow c}\ 1_L \qquad \frac{}{\longrightarrow 1}\ 1_R$$

$$\frac{\Gamma, c_1, c_2 \longrightarrow c}{\Gamma, c_1 \otimes c_2 \longrightarrow c}\ \otimes_L \qquad \frac{\Gamma_1 \longrightarrow c_1 \ \ \Gamma_2 \longrightarrow c_2}{\Gamma_1, \Gamma_2 \longrightarrow c_1 \otimes c_2}\ \otimes_R \qquad \frac{\Gamma, c_i \longrightarrow c}{\Gamma, c_1 \& c_2 \longrightarrow c}\ \&_{Li} \qquad \frac{\Gamma \longrightarrow c_1 \ \ \Gamma \longrightarrow c_2}{\Gamma \longrightarrow c_1 \& c_2}\ \&_R$$

$$\frac{\Gamma_1 \longrightarrow c_1 \ \ \Gamma_2, c_2 \longrightarrow c}{\Gamma_1, \Gamma_2, c_1 \multimap c_2 \longrightarrow c}\ \multimap_L \qquad \frac{\Gamma, c_1 \longrightarrow c_2}{\Gamma \longrightarrow c_1 \multimap c_2}\ \multimap_R \qquad \frac{\Gamma, c_1 \longrightarrow c \ \ \Gamma, c_2 \longrightarrow c}{\Gamma, c_1 \oplus c_2 \longrightarrow c}\ \oplus_L \qquad \frac{\Gamma \longrightarrow c_i}{\Gamma \longrightarrow c_1 \oplus c_2}\ \oplus_{Ri}$$

$$\frac{\Gamma, c \longrightarrow d \ \ x \notin fv(\Gamma, d)}{\Gamma, \exists x.c \longrightarrow d}\ \exists_L \qquad \frac{\Gamma \longrightarrow c[t/x]}{\Gamma \longrightarrow \exists x.c}\ \exists_R \qquad \frac{\Gamma, c[t/x] \longrightarrow d}{\Gamma, \forall x.c \longrightarrow d}\ \forall_L \qquad \frac{\Gamma \longrightarrow c \ \ x \notin fv(\Gamma)}{\Gamma \longrightarrow \forall x.c}\ \forall_R$$

$$\frac{\Gamma \longrightarrow d}{\Gamma, !c \longrightarrow d}\ W \qquad \frac{\Gamma, !c, !c \longrightarrow d}{\Gamma, !c \longrightarrow d}\ C \qquad \frac{\Gamma, c \longrightarrow d}{\Gamma, !c \longrightarrow d}\ D \qquad \frac{!\Gamma \longrightarrow d}{!\Gamma \longrightarrow !d}\ \mathsf{prom}$$

Fig. 4. Rules for intuitionistic linear logic (ILL). $fv(c)$ (resp. $fv(\Gamma)$) denotes the set of free variables of formula c (resp. multiset Γ). Γ, Δ denote multisets of formulas.

`lcc` (Fages *et al.* 2001) is a `ccp`-based calculus that considers constraint systems built from a fragment of Girard's ILL (Girard 1987). The move to a *linear discipline* permits ask agents to *consume* information (i.e., constraints) from the store.

Definition 2 (*Linear constraint systems (Fages et al. 2001)*)
A linear constraint system is a pair $(\mathscr{C}, \vdash)$ where $\mathscr{C}$ is a set of formulas (linear constraints) built from a signature Σ (a set of function and relation symbols), a denumerable set of variables $\mathscr{V}$ and the following ILL operators: *multiplicative* conjunction ($\otimes$) and its neutral element (1), the existential quantifier ($\exists$), and the exponential bang (!). We shall use c, c', d, d', etc., to denote elements of $\mathscr{C}$. Moreover, let Δ be a set of non-logical axioms of the form $\forall \tilde{x}.[c \multimap c']$ where all free variables in c and c' are in $\tilde{x}$. We say that d *entails* c, written as $d \vdash c$, iff the sequent $!\Delta, d \longrightarrow c$ is provable in ILL (Figure 4).

The connective $\otimes$ allows us to conjoin information in the store and 1 denotes the empty store. As usual, existential quantification is used to hide information. The exponential $!c$ represents the arbitrary duplication of the resource c. The entailment $d \vdash c$ means that the information c can be deduced from the information represented by constraint d, possibly using the axioms in the theory Δ. This theory gives meaning to (uninterpreted) predicates. For instance, if R is a transitive relation, Δ may contain the axiom $\forall x, y, z.[R(x, y) \otimes R(y, z) \multimap R(x, z)]$.

We assume that "!" has a tighter binding than $\otimes$ and so, we understand $!c_1 \otimes c_2$ as $(!c_1) \otimes c_2$. For the rest of the operators, we shall explicitly use parenthesis to avoid ambiguities. Given a finite set of indexes $I = \{1, \ldots, n\}$, we shall use $\bigotimes\limits_{i \in I} F_i$ to denote the formula $F_1 \otimes \cdots \otimes F_n$.

We note that, according to Definition 2, constraints are built from the ILL fragment $\otimes$, 1, $\exists$,!. We decided to include all ILL connectives in Figure 4 (linear implication $\multimap$, additive conjunction $\&$ and disjunction $\oplus$, the universal quantifier $\forall$, and the unit $\top$), since those connectives will be used to encode `lcc` processes in Section 5.

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

3.1 The language of processes

Similar to other ccp-based calculi, lcc, in addition to tell and ask agents, provides constructs for parallel composition, hiding of variables, non-deterministic choices, and process definitions and calls. More precisely:

Definition 3 (lcc *agents (Fages et al. 2001)*)
Agents in lcc are built from constraints as follows:

$$P, Q, \ldots ::= c \mid \sum_{i \in I} \forall \tilde{x}_i(c_i \rightarrow P_i) \mid P \parallel Q \mid \exists \tilde{x}(P) \mid p(\tilde{x})$$

An lcc program takes the form $\mathscr{D}.P$, where $\mathscr{D}$ is a set of process definitions of the form $p(\tilde{y}) \stackrel{\Delta}{=} P$ where all free variables of P are in the set of pairwise distinct variables $\tilde{y}$. We assume $\mathscr{D}$ to have a unique process definition for every process name.

Let us give some intuitions about the above constructs. The tell agent c adds constraint c to the current store d producing the new store $d \otimes c$.

Consider the guarded choice $Q = \sum_{i \in I} \forall \tilde{x}_i(c_i \rightarrow P_i)$, where I is a finite set of indexes. Let $j \in I$, d be the current store and θ be the substitution $[\tilde{t}/\tilde{x}_j]$ where $\tilde{t}$ is a sequence of terms. If $d \vdash d' \otimes c_j\theta$ for some d', then Q evolves into $P_j[\tilde{t}/\tilde{x}_j]$ and consumes $c_j\theta$. If none of the guards c_i can be deduced from d, the process Q blocks until more information is added to the store. Moreover, if many guards can be deduced, one of the alternatives is non-deterministically chosen for execution. To simplify the notation, we shall omit "$\sum_{i \in I}$" in $\sum_{i \in I} \forall \tilde{x}_i(c_i \rightarrow P_i)$ when I is a singleton; if the sequence of variables $\tilde{x}$ is empty, we shall write $c \rightarrow P$ instead of $\forall \tilde{x}(c \rightarrow P)$; moreover, if $|I| = 2$, we shall use "+" instead of "$\sum$" as in $c_1 \rightarrow P_1 + c_2 \rightarrow P_2$.

The interleaved parallel composition of P and Q is denoted by $P \parallel Q$. We shall use $\Pi_{i \in I} P_i$ to denote the parallel composition $P_1 \parallel \cdots \parallel P_n$, where $I = \{1, 2, \ldots, n\}$. If $I = \emptyset$, then $\Pi_{i \in I} P_i = 1$.

The agent $\exists \tilde{x}(P)$ behaves like P and binds the variables $\tilde{x}$ to be local to it. The processes $\exists \tilde{x}(P)$ and $\forall \tilde{x}(c \rightarrow P)$, as well as the constraint $\exists \tilde{x}(c)$, bind the variables $\tilde{x}$ in P and c. We shall use $fv(P)$ and $fv(c)$ to denote, respectively, the set of free variables of P and c.

Finally, given a process declaration of the form $p(\tilde{y}) \stackrel{\Delta}{=} P$, $p(\tilde{x})$ evolves into $P[\tilde{x}/\tilde{y}]$.

3.2 Operational semantics

Before giving a formal definition of the operational semantics of lcc agents, let us give an example of how processes evolve. For that, we shall use $\langle P\,;c\rangle \longrightarrow \langle P'\,;c'\rangle$ to denote that the agent P under store c evolves into the agent P' producing the store c'. This notation will be precisely defined shortly.

Example 3.1 (*Consuming permissions*)
Let us assume a constraint system with predicates ref/3, ct/2; constant symbols unq, imm, none, 0, *nil*; function symbol s (successor); and equipped with the axiom:

$$\Delta = \forall x, o.[\mathtt{ref}(x, o, \mathtt{imm}) \otimes \mathtt{ct}(o, s(0)) \multimap \mathtt{ref}(x, o, \mathtt{unq}) \otimes \mathtt{ct}(o, s(0))]$$

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

$$\frac{}{\langle X;\Gamma,c;d\rangle \longrightarrow \langle X;\Gamma;d\otimes c\rangle}\ \textsc{RTell} \qquad \frac{d\vdash \exists\widetilde{y}(d'\otimes c_i[\widetilde{t}/\widetilde{x}_i]),\ \widetilde{y}\cap fv(X,\Gamma,d)=\emptyset,\ mgc(d',\widetilde{t})}{\langle X;\Gamma,\sum_{i\in I}\forall\widetilde{x}_i(c_i\rightarrow P_i);d\rangle \longrightarrow \langle X\cup\widetilde{y};\Gamma,P_i[\widetilde{t}/\widetilde{x}];d'\rangle}\ \textsc{RChoice}$$

$$\frac{\widetilde{y}\cap X=\widetilde{y}\cap fv(\Gamma,d)=\emptyset}{\langle X;\Gamma,\exists\widetilde{y}(P);d\rangle \longrightarrow \langle X\cup\widetilde{y};\Gamma,P;d\rangle}\ \textsc{RLoc} \qquad \frac{p(\widetilde{y})\stackrel{\Delta}{=}P \text{ is a process definition}}{\langle X;\Gamma,p(\widetilde{x});d\rangle \longrightarrow \langle X;\Gamma,P[\widetilde{x}/\widetilde{y}];d\rangle}\ \textsc{RCall}$$

Fig. 5. Operational semantics of `lcc`. $fv(\Gamma,d)$ means $fv(\Gamma)\cup fv(d)$. $fv(X,\Gamma,d)$ means $fv(\Gamma,d)\cup X$. The notion of most general choice ($mgc(d',\tilde{t})$) is in Definition 4.

Informally, Δ says that an `imm` permission can be *upgraded* to `unq` if there is only one reference pointing to o.

Consider now the processes

$$\begin{array}{lcl}
P_1 & = & \texttt{ref}(x,o_x,\texttt{imm})\otimes\texttt{ct}(o_x,s(0))\\
P_2 & = & \texttt{ref}(y,o_y,\texttt{imm})\otimes\texttt{ref}(z,o_y,\texttt{imm})\otimes\texttt{ct}(o_y,s(s(0)))\\
Q & = & \forall o(\texttt{ref}(x,o,\texttt{unq})\otimes\texttt{ct}(o,s(0))\rightarrow Q')\\
Q' & = & \texttt{ref}(x,nil,\texttt{none})\otimes\texttt{ct}(o,0)\\
R & = & \forall o(\texttt{ref}(y,o,\texttt{unq})\rightarrow R')
\end{array}$$

Roughly, P_1 adds to the store the information required to state that x points to o_x with permission `imm` and that there is exactly one reference to o_x. P_2 states that there are two references (y and z) pointing to the same object o_y. Process Q, in order to evolve, requires x to have a unique permission on o_x. Finally, R is asking whether y has a unique permission on a given object o to execute R' (not specified here).

Starting from the configuration $\langle P_1 \parallel P_2 \parallel Q \parallel R;1\rangle$, we observe the derivation below:

$$\begin{array}{lll}
\textbf{(1)} & & \langle P_1\parallel P_2\parallel Q\parallel R;1\rangle\\
\textbf{(2)} & \longrightarrow & \langle P_1\parallel Q\parallel R;1\otimes\texttt{ref}(y,o_y,\texttt{imm})\otimes\texttt{ref}(z,o_y,\texttt{imm})\otimes\texttt{ct}(o_y,s(s(0)))\rangle\\
\textbf{(3)} & \longrightarrow & \langle Q\parallel R;1\otimes\texttt{ref}(y,o_y,\texttt{imm})\otimes\texttt{ref}(z,o_y,\texttt{imm})\otimes\texttt{ct}(o_y,s(s(0)))\otimes\texttt{ref}(x,o_x,\texttt{imm})\otimes\texttt{ct}(o_x,s(0))\rangle\\
\textbf{(4)} & \longrightarrow & \langle Q'[o_x/o]\parallel R;\texttt{ref}(y,o_y,\texttt{imm})\otimes\texttt{ref}(z,o_y,\texttt{imm})\otimes\texttt{ct}(o_y,s(s(0)))\rangle\\
\textbf{(5)} & \longrightarrow & \langle R;\texttt{ref}(y,o_y,\texttt{imm})\otimes\texttt{ref}(z,o_y,\texttt{imm})\otimes\texttt{ct}(o_y,s(s(0)))\otimes\texttt{ref}(x,nil,\texttt{none})\otimes\texttt{ct}(o_x,0)\rangle
\end{array}$$

From the initial store 1, neither Q nor R can deduce their guards and they remain blocked (line 1). Tell processes P_1 (line 3) and P_2 (line 2) evolve by adding information to the store. Let d (resp. d') be the store in the configuration of line 3 (resp. line 4) and c be the guard of the ask agent Q. We note that $d\vdash d'\otimes c[o_x/o]$. Recall that checking this entailment amounts to prove in ILL the sequent $!\Delta,d\longrightarrow d'\otimes c[o_x/o]$ (Definition 2). In this case, the axiom Δ allows us to transform $\texttt{ref}(x,o_x,\texttt{imm})\otimes\texttt{ct}(o_x,s(0))$ into $\texttt{ref}(x,o_x,\texttt{unq})\otimes\texttt{ct}(o_x,s(0))$. Hence, Q reduces to the tell agent $Q'[o_x/o]$ and consumes part of the store leading to the store d' in line 4. In line 5, $Q'[o_x/o]$ adds more information to the store, namely, x points to *null* and there are no references pointing to o_x. Note that R remains blocked since the guard $\texttt{ref}(y,o,\texttt{unq})$ cannot be entailed.

Operational semantics. Let us extend the processes-store configurations used in Example 3.1 to consider configurations of the form $\langle X;\Gamma;c\rangle$. Here, X is the set of local (*hidden*) variables in Γ and c, Γ is a multiset of processes of the form $P_1,\ldots,P_n$ representing the parallel composition $P_1\parallel\cdots\parallel P_n$, and c represents the current store. In what follows, we shall indistinguishably use the notation of multiset as parallel composition of processes.

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

$$\mathcal{C}[\![c]\!]_z = c \otimes \texttt{sync}(z) \qquad \mathcal{C}[\![\sum_{i\in I} \forall \tilde{x}_i(c_i \to P_i)]\!]_z = \sum_{i \in I} \mathcal{C}[\![\forall \tilde{x}_i(c_i \to P_i)]\!]_z$$
$$\mathcal{C}[\![\forall \tilde{y}(c \to P)]\!]_z = \forall \tilde{y}(c \to \mathcal{C}[\![P]\!]_z) \qquad \mathcal{C}[\![\exists y(P)]\!]_z = \exists y(\mathcal{C}[\![P]\!]_z)$$
$$\mathcal{C}[\![P_1 \parallel \ldots \parallel P_n]\!]_z = \exists w_1 \ldots w_n(\mathcal{C}[\![P_1]\!]_{w_1} \parallel \ldots \parallel \mathcal{C}[\![P_n]\!]_{w_n} \parallel \bigotimes_{i \in 1..n} \texttt{sync}(w_i) \to \texttt{sync}(z))$$
$$\mathcal{C}[\![p(\tilde{x})]\!]_z = p(\tilde{x}, z) \qquad \mathcal{C}[\![p(\tilde{y}) \triangleq P]\!]_z = p(\tilde{y}, z) \triangleq \mathcal{C}[\![P]\!]_z$$

Fig. 6. Definition of the sequential composition $P\,;Q$.

The transition relation $\longrightarrow$ defined on configurations is the least relation satisfying the rules in Figure 5. We shall use $\longrightarrow^*$ to denote the reflexive and transitive closure of $\longrightarrow$. It is easy to see that rules R_{TELL}, R_{LOC}, and R_{CALL} realize the behavioral intuition given in the previous section. Let us explain the Rule R_{CHOICE}. Recall that the process $\sum_{i\in I} \forall \tilde{x}_i(c_i \to P_i)$ executes $P_j[\tilde{t}/\tilde{x}_j]$ if $c_j[\tilde{t}/\tilde{x}_i]$ can be deduced from the current store d, i.e., $d \vdash d' \otimes c_j[\tilde{t}/\tilde{x}]$. Moreover, the constraint $c_j[\tilde{t}/\tilde{x}]$ is consumed from d leading to the new store d'. Hence, d' must be the most general choice in the following sense:

Definition 4 (*Most general choice (mgc) (Martinez 2010; Haemmerlé 2011)*)
Consider the entailment $d \vdash \exists \tilde{y}(e \otimes c[\tilde{t}/\tilde{x}])$ and assume that $\tilde{y} \cap fv(d) = \emptyset$. Assume also that $d \vdash \exists \tilde{y}(e' \otimes c[\tilde{t}'/\tilde{x}])$ for an arbitrary e' and $\tilde{t}'$. We say that e and $\tilde{t}$ are the most general choices, notation $mgc(e, \tilde{t})$, whenever $e' \vdash e$ implies $e \vdash e'$ and $c[\tilde{t}/\tilde{x}] \vdash c[\tilde{t}'/\tilde{x}]$.

The *mgc* requirement in rule R_{CHOICE} prevents from an unwanted weakening of the store. For instance, consider the ask agent $Q = c \to P$. We know that $!\,c$ entails $c \otimes 1$ (i.e., $!\,c \vdash c \otimes 1$). Hence, without the *mgc* condition, Q may consume $!\,c$ leading to the store 1. This is not satisfactory since Q did not consume the *minimal information* required to entail its guard. In this particular case, we have to consider the entailment $!\,c \vdash !\,c \otimes c$ where Q can entail its guard and the store remains the same. For further details, please refer to (Haemmerlé 2011).

Sequential composition. In the subsequent sections, we shall use the derived operator $P\,;Q$ that delays the execution of Q until P signals its termination. This operator can be encoded in `lcc` as follows. Let z be a variable that does not occur in P nor in Q and let $\texttt{sync}(\cdot)$ be an uninterpreted predicate symbol that does not occur in the program. The process $P\,;Q$ is defined as $\exists z(\mathscr{C}[\![P]\!]_z \parallel \texttt{sync}(z) \to Q)$ where $[\![\cdot]\!]_z$ is in Figure 6. Intuitively, $\mathscr{C}[\![P]\!]_z$ adds the constraint $\texttt{sync}(z)$ to signal the termination of P. Then, the ask agent $\texttt{sync}(z) \to Q$ reduces to Q. Note that in a parallel composition $P \parallel R$, one has to wait for the termination of both P and R before adding the constraint $\texttt{sync}(z)$. For that, $\mathscr{C}[\![P \parallel R]\!]_z$ creates fresh variables w_1 and w_2 to signal the termination of, respectively, P and R. Then, it adds $\texttt{sync}(z)$ only when both $\texttt{sync}(w_1)$ and $\texttt{sync}(w_2)$ can be deduced. Assume now a process definition of the form $p(\tilde{y}) \stackrel{\text{def}}{=} P$. We require the process P to emit the constraint $\texttt{sync}(z)$ to synchronize with the call $p(\tilde{x})$. We then add an extra parameter to the process definition ($p(\tilde{y}, z) \stackrel{\Delta}{=} \mathscr{C}[\![P]\!]_z$). Hence, the variable z is passed as a parameter and used by $\mathscr{C}[\![P]\!]_z$ to synchronize with the call $p(\tilde{x}, z)$.

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

Constant Symbols	
PER= {unq, shr, imm, none}	Types of access permissions.
GPER = {atm, conc}	Types of data group access permissions.
ndg	Absence of data group.
nst	Absence of statement.
nil	Null reference
c_a	For each field a of a class c.
c_g_i	For each group parameter in the class definition class $c\ \langle g_1, \ldots, g_n\rangle$
$g_1, \ldots, g_n$	For each DG in group$\langle g_1, \cdots, g_n\rangle$
Predicate Symbols	
ref(x, o, p, g)	x points to object o with permission $p \in$ PER and belongs to the data group g
field(u, o, a)	u is the reference to field a of object o.
gparam(c_g, o, gp)	The group parameter g of the object o was instantiated with the data group gp.
sync(z)	Synchronizing on variable z.
act(z)	Activate/start statement z.
run(z)	Statement z is being executed.
end(z)	End of statement z.
ct(o, n)	There are n references pointing to object o.
dg(g, p, z)	Statement z has a data group permission of type $p \in$ GPER on the data group g.
Axioms	
downgrade$_1$	$\forall x, o, g.[\mathtt{ref}(x, o, \mathtt{unq}, \mathtt{ndg}) \multimap \mathtt{ref}(x, o, \mathtt{shr}, g)]$
downgrade$_2$	$\forall x, o.[\mathtt{ref}(x, o, \mathtt{unq}, \mathtt{ndg}) \multimap \mathtt{ref}(x, o, \mathtt{imm}, \mathtt{ndg})]$
upgrade$_1$	$\forall x, o, g.[\mathtt{ref}(x, o, \mathtt{shr}, g) \otimes \mathtt{ct}(o, s(\mathbf{0})) \multimap \mathtt{ref}(x, o, \mathtt{unq}, \mathtt{ndg}) \otimes \mathtt{ct}(o, s(\mathbf{0}))]$
upgrade$_2$	$\forall x, o.[\mathtt{ref}(x, o, \mathtt{imm}, \mathtt{ndg}) \otimes \mathtt{ct}(o, s(\mathbf{0})) \multimap \mathtt{ref}(x, o, \mathtt{unq}, \mathtt{ndg}) \otimes \mathtt{ct}(o, s(\mathbf{0}))]$

Fig. 7. Constraint system for access permissions. **0** denotes the constant “zero” and $s(\cdot)$ successor.

4 AP programs as lcc processes

This section presents an interpretation of APs and DGAPs as processes in lcc. We thus endow AP programs with a declarative semantics which is adequate to verify relevant properties as we show later. We start defining the constraint system we shall use. Constants, predicate symbols, and non-logical axioms are depicted in Figure 7 and explained below.

We shall use c, m, a, g, o to range, respectively, over name of classes, methods, fields, DGs, and objects in the source AP language. For variables, we shall use x, y, and u. We may also use primed and subindexed version of these letters. We shall use the same letters in our encodings. Hence, if x occurs in a constraint (see e.g., predicate ref$(\cdot)$ below), it should be understood as the representation of a variable x in the source language. Finally, we shall use z, w (possible primed or subindexed) to represent identifiers of statements in the source program. Those variables will appear in the scope of constraints used for synchronization in the model as, e.g., in the constraint sync$(\cdot)$.

Permissions and constants: Constant symbols in sets PER and GPER represent the kind of APs and DGAPs available in the language. Since *none*, *unique*, and *immutable* AP are not associated to any DG, we shall use the constant ndg to denote “no-group.” Recall that the split command splits a DGAP into several DGAPs, one per each statement in the block. Then, we require to specify in our model the statement to which the concurrent permission is attached to (see predicate dg$(\cdot)$ below). Since atomic DGAPs are not attached to any particular statement in the program, we use the constant nst to denote “no-statement.” The constant *nil* is used to denote a null reference. Assume a class c with an attribute a and a DG parameter g. We use the constant symbol c_a to make reference to a (see predicate field$(\cdot)$ below) and a constant symbol c_g to make reference to g (see gparam$(\cdot)$ below).

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

We also consider the constant symbols $g_1, \dots, g_n$ to give meaning to the statement $\mathtt{group}\langle g_1, \cdots, g_n\rangle$.

References and fields: We use the predicate symbol $\mathtt{ref}(x, o, p, g)$ to represent that the variable x is pointing to object o and it has a permission p on it. The last parameter of this predicate is used to give meaning to share permissions of the form $\mathtt{shr} : g$. As we already explained, $g = \mathtt{ndg}$ when $p \neq \mathtt{shr}$. The predicate $\mathtt{field}(x_u, o, a)$ associates the variable x_u to the field a of object o. Once an object of a given class c with DG parameters is instantiated, the predicate $\mathtt{gparam}(c\_g, o, gp)$ dictates that the group parameter g of the object o was instantiated with the DG gp. The predicate $\mathtt{sync}(z)$ is used in the definition of $P\,;Q$ as explained in the previous section. Constraints $\mathtt{act}(z)$, $\mathtt{run}(z)$, and $\mathtt{end}(z)$ represent, respectively, that statement z has been called, it is currently being executed or it has finished. We shall use those constraints as witnesses for verification purposes. The number of references (alias) pointing to a given object are modeled with the predicate $\mathtt{ct}(o, n)$. Given a DG g, the predicate $\mathtt{dg}(g, p, z)$ dictates that the statement z has a DG permission $p \in \mathtt{GPER}$ on g. If $p = \mathtt{atm}$, then $z = \mathtt{nst}$.

Non-logical axioms: The entailment of the constraint system allows us to formalize when a given AP can be transformed into another. Assume that x has a unique permission on o. Unique permissions can be downgraded to share or immutable permissions as dictated by axioms $\mathtt{downgrade}_1$ and $\mathtt{downgrade}_2$, respectively. Axiom $\mathtt{upgrade}_1$ (resp. $\mathtt{upgrade}_2$) builds a unique permission from a share (resp. immutable) permission. For that, x needs to be the unique reference with share or immutable permission to the pointed object. Conversions from share permissions into immutable and vice versa require to first upgrade the permission to unique and then applying the appropriate downgrade axiom.

4.1 Modeling statements

Given an AP annotated program, we shall build an lcc program $\mathscr{D}.P$ where $\mathscr{D}$ includes process definitions for each method and constructor of the AP program (Section 4.2 below) and a process definition to encode assignments ($\mathtt{assg}$ in Figure 8). The process P represents the encoding of the main body of the AP program where each statement s is encoded as a lcc process $\mathscr{S}[\![s]\!]_z^G$ that models its behavior.

The process $\mathscr{S}[\![s]\!]_z^G$ adheres to the following schema. The lcc variable z is used to represent the statement s in the model. We assume (by renaming variables if necessary) that z does not occur in s. The encoding uses constrains to signal three possible states in the execution of s. When the program control reaches the statement s, the encoding adds the constraint $\mathtt{act}(z)$ to signal that s is ready to be executed. When the needed permissions for s are successfully acquired, $\mathtt{act}(z)$ is consumed and constraints $\mathtt{sync}(z)$ and $\mathtt{run}(z)$ are added. The first one is used to synchronize with the rest of the model. More precisely, the encoding of the next instruction in the program waits for constraint $\mathtt{sync}(z)$ to be posted before starting its execution. In this way, we model the *data dependencies* resulting from the flow of

$$
\begin{array}{lll}
\mathtt{assg}(x,y,z,gt) & \triangleq & \mathtt{drop}(x);\mathtt{gain}(x,y,gt);\mathtt{act}(z)\rightarrow\mathtt{run}(z);\mathtt{run}(z)\rightarrow\mathtt{sync}(z)\otimes!\mathtt{end}(z)\\
\mathtt{drop}(x) & \stackrel{\text{def}}{=} & \forall o,n,g((\mathtt{ref}(x,nil,\mathtt{none},\mathtt{ndg})\rightarrow 1) + \sum_{p\in\mathtt{PER}\setminus\{\mathtt{none}\}} \mathtt{ref}(x,o,p,g)\otimes\mathtt{ct}(o,s(n))\rightarrow\mathtt{ct}(o,n))\\
\mathtt{gain}(x,y,gt) & \stackrel{\text{def}}{=} & \mathtt{ref}(y,nil,\mathtt{none},\mathtt{ndg})\rightarrow\mathtt{ref}(x,nil,\mathtt{none},\mathtt{ndg})\otimes\mathtt{ref}(y,nil,\mathtt{none},\mathtt{ndg})
\end{array}
$$
$$
\begin{array}{l}
+\ \forall o,n((\mathtt{ref}(y,o,\mathtt{unq},\mathtt{ndg})\otimes\mathtt{ct}(o,s(\mathbf{0}))\rightarrow\mathtt{ref}(y,o,\mathtt{shr},gt)\otimes\mathtt{ref}(x,o,\mathtt{shr},gt)\otimes\mathtt{ct}(o,s(s(\mathbf{0}))))\\
\quad +(\mathtt{ref}(y,o,\mathtt{shr},gt)\otimes\mathtt{ct}(o,n)\rightarrow\mathtt{ref}(y,o,\mathtt{shr},gt)\otimes\mathtt{ref}(x,o,\mathtt{shr},gt)\otimes\mathtt{ct}(o,s(n)))\\
\quad +(\mathtt{ref}(y,o,\mathtt{imm},\mathtt{ndg})\otimes\mathtt{ct}(o,n)\rightarrow\mathtt{ref}(y,o,\mathtt{imm},\mathtt{ndg})\otimes\mathtt{ref}(x,o,\mathtt{imm},\mathtt{ndg})\otimes\mathtt{ct}(o,s(n))))
\end{array}
$$

Fig. 8. Auxiliary definitions for Rule $\mathrm{R}_{\mathrm{ALIAS}}$.

APs. Constraint $\mathtt{run}(z)$ signals that s is currently being executed. Once s has finished and the consumed permissions are restored, the encoding consumes $\mathtt{run}(z)$ and adds the constraint $\mathtt{end}(z)$.

The G in $\mathscr{S}[\![s]\!]_z^G$ stands for the set of DGs on which s must have a concurrent DGAP to be executed. Recall that such permissions are assigned by a `split` command. Then, we use G to control which DGAPs must be consumed and restored by s.

In the following, we define $\mathscr{S}[\![s]\!]_z^G$ for each kind of statement in the syntax in Figure 3. For that, the following shorthand will be useful ($\stackrel{\text{def}}{=}$ must be understood as a shorthand and not as process definition):

$$
\boxed{\mathtt{wrap}(P,\{g_1,\cdots,g_n\},z)\stackrel{\text{def}}{=}\mathtt{act}(z);(\bigotimes_{i\in 1..n}\mathtt{dg}(g_i,\mathtt{conc},z)\rightarrow 1);P\parallel(\mathtt{end}(z)\rightarrow\bigotimes_{i\in 1..n}\mathtt{dg}(g_i,\mathtt{conc},z)))} \qquad \text{(wrap)}
$$

Assume that s is an statement and $P=\mathscr{S}[\![s]\!]_z^G$. The process $\mathtt{wrap}(P,G,z)$ first consumes all the concurrent DGAPs available for s, i.e., those in the set G. If $G=\emptyset$, then $\bigotimes_{i\in 1..n}\mathtt{dg}(g_i,\mathtt{conc},z)$ is defined as 1. Observe that once s has terminated (i.e., the constraint $\mathtt{end}(z)$ is added to the store) such permissions are restored.

Assignments. We have different cases for the assignment $r\langle g\rangle := rhs$ depending whether r and rhs are variables or field selections. Let us start with the case when both are variables as in $x\langle g\rangle := y$ and x is syntactically different from y. We have

$$
\boxed{\mathscr{S}[\![x\langle gt\rangle := y]\!]_z^G \ = \mathtt{wrap}(\mathtt{assg}(x,y,z,gt),G,z)} \qquad (\mathrm{R}_{\mathrm{ALIAS}})
$$

where `assg` is defined in Figure 8. The variable x loses its permission to the pointed object o, and the object o has one less reference pointing to it (Definition `drop`). Thereafter, x and y point to the same object and the permission of y is split between x and y as explained in Section 2.2 (Definition `gain`). Finally, once the permission to y is split, the constraints $\mathtt{sync}(z)$ and $!\mathtt{end}(z)$ are added to the store to, respectively, synchronize with the rest of the program and mark the termination of the statement. Note in `assg` the use of the constraints $\mathtt{act}(\cdot),\mathtt{run}(\cdot)$, and $\mathtt{end}(\cdot)$. Initially, constraint $\mathtt{act}(z)$ is added (by `wrap`). When the permissions on x and y are split (after `drop` and `gain`), $\mathtt{act}(z)$ is consumed to produce $\mathtt{run}(z)$. Finally, $\mathtt{run}(z)$ is consumed to produce $\mathtt{end}(z)$.

Now consider the case $\mathscr{S}[\![x.a\langle g\rangle := y]\!]_z^G$. If the variable x points to the object o of class c, then the field a of o can be accessed via the variable u whenever the constraint $\mathtt{field}(u,o,c\_a)$ holds. Intuitively, u points to $x.a$ and then, a constraint $\mathtt{ref}(u,o',p,g)$ dictates that $x.a$ points to o' with permission p. As we shall show later, the model

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

of constructors adds the constraint $!\mathtt{field}(u, o, c\_a)$ to establish the connection between objects and their fields. The model of the assignment $\mathscr{S}[\![x.a\langle g\rangle := y]\!]_z^G$ is thus obtained from that of $\mathscr{S}[\![u\langle g\rangle := y]\!]_z^G$:

$$\boxed{\mathscr{S}[\![x.a\langle g\rangle := y]\!]_z^G = \forall u, o, p, g(\mathtt{ref}(x, o, p, g) \otimes \mathtt{field}(u, o, c\_a) \rightarrow (\mathtt{ref}(x, o, p, g); \mathscr{S}[\![u\langle g\rangle := y]\!]_z^G))} \quad (\mathrm{R_{ALIAS_F}})$$

The cases $x.a\langle g\rangle := y.a'$ and $x\langle g\rangle := y.a$ are similar.

Let. Local variables in the AP program are defined as local variables in `lcc`:

$$\boxed{\mathscr{S}[\![\mathtt{let}\ \widetilde{T\ x}\ \mathtt{in}\ s\ \mathtt{end}]\!]_z^G = \exists\tilde{x}(\bigotimes_{i\in 1..|\tilde{x}|} \mathtt{ref}(x_i, nil, \mathtt{none}, \mathtt{ndg}); \mathscr{S}[\![s]\!]_z^G \parallel \mathtt{GC})} \quad (\mathrm{R_{LOC}})$$

where $\mathtt{GC} \stackrel{\mathrm{def}}{=} \mathtt{end}(z) \rightarrow \prod_{i\in 1..|\tilde{x}|} \mathtt{drop}(x_i)$. Observe that the freshly created variables point to *nil* with no permissions. Once s ends its execution, the local variables are destroyed (definition GC). We note also that, in this case, we do not add the constraint $\mathtt{sync}(z)$ nor $\mathtt{end}(z)$. The reason is that the creation of the local variable can be considered as "instantaneous" and then, the process $\mathscr{S}[\![s]\!]_z^G$ will be in charge of marking the termination of the statement. Note that we ignore the type T since our model and analyses are concerned only with the flow of APs and we assume that the source program is well typed.

Block of statements. In the block $\{s_1 \cdots\ s_i\ s_j \cdots s_n\}$, the process modeling s_j runs in parallel with the other processes once $\mathscr{S}[\![s_i]\!]_{z_i}^G$ adds the constraint $\mathtt{sync}(z_i)$ to the store. Hence, what we observe is that the execution of s_j is delayed until the encoding of s_i has successfully consumed the required permissions. After that, even if s_i has not terminated, the encoding of s_j can proceed. Once $\mathtt{sync}(z_n)$ can be deduced, constraint $\mathtt{sync}(z)$ is added to the store to synchronize with the rest of the program. Moreover, the constraint $\mathtt{end}(z)$ is added only when all the statements $s_1, \ldots, s_n$ have finished their execution:

$$\boxed{\mathscr{S}[\![\{s_1\ \ldots\ s_i\ \ldots\ s_n\}]\!]_z^G = \mathtt{wrap}(P, G, z)} \quad (\mathrm{R_{COMP}})$$

where P is defined as

$$P \stackrel{\mathrm{def}}{=} \mathtt{act}(z) \rightarrow \mathtt{run}(z); \exists z_1, \ldots z_n(\mathscr{S}[\![s_1]\!]_{z_1}^G \parallel \mathtt{sync}(z_1) \rightarrow \mathscr{S}[\![s_2]\!]_{z_2}^G \parallel \ldots \parallel \mathtt{sync}(z_{n-1}) \rightarrow \mathscr{S}[\![s_n]\!]_{z_n}^G \parallel \mathtt{sync}(z_n) \rightarrow \mathtt{sync}(z) \parallel (\mathtt{run}(z) \otimes \bigotimes_{i\in 1..n} \mathtt{end}(z_i)) \rightarrow !\mathtt{end}(z))$$

Groups of permissions. In order to define DGs, we add a constraint specifying that each of those groups has an atomic DGAP. Recall that the constant `nst` indicates that the atomic permission is not attached to any particular statement in the program:

$$\boxed{\mathscr{S}[\![\mathtt{group}\langle g_1, \ldots, g_n\rangle]\!]_z^G = \bigotimes_{i\in 1..n} \mathtt{dg}(g_i, \mathtt{atm}, \mathtt{nst})} \quad (\mathrm{R_{NEWG}})$$

Similar to the creation of local variables, we do not mark termination of this statement since it can be considered as "instantaneous."

Split. Let $G' = \{g_1, \ldots, g_m\}$. We define the rule for `split` as follows:

$$\boxed{\mathscr{S}[\![\mathtt{split}\langle G'\rangle\{s_1 \cdots s_n\}]\!]_z^G = \mathtt{wrap}(P, G \setminus G', z)} \quad (\mathrm{R_{SPLIT}})$$

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

$$
\begin{array}{rcl}
P & \stackrel{\text{def}}{=} & \exists z_1, \ldots, z_n, z' (\texttt{gainP}; \texttt{act}(z) \rightarrow \texttt{addP}; \texttt{exec}; (\texttt{sync}(z') \rightarrow \texttt{restoreP}); \texttt{run}(z) \rightarrow !\texttt{end}(z)) \\
\texttt{gainP} & \stackrel{\text{def}}{=} & \texttt{dg}(g_1, \texttt{conc}, z) \rightarrow \texttt{env}(g_1, \texttt{conc}, z) + \texttt{dg}(g_1, \texttt{atm}, \texttt{nst}) \rightarrow \texttt{env}(g_1, \texttt{atm}, \texttt{nst}) \parallel \ldots \parallel \\
 & & \texttt{dg}(g_m, \texttt{conc}, z) \rightarrow \texttt{env}(g_m, \texttt{conc}, z) + \texttt{dg}(g_m, \texttt{atm}, \texttt{nst}) \rightarrow \texttt{env}(g_m, \texttt{atm}, \texttt{nst}) \\
\texttt{addP} & \stackrel{\text{def}}{=} & \texttt{run}(z) \otimes \bigotimes_{i \in 1..n} \bigotimes_{j \in 1..m} \texttt{dg}(g_j, \texttt{conc}, z_i) \\
\texttt{exec} & \stackrel{\text{def}}{=} & \mathcal{S}[\![s_1]\!]_{z_1}^{G'} \parallel \texttt{sync}(z_1) \rightarrow \mathcal{S}[\![s_2]\!]_{z_2}^{G'} \parallel \cdots \parallel \texttt{sync}(z_n) \rightarrow \texttt{sync}(z') \\
\texttt{restoreP} & \stackrel{\text{def}}{=} & \bigotimes_{i \in 1..n} \bigotimes_{j \in 1..m} \texttt{dg}(g_j, \texttt{conc}, z_i) \rightarrow \texttt{sync}(z); \\
 & & \texttt{env}(g_1, \texttt{conc}, z) \rightarrow \texttt{dg}(g_1, \texttt{conc}, z) + \texttt{env}(g_1, \texttt{atm}, \texttt{nst}) \rightarrow \texttt{dg}(g_1, \texttt{atm}, \texttt{nst}) \parallel \cdots \parallel \\
 & & \texttt{env}(g_m, \texttt{conc}, z) \rightarrow \texttt{dg}(g_m, \texttt{conc}, z) + \texttt{env}(g_m, \texttt{atm}, \texttt{nst}) \rightarrow \texttt{dg}(g_m, \texttt{atm}, \texttt{nst})
\end{array}
$$

Fig. 9. Auxiliary definitions for Rule $\mathrm{R}_{\mathrm{SPLIT}}$.

where P and definitions `gainP`, `addP`, `exec`, and `restoreP` are in Figure 9. Before explaining those definitions, consider the following code:

```
1 split <g1,g2>{
2   s1
3   split <g2,g3>{
4    s2} }
```

and assume we are encoding the `split` statement in line 3. Then, we consider the process $\mathscr{S}[\![\texttt{split}\langle G'\rangle\{s_2\}]\!]_z^G$ where $G' = \{g_2, g_3\}$. The set $G = \{g_1, g_2\}$ corresponds to the concurrent DGAPs assigned by the external `split` statement in line 1. The process `gainP` consumes either atomic or concurrent permissions for each DG $g_i \in G'$. Since such permissions must be restored once the split command has been executed, we distinguish the case when the consumed permission is concurrent (`conc`) or atomic (`atm`). For that, we use the auxiliary predicate symbol (constraint) $\texttt{env}(\cdot)$ that keeps information of the DGAP consumed. We note that the DGAP $g_2 \in G \cap G'$ is consumed and then split again to be assigned to the statement s_2.

Now consider the DG $g_1 \in G \setminus G'$. Since $g_1 \notin G'$, the DGAP on this group must be consumed and it must not be split to be assigned to s_2. Hence, the consumption of any $g \in G \setminus G'$ is handled by the $\texttt{wrap}(\cdot)$ process as in the encoding of other statements.

Once we have consumed the appropriate DGAPs, we add, for each statement in the block, a concurrent DGAP for each of the DGs in G' (definition `addP`).

The process `exec` is similar to Rule $\mathrm{R}_{\mathrm{COMP}}$ but it uses as parameter G'. In our example, this means that concurrent DGAPs on g_2 and g_3 (and not on g_1) are assigned to s_2. As we already saw in the definition of $\mathrm{R}_{\mathrm{COMP}}$, the constraint $\texttt{sync}(z')$ is added to the store once all the statements in the block were able to consume the required APs. At this point, we wait for all the instructions to reestablish their assigned DGAPs (definition `restoreP`). Recall that this happens only when the statements terminate (see definition `wrap`).

Finally, with the help of the constraints $\texttt{env}(\cdot)$, we restore the DGAPs to the environment and we add the constraint $!\texttt{end}(z)$ to mark the ending of the block.

Method calls and **Object instantiation**. In our encoding, we shall write methods and constructors using functional notation rather than OO notation. For instance, $x.m(\tilde{y})$ is written as $c\_m(x, \tilde{y})$ when x is an object of type c. Similarly, the expression $c\_c(x, \tilde{y})$ corresponds to $x := \texttt{new}\ c(\tilde{y})$. As we shall see, for each method $m(\tilde{y})$ of the class c, we shall generate a process definition $c\_m(x, \tilde{y}, z) \stackrel{\Delta}{=} P$. The extra argument z is used

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

to later add the constraint $\mathtt{sync}(z)$ to synchronize with the rest of the program. If x is of type c, the rule is defined as follows:

$$\boxed{\mathscr{S}[\![x.m(\tilde{y})]\!]_z^G = \mathtt{wrap}(c\_m(x, y_1, .., y_n, z), G, z)} \qquad (\mathrm{R_{CALL}})$$

The case of the call $x.a.m(\tilde{y})$ can be obtained by using the constraint $\mathtt{field}(\cdot)$ as we did in Rule $\mathrm{R_{ALIAS_F}}$ for assignments between fields.

The model of an object initialization is defined similarly but we add also as a parameter the instances of the DGs:

$$\boxed{\mathscr{S}[\![x := \mathtt{new}\ c\langle g_1,\ldots,g_n\rangle(\tilde{y})]\!]_z^G = \mathtt{wrap}(c\_c(x, \tilde{y}, z, g_1,\ldots,g_n), G, z)} \qquad (\mathrm{R_{NEW}})$$

4.2 Modeling class definitions

In this section, we describe function $\mathscr{D}[\![\cdot]\!]$ interpreting method and constructors definitions as $\mathtt{lcc}$ process definitions.

Method definitions. Let $m(\widetilde{c_y\langle \widetilde{g_y}\rangle\ y})\ \ p(\mathtt{this}), \widetilde{p(y)} \Rightarrow p'(\mathtt{this}), \widetilde{p'(y)}\ \{s\}$ be a method in class $c\langle \tilde{g}_x\rangle$. We define

$$\boxed{\mathscr{D}[\![c.m]\!] = c\_m(x, \tilde{y}, z) \stackrel{\Delta}{=} \exists \tilde{y}', x'(\mathit{Consume}; \mathtt{sync}(z); \mathtt{act}(z) \rightarrow \mathtt{run}(z); \mathit{Body})} \qquad (\mathrm{R_{MDEF}})$$

where $n = |\tilde{y}| = |\tilde{y}'|$,

$$\mathit{Consume} \stackrel{\mathrm{def}}{=} \textstyle\prod_{i\in 1..n} \mathtt{consume}(y_i, y_i', p_i, c) \parallel \mathtt{consume}(x, x', p, c)$$
$$\mathit{Body} \stackrel{\mathrm{def}}{=} \exists z'(\mathscr{S}[\![\hat{s}]\!]_{z'} \parallel (\mathtt{sync}(z') \otimes \mathtt{end}(z')) \rightarrow (\mathtt{r\_env}(x, p, x', p', c) \parallel \textstyle\prod_{i\in 1..n} \mathtt{r\_env}(y_i, p_i, y_i', p_i', c))); \mathtt{run}(z) \rightarrow \mathtt{end}(z)$$

and the auxiliary process definitions $\mathtt{consume}(\cdot)$ and $\mathtt{r\_env}(\cdot)$ are in Figure 10.

In the process definition $c\_m(x, \tilde{y}, z)$, the first parameter x represents the object caller $\mathtt{this}$ and the last parameter z is used for synchronization. This definition first declares the local variables $\tilde{y}'$ and x' to replace the formal parameters ($\tilde{y}$) and the caller (x) by the actual parameters. Next, it consumes the required permissions from $\tilde{y}$ and x, and assigns them to the previously mentioned local variables. Finally, the constraint $\mathtt{sync}(z)$ is added and the encoding of the method's body is executed. In the following, we explain the definitions *Consume* and *Body*.

The definition of $\mathtt{consume}(x, x', p, c)$ in Figure 10 can be read as "*consume the permission p on the variable x and assign it to the variable x'.*" If the required permission is share or immutable, the permission is split and restored to allow concurrent executions in the environment that called the method. We recall that in $p = \mathtt{shr} : g$, g must be a DG parameter in the class c. This explains the last parameter in $\mathtt{consume}(\cdot)$. We then use the predicate $!\,\mathtt{gparam}(c\_g, o, g)$, added by the encoding of constructors, as we shall see, to establish the link between the DG parameter and the current DG. Finally, unique and none permissions are consumed and transferred to the local variables.

Now we focus on the definition *Body* where $\hat{s}$ denotes s after replacing y_i by y_i' and x by x'. Once $\hat{s}$ finishes (i.e., it adds $\mathtt{end}(z')$ to the store), the references and permissions of the local variables created to handle the parameters are consumed and restored to the environment according to $\mathtt{r\_env}(x, p, x', p', c)$ in Figure 10 (consume

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

$$\mathtt{consume}(x, x', p, cname) \stackrel{\mathtt{def}}{=}$$
$$\begin{cases} \forall o(\mathtt{ref}(x,o,p,\mathtt{ndg}) \otimes \mathtt{ct}(o,n) \rightarrow \mathtt{ref}(x,o,p,\mathtt{ndg}) \otimes \mathtt{ct}(o,s(n)) \otimes \mathtt{ref}(x',o,p,\mathtt{ndg})) \text{ if } p = \mathtt{imm} \\ \forall g,o(\mathtt{gparam}(cname\_g,o,g) \rightarrow \mathtt{ref}(x,o,p,g) \otimes \mathtt{ct}(o,n) \rightarrow \mathtt{ref}(x,o,p,g) \otimes \mathtt{ct}(o,s(n))) \otimes \mathtt{ref}(x',o,p,g) \text{ if } p = \mathtt{shr}:g \\ \forall o(\mathtt{ref}(x,o,p,\mathtt{ndg}) \rightarrow \mathtt{ref}(x',o,p,\mathtt{ndg})) \text{ if } p \in \{\mathtt{unq},\mathtt{none}\} \end{cases}$$

$$\mathtt{r\_env}(x, p, x', p', cname) \stackrel{\mathtt{def}}{=}$$
$$\begin{cases} \forall o', n(\mathtt{ref}(x',o',p',\mathtt{ndg}) \otimes \mathtt{ct}(o',s(n)) \rightarrow \mathtt{ct}(o',n)) \quad \text{if } p,p' = \mathtt{imm} \\ \forall o', n, g(\mathtt{gparam}(cname\_g,o',g) \rightarrow \mathtt{ref}(x',o',p',g) \otimes \mathtt{ct}(o',s(n)) \rightarrow \mathtt{ct}(o',n)) \quad \text{if } p,p' = \mathtt{shr}:g \\ \forall o'(\mathtt{ref}(x',o',p',\mathtt{ndg}) \otimes \mathtt{ct}(o',s(0)) \rightarrow \mathtt{ref}(x,o',p',\mathtt{ndg}) \otimes \mathtt{ct}(o',s(0)) \quad \text{if } p,p' = \mathtt{unq} \\ \forall o'(\mathtt{ref}(x',o',p',\mathtt{ndg}) \rightarrow \mathtt{ref}(x,o',p',\mathtt{ndg}) \quad \text{if } p = \mathtt{none} \\ \forall o,n,o'((\mathtt{ref}(x,o,p,\mathtt{ndg}) \otimes \mathtt{ct}(o,s(n)) \rightarrow \\ \quad \mathtt{ct}(o,n)); \mathtt{ref}(x',o',p',\mathtt{ndg}) \rightarrow \mathtt{ref}(x,o',p',\mathtt{ndg})) \text{ if } p = \mathtt{imm}, p' \in \{\mathtt{unq},\mathtt{none}\} \\ \forall o,n,o',g((\mathtt{gparam}(cname\_g,o,g) \otimes \mathtt{ref}(x,o,p,g) \otimes \mathtt{ct}(o,s(n)) \rightarrow \\ \quad \mathtt{ct}(o,n)); \mathtt{ref}(x',o',p',\mathtt{ndg}) \rightarrow \mathtt{ref}(x,o',p',\mathtt{ndg})) \text{ if } p = \mathtt{shr}, p' \in \{\mathtt{unq},\mathtt{none}\} \end{cases}$$

Fig. 10. Auxiliary definitions for constructor and method declarations.

the permission p on x and transforms it into a permission p' to the variable x'). Let us give some intuition about the cases considered in this definition. Recall that `consume` *replicates* the `shr` and `imm` permissions for the variables internal to the method. Therefore, we only need to consume those permissions and decrease the number of references pointing to object o'. When the input permissions are `unq` or `none`, `consume` *transfers* those permissions to the local variables and *consumes* the external references. Then, `r_env` needs to restore the external reference and consume the local one (the number of references pointing to o' remains the same). When the method changes the input permission from share or immutable into a unique or none, we need to *consume* first the external reference. Afterwards, we *transfer* the internal permission and reference to the external variable.

Constructor definitions. Let $c(\widetilde{c_y\langle\widetilde{g_x}\rangle\ y})$ $\mathtt{none(this)}, \widetilde{p(y)} \Rightarrow p'(\mathtt{this}), \widetilde{p'(y)}$ $\{s\}$ be a constructor of a parameterized class $c\langle pg_1, \ldots, pg_k\rangle$. We define

$$\begin{aligned} \mathscr{D}[\![C_D]\!] = c(x, \tilde{y}, z, g_1, \ldots, g_k) \stackrel{\Delta}{=} \ & \exists \tilde{y'}, x', o_{new}(\mathtt{gparam\text{-}init}; \mathtt{consume}'; \\ & \exists \tilde{u}(\mathtt{fields\text{-}init}\ ; \mathtt{sync}(z); \mathtt{act}(z) \rightarrow \mathtt{run}(z); \\ & \quad \exists z'(\mathscr{S}[\![\hat{s}]\!]_{z'} \parallel (\mathtt{sync}(z') \otimes \mathtt{end}(z')) \rightarrow \\ & \quad\quad (\mathtt{r\_env}(x, p, x', p', c) \parallel \textstyle\prod_{i \in 1..m} \mathtt{r\_env}(y_i, p_i, y'_i, p'_i, c)))); \\ & \quad\quad \mathtt{run}(z) \rightarrow !\mathtt{end}(z)) \end{aligned} \qquad (\text{R}_{\text{CDEF}})$$

where $n = |\tilde{y}| = |\tilde{y'}|$ and

$$\begin{aligned} \mathtt{consume}' & \stackrel{\mathtt{def}}{=} \textstyle\prod_{i \in 1..m} \mathtt{consume}(y_i, y'_i, p_i, c) \parallel \\ & \quad \mathtt{ref}(x, nil, \mathtt{none}, \mathtt{ndg}) \rightarrow \mathtt{ref}(x', o_{new}, \mathtt{unq}, \mathtt{ndg}) \otimes \mathtt{ct}(o_{new}, s(\mathbf{0})) \\ \mathtt{gparam\text{-}init} & \stackrel{\mathtt{def}}{=} \textstyle\bigotimes_{i \in 1..k} !\mathtt{gparam}(c\_pg_i, o_{new}, g_i) \\ \mathtt{fields\text{-}init} & \stackrel{\mathtt{def}}{=} !\mathtt{field}(u_1, o_{new}, c\_a_1) \otimes \mathtt{ref}(u1, nil, \mathtt{none}, \mathtt{ndg}) \otimes \ldots \otimes \\ & \quad !\mathtt{field}(u_k, o_{new}, c\_a_k) \otimes \mathtt{ref}(u_k, nil, \mathtt{none}, \mathtt{ndg}) \end{aligned}$$

The mechanisms for parameter passing, executing the body $\hat{s}$ and restoring permissions are the same as in method definitions. The definition `consume'` is similar to `consume` in method definitions but, instead of using *consume*(x, x', p, c), we consume the constraint $\mathtt{ref}(x, nil, \mathtt{none}, \mathtt{nst})$, i.e., x in the statement $x := \mathtt{new}\ c\langle\tilde{g}\rangle(\tilde{y})$

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

```
8  main(){
9     let collection c, stats s in
10       c := new collection()
11       s := new stats()
12       c.compStats(s)
13       c.compStats(s)
14       c.removeDuplicates()
15  end}
```

Fig. 11. Main program for Example 4.1. Class definitions are in Figure 1.

is restricted to be a null reference. Moreover, the internal variable x' points to the newly created object o_{new} with permission unique.

The definition `gparam-init` allows us to establish the link between the new object o_{new} and the group parameters. In the constraint $\mathtt{gparam}(c\_pg_i, o_{new}, g_i)$, the constant symbol c_pg_i corresponds to the name defined for the DG parameter pg_i of the class $c\langle pg_1, \ldots, pg_k\rangle$ and g_i is the current DG passed as parameter.

The initialization of fields is controlled by the definition `fields-init`. The added constraint $\mathtt{field}(u_i, o_{new}, c\_a_i)$ establishes the link between the field $o_{new}.a_i$ and the null reference u_i.

Let us present a couple of examples to show the proposed model in action.

Example 4.1 (*Access permission flow*)
Assume the class definitions *stats* and *collection* in Figure 1 and the main body in Figure 11. The `lcc` agent modeling the statement in line 10 calls $collection\_collection(c, z_{10})$, which triggers the execution of the body of the constructor (see Rules $\mathrm{R_{CDEF}}$ and $\mathrm{R_{CALL}}$). Variable z_{10} is the local variable used to synchronize with the rest of the program (see Rule $\mathrm{R_{COMP}}$). Once the agents modeling the statements in lines 10 and 11 are executed, the following store is observed:

$$\exists c, s, o_c, o_s(\mathtt{ref}(c, o_c, \mathtt{unq}, \mathtt{ndg}) \otimes \mathtt{ref}(s, o_s, \mathtt{unq}, \mathtt{ndg}) \otimes \mathtt{ct}(o_c, s(\mathbf{0})) \otimes \mathtt{ct}(o_s, s(\mathbf{0})))$$

Hence, c (resp. s) points to o_c (resp. o_s) with a unique permission. In `c.compStats()`, c requires an immutable permission to o_c. The axiom $downgrade_2$ is used to entail the guard of `consume` in the definition of the method. Let c' be the representation of c inside the method (see Rule $\mathrm{R_{MDEF}}$). We notice that when the body of the method is being executed, both c and c' have an immutable permission to o_c, i.e., the store contains the tokens

$$\mathtt{ref}(c, o_c, \mathtt{imm}, \mathtt{ndg}) \otimes \mathtt{ref}(c', o_c, \mathtt{imm}, \mathtt{ndg}) \otimes \mathtt{ct}(o_c, s(s(\mathbf{0})))$$

Before executing the body of method *compStats* constraint $\mathtt{sync}(z_{12})$ is added, so as to allow possible concurrent executions in the main body (see Rule $\mathrm{R_{COMP}}$). Hence, the agent modeling the statement in line 13 can be executed and we have a store with three references with immutable permission to object o_c, namely, c, c' as before, and c'', the representation of c inside the method *print*. Now, once constraint $\mathtt{sync}(z_{13})$ is added by the definition of *print*, the process representing the statement in line 14 can be executed. However, this call requires c to have a unique permission to o_c which is not possible since the axiom $upgrade_2$ requires that c is the sole reference to o_c. Hence, the guard `consume` for this call is delayed (synchronized) until the permissions on c' and c'' are consumed and restored to the environment (see `r_env`

Line	**Store**	**Observations**
7	$\mathtt{dg}(g, \mathtt{atm}, \mathtt{nst})$	See Rule $\mathrm{R}_{\mathrm{NEWG}}$.
8	$\mathtt{dg}(g, \mathtt{atm}, \mathtt{nst}) \otimes$ $\mathtt{ref}(s, \mathit{nil}, \mathtt{none}, \mathtt{ndg}) \otimes$ $\mathtt{ref}(o1, \mathit{nil}, \mathtt{none}, \mathtt{ndg}) \otimes$ $\mathtt{ref}(o1, \mathit{nil}, \mathtt{none}, \mathtt{ndg})$	s, $o1$ and $o2$ are null references (see Rule $\mathrm{R}_{\mathrm{LOC}}$).
10	$\mathtt{dg}(g, \mathtt{conc}, z_{10}) \otimes$ $\mathtt{dg}(g, \mathtt{conc}, z_{11}) \otimes$ $\mathtt{ref}(s, \mathit{nil}, \mathtt{none}, \mathtt{ndg}) \otimes \cdots$	The $\mathtt{atm}$ DGAP on g is consumed and split into $\mathtt{conc}$ permission for statements in lines 11-12 (see Rule $\mathrm{R}_{\mathrm{SPLIT}}$)
Before 13(1)	$\mathtt{dg}(g, \mathtt{conc}, z_{10}) \otimes \cdots \otimes$ $\mathtt{ref}(s, o_s, \mathtt{shr}, g) \otimes$ $\mathtt{ref}(o1, oo_1, \mathtt{unq}, \mathtt{ndg}) \otimes$ $\mathtt{ref}(o2, oo_2, \mathtt{unq}, \mathtt{ndg})$	Variables s and obs are instantiated. The atomic DGAP has not been restored yet and then, statement in line 13 has to wait.
Before 13(2)	$\mathtt{dg}(g, \mathtt{atm}, \mathtt{nst}) \otimes$ $\mathtt{ref}(s, o_s, \mathtt{shr}, g) \otimes \cdots$	Concurrent DGAPs are consumed and the atomic permission on g is restored (see Rule $\mathrm{R}_{\mathrm{SPLIT}}$).
Before 16	$\mathtt{dg}(g, \mathtt{conc}, z_{14}) \otimes$ $\mathtt{dg}(g, \mathtt{conc}, z_{15}) \otimes$ $\mathtt{ref}(s, o_s, \mathtt{shr}, g) \otimes$ $\mathtt{ref}(s', o_s, \mathtt{shr}, g) \otimes$ $\mathtt{ref}(s'', o_s, \mathtt{shr}, g) \otimes$ $\mathtt{ref}(o1, oo_1, \mathtt{unq}, \mathtt{ndg}) \otimes$ $\mathtt{ref}(o2, oo_2, \mathtt{unq}, \mathtt{ndg})$	There are 3 references to o_s: s, s' and s''. The last two correspond to the internal representation of s in the calls to method *update* (see Rule Rule $\mathrm{R}_{\mathrm{MDEF}}$). Then, such methods can be executed concurrently. We also see that the $\mathtt{atm}$ DGAP was split into $\mathtt{conc}$ DGAP for statements 14 and 15.
16	$\mathtt{dg}(g, \mathtt{atm}, \mathtt{nst}) \otimes$ $\mathtt{ref}(s, o_s, \mathtt{shr}, g) \otimes$ $\mathtt{ref}(o1, oo_1, \mathtt{unq}, \mathtt{ndg}) \otimes$ $\mathtt{ref}(o2, oo_2, \mathtt{unq}, \mathtt{ndg})$	In the end, s is the sole reference to o_s (see $\mathtt{r\_env}$ in Rule $\mathrm{R}_{\mathrm{MDEF}}$) and the atomic DGAP on g is reestablished.

Fig. 12. Constraints added by the processes in Example 4.2 (AP code in Figure 2).

in Rule $\mathrm{R}_{\mathrm{MDEF}}$). We then observe that statements in lines 12 and 13 can be executed concurrently but the statement in line 14 is delayed until the termination of the previous ones.

Example 4.2 (*Data group permissions flow*)
Now consider the program in Figure 2. Figure 12 shows the stores generated by the model of this program. We omit some tokens for the sake of readability.

Example 4.3 (*Deadlocks*)
Let us consider the following implementation for the method *compStats* in the class *collection* (see Figure 1)

```
compStats(s)  imm(this), unq(s) => imm(this), unq(s) {
   ...
   c.sort()
   ... }
```

Consider the call `c.compStats(s)` and suppose that, in the `lcc` model, variable c points to the object o_c. When the *compStats* method is invoked, the immutable permission is divided between the external reference c and the internal reference c'. For this reason, inside the method, reference c' cannot acquire a unique permission for the invocation of method *sort* which then blocks. Our analysis will thus inform that there is a deadlock, unless, e.g., the program includes the statement $c\langle g\rangle := \mathit{nil}$ to discard the permission of c to o_c.

Consider now the following definition of the same method:

```
compStats(s)  unq(this), unq(s) => unq(this), unq(s) {
  ...
```

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

```
 c.sort()
...}
```

When *compStats* is invoked, the unique permission is transferred from reference c to (the internal) reference c'. The invocation of method *sort* has thus the right permissions to be executed and it does not block.

4.3 The model as a runnable specification

Models based on the `ccp` paradigm can be regarded as runnable specifications, and so we can observe how permissions evolve during program execution by running the underlying `lcc` model. We implemented an interpreter of `lcc` in Java and used Antlr (`http://www.antlr.org`) to generate a parser from AP programs into `lcc` processes following our encoding. The resulting `lcc` process is then executed and a program trace is output. The interpreter and the parser have been integrated into Alcove (AP Linear COnstraints VErifier) Animator, a web application freely available at `http://subsell.logic.at/alcove2/`. The URL further includes all the examples presented in this section. In the following, we explain some outputs of the tool.

Example 4.4 (*Trace of access permissions*)
The program in Figure 1 generates the trace depicted in Figure 13. For verification purposes, the implementation extends the predicates `act`(·), `run`(·), and `end`(·) to include also the variable that called the method, the name of the method, and the number of line of the call. Note for instance that the call to *print* (line 9 in Figure 13) was marked while the method *sort* was running (line 7). Nevertheless, the execution of *print* (line 11) must wait until *sort* terminates (line 10). In this trace, the constructor *stats* (line 5) runs in parallel with *sort* (line 7). Finally, the execution of *removeDuplicates* (line 17) is delayed until the methods *print* (line 13) and *compStats* (line 16) terminate. Lines 20 and 21 show that both c and s end with a unique permission to objects `o_4774` and `o_79106`, respectively (the numbers that follow the variable names are generated each time a local variable is created to avoid clash of names).

Example 4.5 (*Deadlock detection*)
Let us assume now the class definitions in Figure 1 and the following `main`:

```
8  main(){
9   group<g>
10  let collection c, stats s, stats svar in
11     c := new collection()
12     s := new stats()
13     svar<g> := s
14     c.compStats(s)
15     c.compStats(svar)
16 end}
```

The assignment in line 13 aliases *svar* and s so they share the same permission afterwards. Therefore, s cannot recover the unique permission to execute the statement in line 14, thus leading to a permission deadlock. This bug is detected by Alcove as depicted in Figure 14 (line 13). Observe in the trace that *compstats*

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

```
act(C_628,collection_collection,line 10 (Z_PAR_814))
run(C_628,collection_collection,line 10 (Z_PAR_814))
act(S_729,stats_stats,line 11 (Z_PAR_915))
end(C_628,collection_collection,line 10 (Z_PAR_814))
run(S_729,stats_stats,line 11 (Z_PAR_915))
act(C_628,collection_sort,line_12 (Z_PAR_1016))
run(C_628,collection_sort,line_12 (Z_PAR_1016))
end(S_729,stats_stats,line 11 (Z_PAR_915))
act(C_628,collection_print,line_13 (Z_PAR_1117))
end(C_628,collection_sort,line_12 (Z_PAR_1016))
run(C_628,collection_print,line_13 (Z_PAR_1117))
act(C_628,collection_compStats,line_14 (Z_PAR_1218))
end(C_628,collection_print,line_13 (Z_PAR_1117))
run(C_628,collection_compStats,line_14 (Z_PAR_1218))
act(C_628,collection_removeDuplicates,line_15 (Z_PAR_1319))
end(C_628,collection_compStats,line_14 (Z_PAR_1218))
run(C_628,collection_removeDuplicates,line_15 (Z_PAR_1319))
end(C_628,collection_removeDuplicates,line_15 (Z_PAR_1319))

[ref(C_628,O_4774,unq,ng), ct(O_4774,1)]
[ref(S_729,O_79106,unq,ng), ct(O_79106,1)]
ok()
567 processes Created
```

Fig. 13. Trace generated by the program in Figure 1 (Example 4.4).

```
1 act(C_644,collection_collection,line 10 (Z_PAR_928))
2 run(C_644,collection_collection,line 10 (Z_PAR_928))
3 act(S_745,stats_stats,line 11 (Z_PAR_1029))
4 run(S_745,stats_stats,line 11 (Z_PAR_1029))
5 end(C_644,collection_collection,line 10 (Z_PAR_928))
6 end(S_745,stats_stats,line 11 (Z_PAR_1029))
7 act(C_644,collection_compStats,line_13 (Z_PAR_1231))
8 [Killed] ask endc(line_13 (Z_PAR_1231)) then ...
9 [Killed] ask sync(line_14 (Z_PAR_1332)) then ...
10 [Killed] ask ref(S_745,O_142,unq,ng) then ... + ask
11 ...
12 404 processes Created
13 [FAIL] Token ok not found. End of the program not reached.
14
15 VARIABLES
16 C_644 -> O_6491. imm:ng
17 S_745 -> O_96123. shr:GRP_461
18 SVAR_846 -> O_96123. shr:GRP_461
19 INNER_136172 -> O_6491. imm:ng
```

Fig. 14. Trace generated by the program in Example 4.5.

is called (line 7 in the trace) but not executed. Furthermore, both *s* and *svar* have a share permission on the same pointed object (lines 17 and 18). Moreover, both *c* (`c_644`) and its internal representation inside *compStats* (`inner_136172`) have an immutable permission on object `o_6491` (lines 16 and 19). Lines 8–11 show the suspended `lcc` processes in the end of the computation that were killed by the scheduler. Particularly, line 10 shows that there is an ask agent trying to consume a unique permission on object `O_142` pointed by `S_745`.

4.4 Adequacy of the encoding

In this section, we present some invariant properties of the encoding and prove it correct. There are three key arguments in our proofs:

Observation 4.1 (*Ask agents*)
(1) The ask agents controlling both the APs (Proposition 4.1) and the state of statements (Proposition 4.2) are of the form $c \rightarrow P$ where P is a tell agent (and not, e.g., a parallel composition). Hence, in one single transition, the encoding consumes and produces the needed tokens to maintain the invariants (ruling out intermediate states where the property might not hold). Moreover, (2) such ask agents are preceded by the sequential composition operator ";." This means that, before consuming the needed constraints, some action must have been finished. In particular, (3) the ask agent $\texttt{act}(z) \rightarrow \texttt{run}(z)$ is executed only when the needed permissions are consumed and the ask agent $\texttt{run}(z) \rightarrow \texttt{end}(z)$ is executed only after restoring the consumed permissions (Rules R$_{\text{ALIAS}}$, R$_{\text{CDEF}}$ and R$_{\text{MDEF}}$).

The following invariants show that the lcc model correctly keeps track of the variables and their corresponding pointed objects.

Proposition 4.1 (*Invariants on references*)
Let S be an AP program and $\mathscr{D}.P$ its corresponding translation into lcc. Assume that $(\emptyset; P; 1) \longrightarrow^* (X; \Gamma; c)$. The following holds:

(1) If $c \vdash \texttt{ref}(x, o, \texttt{unq}, \texttt{ndg})$, then $c \vdash \texttt{ct}(o, s(0))$.
(2) If $c \vdash \texttt{ref}(x, o, p, g)$ and $p \in \{\texttt{shr}, \texttt{imm}\}$, then there exists $n > 0$ s.t. $c \vdash \texttt{ct}(o, n)$.
(3) If $c \vdash \texttt{ref}(x, o, p, g)$ and $c \vdash \texttt{ref}(x, o', p', g')$, then $o' = o$, $p' = p$ and $g' = g$.
(4) If $c \vdash \texttt{ref}(x, nil, p, g)$, then $p = \texttt{none}$ and $g = \texttt{ndg}$.
(5) (**counting**) if $c \vdash \texttt{ct}(o, n)$, then
    (a) for all $m \leqslant n$, $c \vdash \exists x_1, p_1, g_1 \ldots, x_m, p_m, g_m \bigotimes_{i \in 1..m} \texttt{ref}(x_i, o, p_i, g_i)$, and
    (b) for all $m > n$, $c \not\vdash \exists x_1, p_1, g_1 \ldots, x_m, p_m, g_m \bigotimes_{i \in 1..m} \texttt{ref}(x_i, o, p_i, g_i)$.

Proof
An inspection of the encoding reveals that the rules R$_{\text{ALIAS}}$ and R$_{\text{LOC}}$ and the definitions consume, r_env, and fields-init are the only ones that consume/produce $\texttt{ref}(\cdot)$ and $\texttt{ct}(\cdot)$ constraints. For any newly created variable, R$_{\text{LOC}}$ and fields-init add the needed $\texttt{ref}(\cdot)$ token adhering to item 4. Moreover, the ask agents in the above rules/definitions adhere to the conditions in Observation 4.1. Therefore, if the agent $c \rightarrow P$ consumes a constraint of the form $\texttt{ref}(x, o, p, g)$, the **tell** process P adds the needed constraints to maintain correct the counting of references to o. □

The next proposition shows that the encoding correctly captures the state of statements.

Proposition 4.2 (*States*)
Let $State = \{\texttt{act}, \texttt{run}, \texttt{end}\}$, S be an AP program and $\mathscr{D}.P$ its corresponding lcc translation. Consider an arbitrary execution starting at P:

$$(\emptyset; P; 1) \longrightarrow (X_1; \Gamma_1; c_1) \longrightarrow (X_2; \Gamma_2; c_2) \longrightarrow \cdots \longrightarrow (X_n; \Gamma_n; c_n)$$

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

Let $z \in X_n$, $st \in State$, $x \in 1 \dots n$, and assume that $c_x \vdash st(z)$. Then,

(1) (**no confusion**) for all $st' \in State \setminus \{st\}$, $c_x \nvdash st'(z)$;
(2) (**state ordering**) there exists $i \in 1..x$ such that
  (a) (**init**) for all $k \in [1, i)$ and $st' \in State$, $c_k \nvdash st'(z)$;
  (b) (**continuity**) for all $k \in [i, n]$, $c_k \vdash st'(z)$ for some $st' \in State$;
  (c) (**act**) if $c_n \vdash \texttt{act}(z)$, then for all $k \in [i, n]$, $c_k \vdash \texttt{act}(z)$;
  (d) (**act until run**) if $c_n \vdash \texttt{run}(z)$, then there exist two non-empty intervals $A = [i \dots j_r)$ and $R = [j_r, n]$ s.t. for all $k \in A$, $c_k \vdash \texttt{act}(k)$ and for all $k \in R$, $c_k \vdash \texttt{run}(k)$;
  (e) (**run until end**) if $c_n \vdash \texttt{end}(z)$, then there are three non-empty intervals $A = [i..j_r)$, $R = [j_r, \dots, j_e)$, $E = [j_e, \dots, j_n]$ s.t. A and R are as above and for all $k \in E$, $c_k \vdash \texttt{end}(z)$.

Proof
Note that the token $\texttt{act}(z)$ is added when the encoding of a statement is activated (`wrap`). An inspection of the encoding shows that the ask agents controlling the state of statements adhere to conditions in Observation 4.1. Since each executed statement uses a freshly created variable z (see R_{COMP}), we can show that, for any z and multiset Γ_x, Γ_x can contain at most one of each of such ask agents (using z). Hence, for all $st \in State$, if $s(z)$ is consumed from the store c_x, the store c_{x+1} must contain the next state $st'(z)$. This guarantees the correct ordering of states. □

We conclude by showing that the encoding enforces the execution of statements according to the AP specification. More precisely, the activation of a statement s is delayed until its (lexical) predecessor has successfully consumed the needed permissions, the execution of s is delayed until its required permissions are available (and consumed), signaling the termination of s is delayed until all the consumed permissions are restored.

Theorem 4.1 (*Adequacy*)
Let S be an AP program and $\mathscr{D}.P$ its corresponding `lcc` translation. Let s_i and s_j be two sentences that occur in the same block and s_j is lexically after s_i. Then,

(1) (safety) s_i and s_j are in conflict iff for any reachable configuration $(X; \Gamma; c)$ from $(X; P; 1)$, $c \vdash \texttt{run}(z_{s_j})$ implies $c \vdash \texttt{end}(z_{s_i})$;
(2) (concurrency) s_i is not in conflict with s_j iff there exists a reachable configuration $(X; \Gamma; c)$ from $(X; P; 1)$ s.t. $c \vdash \texttt{run}(z_{s_i})$ and $c \vdash \texttt{run}(z_{s_j})$.

Proof
The execution of assignments, the call to methods/constructors and the beginning of blocks are the statements we have to synchronize in the encoding. Note that rules R_{ALIAS}, R_{CDEF}, R_{MDEF}, R_{SPLIT}, and R_{COMP} adhere to conditions in Observation 4.1. In particular, condition (3) shows that the changes of states are controlled by acquiring/releasing permissions.

($\Rightarrow$) (1) Assume that s_i and s_j both require a unique permission on the same object (the other kind of conflicts are similar). From rule R_{COMP}, we know that s_i first consumes its permissions (before enabling s_j). From Propositions 4.1 and 4.2, we can show that s_j cannot move to the state $\mathtt{run}$ until s_i moves to state $\mathtt{end}$.

(2) If there are no conflicting resources, then both processes may successfully consume the needed permissions from the store. Consider the following trace: the encoding of s_i consumes the needed permissions, adds $\mathtt{run}(z_{s_i})$, and the $\mathtt{sync}(z_{s_i})$ token. Then, the encoding of s_2 can start its execution (consuming $\mathtt{sync}(z_{s_i})$), consumes the needed permissions, and adds $\mathtt{run}(z_{s_j})$ to the store.

($\Leftarrow$) For (1), assume that in any reachable configuration $(X;\Gamma;c)$, $c \vdash \mathtt{run}(z_{s_j})$ implies $c \vdash \mathtt{end}(z_{s_i})$. By Proposition 4.2, we know that $c \nvdash \mathtt{run}(z_{s_i})$. Since the encoding maintains correct the number of references (in the sense of Proposition 4.1), there is no reachable store able to entail the permissions needed for both s_i and s_j. Hence, there is a conflicting access in s_i and s_j. The case (2) follows from a similar argument. □

5 Logical meaning of access permissions

Besides playing the role of executable specifications, ccp-based models can be declaratively interpreted as formulas in logic (Saraswat 1993; de Boer *et al.* 1997; Fages *et al.* 2001; Nielsen *et al.* 2002; Olarte and Pimentel 2017). This section provides additional mechanisms and tools for verifying properties of AP-based programs. More concretely, we take the `lcc` agents generated from the AP program and translate them as an ILL formula. Then, a property specified in ILL is verified with the Alcove LL Prover, a theorem prover implemented on top of Teyjus (`http://teyjus.cs.umn.edu`), an implementation of λ-Prolog (Nadathur and Miller 1988; Miller and Nadathur 2012).

Our analyses are based on reachability properties, i.e., we verify the existence of reachable `lcc` configurations satisfying some conditions. It turns out that this is enough for verifying interesting properties of AP programs. For instance, we can check whether a program is dead-lock free or whether two statements can be executed concurrently.

5.1 Agents as formulas

The logical interpretation of `lcc` agents as formulas in ILL is defined with the aid of a function $\mathscr{L}[\![\cdot]\!]$ defined in Figure 15 (Fages *et al.* 2001). As expected, parallel composition is identified with multiplicative conjunction and ask processes correspond to linear implications. Moreover, process definitions are (universally quantified) implications to allow the unfolding of its body.

In what follows, we will show how to use logic in order to have a better control of the operational flow and, therefore, be able to verify properties of AP programs.

The first step consists of interpreting the `lcc` model in Section 4 as ILL formulas via $\mathscr{L}[\![\cdot]\!]$. We shall call *definition clauses* to the encoding of process definitions

$$
\begin{array}{lll}
\mathcal{L}[\![c]\!] & = & c \\
\mathcal{L}[\![\sum_{i\in I} \forall \widetilde{x}_i(c_i \to P_i)]\!] & = & \&_{i\in I}(\forall \widetilde{x}_i(c_i \multimap \mathcal{L}[\![P]\!])) \\
\mathcal{L}[\![p(\widetilde{x}) \stackrel{\Delta}{=} P]\!] & = & \forall \widetilde{x}.p(\widetilde{x}) \multimap \mathcal{L}[\![P]\!]
\end{array}
\qquad
\begin{array}{lll}
\mathcal{L}[\![P \parallel Q]\!] & = & \mathcal{L}[\![P]\!] \otimes \mathcal{L}[\![Q]\!] \\
\mathcal{L}[\![\exists x(P)]\!] & = & \exists x.(\mathcal{L}[\![P]\!]) \\
\mathcal{L}[\![p(\widetilde{x})]\!] & = & p(\widetilde{x})
\end{array}
$$

Fig. 15. Interpretation of `lcc` processes as ILL formulas.

$$
\begin{array}{l}
assg(x,y,z,gt) \multimap \\
\qquad \exists z_1.(\forall o,n,g.(\mathtt{ref}(x,o,\mathtt{none},\mathtt{ndg}) \multimap 1 \otimes \mathtt{sync}(z_1)\& \\
\qquad \mathtt{ref}(x,o,\mathtt{unq},\mathtt{ndg}) \otimes \mathtt{ct}(o,s(n)) \multimap \mathtt{ct}(o,n) \otimes \mathtt{sync}(z_1)\& \\
\qquad \mathtt{ref}(x,o,\mathtt{shr},g) \otimes \mathtt{ct}(o,s(n)) \multimap \mathtt{ct}(o,n) \otimes \mathtt{sync}(z_1)\& \\
\qquad \mathtt{ref}(x,o,\mathtt{imm},\mathtt{ndg}) \otimes \mathtt{ct}(o,s(n)) \multimap \mathtt{ct}(o,n) \otimes \mathtt{sync}(z_1))\otimes \\
\mathtt{sync}(z_1) \multimap \exists z_2.(\mathtt{ref}(y,nil,\mathtt{none},\mathtt{ndg}) \multimap \\
\qquad \mathtt{ref}(x,nil,\mathtt{none},\mathtt{ndg}) \otimes \mathtt{ref}(y,nil,\mathtt{none},\mathtt{ndg}) \otimes \mathtt{sync}(z_2) \\
\qquad \& \forall o,n.(\mathtt{ref}(y,o,\mathtt{unq},\mathtt{ndg}) \otimes \mathtt{ct}(o,s(\mathbf{0})) \multimap \\
\qquad\quad \mathtt{ref}(y,o,\mathtt{shr},gt) \otimes \mathtt{ref}(x,o,\mathtt{shr},gt) \otimes \mathtt{ct}(o,s(s(\mathbf{0}))) \otimes \mathtt{sync}(z_2)) \\
\qquad \& \forall o,n.(\mathtt{ref}(y,o,\mathtt{shr},gt) \otimes \mathtt{ct}(o,n) \multimap \\
\qquad\quad \mathtt{ref}(y,o,\mathtt{shr},gt) \otimes \mathtt{ref}(x,o,\mathtt{shr},gt) \otimes \mathtt{ct}(o,s(n)) \otimes \mathtt{sync}(z_2)) \\
\qquad \& \forall o,n.(\mathtt{ref}(y,o,\mathtt{imm},\mathtt{ndg}) \otimes \mathtt{ct}(o,n) \multimap \\
\qquad\quad \mathtt{ref}(y,o,\mathtt{imm},\mathtt{ndg}) \otimes \mathtt{ref}(x,o,\mathtt{imm},\mathtt{ndg}) \otimes \mathtt{ct}(o,s(n)) \otimes \mathtt{sync}(z_2))\otimes \\
\mathtt{sync}(z_2) \multimap \exists z_3.(\mathtt{act}(z) \multimap \mathtt{run}(z) \otimes \mathtt{sync}(z_3)\otimes \\
\mathtt{sync}(z_3) \multimap \exists z_4.(\mathtt{run}(z) \multimap \mathtt{sync}(z_4) \otimes !\,\mathtt{end}(z))))). \\
\mathcal{L}[\![P]\!] = \exists c,s,svar,z,z_1,z_2,z_3,z_4,z_5.(\mathtt{ref}(c,nil,\mathtt{none},\mathtt{ndg})\otimes \\
\qquad \mathtt{ref}(s,nil,\mathtt{none},\mathtt{ndg}) \otimes \mathtt{ref}(svar,nil,\mathtt{none},\mathtt{ndg}) \otimes \mathtt{sync}(z_1) \otimes !\,\mathtt{end}(z_1)\otimes \\
\qquad \mathtt{sync}(z_1) \multimap collection\_collection(c,z_2)\otimes \\
\qquad \mathtt{sync}(z_2) \multimap stats\_stats(s,z_3)\otimes \\
\qquad \mathtt{sync}(z_3) \multimap assig(svar,s,z_4)\otimes \\
\qquad \mathtt{sync}(z_4) \multimap collection\_compStats(c,s,z_5)\otimes \\
\qquad \mathtt{sync}(z_5) \multimap \mathtt{sync}(z)) \otimes (\bigotimes_{i\in 1..5} \mathtt{end}(z_i)) \multimap !\,\mathtt{end}(z)
\end{array}
$$

Fig. 16. Encoding of `assg` definition and the `main` body in Example 4.5.

of the form $p(\tilde{x}) \stackrel{\Delta}{=} P$ (i.e., assignment and constructor and method definitions in our encoding) and we shall include them in a theory Δ, together with the axioms of upgrade and downgrade in Figure 7. The next example illustrates this translation. For the sake of readability, we shall omit empty synchronizations such as $\mathtt{sync}(z) \otimes (\mathtt{sync}(z) \to 1)$.

Example 5.1 (*Agents as formulas*)

Consider the following `lcc` process definition resulting from the encoding of the constructor of class collection in Figure 1:

$$
\begin{array}{l}
collection\_collection(x,z) \stackrel{\Delta}{=} \exists x', o_{new}(1; \mathtt{ref}(x,nil,\mathtt{none},\mathtt{ndg}) \to \mathtt{ref}(x',o_{new},\mathtt{unq},\mathtt{ndg}) \otimes \mathtt{ct}(o_{new},s(0)); 1; \mathtt{sync}(z); \\
\qquad \mathtt{act}(z) \to \mathtt{run}(z); \exists z'(\mathtt{sync}(z') \otimes !\,\mathtt{end}(z') \parallel (\mathtt{sync}(z') \otimes \mathtt{end}(z')) \to \\
\qquad\quad \forall o'(\mathtt{ref}(x',o',\mathtt{unq},\mathtt{ndg}) \otimes \mathtt{ct}(o',s(0)) \to \mathtt{ref}(x,o',\mathtt{unq},\mathtt{ndg}) \otimes \mathtt{ct}(o',s(0))); \\
\qquad\quad \mathtt{run}(z) \to !\,\mathtt{end}(z)))
\end{array}
$$

where the first 1 corresponds to the empty parallel composition in `gparam-init`. From now on, for the sake of readability, we will identify $A \equiv A \otimes 1$. This process definition gives rise to the following (universally quantified) definition clause:

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underbrace{\mathit{collection\_collection}(x,z) \multimap}_{1} \exists x', o_{new}, w_1.(\underbrace{\mathtt{ref}(x, \mathit{nil}, \mathtt{none}, \mathtt{ndg}) \multimap}_{2}(\mathtt{ref}(x', o_{new}, \mathtt{unq}, \mathtt{ndg}) \otimes \mathtt{ct}(o_{new}, s(\mathbf{0})) \otimes \mathtt{sync}(w_1) \otimes} \\
\quad \underbrace{\mathtt{sync}(w_1) \multimap}_{3} & \exists w_2.\mathtt{sync}(z) \otimes \mathtt{sync}(w_2) \otimes \\
\quad \underbrace{\mathtt{sync}(w_2) \multimap}_{4} & \exists w_3. \underbrace{\mathtt{act}(z) \multimap}_{5}(\mathtt{run}(z) \otimes \mathtt{sync}(w_3)) \otimes \\
\qquad \underbrace{\mathtt{sync}(w_3) \multimap}_{6} & \exists z', w_4.\mathtt{sync}(z') \otimes\, !\,\mathtt{end}(z') \otimes \underbrace{(\mathtt{sync}(z') \otimes \mathtt{end}(z') \multimap}_{7} \\
 & \forall o'.(\mathtt{ref}(x', o', \mathtt{unq}, \mathtt{ndg}) \otimes \mathtt{ct}(o', s(\mathbf{0})) \multimap \mathtt{ref}(x, o', \mathtt{unq}, \mathtt{ndg}) \otimes \mathtt{ct}(o', s(\mathbf{0})) \otimes \mathtt{sync}(w_4))) \otimes \\
\qquad \underbrace{\mathtt{sync}(w_4) \multimap}_{8} & \mathtt{run}(z) \multimap\, !\,\mathtt{end}(z))).
\end{array}
$$

The underlying brackets will be used in Section 5.4 for determining the complexity of decomposing this formula. The theory Δ contains the definition clause above and the definition clauses for the other methods and constructors in Figure 1 (i.e., *collection_sort*, *collection_print*, etc). Δ also contains the axioms for upgrading and downgrading permissions and the definition clause resulting from the process definition `assg` in Figure 8. In Figure 16, we show the encoding for `assg` as well as the encoding $\mathscr{L}[\![P]\!]$ of the main program in Example 4.5.

5.2 *Focusing and adequacy*

In this section, we show that the translations presented in the last section are neat, in the sense that one computational step corresponds to one focused phase in proofs (Andreoli 1992). This will not only guarantee that our encodings are *adequate* (in the sense that logical proofs mimics *exactly* computations), but also it will provide an elegant way of measuring the complexity of computations via complexity of derivations (see Section 5.4).

The approach for this section will be intuitive. The reader interested in the formalization of focusing and various levels of adequacy between ILL and `lcc` can check the details in (Olarte and Pimentel 2017).

Let us start by analyzing the following two right rules in ILL (for the additive and multiplicative conjunctions):

$$
\frac{\Gamma \longrightarrow F \quad \Gamma \rightarrow G}{\Gamma \longrightarrow F \& G} \;\&_R \qquad \frac{\Gamma_1 \longrightarrow F \quad \Gamma_2 \rightarrow G}{\Gamma_1, \Gamma_2 \longrightarrow F \otimes G} \;\otimes_R
$$

Reading these rules bottom-up, while the first copies the contexts, the second involves a choice of which formulas should go to left or right premises. Computationally, these behaviors are completely different: while the price to pay on applying $\&_R$ is just the duplication of memory needed to store formulas in the context, in $\otimes_R$ one has to decide on how to split the context, and this has exponential cost. These rules are very different from the proof theoretical point of view as well: the first rule turns out to be *invertible* in ILL, while the second is not. This implies that the rule $\&_R$ can be applied *anywhere* in the proof, and this will not affect provability. On the other hand, $\otimes_R$ is not invertible and its application may involve backtracking.

The same analysis can be done to all other rules in ILL, giving rise to two disjoint classes of rules: the invertible ones, that can be applied eagerly, $\{\top_R, 1_L, \otimes_L, \&_R, \multimap_R, \oplus_L, \exists_L, \forall_R, C\}$ and the non-invertible ones $\{1_R, \otimes_R, \&_L, \multimap_L, \oplus_R, \exists_R, \forall_L, W, D, \mathsf{prom}\}$.

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

This separation induces a two-phase proof construction: a *negative*, where *no backtracking* on the selection of inference rules is necessary, and a *positive*, where choices within inference rules can lead to failures for which one may need to backtrack.

An intuitive notion of focusing can be then stated as a proof is *focused* if, seen bottom-up, it is a sequence of alternations between maximal negative and positive phases.

Focusing is enough for assuring that the encoding presented in Section 5.1 is, indeed, adequate.

Theorem 5.1 (*Adequacy, (Olarte and Pimentel 2017)*)
Let P be a process, Ψ be a set of process definitions, and Δ be a set of non-logical axioms. Then, for any constraint c, $(\emptyset; P; 1) \longrightarrow^* (X; \Gamma; d)$ with $\exists X.d \vdash c$ iff there is a proof of the sequent $!\mathscr{L}[\![\Psi, \Delta]\!], \mathscr{L}[\![P]\!] \longrightarrow c \otimes \top$ in ILLF. Moreover, one focused logical phase corresponds exactly to one operational step.

This result, together with Theorem 4.1, shows that AP can be adequately encoded in ILL in a natural way. In the present work, we are more interested in using logic in order to verify properties of the computation, as clarified in the next example.

Example 5.2 (*Traces, proofs, and focusing*)
Let $A_1 = a \to b \to (a \otimes b)$, $A_2 = b \to a \to \texttt{ok}$, and $P = a \otimes b \parallel A_1 \parallel A_2$. The operational semantics of `lcc` dictates that there are two possible transitions leading to the store `ok`. Both of such transitions start with the tell action $a \otimes b$:

$$\begin{array}{ll}\texttt{Derivation 1:}\ \langle \emptyset; P; 1\rangle & \longrightarrow^* \langle \emptyset; A_1 \parallel A_2; a\otimes b\rangle \longrightarrow^* \langle \emptyset; b \to (a\otimes b) \parallel A_2; b\rangle \\ & \longrightarrow^* \langle \emptyset; (a\otimes b) \parallel A_2; 1\rangle \longrightarrow^* \langle \emptyset; A_2; a\otimes b\rangle \longrightarrow^* \langle \emptyset; \cdot; \texttt{ok}\rangle \not\longrightarrow \\ \texttt{Derivation 2:}\ \langle \emptyset; P; 1\rangle & \longrightarrow^* \langle \emptyset; A_1 \parallel A_2; a\otimes b\rangle \longrightarrow^* \langle \emptyset; A_1 \parallel a \to \texttt{ok}; b\rangle \\ & \longrightarrow^* \langle \emptyset; A_1 \parallel \texttt{ok}; 1\rangle \longrightarrow^* \langle \emptyset; A_1; \texttt{ok}\rangle \not\longrightarrow \end{array}$$

Each of these transitions corresponds exactly to a focused proof of the sequent $\mathscr{L}[\![P]\!] \longrightarrow \texttt{ok} \otimes \top$: one focusing first on $\mathscr{L}[\![A_1]\!]$ and the other focusing first on $\mathscr{L}[\![A_2]\!]$.

On the other hand, there is also an interleaved execution of A_1 and A_2 that does not lead to the final store `ok`:

$$\begin{array}{ll}\texttt{Detivation 3:}\ \langle \emptyset; P; 1\rangle & \longrightarrow^* \langle \emptyset; A_1 \parallel A_2; a\otimes b\rangle \longrightarrow^* \langle \emptyset; b \to (a\otimes b) \parallel A_2; b\rangle \\ & \longrightarrow^* \langle \emptyset; b \to (a\otimes b) \parallel a \to \texttt{ok}; 1\rangle \not\longrightarrow \end{array}$$

This trace does not have any correspondent derivation in ILLF (see (Olarte and Pimentel 2017) for details).

This example is a good witness of a need for Alcove's *verifier*, other than just having an *animator*: an animator exhibits traces of possible executions without any pre-defined scheduling policy. One of such traces may not lead to the expected final store (as the `ok` above). On the other hand, the verifier would either fail (if a property is not provable) or succeed. In this last case, the proof produced by the prover corresponds *exactly* to a valid trace from the operational point of view.

Let us show an example of how focusing can control executions on a sequential composition.

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

Example 5.3 (*Focusing on a sequential composition*)
Consider the ILL interpretation of the sequential composition $P\,;Q$:

$$\mathscr{L}[\![P\,;Q]\!] = \exists z((\mathscr{L}[\![(\mathscr{C}[\![P]\!]_z)]\!]) \otimes (\mathtt{sync}(z) \multimap \mathscr{L}[\![Q]\!]))$$

This is a positive formula which will be on the left side of the sequent and $\exists$ and $\otimes$ will be decomposed in a negative phase. Once P is executed, we observe the invertible action of adding the atom $\mathtt{sync}(z)$ to the context. Then, one could change to a positive phase and focus on the negative formula $\mathtt{sync}(z) \multimap \mathscr{L}[\![Q]\!]$. This positive action needs to be synchronized with the context, consuming $\mathtt{sync}(z)$ in order to produce $\mathscr{L}[\![Q]\!]$.

In the following sections, we shall show that an ILLF prover is a complete decision procedure for reachability properties of the `lcc` agents resulting from our encodings. This will be useful to verify properties of the encoded AP program.

5.3 *Linear logic as a framework for verifying AP properties*

Let P be an agent and $\mathscr{L}[\![P]\!]$ its translation into ILL, producing a formula F together with a theory Δ. In order to verify a certain property $\mathscr{G}$, specified by an ILL formula G, we test if the sequent $!\,\Delta, F \longrightarrow G$ is provable.

First of all, observe that the fragment of ILL needed for encoding APs is given by the following grammar for guards/goals G and processes P:

$$\begin{aligned} G &:= a \mid G \otimes G \mid \exists x.G \\ P &:= a \mid !a \mid 1 \mid P \otimes P \mid P \& P \mid \forall x.G \multimap P \mid \exists x.P \mid !\forall \tilde{x}.(p(\tilde{x}) \multimap P) \end{aligned}$$

where a is an atomic formula. Observe that guards G do not consider banged formulas, i.e., agents are not allowed to *ask* banged constraints. A simple inspection on the encoding of Section 4 shows that processes in our case indeed belong to such fragment. We note also that formulas generated from this grammar exhibit the following properties:

(1) The left context in the sequent $!\,\Delta, F \longrightarrow G$ will be formed by P formulas.
(2) The right context will have only G formulas.
(3) Implications on the left can only introduce guards on the right side of a sequent. In fact, on examining a proof bottom-up, decomposing the implication on the sequent $\Gamma_1, \Gamma_2, B \multimap C \longrightarrow D$ will produce the premises $\Gamma_1, C \longrightarrow D$ and $\Gamma_2 \longrightarrow B$. Hence, it is important to guarantee that B (a guard) is a G (goal) formula.

Finally, notice that the fragment described above is undecidable in general, due to the presence of processes declarations (Lincoln *et al.* 1992). However, since we are considering AP programs adhering to the condition in Remark 2.1, our base language does not lead to cyclic recursive definitions. In the next section, we determine an upper bound for the complexity of proofs in Alcove's verifier. Therefore, we can show that provability in the resulting ILL translation is decidable (see Theorem 5.2).

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

5.4 Complexity analysis

Note that, when searching for proofs in the focused system, the only non-deterministic step is the one choosing the *focus formula* in a positive phase. This determines completely the complexity of a proof in ILLF and it justifies the next definition.

Definition 5 (*Proof depth*)
Let π be a proof in ILLF. The *depth* of π is the maximum number of positive phases along any path in π from the root.

Example 5.4 (*Complexity of formulas*)
Consider the formulas in Example 5.1. The depth of decomposing the definition clause $collection\_collection(x, z)$ into its literal or purely positive subformulas is 8. To see that, note that focusing in such a negative formula on the left will produce seven more nested positive phases in one of the branches of the proof: each one of these phases is signaled in the formula with an underlying bracket containing the respective number of the focused phase. The same holds when decomposing the clauses for $stats\_stats(s, z)$ and $collection\_compStats(c, s, z)$. As we will see later, decomposing $assig(svar, s, z)$ has a fixed depth equal to 7. Hence, the depth of a derivation for decomposing the formula F (the model of the `main` program) is $8 + 8 + 7 + 8 = 31$.

We will now proceed with a careful complexity analysis of decomposing the formulas produced by the specification of AP programs. These will be placed on the left of the sequent. This is done by counting the changes of nested polarities, as in the example above. The complexity of decomposing a process P will be denoted by $comp(\mathscr{L}[\![P]\!])$.

– **Base cases.** We will start by presenting the complexity for decomposing the different kinds of `lcc` processes:

$$\begin{array}{ll} comp(\mathscr{L}[\![c]\!]) & = 0 \\ comp(\mathscr{L}[\![p(x)]\!]) & = 1 + comp(\mathscr{L}[\![P]\!]) \quad \text{if} \quad \forall \tilde{x}.p(\tilde{x}) \stackrel{\Delta}{=} P \\ comp(\mathscr{L}[\![P \parallel Q]\!]) & = comp(\mathscr{L}[\![P]\!]) + comp(\mathscr{L}[\![Q]\!]) \\ comp(\mathscr{L}[\![\sum_{i \in I} \forall \tilde{x}_i(c_i \to P_i)]\!]) & = 1 + max_{i \in I}\{comp(\mathscr{L}[\![P_i]\!])\} \\ comp(\mathscr{L}[\![\exists x(P)]\!]) & = comp(\mathscr{L}[\![P]\!]). \end{array}$$

– **Sequential composition.** Recall that the process $P\,;Q$ was defined in Figure 6 with the aid of the function $\mathscr{C}[\![\cdot]\!]$. The complexity of decomposing $\mathscr{L}[\![P\,;Q]\!]$ will be given with the help of the auxiliary function $comp_{sc}$, that differs from $comp$ only in the case of the parallel composition:

$$\begin{array}{ll} comp(\mathscr{L}[\![P\,;Q]\!]) & = 1 + comp_{sc}(\mathscr{L}[\![\mathscr{C}[\![P]\!]_z]\!]) + comp(\mathscr{L}[\![Q]\!]) \\ comp_{sc}(\mathscr{L}[\![\mathscr{C}[\![P_1 \parallel \ldots \parallel P_n]\!]_z]\!]) & = 1 + \sum_{i \in 1..n} comp_{sc}(\mathscr{L}[\![\mathscr{C}[\![P_i]\!]_{w_i}]\!]) \\ comp_{sc}(\mathscr{C}[\![P]\!]_z) & = comp(\mathscr{L}[\![\mathscr{C}[\![P]\!]_z]\!]) \quad \text{in any other case} \end{array}$$

In the definition of $P\,;Q$, the constraint $\texttt{sync}(z)$ will always be produced before executing Q. As already said, these are negative actions and hence do not interfere with the proof's complexity. However, if P is a parallel composition $P = P_1 \parallel \ldots \parallel, P_n$, then each process P_i will produce its own synchronization token, and all of them will be consumed at once in order to produce the constraint $\texttt{sync}(z)$. Hence, the complexity of decomposing $P\,;Q$ takes into account nested

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

parallel compositions inside P.

– **Wrap.** The complexity of decomposing the subformula $\mathtt{wrap}(P, \{g_1, \cdots, g_n\}, z)$ is

$$\begin{aligned} comp(\mathscr{L}[\![\mathtt{wrap}(P, \{g_1, \cdots, g_n\}, z)]\!]) &= 1 + comp(\textstyle\bigotimes_{i\in 1..n} \mathtt{dg}(g_i, \mathtt{conc}, z) \multimap 1) \\ &\quad + comp(\mathscr{L}[\![P]\!] \otimes (\mathtt{end}(z) \multimap \textstyle\bigotimes_{i\in 1..n} \mathtt{dg}(g_i, \mathtt{conc}, z))) + 1 \\ &= n + 3 + comp(\mathscr{L}[\![P]\!]) \end{aligned}$$

– **Assignment.** It is immediate to see that

$$\begin{aligned} comp(\mathscr{L}[\![\mathtt{gain}(x, y, gt)]\!]) &= comp(\mathscr{L}[\![\mathtt{drop}(x)]\!]) = 1 \\ comp(\mathscr{L}[\![\mathtt{assg}(x, y, z, gt)]\!]) &= 7 \end{aligned}$$

Hence, $comp(assign) = comp(\mathscr{L}[\![\mathtt{wrap}(\mathtt{assg}(x, y, z, gt), G, z)]\!]) = 7 + n + 3 = n + 10$, where n is the number of elements in G. Observe that, when there are no group permissions, the wrap is not necessary and the complexity is the same as for decomposing $\mathscr{L}[\![\mathtt{assg}(x, y, z, gt)]\!]$, which is 7.

– **Axioms.** The upgrade and downgrade axioms are negative formulas. Decomposing them has depth 1.

– **Method definition.** Let m be the number of parameters of a method and suppose that, when consuming APs, one has to *upgrade* or *downgrade* r of them. Then, $comp(\mathtt{consume}) = r + (m + 1) + 1$, and

$$comp(\mathscr{L}[\![c\_m(x, \tilde{y}, z) \stackrel{\Delta}{=} P_M]\!]) = r + m + 4 + comp(\mathscr{L}[\![Body]\!])$$

where $Body$ is the body of the method (see Rule $\mathrm{R}_{\mathrm{MDEF}}$). On the other hand,

$$comp(\mathscr{L}[\![Body]\!]) = comp(\mathscr{L}[\![\mathscr{S}[\![\hat{s}]\!]_z]\!]) + m + r + 4.$$

– **Constructor.** With r and m as before, we have

$$comp(\mathscr{L}[\![c(x, \tilde{y}, z, g_1, \ldots, g_k) \stackrel{\Delta}{=} P_C]\!]) = 2r + 2m + comp(\mathscr{L}[\![\mathscr{S}[\![\hat{s}]\!]_z]\!]) + 11.$$

Theorem 5.2 (*Complexity*)
Let Δ be a theory containing the definition clauses for method and constructor definitions, the definition of $\mathtt{assg}$ and the upgrade and downgrade axioms. Let F be the formula interpreting the main program and G be a formula interpreting a property to be proven. It is decidable whether or not the sequent $!\Delta, F \longrightarrow G$ is provable. In fact, if such a sequent is provable, then its proof is bounded in ILLF by the depth $comp(F) + 1$.

Proof
First of all, note that, since there are no circular recursive definitions (see Remark 2.1), methods are simply unfolded. Moreover, as carefully described above, the complexity of such method calls is taken into account in the complexity of the outer method definition (see $\mathscr{L}[\![\mathscr{S}[\![\hat{s}]\!]_z]\!]$). This means that, whenever a method, constructor, or an axiom is called in Δ via F, its complexity is already computed in the complexity procedure we have just described. Due to the focusing discipline, proving a sequent in AP is equivalent to decomposing its formulas completely. Therefore, the complexity of the proof of the sequent $!\Delta, F \longrightarrow G$ is completely

determined by the complexity of decomposing F plus the final focusing in G, which is a purely positive formula. □

5.5 *Alcove prover and verification of properties*

In the following, we explain our verification technique for three different kinds of properties: deadlock detection; the ability of methods to run concurrently; and correctness (whether programs adhere to their specifications or not). Recall, from Example 4.4, that we have added to the predicates $\mathtt{act}(\cdot), \mathtt{run}(\cdot), \mathtt{end}(\cdot)$ extra parameters to signalize the variable that called the method, the name of the method, and the number of line of the source program. Then, for instance, in Example 5.1, the definition of the constructor looks like

$$collection\_collection(x,z,l) \stackrel{\Delta}{=} \exists x \ldots \mathtt{act}(x,' collection\_collection', l, z) \rightarrow \mathtt{run}(x,' collection\_collection', l, z); \\ \ldots .\mathtt{run}(x,' collection\_collection', l, z) \rightarrow\ !\,\mathtt{end}(x,' collection\_collection', l, z)$$

and the encoding of, e.g, line 10 in Figure 1, is $collection\_collection(x, z, 10)$.

Deadlock detection. Consider Example 4.5. We already showed that this code leads to a deadlock since s cannot upgrade its unique permission to execute `c.compStats(s)`. We are then interested in providing a proof to the programmer showing that the code leads to a deadlock. For doing this, let $\mathscr{D}[\![Def]\!]$ be the process definitions for the methods and constructors of the example plus the definition of assignment. Let st be the `main` program and consider the `lcc` program $\mathscr{D}[\![Def]\!].\mathscr{S}[\![st]\!]_z$. According to the definition of $\mathscr{S}[\![\cdot]\!]$ and $\mathscr{D}[\![\cdot]\!]$, we know that, for some z and c, $\mathtt{end}(c,' collection\_compStats', 15, z)$ will be added to the store only when the statement `c.compStats(svar)` (in line 15) is successfully executed. The translation of this program will give rise to the theory Δ and the formula F described in Example 5.1. The verification technique consists in showing that the sequent $!\Delta, F \longrightarrow \exists z, c.\mathtt{end}(c,' collection\_compStats', 15, z) \otimes \top$ is not provable. This verification is done automatically by using Alcove prover, a theorem prover for ILLF developed in Teyjus and integrated to the tool described in Section 4.3. Basically, we look for proofs with depth less or equal to 38, given by the depth of F. In this case, the prover fails, thus showing that the process $\mathscr{S}[\![st]\!]_z$ cannot reach a store entailing the constraint $\exists z, c.\mathtt{end}(c,' collection\_compStats', 15, z)$.

The URL of the Alcove tool includes the output of the theorem prover and the `lcc` interpreter for this example. It is worth noticing that the `lcc` interpreter only computes a possible trace of the program, while the theorem prover is able to check all the reachable configurations for the same program. The Alcove prover is completely faithful to the ILLF fragment presented in Section 5.3.

The use of "animators" and provers is complementary. Existing formal models for system construction, such as the *Rodin* (Abrial *et al.* 2010) tool for the event B modeling language, usually include both. The idea is that by using the animator, the user gain a global understanding of the behavior of the program before attempting the proof of more precise desirable properties. This usually avoids frustrations in trying to figure out corrections of the model to discharge unproved properties.

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

Concurrency analysis. Consider the following `lcc` agents:

$$\begin{array}{rcl} P & = & \mathtt{act}(z_1) \parallel \mathtt{act}(z_1) \rightarrow (\mathtt{run}(z_1) \otimes \mathtt{sync}(z_1)) \parallel \mathtt{run}(z_1) \rightarrow \,!\,\mathtt{end}(z_1) \\ Q & = & \mathtt{sync}(z_1) \rightarrow \mathtt{act}(z_2) \parallel \mathtt{act}(z_2) \rightarrow \mathtt{run}(z_2) \parallel \mathtt{run}(z_2) \rightarrow \,!\,\mathtt{end}(z_2) \end{array}$$

These processes represent an abstraction of the encoding of two statements s_1 and s_2 such that s_2 must wait until s_1 releases the program control by adding $\mathtt{sync}(z_1)$. It is easy to see that from the initial configuration $\gamma = \langle \emptyset; P \parallel Q; 1 \rangle$, we always end up in the final configuration $\gamma' = \langle \emptyset; \emptyset; !\,\mathtt{end}(z_1) \otimes !\,\mathtt{end}(z_2) \rangle \rangle$ showing that both s_1 and s_2 were successfully executed. Nevertheless, depending on the scheduler, we may observe different intermediate configurations. For instance, if all the processes in P are first selected for execution, we shall observe the derivation:

$$\begin{array}{rl} \gamma & \longrightarrow^* \langle \emptyset; \mathtt{run}(z_1) \otimes \mathtt{sync}(z_1) \parallel \mathtt{run}(z_1) \rightarrow \,!\,\mathtt{end}(z_1) \parallel Q; 1 \rangle \\ & \longrightarrow^* \langle \emptyset; \mathtt{run}(z_1) \rightarrow \,!\,\mathtt{end}(z_1) \parallel Q; \mathtt{run}(z_1) \otimes \mathtt{sync}(z_1) \rangle \\ & \longrightarrow^* \langle \emptyset; Q; !\,\mathtt{end}(z_1) \rangle \longrightarrow^* \gamma' \end{array}$$

On the other side, an interleaved execution of P and Q may be

$$\begin{array}{rl} \gamma & \longrightarrow^* \langle \emptyset; \mathtt{run}(z_1) \otimes \mathtt{sync}(z_1) \parallel \mathtt{run}(z_1) \rightarrow \,!\,\mathtt{end}(z_1) \parallel Q; 1 \rangle \\ & \longrightarrow^* \langle \emptyset; P' \parallel Q'; \mathtt{run}(z_1) \otimes \mathtt{act}(z_2) \rangle \\ & \longrightarrow^* \langle \emptyset; P' \parallel Q''; \mathtt{run}(z_1) \otimes \mathtt{run}(z_2) \rangle \longrightarrow^* \gamma' \end{array}$$

where $P' = \mathtt{run}(z_1) \rightarrow \,!\,\mathtt{end}(z_1)$, $Q' = \mathtt{act}(z_2) \rightarrow (\mathtt{run}(z_2) \otimes \mathtt{sync}(z_2) \parallel Q'')$, and $Q'' = \mathtt{run}(z_2) \rightarrow \,!\,\mathtt{end}(z_2)$. Unlike the first derivation, in the second one, we were able to observe the store $\mathtt{run}(z_1) \otimes \mathtt{run}(z_2)$ representing the fact that both s_1 and s_2 were executed concurrently.

From the point of view of the `lcc` interpreter, the two derivations above are admissible. This means that the fact of not observing in a trace the concurrent execution of two statements does not imply that they have to be sequentialized due to the AP dependencies.

We can rely on the logical view of processes to verify whether it is possible for two statements to run concurrently. For instance, consider the Example 4.4 and let F be the resulting ILL formula. The following sequent turns out to be provable:

$$!\,\Delta, F \longrightarrow \exists z_1, z_2, c, s(\mathtt{run}(c, 'collection\_print', 13, z_1) \otimes \mathtt{run}(c, 'collection\_compStats', 14, z_2)) \otimes \top$$

while the following one is not:

$$!\,\Delta, F \longrightarrow \exists z_1, z_2, c, s(\mathtt{run}(c, 'collection\_compStats', 14, z_1) \otimes \mathtt{run}(c, 'collection\_removeDuplicates', 15, z_2)) \otimes \top$$

i.e., regardless the scheduling policy, the program will not generate a trace where *compStats* and *removeDuplicates* run concurrently.

Verifying a method specification. Finally, assume that class *collection* has a field a and we define the following method:

```
mistake() unq(this)=>unq(this){
   this.a<g>:=this}
```

This method requires that the unique permission to the caller must be restored to the environment. Nevertheless, the implementation of the method splits the unique permission into two share permissions, one for the field a and another for the caller

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

(Rule R_{ALIAS}). Then, the axiom $upgrade_1$ cannot be used to recover the unique permission and the ask agent in definition `r_env` remains blocked. An analysis similar to that of deadlocks will warn the programmer about this. In general, what we need is to prove sequents of the shape $!\,\Delta, \Gamma \longrightarrow \exists c, z, l.\mathtt{end}(c, 'method', l, z) \otimes \top$ where Γ contains an atomic formula needed to start the execution of the method (i.e., a formula of the shape $c\_method(x, \cdots)$) and also the atomic formulas guaranteeing that the method can be executed ($\mathtt{ref}(x, o, \mathtt{unq}, \mathtt{ndg})$, $\mathtt{ct}(o, s(0))$ for the method *mistake*). This can be done, for instance, by letting $\Gamma = \mathscr{L}[\![\mathscr{S}[\![st]\!]]\!]$ where st is a dummy main program that creates an instance of `collection` and then calls the method *mistake*. In this case, the prover answers negatively to the query $!\,\Delta, \mathscr{L}[\![\mathscr{S}[\![st]\!]]\!] \longrightarrow \exists c, z, l.\mathtt{end}(c, 'collection\_mistake', l, z) \otimes \top$, showing that, even satisfying the preconditions of the method *mistake*, it cannot finish its execution.

6 Applications

In this section, we present two compelling examples of the use of our verification techniques. One is the well-known mutual exclusion problem where two (or more) processes compete for access to a critical section. In our example, there are two critical sections with exclusive access. The other models a producer and a consumer processes concurrently updating a data structure.

6.1 Two critical zones management system

Assume the class definitions for a two critical zones management system in Figure 17. There are three classes, `lock` (line 1), `process` (line 4), and `cs` (line 7). Each critical section has a private lock managed by an object of the class `cs`. When a process wants to enter the critical section $i \in \{1, 2\}$, it tries first to invoke the method `acq`*i* (lines 11 and 15) of the `cs` manager. If successful, the process obtains a lock (i.e., an object of class `lock`) that it uses then to enter that critical zone (lines 12 and 16). When the process wants to leave the critical zone, it invokes the method `release`*i* (lines 19 and 23). This releases ownership of the critical section lock.

Method `acq`*i* has three parameters: `this` (i.e., the `cs` manager), `b` the process wanting to enter the critical zone and `l`, a field of `b` that will hold the lock of the `cs` supplied by the manager. Since `this` has unique permission, only one reference to the manager object can exist for `acq`*i* to be invoked. The body of method `acq`*i* stores the lock in `l` and a reference to the manager in field `cs1` or `cs2` of `b`, depending on whether the lock for `cs1` or for `cs2` is requested. Storing this reference to the manager implies that it cannot longer have unique permission, so the output permission for `this` becomes shared. Moreover, `l` holds now the only reference to the private lock of the manager, so its output permission becomes unique. The effect is that field `lock1` or `lock2` of object `b` uniquely acquires the section lock. The method `enter` (line 3) requires a unique permission on the lock. This ensures that only one process has a reference to the lock at any given time when entering the critical section. The method `release`*i* restores conditions as they were before invocation to `acq`*i*, i.e., the

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

```
class lock <g> {
  lock() none(this) => unq(this) {}
  enter(process b) unq(this), shr : g(b) => unq(this), shr : g(b){}  }
class process <g>{
  attr  lock<g> lock1,  lock<g> lock2, cs<g> cs1, cs<g> cs2
  process() none(this) => unq(this) {}  }
class cs <g>{
  attr  lock<g> mylock
  cs() none(this) => unq(this) {
     this.mylock := new lock<g>()}
  acq1(process b,lock l)unq(this),shr:g(b),none(l)=>shr:g(this),shr:g(b),unq(l){
        l <g>:= this.mylock
        b.cs1 <g> := this
        this.mylock <g>:= null }
  acq2(process b,lock l)unq(this),shr:g(b),none(l)=>shr:g(this),shr:g(b),unq(l){
        l<g> := this.mylock
        b.cs2 <g>:= this
        this.mylock <g>:= null  }
  release1(lock a,process b)shr:g(this),unq(a),shr:g(b) => unq(this),none(a),shr
       :g(b){
        this.mylock <g>:= a
        b.cs1<g> := null
        a <g>:= null }
  release2(lock a, process b) shr:g(this),unq(a),shr:g(b) => unq(this),none(a),
       shr:g(b){
        this.mylock <g>:= a
        b.cs2 <g>:= null
        a <g>:= null } }
```

Fig. 17. Class definitions for a two critical zones management system.

```
1 main () {
2  group<g>
3  let cs x, cs w,
4       process y, process z in
5        cs1:= new cs<g>()
6        cs2 := new cs<g>()
7        p1  := new process<g>()
8        p2  := new process<g>()
9        cs1.acq1(p1, p1.lock1)
10       p1.lock1.enter(p1)
11       cs2.acq2(p2, p2.lock2)
12       p2.lock2.enter(p2)
13       cs1.acq1(p2, p2.lock1)
14       p2.lock1.enter(p2)
15       cs2.acq2(p1, p1.lock2)
16       p1.lock2.enter(p1)
17     end }
```

(a) Deadlock code

```
    ... // constructors
    cs1.acq1(p1, p1.lock1)
    p1.lock1.enter(p1)
    cs2.acq2(p2, p2.lock2)
    p2.lock2.enter(p2)
    cs1.release1(p1.lock1, p1)
    cs1.acq1(p2, p2.lock1)
    p2.lock1.enter(p2)
    cs2.release2(p2.lock2, p2)
    cs2.acq2(p1, p1.lock2)
    p1.lock2.enter(p1)
    cs1.release1(p2.lock1, p2)
    cs2.release2(p1.lock2, p1)
  end }
```

(b) Deadlock free code

Fig. 18. Main codes for the critical zone management system.

manager regains the unique permission and stores a unique reference to its private lock. Process object fields loose the lock and the reference to the manager.

Assume now the main code in Figure 18(a) where there are two section manager objects `cs1` and `cs2`. There are also two processes, `p1` and `p2`. Consider the situation where `p1` acquires the lock from `cs1` (line 9) and enters (line 10). Then, `p2` acquires the lock from `cs2` and enters (lines 11–12). Now, `p2` tries to acquire the lock from

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

cs2 (line 13), but this is not possible because cs1 has no longer a unique permission and execution blocks. Alcove reports this situation:

```
...
calling(X_6136,cs_acq1,line_71 (Z_PAR_18116))
...
[FAIL] Token ok not found. End of the program not reached.
```

Consider now the program in Figure 18(b) where processes leave the critical section before attempting to acquire another lock. In this case, all invocations run without blockage and Alcove successfully finishes the analysis:

```
...
ended(X_6152,cs_release1,line_113 (Z_PAR_24136))
ended(W_7153,cs_release2,line_114 (Z_PAR_25137))
ok()
3517 processes Created

[OK] Token ok found. End of the program reached.
```

6.2 *Concurrent producer–consumer system*

Figure 19 shows the class definitions for a producer–consumer system working concurrently over a buffer. Class *buffer* (line 1) represents the data structure with operations for reading (line 3), writing (line 4), and removing the content of the buffer (line 5). Class *producerConsummer* (line 14) provides methods for adding (*produce*) – line 19 – and remove (*consume*) – line 24 – elements from the data structure. Since these could be invoked concurrently, the class implements a critical section (line 8) representing access to the element of the data structure the consumer or producer is working on. That is, producing or consuming could in principle be simultaneous over different elements of the structure. To keep the example simple, we assume a single critical section over the whole data structure.

Class *producerConsumer* defines a group g for processes operating over the data structure (line 14). The group is used to manage permissions of all processes invoking methods of the class. Since callers of *produce* and *consume* both have share group permissions on g, they can be invoked concurrently. This can be seen in the main program (line 30). Variable PC has unique permission over the *producerConsumer* object. This unique permission is split (line 36) into share permissions for the group to allow producer and consumer calls to run in parallel. Note, however, that simultaneous access to the buffer is precluded by the need for each process to acquire the lock before (lines 20 and 25).

As shown, in the excerpt of the Alcove's output in Figure 20, the call to *consume* (line 4) is done while *produce* is still running (line 2). Note also that before executing *write* (line 11), the method *produce* has to acquire the lock on the data structure (lines 5 and 9). Similarly, the execution of *read* (line 23) (called by the consumer in line 20) has to wait until the lock is released by the consumer (line 17) and acquired by the producer (lines 18 and 21).

AP-based languages like Æminium (Stork *et al.* 2009) provides abstractions to simplify the (concurrent) access to share objects. For instance, in the example above, we locked the buffer before executing the methods *write* and *read* (lines 21 and 26 in

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

```
1 class buffer<g>{
2    buffer() none(this) => unq(this) { }
3    read()  shr : g(this) => shr : g(this) { }
4    write()   shr : g(this) => shr : g(this) { }
5    dispose() unq(this)  => unq(this) { }}
6 class lock {
7     lock() none(this)  => unq(this) {} }
8 class cs <g>{
9      cs() none(this) => unq(this) {  }
10     acq(lock l) shr : g(this), none(l) => shr : g(this),  unq(l) {
11            l := new lock()  }
12     release(lock l) shr:g(this), unq(l) => shr:g(this),  none(l) {
13             l<g> := null  }  }
14 class producerConsumer<g>  {
15   attr  lock l, cs<g> c
16   producerconsumer() none(this) => unq(this) {
17     this.c := new cs<g>() // Initializing the critical section
18   }
19   produce(buffer<g> B) shr:g(this),shr:g(B) => shr:g(this),  shr:g(B) {
20       this.c.acq(this.l)  // Getting a lock on the data structure
21       B.write()
22       this.c.release(this.l) //releasing the lock
23        }
24   consume(buffer<g> B)shr:g(this),shr:g(B)=>shr:g(this),  shr:g(B) {
25       this.c.acq(this.l) // Getting a lock on the data structure
26       B.read()
27       this.c.release(this.l)//releasing the lock
28        }
29   }
30 main{
31   group <g>
32   let producerConsumer<g> PC, buffer<g> B  in {
33   B  := new buffer <g>()
34   PC := new producerConsumer<g>()
35   // produce and consume running in parallel
36   split<g>{
37      PC.produce(B)
38      PC.consume(B)
39   } }}
```

Fig. 19. AP program for a concurrent producer–consumer system.

Figure 19). In Æminium, it suffices to wrap the call to these methods into a atomic block of the form:

```
atomic<g>{
 B.write()
}
```

The Æminium runtime system guarantees that the execution of *write* on the object pointed by *B* is isolated, i.e., other methods invoked on the same object must wait until the termination of *write*. We note that the behavior of `atomic` blocks relies completely on the runtime system. Since we are interested in the static analysis of AP programs, we did not considered atomic blocks in the grammar of Figure 3. Note also that what we can analyze statically is whether methods *produce* (line 19) and *consume* (line 24) can acquire a share permission on the buffer *B*.

7 Concluding remarks

We presented an approach based on `lcc` for specifying and verifying programs annotated with APs. Program statements are modeled as `lcc` agents that faithfully

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press

```
[...] calling(PC_660,producerconsumer_produce,line_58 (z_2347))
running(PC_660,producerconsumer_produce,line_58 (z_2347))
calling(u_117119,cs_acq,line_41 (z_par_219246))
calling(PC_660,producerconsumer_consume,line_59 (z_2448))
running(u_117119,cs_acq,line_41 (z_par_219246))
calling(inner_294344,lock_lock,line 24 (z_par_307308))
calling(inner_200289,buffer_write,line_42 (z_par_220247))
running(PC_660,producerconsumer_consume,line_59 (z_2448))
running(inner_294344,lock_lock,line 24 (z_par_307308))
calling(u_117119,cs_acq,line_46 (z_par_368395))
running(inner_200289,buffer_write,line_42 (z_par_220247))
ended(inner_294344,lock_lock,line 24 (z_par_307308))
calling(u_117119,cs_release,line_43 (z_par_221248))
ended(inner_200289,buffer_write,line_42 (z_par_220247))
running(u_117119,cs_release,line_43 (z_par_221248))
ended(u_117119,cs_acq,line_41 (z_par_219246))
ended(u_117119,cs_release,line_43 (z_par_221248))
running(u_117119,cs_acq,line_46 (z_par_368395))
calling(inner_516566,lock_lock,line 24 (z_par_529530))
calling(inner_349438,buffer_read,line_47 (z_par_369396))
running(inner_516566,lock_lock,line 24 (z_par_529530))
ended(PC_660,producerconsumer_produce,line_58 (z_2347))
running(inner_349438,buffer_read,line_47 (z_par_369396)) [...]
```

Fig. 20. Excerpt of Alcove's output for the producer–consumer program.

represent the statement permissions flow. The declarative reading of `lcc` agents as formulas in ILL permits verifying properties such as deadlocks, the admissibility of parallel executions, and whether methods are correct w.r.t. their AP specifications. Central to our verification approach is the synchronization mechanism based on constraints, combined with the logical interpretation of `lcc` into the focused system ILLF.

A good strategy for understanding the behavior of a concurrent program is running a simulator able to observe the evolution of its processes, hence having a better glance of the global program behavior. Then, a prover able to verify formally various properties can be executed. For this reason, we have automated our specification and verification approach as the Alcove tool. Using this tool, we were able, for instance, to verify the critical zone management system and the producer–consumer system presented in Section 6. The reader can find these and other examples at the Alcove tool website. The results and techniques presented here are certainly a novel application for `ccp`, and they will open a new window for the automatic verification of (OO) concurrent programs.

Related and future work. `ccp`-based calculi have been extensively used to reason about concurrent systems in different scenarios such as system biology, security protocols, multimedia interaction systems, just to name a few. The reader may find in (Olarte *et al.* 2013) a survey of models and applications of `ccp`. A work related to ours is (Jagadeesan *et al.* 2005), where the authors propose a timed-`ccp` model for role-based access control in distributed systems. The authors combine constraint reasoning and temporal logic model checking to verify when a resource (e.g., a directory in a file system) can be accessed. We should also mention the work in

(Nigam 2012) where linear authorization logics are used to specify access control policies that may mention the affirmations, possessions, and knowledge of principals. In the above-mentioned works, access policies are used to control and restrict the use of resources in a distributed environment but they do not deal with the verification of a (concurrent) programming language.

Languages like Æminium (Stork *et al.* 2009) and Plaid (Sunshine *et al.* 2011) offer a series of guarantees such as (1) absence of AP usage protocol violation at run time; (2) when a program has deterministic results; and (3) whether programs are free of race conditions on the abstract state of the objects (Boyland 2003; Bierhoff and Aldrich 2007). Roughly, type-checking rules generate the needed information to build the graph of dependencies among the statements in the program. Such annotations are then used by the runtime environment to determine the pieces of code that can be executed in parallel (Stork *et al.* 2014). Well-typed programs are free of race condition by either enforcing synchronization when accessing shared data or by correctly computing dependencies. However, well-typed programs are not necessarily deadlock free. Hence, our developments are complementary to those works and provide additional reasoning techniques for AP programs.

Somewhat surprisingly, even though in (Stork *et al.* 2014), it is mentioned that *access permissions follow the rules of linear logic*, the authors did not go further on this idea. Our linear logic encodings can be seen as the first logic semantics for AP. As showed in this paper, such declarative reading of AP allows to perform interesting static analyses on AP-based programs.

The constraint system we propose to model the downgrade and upgrade of axioms was inspired by the work of *fractional* permissions in (Boyland 2003) (see also (Bierhoff and Aldrich 2007)). *Fractional* in this setting means that an AP can be split into several more *relaxed* permissions and then joined back to form a more *restrictive* permission. For instance, a unique permission can be split into two share permissions of weight $k/2$. Therefore, to recover a unique permission, it is necessary to have two $k/2$-share permissions. The constraint system described in this paper keeps explicitly the information about the fractions by using the predicate $\mathtt{ct}(\cdot)$.

Chalice (Leino 2010) is a program verifier for OO concurrent programs that uses permissions to control memory accesses. Unlike Æminium and Plaid, concurrency in Chalice is explicitly stated by the user by means of execution threads.

The language Rust (`https://www.rust-lang.org/`) provides mechanisms to avoid data races. These do not use APs but rely on types. Type *mut* (mutable) works similarly as a *unq* permission. A data structure defined with type *mut* is claimed ownership by the first thread using it, so it cannot be taken concurrently by another thread. The compiler checks this statically. Type *Arc* allows the data structure to be shared among threads, but then it cannot be a mutable structure. This is then similar to *imm* permissions. Type *Arc* can be combined with *mutex* to have a mutable structure that can be shared. A lock mechanism is available for the user to control simultaneous accesses. As opposed to permissions, however, there is no upgrading/downgrading of types. Hence, AP and DGAP provide, in principle, more flexible mechanisms to express concurrent behaviors. In (Ullrich 2016), a translation of Rust programs into the *Lean* prover to verify program correctness is described. As far as we know, however, no verification of the kind we presented here is available for Rust.

AP annotations in concurrent-by-default OO languages can be enhanced with the notion of *typestates* (Bierhoff and Aldrich 2007; Beckman *et al.* 2008). Typestates describe abstract states in the form of state- machines, thus defining a usage protocol (or *contract*) of objects. For instance, consider the class *File* with states `opened` and `closed`. The signature of the method *open* can be specified as the agent $\mathtt{unq(this)} \otimes \mathtt{closed(this)} \rightarrow \mathtt{unq(this)} \otimes \mathtt{opened(this)}$. The general idea is to verify whether a program follows correctly the usage protocol defined by the class. For example, calling the method *read* on a `closed` file leads to an error. Typestates then impose certain order in which methods can be called. The approach our paper defines can be extended to deal with typestates annotations, thus widening its applicability.

The work in (Naden *et al.* 2012) defines more specific systems and rules for APs to provide for *borrowing permissions*. This approach aims at dealing more effectively with local variable aliasing, and with how permissions flow from the environment to method formal parameters. Considering these systems in Alcove amounts to refine our model of permissions in Section 4. Verification techniques should remain the same.

Acknowledgements

We thank the anonymous reviewers for their valuable comments on an earlier draft of this paper.

References

Abrial, J.-R., Butler, M. J., Hallerstede, S., Hoang, T. S., Mehta, F. and Voisin, L. 2010. Rodin: An open toolset for modelling and reasoning in event-b. *International Journal on Software Tools for Technology Transfer 12,* 6, 447–466.

Andreoli, J.-M. 1992. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation 2,* 3, 297–347.

Beckman, N. E., Bierhoff, K. and Aldrich, J. 2008. Verifying correct usage of atomic blocks and typestate. In *OOPSLA*, G. E. Harris, Ed. ACM, 227–244.

Bierhoff, K. and Aldrich, J. 2007. Modular typestate checking of aliased objects. In *OOPSLA*, R. P. Gabriel, D. F. Bacon, C. V. Lopes and G. L. S. Jr., Eds. ACM, 301–320.

Boyland, J. 2003. Checking interference with fractional permissions. In *SAS*, R. Cousot, Ed. Lecture Notes in Computer Science, vol. 2694. Springer, 55–72.

Boyland, J., Noble, J. and Retert, W. 2001. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, J. L. Knudsen, Ed. Lecture Notes in Computer Science, vol. 2072. Springer, 2–27.

de Boer, F. S., Gabbrielli, M., Marchiori, E. and Palamidessi, C. 1997. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems 19,* 5, 685–725.

Fages, F., Ruet, P. and Soliman, S. 2001. Linear concurrent constraint programming: Operational and phase semantics. *Information and Computation 165,* 1, 14–41.

Girard, J.-Y. 1987. Linear logic. *Theoretical Computer Science 50*, 1–102.

Haemmerlé, R. 2011. Observational equivalences for linear logic concurrent constraint languages. *Theory and Practice of Logic Programming 11,* 4–5, 469–485.

Igarashi, A., Pierce, B. C. and Wadler, P. 2001. Featherweight java: A minimal core calculus for java and GJ. *ACM Transactions on Programming Languages and Systems 23,* 3, 396–450.

Jagadeesan, R., Marrero, W., Pitcher, C. and Saraswat, V. A. 2005. Timed constraint programming: A declarative approach to usage control. In *PPDP*, P. Barahona and A. P. Felty, Eds. ACM, 164–175.

Leino, K. R. M. 1998. Data groups: Specifying the modification of extended state. In *OOPSLA*, B. N. Freeman-Benson and C. Chambers, Eds. ACM, 144–153.

Leino, K. R. M. 2010. Verifying concurrent programs with Chalice. In *VMCAI*, G. Barthe and M. V. Hermenegildo, Eds. Lecture Notes in Computer Science, Vol. 5944. Springer, 2.

Lincoln, P., Mitchell, J. C., Scedrov, A. and Shankar, N. 1992. Decision problems for propositional linear logic. *Annals of Pure and Applied Logic 56,* 1–3, 239–311.

Martinez, T. 2010. Semantics-preserving translations between linear concurrent constraint programming and constraint handling rules. In *PPDP*, T. Kutsia, W. Schreiner and M. Fernández, Eds. ACM, 57–66.

Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic*. Cambridge University Press.

Nadathur, G. and Miller, D. 1988. An overview of lambda-prolog. In *Logic Programming, Proc. of the 5th International Conference and Symposium*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Seattle, Washington, August 15–19, 1988 (2 Volumes), 810–827.

Naden, K., Bocchino, R., Aldrich, J. and Bierhoff, K. 2012. A type system for borrowing permissions. In *POPL*, J. Field and M. Hicks, Eds. ACM, 557–570.

Nielsen, M., Palamidessi, C. and Valencia, F. D. 2002. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing 9,* 1, 145–188.

Nigam, V. 2012. On the complexity of linear authorization logics. In *LICS*. IEEE, 511–520.

Olarte, C. and Pimentel, E. 2017. On concurrent behaviors and focusing in linear logic. *Theoretical Computer Science 685*, 46–64.

Olarte, C., Pimentel, E., Rueda, C. and Cataño, N. 2012. A linear concurrent constraint approach for the automatic verification of access permissions. In *PPDP*, D. D. Schreye, G. Janssens and A. King, Eds. ACM, 207–216.

Olarte, C., Rueda, C. and Valencia, F. D. 2013. Models and emerging trends of concurrent constraint programming. *Constraints 18*, 535–578.

Pottier, F. and Protzenko, J. 2013. Programming with permissions in mezzo. *Special Interest Group on Programming Languages Notices 48,* 9, 173–184.

Saraswat, V. A. 1993. *Concurrent Constraint Programming*. MIT Press.

Saraswat, V. A. and Rinard, M. C. 1990. Concurrent constraint programming. In *POPL*, F. E. Allen, Ed. ACM Press, 232–245.

Saraswat, V. A., Rinard, M. C. and Panangaden, P. 1991. Semantic foundations of concurrent constraint programming. In *POPL*, D. S. Wise, Ed. ACM Press, 333–352.

Stork, S., Marques, P. and Aldrich, J. 2009. Concurrency by default: Using permissions to express dataflow in stateful programs. In *OOPSLA Companion*, S. Arora and G. T. Leavens, Eds. ACM, 933–940.

Stork, S., Naden, K., Sunshine, J., Mohr, M., Fonseca, A., Marques, P. and Aldrich, J. 2014. Æminium: A permission-based concurrent-by-default programming language approach. *ACM Transactions on Programming Languages and Systems 36,* 1, 2.

Sunshine, J., Naden, K., Stork, S., Aldrich, J. and Tanter, É. 2011. First-class state change in plaid. In *OOPSLA*, C. V. Lopes and K. Fisher, Eds. ACM, 713–732.

Ullrich, S. A. 2016. *Simple Verification of Rust Programs via Functional Purification.* Master's Thesis, Karlsruher Institut für Technologie (KIT).

https://doi.org/10.1017/S1471068418000017 Published online by Cambridge University Press