Erschienen in: Science of Computer Programming ; 133 (2017), 2. - S. 194-215
https://dx.doi.org/10.1016/j.scico.2016.08.005

On the verification of SCOOP programs

Georgiana Caltais [a,b,*], Bertrand Meyer [a,c,*]

[a] *Department of Computer Science, ETH Zürich, Switzerland*
[b] *Department for Computer and Information Science, University of Konstanz, Germany*
[c] *Software Engineering Lab, Innopolis University, Russia*

A B S T R A C T

In this paper we focus on the development of a unifying framework for the formal modeling of an object oriented-programming language, its underlying concurrency model and their associated analysis tools. More precisely, we target SCOOP – an elegant concurrency model, recently formalized based on Rewriting Logic (RL) and Maude. SCOOP is implemented in Eiffel and its applicability is demonstrated also from a practical perspective, in the area of robotics programming. Our contribution consists of devising and integrating an alias analyzer and a Coffman deadlock detector under the roof of the same RL-based semantic framework of SCOOP. This enables using the Maude rewriting engine and its LTL model-checker "for free," in order to perform the analyses of interest. We discuss the limitations of our approach for model-checking deadlocks and provide possible workarounds for the state space explosion problem. On the aliasing side, we propose an extension of a previously introduced alias calculus based on program expressions, to the setting of unbounded program executions. Moreover, we devise a corresponding executable specification easily implementable on top of the SCOOP formalization. An important property of our extension is that, in non-concurrent settings, the corresponding alias expressions can be over-approximated in terms of a notion of regular expressions. This further enables us to derive an algorithm for computing a sound over-approximation of the "may aliasing" information, where soundness stands for the lack of false negatives.

1. Introduction

In light of the widespread deployment and complexity of concurrent systems, the development of corresponding frameworks for rigorous design and analysis has been a great challenge. In this paper we focus on the development of a unifying framework for the formal modeling of an object oriented-programming language, its underlying concurrency model and their associated analysis tools.

We are targeting SCOOP [1], a simple object-oriented programming model for concurrency. Two main characteristics make SCOOP simple: 1) just one keyword programmers have to learn and use in order to enable concurrent executions, and 2) the burden of orchestrating concurrent executions is handled within the model, therefore reducing the risk of correctness issues. The reference implementation is Eiffel [2], but implementations have also been built on top of Java-like languages [3]. The success of SCOOP is demonstrated not only from a research perspective, but also from a practical perspective, with applications appearing, for instance, in the area of robotics programming [4].

* Corresponding authors.
E-mail addresses: gcaltais@gmail.com (G. Caltais), bertrand.meyer@inf.ethz.ch (B. Meyer).

The basis of a framework for the design and analysis of the SCOOP model has already been set. In this respect, we refer to the recent formalization of SCOOP in [1] based on Rewriting Logic (RL) [5], which is executable and straightforwardly implementable in the programming language Maude [6]. In [1] these capabilities have been successfully exploited in order to reason about the original SCOOP model and to identify a number of design flaws.

Moreover, an executable semantics can be exploited in order to formalize and "run" analysis tools for SCOOP programs as well. This facilitates the extension of the aforementioned SCOOP formalization to the level of a unifying executable semantic framework for the design and analysis of both the model and its concurrent applications. In this paper we focus on the *development of a RL-based toolbox for the analysis of SCOOP programs* on top of the formalization in [1]. We are interested in constructing an alias analyzer and a deadlock detector.

Alias analysis has been an interesting research direction for the verification and optimization of programs. One of the challenges along this line of research has been the undecidability of determining whether two expressions in a program *may* reference the same object. A rich suite of approaches aiming at providing a satisfactory balance between scalability and precision has already been developed in this regard. Examples include: (i) intra-procedural frameworks [7,8] that handle isolated functions only, and their inter-procedural counterparts [8–10] that consider the interactions between function calls; (ii) type-based techniques [11]; (iii) flow-based techniques [12,13] that establish aliases depending on the control-flow information of a procedure; (iv) context-(in)sensitive approaches [14,15] that depend on whether the calling context of a function is taken into account or not; (v) field-(in)sensitive approaches [16,17] that depend on whether the individual fields of objects in a program are traced or not.

There is a huge literature on heap analysis for aliasing [18], but hardly any paper that presents a calculus allowing the derivation of alias relations as the result of applying various instructions of a programming language. Hence, of particular interest for the work in this paper is the untyped, flow-sensitive, field sensitive, inter-procedural and context-sensitive calculus for may aliasing, introduced in [19]. The calculus covers most of the aspects of a modern object-oriented language, namely: object creation and deletion, conditionals, assignments, loops and (possibly recursive) function calls. The approach in [19] abstracts the aliasing information in terms of explicit access paths [20] referred to as *alias expressions* straightforwardly computed in an equational fashion, based on the language constructs. As we shall see later on in this paper, the language-based expressions can be exploited in order to reason about "may aliasing" in a finite number of steps in non-concurrent settings and, moreover, can be easily incorporated in the semantic rules defining SCOOP in [1].

The work most similar to our contribution is the one in [21], where aliasing information is identified by exploiting regular behavior of (non-concurrent) programs, in a RL setting. The main difference with the results in [21] consists of how the abstract memory addresses corresponding to pointer variables are represented. In [21] these range over a finite set of natural numbers. In this paper we consider alias expressions build according to the calculus in [19], based on program constructs.

In [22] the authors focus on alias analysis and type inference for PHP and sketch a hypothetical solution of their approach in Maude. The alias analysis in [22] is inter-procedural as well, and can handle function pointers, recursive calls and references. One of the major differences with our work is that in [22], in order to derive correct analysis results, the authors run aliasing and type inference analysis in "tandem" until a fixpoint is reached. We guarantee the termination our analysis by exploiting a sound over-approximation of aliasing via the so-called regular alias expressions.

Deadlock is one of the most serious problems in concurrent systems. It occurs when two or more executing threads are each waiting for the other to finish. Along time, the complexity of the problem determined various approaches to combat deadlocks [23]. Examples include: (i) deadlock prevention [24] which ensures that at least one of the deadlock conditions cannot hold, (ii) deadlock avoidance [25] that provides a priori information so that the system can predict and avoid deadlock situations, (iii) deadlock detection [26,27] that detects and recovers from a deadlock state.

Our focus is on *deadlock detection* for SCOOP programs. We base our work on the fact that this type of analysis is in strict connection with the underlying model of interest. Consequently, as described in the corresponding subsequent sections, our approach consists of formalizing deadlocks in the context of the SCOOP concurrency model and enriching its semantics in [1] with the equivalent operational-based definition of deadlocks. This enables using the Maude rewriting capabilities "for free" in order to test SCOOP programs for deadlock. Nevertheless, the more ambitious goal of using the Maude LTL model-checker for deadlock detection is not straightforward. As discussed in more detail later on in this paper, verification of deadlocks was possible after reducing the SCOOP semantics in [1] and abstracting it based on aliasing information, and modifying a series of implementation aspects (such as indexed-based parameterizations) that determined state space explosion issues.

The literature on using static analysis [8] and abstracting techniques for (related) concurrency models is considerable. We refer, for instance, to the recent work in [28] that introduces a framework for detecting deadlocks by identifying circular dependencies in the (finite state) model of so-called contracts that abstract methods in an OO-language. Nevertheless, the integration of a deadlock analyzer in SCOOP on top of Maude is an orthogonal approach that aims at constructing a RL-based toolbox for SCOOP programs laying over the same semantic framework.

In [29] SCOOP programs are verified for deadlocks and other behavioral properties using GROOVE [30]. The work in [29] proposes a redefinition of the most common features of the SCOOP semantics based on graph transformation systems (GPSs). This is a bottom-up approach, as it aims at redefining the SCOOP semantics from scratch via GPSs, orthogonal to our rather top-down strategy of narrowing the original semantics proposed in [1].

There are several RL-based approaches to deadlock detection in the literature. In [31,32], for instance, the authors use the Maude LTL model checker to show deadlock-freedom in a dining philosophers implementation in C. In [33], the Maude

LTL model checker is used to verify safety and liveness properties of Orc [34] programs such as a dining philosophers implementation and an online auction system. Nevertheless, none of the aforementioned results are used in the context of a unifying framework for specifying and clarifying a concurrency model [1], and analyzing its corresponding applications.

The results in this paper centre around SCOOP, as tackling aliasing and deadlock related aspects is particularly challenging in the context of a concurrency model as complex as SCOOP. However, exploiting a unifying RL-based framework for both designing the concurrency model and analyzing its corresponding applications raises certain issues that were not obvious at the stage of just formalizing SCOOP in [1]. Some of the identified design and implementation decision affecting the verification of SCOOP applications in a negative way can serve as guidelines in the context of other RL-based models as well.

This paper is an extended version of [35] where we proposed:

1. a translation of the (finite) alias calculus in [19] to the setting of unbounded program executions determined by finite executions of any length, together with a sound over-approximation technique based on (finitely representable) "regular alias expressions" capturing unbounded executions in non-concurrent settings;
2. a RL-based specification of the extended calculus suitable for integration within the SCOOP formalization in [1] (for this purpose we use a notation inspired by the $\mathbb{K}$-framework enabling compact and modular definitions);
3. an algorithm for "may aliasing" (exploiting the finiteness property in 1.) calculating over-approximating aliases in non-concurrent settings.

The current work adds to 1.–3. above:

4. the full, revised RL-based specification in 2. and the complete formal proofs showing the soundness of the over-approximating technique based on "regular alias expressions";
5. examples of exploiting the algorithm in 3. and its implementation on top of the SCOOP formalization in Maude [1];
6. the formalization and integration of a deadlock detection mechanism on top of the SCOOP operational semantics [1], together with discussions on the limitations of our approach and associated workarounds.

Paper structure The paper is organized as follows. In Section 2 we provide a brief overview of SCOOP. In Section 3 we introduce the extension of the alias calculus in [19] to unbounded executions. In Section 4 we provide the full RL-based executable specification of the calculus and a termination result. The implementation in SCOOP and further applications are discussed in Section 5. Section 6 is dedicated to deadlocking in SCOOP. In Section 7 we draw some conclusions and provide pointers to future developments.

2. Brief introduction to SCOOP

As already stated, the purpose of the current work is the development of a toolbox for the analysis of SCOOP programs by exploiting the semantics proposed in [1]. SCOOP is particularly attractive due to its simplicity and elegance, as it allows the switch from sequential to concurrent programming in a rather straightforward fashion, by means of just one keyword, namely, *separate*. Transparent to the user, the key notion in SCOOP is the processor, or *handler* (that can be a CPU, or it can also be implemented in software, as a process or thread). Handlers are in charge of executing the routines of "separate" objects, in a concurrent fashion.

For an example, assume a processor p that performs a call $o.f(a_1, a_2, \ldots)$ on an object o. If o is declared as "separate", then p sends a request for executing $f(a_1, a_2, \ldots)$ to q – the handler of o (note that p and q can coincide). Meanwhile, p can continue. Moreover, assume that $a_1, a_2, \ldots$ are of "separate" types. According to the SCOOP semantics, the application of the call $f(\ldots)$ will *wait* until it has been able to *lock* all the separate objects associated to $a_1, a_2, \ldots$. This mechanism guarantees exclusive access to these objects. Given a processor p, by $W(p)$ we denote the set of processors p *waits* to release the resources p needs for its asynchronous execution. Orthogonally, by $H(p)$ we represent the set of resources (more precisely, resource handlers) that p already acquired.

The semantics of SCOOP in [1] is defined over tuples of shape

$$\langle p_1 :: St_1 \mid \ldots \mid p_n :: St_n, \sigma \rangle \tag{1}$$

where, p_i denotes a processor (for $i \in \{1, \ldots, n\}$), St_i is the call stack of p_i and σ is the *state* of the system. States hold information about the *heap* (which is a mapping of references to objects) and the *store* (which includes the binding of the formal parameters to actual arguments, local variables, *etc.*). Processors communicate via *channels*.

Roughly speaking, one could classify the operational rules formalizing SCOOP in [1] in: a) *language rules* that provide the semantics of language constructs such as "**if** ... **then** ... **else** ... **end**" or "**until** ... **loop** ... **end**", and b) *control rules* implementing mechanisms such as locking or scheduling.

For an example in category a) above, consider the rules specifying "**if**" instructions:

$$\frac{a \text{ is fresh}}{\begin{array}{c}\langle p :: \textbf{if } e \textbf{ then } St_1 \textbf{ else } St_2 \textbf{ end}\,; St,\ \sigma\rangle \to \\ \langle p :: \text{eval}(a, e); \text{wait}(a); \mathit{provided\ a.data\ then\ } St_1 \mathit{\ else\ } St_2; St,\ \sigma\rangle\end{array}} \tag{2}$$

$$\frac{\cdot}{\langle p :: provided\ true\ then\ St_1\ else\ St_2; St,\ \sigma \rangle \rightarrow \langle p :: St_1; St,\ \sigma \rangle} \tag{3}$$

$$\frac{\cdot}{\langle p :: provided\ false\ then\ St_1\ else\ St_2; St,\ \sigma \rangle \rightarrow \langle p :: St_2; St,\ \sigma \rangle} \tag{4}$$

Intuitively, "eval(a, e)" evaluates e and puts the result on a fresh channel a and "wait(a)" enables processor p to use the evaluation result stored in $a.data$. It is straightforward to see that, according to (3), in case the condition e is evaluated to $true$ then the "**if** branch" St_1 is placed on top of the call stack of p. Otherwise, based on (4), if e is evaluated to $false$, the "**else** branch" is executed.

As we shall see in Section 5, an operational view on the alias calculus in [19] exploiting the instructions of a programming language will enable a straightforward implementation on top of the "language rules" of SCOOP.

For the case b) above we refer to the locking rule:

$$\frac{\cdot}{\begin{array}{c}\langle p :: lock(\{q_1, \dots q_m\}); St,\ \sigma \rangle \rightarrow \\ \langle p :: St,\ \sigma.lock\_rqs(p, \{q_1, \dots q_m\}) \rangle\end{array}} \quad \forall q_i \in \{q_1, \dots, q_m\} : \sigma.rq\_locked(q_i) = false \tag{5}$$

stating that a processor p can lock a set of handlers $\{q_1, \dots, q_m\}$ by calling $lock\_rqs$ on the state σ whenever none of the handlers q_i has previously been acquired by other processors, *i.e.*, $\sigma.rq\_locked(q_i) = false$.

As will become clear in Section 6, "control rules" pave the way to an immediate implementation of a corresponding "deadlock rule" on top of the Maude formalization of SCOOP in [1].

3. The alias calculus

The calculus for *may aliasing* introduced in [19] abstracts the aliasing information in terms of explicit access paths referred to as "alias expressions". Consider, for an example, the case of a linked list. We write x_i $(i \geq 0)$ to represent node i in the list, and use a setter to assign the next node of the list:

```
**create** $x_0$
**loop**
  $i := i + 1$
  **create** $x_i$
  $x_i.set\_next(x_{i-1})$
**end**                                                    (6)
```

The result of the execution of the code above can be intuitively depicted as the infinite sequence:

$$x_0 \xleftarrow{next} x_1 \xleftarrow{next} \dots x_{k-1} \xleftarrow{next} x_k \xleftarrow{next} x_{k+1} \dots$$

Hence, x_0 becomes aliased to $x_1.next$, $x_2.next.next$, $x_3.next.next.next$, so on and so on. In short, the set of associated alias expressions can be equivalently written as:

$$\{[x_i,\ x_{i+k}.next^k] \mid i \geq 0 \wedge k \geq 1\}. \tag{7}$$

The sources of imprecision introduced by the calculus in [19] are limited to ignoring tests in conditionals, and to "cutting at length L" for the case of possibly infinite alias relation corresponding to unbounded executions as in (6). The cutting technique considers sequences longer than a given length L as aliased to all expressions.

In this section we define an extension of the calculus in [19], to unbounded program executions. Moreover, we devise a corresponding sound over-approximation of "may aliasing" in terms of regular expressions, applicable in sequential contexts. This paves the way to developing an algorithm for the aliasing problem, as presented in Section 4. The machinery for collecting aliases is defined using a notation inspired by the $\mathbb{K}$ framework [36], as its operational flavor enables a straightforward integration within the SCOOP formalization in [1].

We proceed by recalling the notion of *alias relation* and a series of associated notations and basic operations, as introduced in [19].

Definition 1 *(Alias expressions [19]).* We call an *alias expression* a (possibly infinite) path of shape $x.y.z. \dots$, where x is a local variable, class attribute or Current, and $y, z, \dots$ are attributes. Here, *Current*, also known as *this* or *self*, stands for the current object. For an arbitrary alias expression e, it is the case that $e.Current = Current.e = e$. *Intuitively, an expression $e.Current$ makes sense if, for instance, the type of Current is the class of the objects reached by e.*

Definition 2 *(Alias relations [19]).* Let E represent the set of all expressions of a program. An *alias relation* is a symmetric and irreflexive binary relation $r \subseteq E \times E$. Let $[e_1, e_2] \in r$; we call $[e_1, e_2]$ an *alias pair* in r.

Given an alias relation r and an expression e, we define

$$r/e = \{e\} \cup \{x \colon E \mid [x, e] \in r\}$$

denoting the set consisting of all elements in r which are aliased to e, plus e itself.

Let x be an expression; we write $r - x$ to represent r without the pairs with one element of shape $x.e$:

$$r - x = \{[e_1, e_2] \mid \forall e{:}E\ .\ [e_1, e_2] \in r \wedge e_1 \neq x.e \wedge e_2 \neq x.e\}.$$

By $a \in dom(t)$ we refer to an attribute in the class corresponding to the object referred by the alias expression associated to t. For instance, given a class NODE with a field *next* of type NODE, and a NODE object x, we say that *next* is in the domain of $t = x.next.next$.

Intuitively, we say that an alias relation is *dot-complete*, whenever its elements are soundly extended via fields in their corresponding domains. The operation making a relation dot-complete is defined as follows:

$$\textit{dot-complete}(r) = r \cup \{[t.a, s.a] \mid [t, s] \in r \wedge a \in dom(t)\} \cup \{[s.a, v] \mid [t, s] \in r \wedge [t.a, v] \in r\}.$$

Let x be an alias expression and u be a set of expressions in E. We define the relation r augmented with pairs $[x, y]$, for $y \in u$, and made dot complete as

$$r[x = u] = \textit{dot-complete}(r') \text{ where } r' = r \cup \{[x, y] \mid y \in u\}.$$

3.1. Extension to unbounded executions

We further introduce an extension of the alias calculus in [19] to infinite alias relations corresponding to unbounded executions. The main difference in our approach is reflected by the definition of loops, which now complies to the usual fixed-point denotational semantics.

The alias calculus is defined by a set of axioms "describing" how the execution a program affects the aliasing between expressions. As in [19], the calculus ignores tests in conditionals and loops.

Definition 3 *(Program instructions [19]).* Let x, x' be variable names and s a valid sequence $x'.y_1.y_2 \ldots y_n$ with $y_1 \in dom(x')$, $y_2 \in dom(x'.y_1), \ldots, y_n \in dom(x'.y_1 \ldots y_{n-1})$. Let f be a routine name and l the actual parameters of a call of f. The *program instructions* are defined as follows:

$$\begin{aligned} p \ ::= \ & p\,;p \mid \textbf{then}\ p\ \textbf{else}\ p\ \textbf{end} \mid \\ & \textbf{create}\ x \mid \textbf{forget}\ x \mid x := s \mid \\ & \textbf{loop}\ p\ \textbf{end} \mid \textbf{call}\ f(l) \mid x.\textbf{call}\ f(l). \end{aligned} \tag{8}$$

We write $r \text{ » } p$ to represent the alias information obtained by executing p when starting with the initial alias relation r.

We further give the axioms of the extended alias calculus. The axiom for sequential composition is defined in the obvious way:

$$r \text{ » } (p\,;q) = (r \text{ » } p) \text{ » } q. \tag{9}$$

Conditionals are handled by considering the union of the alias pairs resulted from the execution of the instructions corresponding to each of the two branches, when starting with the same initial relation:

$$r \text{ » } (\textbf{then}\ p\ \textbf{else}\ q\ \textbf{end}) = r \text{ » } p\ \cup\ r \text{ » } q. \tag{10}$$

As previously mentioned, we define $r \text{ » } \textbf{loop}\ p\ \textbf{end}$ according to its informal semantics: "execute p repeatedly any number of times, including zero". The corresponding rule is:

$$r \text{ » } (\textbf{loop}\ p\ \textbf{end}) = \bigcup_{n \in \mathbb{N}} (r \text{ » } p^n) \tag{11}$$

where p^n is the sequential composition of p with itself for n times and $\bigcup$ stands for the union of alias relations, as above. This way, our calculus is extended to finite executions of any length. This is the main difference with the approach in [19] that proposes a "cutting" technique restricting the model to a maximum length L. In [19], sequences longer than L are considered as aliased to all expressions. Orthogonally, for sequential settings, we provide finite representations of infinite alias relations based on over-approximating regular expressions, as we shall see in Section 3.2.

Both the creation and the deletion of an object x eliminate from the current alias relation all the pairs having one element prefixed by x:

$$\begin{aligned} r \text{ » } (\textbf{create}\ x) &= r - x \\ r \text{ » } (\textbf{forget}\ x) &= r - x. \end{aligned} \tag{12}$$

The (qualified) function calls comply to their initial definitions in [19]:

$$\begin{aligned} r \text{ » } (\textbf{call}\ f(l)) &= (r[f^\bullet{:}l]) \text{ » } \mid f \mid \\ r \text{ » } (x.\textbf{call}\ f(l)) &= x.((x'.r) \text{ » } \textbf{call}\ f(x'.l)). \end{aligned} \tag{13}$$

Here $f^\bullet$ and $\mid f \mid$ stand for the formal argument list and the body of f, respectively, whereas $r[u{:}v]$ is the relation r in which every element of the list v is replaced by its counterpart in u.

Consider a call $x.\mathbf{call}\, f(l)$ executed on behalf of a certain client object, which applies f to a supplier object referenced by x. In order to compute the aliasing induced by $x.\mathbf{call}\, f(l)$ when starting with an aliasing environment r, we need to compute the aliasing induced by the execution of f from r as seen by the supplier object x. This supplier view is $x'.r$ with both elements of every pair in r prefixed by the negative reference x'. Intuitively, x' can be seen as a back pointer giving access to the client. In accordance, the alias calculus evaluates $x'.x$ to $Current$. Intuitively, the negative variable x' is meant to transpose the context of the qualified call to the context of the caller.

For an example, consider a class C in an OO-language, and an associated procedure f that assigns a local variable y, defined as: $f(x)\{\ y := x\ \}$. Then, for instance, the aliasing for $a.\mathbf{call}\, f(a)$ computes as follows:

$$\begin{aligned} \emptyset \gg a.\mathbf{call}\, f(a) &= \\ a.(a'.\emptyset \gg y := a'.a) &= \\ a.(\emptyset \gg y := Current) &= \\ \mathit{dot\text{-}complete}(\{[a.y, a]\}). \end{aligned}$$

Recursive function calls can lead to infinite alias relations if they do not terminate. In sequential settings, as for the case of loops, the mechanism exploiting sound regular over-approximations in order to derive finite representations of such relations is presented in the subsequent sections.

The axiom for assignment is as well in accordance with its original counterpart in [19]:

$$\begin{aligned} r \gg (x := s) \;=\; & (r_1 - x)[x = (r_1/s - x)] - ox \\ & \text{where } r_1 = r[ox = x] \end{aligned} \tag{14}$$

where ox is a fresh variable (that stands for "old x"). Intuitively, the aliasing information w.r.t. the initial value of x is "saved" by associating x and ox in r and closing the new relation under dot-completeness, in r_1. Then, the initial x is "forgotten" by computing $r_1 - x$ and the new aliasing information is added in a consistent way. Namely, we add all pairs (x, s'), where s' ranges over $r_1/s - x$ representing all expressions already aliased with s in r_1, including s itself, but without x. Recall that alias relations are not reflexive, thus by eliminating x we make sure we do not include pairs of shape $[x, x]$. Then, we consider again the closure under dot-completeness and forget the aliasing information w.r.t. the initial value of x, by removing ox.

Remark 4. It is worth discussing the reason behind *not* considering transitive alias relations. Assume the following program:

$\mathbf{then}\; x := y\; \mathbf{else}\; y := z\; \mathbf{end}$

Based on the equations (10) and (14) handling conditionals and assignments, respectively, the calculus correctly identifies the alias set: $\{[x, y], [y, z]\}$. Including $[x, z]$ would be semantically equivalent to the execution of the two branches in the conditional at the same time, which is not what we want.

3.2. A sound over-approximation

In a non-concurrent or, sequential setting, the challenge of computing the alias information in the context of (infinite) loops and recursive calls reduces to evaluating their corresponding "unfoldings", captured by expressions of shape

$r \gg p^n,$

with n ranging over naturals $\mathbb{N}$, r an (initial) alias relation ($r = \emptyset$), and p a *basic block* defined by:

$$\begin{aligned} p \;::=\; & p\,;p \mid \mathbf{then}\; p\; \mathbf{else}\; p\; \mathbf{end} \mid \\ & \mathbf{create}\; x \mid \mathbf{forget}\; x \mid \\ & x := s. \end{aligned} \tag{15}$$

The value $r \gg p^n$ refers to the alias relation obtained by recursively executing the control block p for n times, and it is calculated in the expected way:

$$\begin{aligned} r \gg p^0 &= r \\ r \gg p^{k+1} &= (r \gg p^k) \gg p. \end{aligned}$$

Consider again the code in (6):

```
**create** $x_0$
**loop**
   $i := i + 1$
   **create** $x_i$
   $x_i.set\_next(x_{i-1})$
**end**
```

Its execution generates an alias relation including an infinite number of pairs of shape:

$$[x_i, x_{i+1}.next],\ [x_i, x_{i+2}.next.next],\ [x_i, x_{i+3}.next.next.next] \ldots\ . \tag{16}$$

In what follows we provide a way to compute finite representations of infinite alias relations in sequential settings. The key observation is that alias expressions corresponding to unbounded program executions grow in a regular fashion. See, for instance, the aliases in (16), which are pairs of type $[x_i, x_{i+k}.next^{k \geq 1}]$.

Regular alias expressions are defined similarly to the regular languages over an alphabet. We say that an alias expression is *regular* if it is a local variable, class attribute or *Current*. Moreover, the concatenation $e_1 \,.\, e_2$ and the choice $e_1 + e_2$ of two regular alias expressions e_1 and e_2 are also regular. Given a regular alias expression e, the expression e^* is also regular; here $(-)^*$ denotes the Kleene star. We call an alias relation *regular* if it consists of pairs of regular alias expressions.

Proposition 5. *Let r be a regular alias relation and*

$$p ::= \textbf{\textit{create}}\ x \mid \textbf{forget}\ x \mid x := s.$$

In a sequential setting, it is the case that r » p is regular.

Proof. First, observe that the name of a variable x and the right-hand side of an assignment $x := s$ as introduced in Definition 3, in Section 3, are regular alias expressions. Hence, by Definition 2, it follows that r/s, $r - x$, *dot-complete*(r) and $r[x = s]$ preserve regularity of alias relations. Consequently, by (12) and (14) r » **create** x, r » **forget** x and r » $x := s$ are regular alias relations. □

Lemma 6. *Let r be a regular alias relation and p a basic block as in (15). In a sequential setting, the following hold:*

(a) *r » p is regular;*
(b) *r » p^n, with $n \geq 1$, is regular.*

Proof. The proof of (a) follows by induction on the structure of p.

Base case: $p ::=$ **create** x | **forget** x | $x := s$. The result follows by Proposition 5.
Induction step. Assume r' » p' is regular for all regular relations r' and all basic blocks p' with simpler structure than p.
Let $p =$ **then** p_1 **else** p_2 **end**. By definition, $\bar{r} = r$ » **then** p_1 **else** p_2 **end** $= (r$ » $p_1) \cup (r$ » $p_2)$. Hence, by the induction hypothesis, it follows that $\bar{r}$ is regular.
Let $p = p_1\,;p_2$. By definition, $\bar{r} = r$ » $p_1\,; p_2 = (r$ » $p_1)$ » p_2. By the induction hypothesis, it holds that $\tilde{r} = (r$ » $p_1)$ is regular. Hence, by the induction hypothesis, it follows that $\bar{r} = \tilde{r}$ » p_2 is regular as well.

The proof of (b) is by induction on n. The base case, when $n = 1$, follows by Lemma 6(a). The induction step follows immediately by the induction hypothesis, as r » $p^{n+1} = (r$ » $p^n)$ » p. □

Lemma 7. *Let r be a regular alias relation and*

$$\begin{aligned} p \ ::= \ & p\,; p \mid \textbf{then}\ p\ \textbf{else}\ p\ \textbf{end} \mid \\ & \textbf{\textit{create}}\ x \mid \textbf{forget}\ x \mid x := s \mid \\ & \textbf{loop}\ p\ \textbf{end} \end{aligned} \tag{17}$$

In a sequential setting, the following hold:

(a) *r » p is regular;*
(b) *r » p^n, with $n \geq 1$, is regular.*

Proof. The proof of (a) is by induction on n – the number of *nested* loops in p.

Base case: $n = 0$. The result follows immediately by Lemma 6(a), for p a basic block as in (15).
Let $p = p_1\,;$ **loop** q **end**$\,; p_2$, with p_1, p_2 and q basic blocks as in (15). The following holds:

$$r' = r \text{ » } p_1\,; \textbf{loop}\ q\ \textbf{end}\,; p_2 = \bigcup_{n \in \mathbb{N}} r \text{ » } p_1 \text{ » } q^n \text{ » } p_2.$$

Hence, by Lemma 6(b), r' is regular.
Induction step. Assume r' » p' is regular for all regular relations r' and p' instruction blocks with maximum n nested loops, built as in (17). Let $p = p_1\,;$ **loop** q **end**$\,; p_2$, with p_1, p_2 and q instruction blocks with maximum n nested loops, built as in (17). It is the case that $\bar{r} = r$ » $p_1\,;$ **loop** q **end**$\,; p_2 = \bigcup_{n \in \mathbb{N}} r$ » p_1 » q^n » p_2. Hence, $\bar{r}$ is regular by the induction hypothesis.

The proof of (b) is by induction on n. The base case, when $n = 1$, follows by Lemma 7(a). The induction step follows immediately by the induction hypothesis, as $r \gg p^{n+1} = (r \gg p^n) \gg p$. $\square$

Lemma 8. *Let r be a regular alias relation and f a function with $|\, f\,|$ as in (17). In a sequential setting, the following hold:*

(a) $r \gg \textbf{call}\, f(l_0)$ *and* $r \gg x.\textbf{call}\, f(l_0)$*;*
(b) $r \gg \textbf{call}\, f(l_0) \gg \ldots \gg \textbf{call}\, f(l_n)$ *and* $r \gg x.\textbf{call}\, f(l_0) \gg \ldots \gg x.\textbf{call}\, f(l_n)$

are regular, where $n \in \mathbb{N}$ and l_i $(0 \leq i \leq n)$ represent valid lists of actual arguments of f.

Proof sketch. The result is a direct consequence of Lemma 7 and exploits the definition of (qualified) function calls in (13). $\square$

We call *non-recursive control blocks* the control blocks built as in (8), that lack (mutual) recursion.

Lemma 9. *Let r be a regular alias relation and p a non-recursive control block. In a sequential setting, the following hold:*

(a) *$r \gg p$ is regular;*
(b) *$r \gg p^n$, with $n \geq 1$, is regular.*

Proof sketch. The result is a direct consequence of Lemma 7 and Lemma 8. $\square$

We call *non-qualified recursive functions* the (mutually) recursive functions built as in (8), that lack qualified (mutually) recursive calls.

Next, we show that the execution of non-qualified recursive calls lead to regular relations as well. In short, our approach consists of providing a *flattening* of the recursive functions by extracting the control blocks that do not include (mutually) recursive calls. Eventually, we combine these blocks via sequencing ; and exponentiation $(-)^n$ in order to derive over-approximating aliases. This technique cannot be applied in the context of qualified function calls, as it looses the necessary information for transposing the context of the qualified call to the context of the caller.

We first provide an over-approximating result for the (instructive) case of a simple recursive function, where, intuitively, the "flattening" procedure returns two non-recursive blocks (p_1 and p_2).

Lemma 10. *Let r be a regular alias relation and*

$$f(l') = \{p_1\, ;\, \textbf{call}\, f(l'')\, ;\, p_2\}$$

be a non-qualified recursive function, with p_1, p_2 non-recursive control blocks and l', l'' valid argument lists. In a sequential setting, it is the case that

$$r \gg \textbf{call}\, f(l_0)$$

is regular, where l_0 is a valid list of actual parameters of f.

Proof. First, for simplicity of notation, we write l_i, $i \geq 1$, in order to refer to the actual parameters of f corresponding to its i'th recursive call. Then:

$$\begin{aligned}
&r \gg \textbf{call}\, f(l_0) \;=\; r[f^\bullet{:}l_0] \gg |\, f\,| \;=\; r[f^\bullet{:}l_0] \gg p_1 \gg \textbf{call}\, f(l_1) \gg p_2 \;=\\
&(r[f^\bullet{:}l_0] \gg p_1)[f^\bullet{:}l_1] \gg p_1 \gg \textbf{call}\, f(l_2) \gg p_2 \gg p_2 \;=\; \ldots \;=\\
&((r[f^\bullet{:}l_0] \gg p_1)[f^\bullet{:}l_1] \gg p_1 \ldots)[f^\bullet{:}l_n] \gg p_1 \gg \textbf{call}\, f(l_n) \gg p_2^n,
\end{aligned}$$

for $n \in \mathbb{N}$. Let $\bar{r}$ be the alias relation obtained from r by repeatedly replacing the actual arguments of f with their formal counterparts, in the reasoning above. Thus, also based on the results in Lemma 7, it follows that $\bar{r}$ is regular. Additionally, the aliasing information corresponding to $r \gg \textbf{call}\, f(l_0)$ can be over-approximated by $\bigcup_{i,j \in \mathbb{N}} \bar{r} \gg p_1^i \gg p_2^j$. Hence, as $\bar{r}$ is regular and p_1, p_2 are non-recursive control blocks, the statement in Lemma 10 is a consequence of Lemma 9. $\square$

Definition 11 *(Flattening).* Consider f, g two non-qualified recursive functions, S a non-recursive control block, and C a possibly recursive, but non-qualified recursive, control block. S and C can also be empty, *i.e.*, they are semantically equivalent with an empty sequencing (;). Moreover, f and g might refer to the same function.

By abuse of notation, we write $S^{[f^\bullet:u]}$ in order to refer to the block S "enriched" with the information $[f^\bullet{:}u]$ necessary according to (13) in order to replace the list of actual arguments u of the function f with its formal counterpart $f^\bullet$.

The flattening of $f(u)$ is recursively defined as follows:

$$\begin{aligned}
flat(f(u)) &= flat(|\,f\,|, [f^{\bullet}{:}u]) \\
flat(;, [f^{\bullet}{:}u]) &= \emptyset \\
flat(S, [f^{\bullet}{:}u]) &= \{S^{[f^{\bullet}:u]}\} \text{ if } S \neq ; \\
flat(S;g(v);C, [f^{\bullet}{:}u]) &= flat(S, [f^{\bullet}{:}u]) \cup flat(C, [f^{\bullet}{:}u]) \cup flat(g(v))
\end{aligned}$$

As expected, the execution $r \,\text{»}\, S^{[f^{\bullet}:u]}$ is defined as $r[f^{\bullet}{:}u] \,\text{»}\, S$.

Lemma 12. *Let r be a regular alias relation and f a non-qualified (possibly mutually) recursive function. In a sequential setting, it is the case that*

$$r \,\text{»}\, \textbf{call}\ f(l)$$

is regular, where l is a valid list of actual parameters of f.

Proof sketch. First, note that the alias relation $r \,\text{»}\, \textbf{call}\ f(a)$ can be soundly over-approximated by

$$\bigcup_{\substack{k \in \{1,\ldots,n\} \\ h_k,\, w \in \mathbb{N} \\ q_k \in flat(f(l))}} \bar{r} \,\text{»}\, (q_1^{h_1}\,;\,\ldots\,;q_n^{h_n})^w \tag{18}$$

where n stands for the size of $flat(f(l))$, and $\bar{r}$ is obtained from r by repeatedly replacing the actual arguments of the (possibly mutually) recursive calls with their formal counterparts, according to the semantics of the "enriched" control blocks q_i. As $\bar{r}$ is regular, the statement in Lemma 12 is a consequence of Lemma 9. □

Theorem 13. *Assume p a program built according to the rules in (8) that does not contain qualified recursive calls. Let r be a regular relation. Then, in a sequential setting, the relations $r \,\text{»}\, p$ and $r \,\text{»}\, p^n$, with $n \in \mathbb{N}$, are regular.*

Proof sketch. The proof follows by induction on the structure of p and Lemmas 6–12. □

Inspired by the idea behind the *pumping lemma for regular languages* [37], in what follows, we show that in sequential settings, executions of shape $r \,\text{»}\, p^n$, for $n \in \mathbb{N}$, can be over-approximated by a regular alias relation "closed under the application of p". More precisely, there exists a regular alias relation R that includes all aliases produced by $r \,\text{»}\, p^n$, for all $n \in \mathbb{N}$.

Definition 14 *(Closure under program executions).* We write $words(e)$ in order to refer to the set of words corresponding to the regular alias expression e. Let r be a regular alias relation. We write $reg^*(r)$ in order to represent a finite regular alias relation R such that:

$$\text{if } [e_1, e_2] \in r \text{ then } \exists [E_1, E_2] \in R \text{ s.t. } words(e_1) \subseteq words(E_1) \text{ and } words(e_2) \subseteq words(E_2)$$

By abuse of notation, we write $r \subseteq R$ whenever if $[e_1, e_2] \in r$ then $\exists [E_1, E_2] \in R$ s.t. $words(e_1) \subseteq words(E_1)$ and $words(e_2) \subseteq words(E_2)$. Let p be a control block as in (8). We say that $reg^*(r) = R$ is *closed under the execution of p* whenever $R \,\text{»}\, p^n \subseteq R$, for all $n \in \mathbb{N}$.

Lemma 15. *Consider r a regular alias relation, and p a block as in (8) that does not contain qualified recursive calls. In a sequential setting, it is the case that there exists $m \in \mathbb{N}$ and R a regular alias relation such that $reg^*(r \,\text{»}\, p^m) = R$ is closed under the execution of p.*

Proof sketch. The result follows as a consequence of the "pumping lemma for regular languages". Recall that, due to the execution of loops and (unqualified) recursive function calls, alias expressions $[e_1, e_2]$ in $r \,\text{»}\, p^i$, with $i \in \mathbb{N}$, grow in a regular fashion. Hence, there must be a "sufficiently large" $m \in \mathbb{N}$ and R a regular alias relation such that expressions e_1 and e_2 may be "pumped". That is, e_1 and e_2 have a middle sections repeated an arbitrary number of times, to produce new expressions that also belong to R. □

4. A RL-based procedure for collecting aliases

In this section we provide the specification of a RL-based mechanism collecting the alias information. The RL rules are specified using the $\mathbb{K}$ notation for configurations and are implemented in Maude on top of the formalization of SCOOP [1] (we refer to Section 5 for more details on our approach).

In short, our strategy is to start with a program built on top of the control structures in (8), then to apply their corresponding $\mathbb{K}$-rules in order to get the "may aliasing" information in a designated $\mathbb{K}$-cell ($\langle - \rangle_{\text{al}}$). Independently of the setting (sequential or concurrent) one can exploit this approach in order to evaluate the aliases of a given finite length L. We also show that for sequential contexts, for control blocks not including qualified recursive calls, the application of the $\mathbb{K}$-like rules is finite and the aliases in the final configuration soundly over-approximate the (infinite) “may alias” relations of the calculus.

$\mathbb{K}$-like notations for configurations and (RL) rules $\mathbb{K}$ [38,36] is an executable semantic framework originating from Rewriting Logic [5], suitable for defining (concurrent) languages and corresponding formal analysis tools. $\mathbb{K}$-definitions make use of the so-called *cells*, which are labeled and can be nested, and (rewriting) *rules* describing the intended (operational) semantics. In this section we adopt a $\mathbb{K}$-like notation in order to provide a RL-based specification of the alias calculus.

A *cell* is denoted by $\langle - \rangle_{[\mathrm{name}]}$, where [name] stands for the *name of the cell*. Intuitively, a construction $\langle\ .\ \rangle_{\mathrm{n}}$ stands for an *empty cell* named n. We use “pattern matching” and write $\langle\ c\ \ldots\rangle_{\mathrm{n}}$ for a cell with content c at the top, followed by an arbitrary content $(\ldots)$. We can utilize cells of shape $\langle \ldots\ c\ \rangle_{\mathrm{n}}$ and $\langle\ \ldots c \ldots\ \rangle_{\mathrm{n}}$, defined in the obvious way.

Of particular interest is $\langle - \rangle_{\mathrm{k}}$ – the *continuation cell* holding the stack of program instructions, in the context of a programming language formalization. We write

$$\langle\ i_1 \curvearrowright i_2\ \ldots\rangle_{\mathrm{k}}$$

for a set of instructions to be “executed”, starting with instruction i_1, followed by i_2. The associative operation $\curvearrowright$ is the instruction sequencing.

A rule

$$\langle\ \frac{c}{c'}\ \ldots\rangle_{n_1}\ \langle\ c'\ \rangle_{n_2}\ \langle \ldots\ \frac{.}{c'}\ \rangle_{n_3}$$

reads as: if cell n_1 has c at the top and cell n_2 contains value c', then c is replaced by c' in n_1 and c' is added at the end of the cell n_3. The content of n_2 remains unchanged.

We further provide the details behind the $\mathbb{K}$-like specification of the alias calculus. As expected, the k-cell retains the instruction stack of the object-oriented program. We utilize cells $\langle - \rangle_{\mathrm{al}}$ to enclose the current alias information, and the so-called *back-tracking cells* $\langle - \rangle_{\text{bkt-}\ldots}$ enabling the sound computation of aliases for the case of **then** – **else** – **end** and, in non-concurrent contexts, for **loop** – **end** structures and (possibly recursive) function calls. As a convention, we mark with (♣) the rules that are sound only for non-concurrent applications, based on Lemma 15.

The following $\mathbb{K}$-like rules are straightforward, based on the axioms (9)–(14) in Section 3.1. Namely, the rule implementing an instruction $p\,;q$ simply forces the sequential execution of p and q by positioning $p \curvearrowright q$ at the top of the continuation cell:

$$\langle\ \frac{p\,;q}{p \curvearrowright q}\ \ldots\rangle_{\mathrm{k}} \tag{19}$$

Handling **create** x and **forget** x complies to the associated definitions. Namely, it updates the current alias relation by removing all the pairs having (at least) one element with x as prefix. In addition, it also pops the corresponding instruction from the continuation stack:

$$\langle\ \frac{r}{r-x}\ \rangle_{\mathrm{al}}\ \langle\ \frac{\textbf{create}\ x}{.}\ \ldots\rangle_{\mathrm{k}} \qquad \langle\ \frac{r}{r-x}\ \rangle_{\mathrm{al}}\ \langle\ \frac{\textbf{forget}\ x}{.}\ \ldots\rangle_{\mathrm{k}} \tag{20}$$

The assignment rule restores the current alias relation according to its axiom in (14), and removes the assignment instruction from the top of the k-cell:

$$\langle\ \frac{r}{(r_1 - t)[t = (r_1/s - t)] - ot}\ \rangle_{\mathrm{al}}\ \langle\ \frac{t := s}{.}\ \ldots\rangle_{\mathrm{k}} \quad \text{with } r_1 = r[ot = t] \tag{21}$$

Handling **then** p **else** q **end** statements is more sophisticated, as it requires instrumenting a stack-based mechanism enabling the computation of the union of alias relations $r \gg p \cup r \gg q$ in three steps. First, we define the $\mathbb{K}$-like rule:

$$\langle\ r\ \rangle_{\mathrm{al}}\ \langle\ \frac{\textbf{then}\ p\ \textbf{else}\ q\ \textbf{end}}{p\ \boxed{\mathrm{et}}\ q\ \boxed{\mathrm{ee}}}\ \ldots\rangle_{\mathrm{k}}\ \langle\ \frac{.}{\langle\ r, p\ \rangle_{\mathrm{t}}\ \langle\ r, q\ \rangle_{\mathrm{e}}}\ \ldots\rangle_{\text{bkt-te}} \tag{22}$$

saving at the top of the back-tracking stack $\langle - \rangle_{\text{bkt-te}}$ the initial alias relation r to be modified by both p and q, via two cells $\langle r, p\rangle_{\mathrm{t}}$ and $\langle r, q\rangle_{\mathrm{e}}$, respectively. Note that the original instruction in the k-cell is replaced by a construction marking the end of the executions corresponding to the **then** and **else** branches with $\boxed{\mathrm{et}}$ and $\boxed{\mathrm{ee}}$, respectively.

Second, whenever the successful execution of p (signaled by $\boxed{\mathrm{et}}$) at the top of the k-cell) builds an alias relation r', the execution of q starting with the original relation r is forced by replacing r' with r in $\langle - \rangle_{\mathrm{al}}$, and by positioning $q\ \boxed{\mathrm{ee}}$ at the top of the k-cell. The new alias information after p, denoted by $\langle r', p\rangle_{\mathrm{t}}$, is updated in the back-tracking cell:

$$\langle\ \frac{r'}{r}\ \rangle_{\mathrm{al}}\ \langle\ \frac{\boxed{\mathrm{et}}\ q\ \boxed{\mathrm{ee}}}{q\ \boxed{\mathrm{ee}}}\ \ldots\rangle_{\mathrm{k}}\ \langle\ \frac{\langle\ r, p\ \rangle_{\mathrm{t}}}{\langle\ r', p\ \rangle_{\mathrm{t}}}\ \langle\ r, q\ \rangle_{\mathrm{e}}\ \ldots\rangle_{\text{bkt-te}} \tag{23}$$

Eventually, if the successful execution of q (marked by $\boxed{\mathrm{ee}}$ at the top of $\langle - \rangle_{\mathrm{k}}$) produces an alias relation r'', then the final alias information becomes $r' \cup r''$, where r' is the aliasing after p, stored as showed in (23). The corresponding back-tracking information is removed from $\langle - \rangle_{\mathrm{bkt\text{-}te}}$, and the next program instruction is enabled in the k-cell:

$$\langle \frac{r''}{r' \cup r''} \rangle_{\mathrm{al}} \ \langle \frac{\boxed{\mathrm{ee}}}{\cdot} \ldots \rangle_{\mathrm{k}} \ \langle \frac{\langle r', p \rangle_{\mathrm{t}} \ \langle r, q \rangle_{\mathrm{e}}}{\cdot} \ldots \rangle_{\mathrm{bkt\text{-}te}} \tag{24}$$

For **loop** p **end**, we utilize a meta-construction $p \ \boxed{\mathrm{l}}$ **loop** p **end** simulating the set union in (11), and a back-tracking stack $\langle - \rangle_{\mathrm{bkt\text{-}l}}$ collecting the alias information closed under the repeated execution of p. Moreover, the $\mathbb{K}$-like specification exploits the result in Lemma 15. Whenever a "closed" regular alias relation is reached, the infinite rewriting is prevented by resuming the infinite application of p in terms of a sound over-approximating alias relation. The $\mathbb{K}$-rules are as follows.

First, the aforementioned unfolding step is performed and the "back-tracking" information is set to $\emptyset$:

$$\langle\, r \,\rangle_{\mathrm{al}} \ \langle \frac{\mathbf{loop}\ p\ \mathbf{end}}{p\ \boxed{\mathrm{l}}\ \mathbf{loop}\ p\ \mathbf{end}} \ldots \rangle_{\mathrm{k}} \ \langle \frac{\cdot}{\emptyset} \ldots \rangle_{\mathrm{bkt\text{-}l}} \tag{25}$$

The process of storing/building the regular alias relation closed under the application of p starts after the first execution of p, marked by $\boxed{\mathrm{l}}$ at the top of the k-cell and $\emptyset$ at the top of $\langle - \rangle_{\mathrm{bkt\text{-}l}}$:

$$\langle\, r \,\rangle_{\mathrm{al}} \ \langle \frac{\boxed{\mathrm{l}}\ \mathbf{loop}\ p\ \mathbf{end}}{p\ \boxed{\mathrm{l}}\ \mathbf{loop}\ p\ \mathbf{end}} \ldots \rangle_{\mathrm{k}} \ \langle \frac{\emptyset}{reg^{*}(r)} \ldots \rangle_{\mathrm{bkt\text{-}l}} \tag{26}$$

If the alias relation r' obtained after the successful execution of p (marked by $\boxed{\mathrm{l}}$ at the top of the continuation) is not in the "closure" of R – the aliasing previously stored in $\langle - \rangle_{\mathrm{bkt\text{-}l}}$ – then p is constrained to a new execution by becoming the top of the k-cell, and $reg^{*}(r' \cup R)$ is memorized for back-tracking:

$$\langle\, r' \,\rangle_{\mathrm{al}} \ \langle \frac{\boxed{\mathrm{l}}\ \mathbf{loop}\ p\ \mathbf{end}}{p\ \boxed{\mathrm{l}}\ \mathbf{loop}\ p\ \mathbf{end}} \ldots \rangle_{\mathrm{k}} \ \langle \frac{R}{reg^{*}(R \cup r')} \ldots \rangle_{\mathrm{bkt\text{-}l}} \quad \text{if } r' \not\subseteq R \quad (\clubsuit) \tag{27}$$

Last, if a closure is reached after the execution of p, then the current aliasing is soundly replaced by a "regular" over-approximation R, the corresponding back-tracking information is removed from $\langle - \rangle_{\mathrm{bkt\text{-}l}}$ and the **loop** instruction is eliminated from the k-cell:

$$\langle \frac{r'}{R} \rangle_{\mathrm{al}} \ \langle \frac{\boxed{\mathrm{l}}\ \mathbf{loop}\ p\ \mathbf{end}}{\cdot} \ldots \rangle_{\mathrm{k}} \ \langle \frac{R}{\cdot} \ldots \rangle_{\mathrm{bkt\text{-}l}} \quad \text{if } r' \subseteq R \quad (\clubsuit) \tag{28}$$

For handling function calls such as **call** $f(l)$ we use a meta-construction $\mid f \mid \boxed{\mathrm{f}}$. Here $\mid f \mid$ stands for the body of f and $\boxed{\mathrm{f}}$ marks the end of the corresponding execution.

The first $\mathbb{K}$-like rule for handling function calls **call** $f(l)$ matches the associated axiom in (13): the alias information is set to $r[f^{\bullet}{:}l]$, whereas the next instructions to be executed are given by $\mid f \mid$:

$$\langle \frac{r}{r[f^{\bullet}{:}l]} \rangle_{\mathrm{al}} \ \langle \frac{\mathbf{call}\ f(l)}{\mid f \mid\ \boxed{\mathrm{f}}} \ldots \rangle_{\mathrm{k}} \tag{29}$$

A successful execution of **call** $f(l)$ is distinguished by the occurrence of $\boxed{\mathrm{f}}$ at the top of the continuation stack. If this is the case, then the next program instruction (if any) is enabled at the top of the k-cell:

$$\langle\, r' \,\rangle_{\mathrm{al}} \ \langle \frac{\boxed{\mathrm{f}}}{\cdot} \ldots \rangle_{\mathrm{k}} \tag{30}$$

Qualified calls $x.$**call** $f(l)$ are handled by two rules as follows. First, based on the definition in (13), the "negative variable" x' transposing the context of the call to the context of the caller is distributed to the elements of the initial alias relation r, and to l – the argument list of f. Moreover, a meta-construction $\boxed{\mathrm{qf}}$ is utilized in order to mark the end of the qualified call in the continuation cell, similarly to the rule (29). The caller is stored in a back-tracking stack $\langle\, .\, \rangle_{\mathrm{bkt\text{-}qf}}$ also parameterized by f – the name of the function. The current instruction in the k-cell becomes **call** $f(x'.l)$, as expected:

$$\langle \frac{r}{x'.r} \rangle_{\mathrm{al}} \ \langle \frac{x.\mathbf{call}\ f(l)}{\mathbf{call}\ f(x'.l)\ \boxed{\mathrm{qf}}} \ldots \rangle_{\mathrm{k}} \ \langle \frac{\cdot}{\langle\, x \,\rangle_{\mathrm{f}}} \ldots \rangle_{\mathrm{bkt\text{-}qf}} \tag{31}$$

Second, when the successful termination of the qualified call is signaled by $\boxed{\mathrm{qf}}$ at the top of the k-cell, the corresponding stored caller is distributed to the current alias relation and removed from the back-tracking cell. The next instruction in the continuation cell is released by eliminating the top $\boxed{\mathrm{qf}}$:

$$\langle \frac{r}{x.r} \rangle_{\mathrm{al}} \ \langle \frac{\boxed{\mathrm{qf}}}{\cdot} \ldots \rangle_{\mathrm{k}} \ \langle \frac{\langle\, x \,\rangle_{\mathrm{f}}}{\cdot} \ldots \rangle_{\mathrm{bkt\text{-}qf}} \tag{32}$$

In sequential settings, non-qualified recursive calls **call** $f(l)$ are handled in the spirit of (18) in Lemma 12. More precisely, we are combining the non-recursive control blocks obtained by “flattening” $f(l)$ via **loop** $-$ **end** and **then** $-$ **else** $-$ **end** constructions, in order to derive a sound over-approximation of the corresponding alias information. This is achieved via the **TE** function:

$$\begin{aligned}\mathbf{TE}(\emptyset) &= ;\\ \mathbf{TE}(S^{[f^{\bullet}:u]} \cup Set) &= \mathbf{then}\ S^{[f^{\bullet}:u]}\ \mathbf{else}\ \mathbf{TE}(Set)\ \mathbf{end}\end{aligned}$$

where *Set* stands for a set of “enriched” non-recursive control blocks. The associated RL-rule reducing non-qualified recursive calls to loop-like behavior is:

$$\langle\ r\ \rangle al\ \langle \frac{\mathbf{call}\ f(l)}{\mathbf{loop}\ \mathbf{TE}(flat(l))\ \mathbf{end}} \ldots\rangle_{\mathsf{k}} \qquad (\clubsuit) \tag{33}$$

Note that, in sequential settings, for the case of non-qualified recursive calls **call** $f(l)$, rule (33) replaces rule (29).

Remark 16. Observe that the RL-based machinery can simulate precisely the “cutting at length L” technique in [19]. It suffices to: eliminate the back-tracking cell $\langle\ .\ \rangle_{\mathsf{bkt\text{-}l}}$ in the rules (25) and (26) defining **loop** $-$ **end**, disable the rules ($\clubsuit$) and stop the rewriting after L steps.

The naturalness of applying the resulted aliasing framework is illustrated in the example in Section 4.2.

4.1. Termination

In this section, we show that the proposed RL-machinery can terminate with a sound over-approximation of “may aliasing”. The result is in close connection with Lemma 15, stating that, in sequential settings, an over-approximating alias relation closed under the application of control blocks that do not contain qualified recursive calls can always be constructed.

For instance, for identifying over-approximating regular expressions $reg^*(-)$ as in (27), we can draw inspiration from the *pumping lemma for regular languages* [37]. We proceed by defining a *lasso* property for alias relations, which identifies the repetitive patterns within the structure of the corresponding alias expressions. The intuition is that such patterns will occur for an “infinite” number of times due to unbounded executions.

Definition 17 *(Lassos).* Consider r and r' two alias relations, and x_i, y_i and z_i a set of possibly empty expressions, for $i \in \{1, 2\}$. A *lasso* property over r and r' is defined as:

$$\text{lasso}(r, r') = ([x_1.y_1.z_1, x_2.y_2.z_2] \in r \ \text{ iff } \ [x_1.y_1.y_1.z_1, x_2.y_2.y_2.z_2] \in r'). \tag{34}$$

Intuitively, there is a lasso between r and r' whenever the elements in r' correspond to elements in r for which tails of prefixes are repeated.

Definition 18 *(Over-approximating regular aliases).* Consider r and r' such that $\text{lasso}(r, r') = true$. Let x_i, y_i and z_i be possibly empty regular expressions, for $i \in \{1, 2\}$. The set of regular alias expressions over-approximating r and r' is defined as:

$$\begin{aligned}\text{reg}(r, r') = \{&[x_1.y_1^*.z_1, x_2.y_2^*.z_2] \mid \\ &[x_1.y_1.z_1, x_2.y_2.z_2] \in r \land \\ &[x_1.y_1.y_1.z_1, x_2.y_2.y_2.z_2] \in r'\}\end{aligned} \tag{35}$$

Intuitively, $\text{reg}(r, r')$ plays the rôle of $reg^*(R \cup r')$ in (27), where $R = r$.

A lasso property can be identified, for instance, in Fig. 1, based on the relations

$$r_1 = \{[x, x.a.b],\ [x.a, x.a.b.a],\ [x.b, x.a.b.b]\} \text{ and } r_2 = \{[x, x.a.b.a.b],\ [x.a, x.a.b.a.b.a],\ [x.b, x.a.b.a.b.b]\}.$$

The over-approximating regular aliasing is $\{[x, x.(a.b)^*],\ [x.a, x.(a.b)^*.a],\ [x.b, x.(a.b)^*.b]\}$.

In accordance with Lemma 15, lassos are closed under the application of control blocks as in (15), necessary to implement the rules (19)–(33).

Lemma 19. *Consider r and r′ two alias relations, and p a basic block as in (15). In a sequential setting, if $r \gg p = r'$ and $\text{lasso}(r, r') =$ true, then the following holds for all $n \geq 1$:*

$$r \gg p^n \subseteq \text{reg}(r, r').$$

Proof. We proceed by induction on n.

- *Base case*: $n = 1$. By hypothesis it holds that $\mathrm{lasso}(r, r') = \mathit{true}$. Hence, according to the definition of $\mathrm{lasso}(-, -)$ in (34), the following holds for r and r':

  $$[x_1.y_1.z_1, x_2.y_2.z_2] \in r \text{ iff } [x_1.y_1.y_1.z_1, x_2.y_2.y_2.z_2] \in r'.$$

  Consequently, by the definition of $\mathrm{reg}(-, -)$ in (35), it is easy to see that

  $$r' \subseteq \mathrm{reg}(r, r').$$

- *Induction step.* Fix a natural number n and suppose that

  $$r_k = r \gg p^k \subseteq \mathrm{reg}(r, r') \tag{36}$$

  for all $k \in \{1, \ldots, n\}$. We want to prove that (36) holds also for $k = n + 1$.
  We continue by "reductio ad absurdum". Consider

  $$r_n = r \gg p^n \subseteq \mathrm{reg}(r, r'),$$

  and assume that

  $$r_{n+1} = r_n \gg p \not\subseteq \mathrm{reg}(r, r') \tag{37}$$

  Clearly, the execution of p when starting with r_n introduces an alias pair which is not in $\mathrm{reg}(r, r')$. According to (12), only instructions **create** x and **forget** x that appear within p, can remove alias pairs from r_n. Hence, it must be the case that the regular structure of the alias information is broken via a newly added pair $[t', s']$ associated to an assignment $x := s$ within p.

  Let $p = C[x := s]$, where C is a context built according to (15), and $x := s$ is the upper-most assignment instruction in the syntactic tree associated to p that introduces a pair $[t', s'] \notin \mathrm{reg}(r, r')$. Assume that $\widetilde{r}_{n-1}$ and $\widetilde{r}_n$, respectively, are the intermediate alias relations obtained by reducing $r_{n-1} \gg C[x := s]$ and $r_n \gg C[x := s]$, respectively, according to (9)–(14), before the application of the assignment axiom corresponding to $x := s$. As only **create** x and **forget** x can remove alias pairs from r_{n-1} and r_n, respectively, and as $x := s$ is the first assignment within p, it follows that $\widetilde{r}_{n-1} \subseteq r_{n-1} \subseteq \mathrm{reg}(r, r')$ and $\widetilde{r}_n \subseteq r_n \subseteq \mathrm{reg}(r, r')$.

  Let $\widetilde{r} \subseteq \mathrm{reg}(r, r')$. Based on axiom (14) it follows that $\widetilde{r} \gg x := s \not\subseteq \mathrm{reg}(r, r')$ if and only if:
  (i) $[x, s] \notin \mathrm{reg}(r, r')$, or, according to dot-completeness,
  (ii.1) $[x.a, s.a] \notin \mathrm{reg}(r, r')$, where $a \in \mathit{dom}(x)$, or,
  (ii.2) $[s.a, x_2.y_2^*.z_2] \notin \mathrm{reg}(r, r')$, where $[x.a, x_2.y_2^*.z_2] \in \widetilde{r}$.

  Let $[t', s'] \notin \mathrm{reg}(r, r')$ be the pair introduced by $x := s$ when starting from $\widetilde{r}_n$. Assume, based on (i) above, that $[t', s'] = [x, s] \notin \mathrm{reg}(r, r')$. However, $[x, s]$ must have been added when handling $\widetilde{r}_{n-1} \gg x := s$ as well, without affecting the regularity of r_n. Hence, $[x, s]$ must have been removed from r_n via **forget** x or **create** x within p. Consequently, $[x, s]$ will not occur in $\widetilde{r}_n \gg C[x := s]$ either. Therefore, we conclude that all pairs $[x, s] \notin \mathrm{reg}(r, r')$ added to $\widetilde{r}_n$ via assignments $x := s$ within p will be eventually removed via **forget** x or **create** x within p. The reasoning is similar for $[t', s'] = [x.a, s.a]$ as in (ii.1) and $[t', s'] = [s.a, x_2.y_2^*.z_2]$ as in (ii.2) above.

  So, it follows that $r_{n+1} = r_n \gg p \subseteq \mathrm{reg}(r, r')$. This contradicts our assumption in (37). We conclude that $r \gg p^n \subseteq \mathrm{reg}(r, r')$. □

More generally: in a non-concurrent setting, the machinery orchestrating the $\mathbb{K}$-like rules introduced in this section implements an algorithm that can terminate for the case of control blocks that do not contain qualified recursive functions, and provides a sound over-approximation of "may aliasing".

Theorem 20. *Consider p a program built on top of the control structures in (8), that does not contain qualified recursive calls, and executes in a sequential setting. Then, the application of the corresponding $\mathbb{K}$-like rules when starting with p and an empty alias relation, can always determine a finite rewriting of shape*

$$\langle\ \emptyset\ \rangle_{\mathrm{al}} \langle\ p\ \rangle_{\mathrm{k}} \overset{(*)}{\Longrightarrow} \langle\ r\ \rangle_{\mathrm{al}} \langle\ .\ \rangle_{\mathrm{k}},$$

with r a sound over-approximation of the aliasing information corresponding to the execution of p, and $\overset{(*)}{\Longrightarrow}$ the rewrite relation defined by the rules (19)–(33).

Proof sketch. First, note that the rewrite system in (19)–(33) is confluent, as there is only one way of performing the rewritings starting with program p, by matching at top (and at the top of the cells $\langle - \rangle_{[\mathrm{name}]}$). Then, observe that due to the execution of loops, expressions can infinitely grow in a *regular* fashion. Hence, an over-approximating alias relation closed under the execution of p can always be reached, according to Lemma 15 (and as illustrated in Lemma 19, in the context of "lassos"). Consequently, the control structure generating the infinite behavior is removed from the k-cell, according to the associated $\mathbb{K}$-like specification for loops in (28). This guarantees termination. □

$$\langle\, r \,\rangle_{\text{al}} \quad \langle\, \textbf{loop}\ x := x.a,\ x := x.b\ \textbf{end} \,\rangle_{\text{k}}\ \langle\, . \,\rangle_{\text{bkt-l}}$$

$$\Downarrow (25)$$

$$\langle\, r \,\rangle_{\text{al}} \quad \langle\, x := x.a; x := x.b\ \boxed{1}\ \textbf{loop}\ x := x.a,\ x := x.b\ \textbf{end} \,\rangle_{\text{k}}\ \langle\, \emptyset \,\rangle_{\text{bkt-l}}$$

$$\Downarrow (*)(21)$$

$$\langle\, r_1 \,\rangle_{\text{al}} \quad \langle\ \boxed{1}\ \textbf{loop}\ x := x.a,\ x := x.b\ \textbf{end} \,\rangle_{\text{k}}\ \langle\, \emptyset \,\rangle_{\text{bkt-l}}$$
$$\text{where } r_1 = \{[x, x.a.b],\ [x.a, x.a.b.a],\ [x.b, x.a.b.b]\}$$

$$\Downarrow (26)$$

$$\langle\, r_1 \,\rangle_{\text{al}} \quad \langle\, x := x.a; x := x.b\ \boxed{1}\ \textbf{loop}\ x := x.a,\ x := x.b\ \textbf{end} \,\rangle_{\text{k}}\ \langle\, r_1 \,\rangle_{\text{bkt-l}}$$

$$\Downarrow (*)(21)$$

$$\langle\, r_2 \,\rangle_{\text{al}} \quad \langle\ \boxed{1}\ \textbf{loop}\ x := x.a,\ x := x.b\ \textbf{end} \,\rangle_{\text{k}}\ \langle\, r_1 \,\rangle_{\text{bkt-l}}$$
$$\text{where } r_2 = \ \{[x, x.a.b.a.b],\ [x.a, x.a.b.a.b.a],\ [x.b, x.a.b.a.b.b]\}$$

$$\Downarrow (27)$$

$$\langle\, r_2 \,\rangle_{\text{al}} \quad \langle\, x := x.a; x := x.b\ \boxed{1}\ \textbf{loop}\ x := x.a,\ x := x.b\ \textbf{end} \,\rangle_{\text{k}}\ \langle\, r_2^* \,\rangle_{\text{bkt-l}}$$
$$\text{where } r_2^* = \{[x, x.(a.b)^*],\ [x.a, x.(a.b)^*.a],\ [x.b, x.(a.b)^*.b]\}$$

$$\Downarrow (*)(21)$$

$$\langle\, r_3 \,\rangle_{\text{al}} \quad \langle\ \boxed{1}\ \textbf{loop}\ x := x.a,\ x := x.b\ \textbf{end} \,\rangle_{\text{k}}\ \langle\, r_2^* \,\rangle_{\text{bkt-l}}$$
$$\text{where } r_3 = \ \{[x, x.a.b.a.b.a.b],\ [x.a, x.a.b.a.b.a.b.a],\ [x.b, x.a.b.a.b.a.b.b]\}$$

$$\Downarrow (28)$$

$$\langle\, r_2^* \,\rangle_{\text{al}} \quad \langle\, . \,\rangle_{\text{k}}\ \langle\, . \,\rangle_{\text{bkt-l}}$$

Fig. 1. Over-approximating aliasing.

4.2. The $\mathbb{K}$-machinery by example

For an example, in this section we show how the $\mathbb{K}$-machinery developed in Section 4 can be used in order to extract the alias information for the case of a **loop** $-$ **end** structure as follows:

loop $x{:} = x.a$; $x{:} = x.b$; **end**

We assume that x is an object of a class with two fields a and b, respectively. We consider a sequential setting.

The execution of the loop above, when starting with an empty alias relation r, produces the alias expressions:

$$[x,\ x.(a.b)^*]\quad [x.a,\ x.(a.b)^*.a]\quad [x.b,\ x.(a.b)^*.b \tag{38}$$

The associated rewriting is depicted in Fig. 1. The whole procedure starts with an empty alias relation $r = \emptyset$, and **loop** $x{:} = x.a$; $x{:} = x.b$; **end** in the continuation stack. Then, the corresponding $\mathbb{K}$ rules (for handling assignments and loops) are applied in the expected way.

A regular structure of the alias relation becomes obvious after two unfoldings of the loop structure. This triggers the application of rule (28) enabling the "regular" aliasing in (38), in accordance with Lemma 15.

5. Aliasing in SCOOP

In this section we provide a brief overview on the integration and applicability of the alias calculus in SCOOP. First, recall from Section 2 that the Maude semantics of SCOOP in [1] is defined over tuples of shape

$$\langle p_1 :: St_1 \mid \ldots \mid p_n :: St_n, \sigma \rangle$$

where, p_i and St_i stand for processors and their call stacks, respectively. σ is the state of the system and it holds information about the heap and the store.

The assignment instruction, for instance, is formally specified as the transition rule:

$$\frac{a \text{ is fresh}}{\Gamma \vdash \langle p :: t := s; St,\ \sigma\rangle \to \langle p :: \text{eval}(a, s); \text{wait}(a); \text{write}(t, a.data); St,\ \sigma\rangle} \tag{39}$$

where, intuitively, “eval(a,s)” evaluates s and puts the result on channel a, “wait(a)” enables processor p to use the evaluation result, “write$(t, a.data)$” sets the value of t to $a.data$, St is a call stack, and Γ is a typing environment [39] containing the class hierarchy of a program and all the type definitions.

The $\mathbb{K}$-like rule for assignments

$$\langle \frac{r}{(r_1 - t)[t = (r_1/s - t)] - ot} \rangle_{\mathsf{al}} \; \langle \frac{t := s}{.} \ldots \rangle_{\mathsf{k}} \quad \text{with } r_1 = r[ot = t] \qquad (21)$$

can be straightforwardly integrated in (39) by enriching the SCOOP configuration with a new component *alias_* encapsulating the alias information, and considering instead the transition:

$$\begin{aligned} &\Gamma \vdash \langle p :: t := s; St,\ \sigma, alias_{old} \rangle \to \\ &\langle p :: \text{eval}(a,s); \text{wait}(a); \text{write}(t, a.data); St,\ \sigma, alias_{new} \rangle \end{aligned}$$

where

$$alias_{old} = r \qquad alias_{new} = (r_1 - t)[t = (r_1/s - t)] - ot$$

with r and r_1 as in (21). The integration of all the $\mathbb{K}$-rules of the alias calculus on top of the Maude formalization of SCOOP is achieved by following a similar approach.

Nevertheless, it is important to point out that in order of the sound over-approximating aliases to be constructed as in Section 4, Theorem 20, one needs to simulate a SCOOP *sequential* setting. This is possible in SCOOP by creating all the objects in the program to be analyzed as objects handled by the same processor. An example computing the aliasing information is provided at: `https://dl.dropboxusercontent.com/u/1356725/SCOOP-SCP.zip`. Simply run the command `> maude SCOOP.maude ..\examples\linked_list-test.maude` corresponding to the execution of the code in (6), Section 3

```
**create** $x_0$
**loop**
  $i := i + 1$
  **create** $x_i$
  $x_i.set\_next(x_{i-1})$
**end**
```

for five iterations of the loop. As can be observed based on the code in `linked_list-test.maude`, in order to implement our applications in Maude, we use intermediate (but still intuitive) representations. For instance, the class structure defining a node in a simple linked list, with filed *next* and setter *set_next* is declared as:

```
class 'NODE
    create {'make} (
    attribute { 'ANY } 'next : [?, . , 'NODE] ;
    procedure { 'ANY } 'set_next ( 'a_next : [?, ., 'NODE] ;) [...]
   )
end ;
```

where `'next : [?, . , 'NODE]` stands for an object of type `NODE`, that is handled by the current processor (`.`) and that can be Void (`?`), and `'make` plays the role of a constructor. The “entry point” of the program corresponds to the function `'make` in the (main) class `'LINKED_LIST_TEST` and is set via:

```
settings('LINKED_LIST_TEST, 'make, aliasing-on) .
```

Observe that the flag for performing the alias analysis is switched to “on”.

The relevant parts of the corresponding Maude output after executing the aforementioned command are as follows:

```
rewrite in SYSTEM :
[...] settings('LINKED_LIST_TEST, 'make, aliasing-on))

|-
  {0}proc(0) :: nil | {0}proc(1) :: nil,
  {['x0 ; 'x0]} U {['x0 ; 'x1.'next]} U
  {['x0 ; 'x2.'next.'next]} U {['x0 ; 'x3.'next.'next.'next]} U
  {['x0 ; 'x4.'next.'next.'next.'next]} U
  [...]
  {['x3 ; 'x3]} U {['x3 ; 'x4.'next]}
```

```
state
  heap [...]
  store [...]
end
```

In short, one can see that two processors that were created finished executing the instructions on their corresponding call stacks: `{0}proc(0) :: nil` and `{0}proc(1) :: nil`. The aliased expressions include, as expected based on (7), pairs of shape $[x_i\,;x_{i+k}.next^k]$. Moreover, the output displays the contents of the current system state, by providing information on the *heap* and *store*, as formalized in [1].

6. Deadlocks in SCOOP

In what follows we provide the details behind the formalization and the implementation of a deadlock detection mechanism for SCOOP. We discuss how Maude can be exploited in order to test and, respectively, model-check deadlocks in the context of SCOOP programs, we analyze the associated challenges and propose a series of corresponding solutions.

6.1. Formalizing deadlocks in SCOOP

Recall that the key idea of SCOOP is to associate to each object declared as *separate* a processor, or handler in charge of executing the routines of that object. Assume a processor p that performs a call $o.f(a_1, a_2, \ldots)$ on a separate object o, with separate arguments a_i $(i \geq 1)$. As previously stated in Section 2, according to the SCOOP semantics, the application of the call $f(\ldots)$ will *wait* until it has been able to *lock* all the separate objects associated to $a_1, a_2, \ldots$. This reservation mechanism enables deadlocks to occur whenever a set of processors reserve each other circularly.

Definition 21 *(Deadlock).* For a processor p, let $W(p)$ be the set of handlers p *waits* for its asynchronous execution, and $H(p)$ represent the set of resources p already acquired. A *deadlock* exists if for some set D of processors the following holds:

$$(\forall p \in D).(\exists p' \in D).(p \neq p') \wedge (W(p) \cap H(p') \neq \emptyset). \tag{40}$$

A deadlock can happen, for instance, in a Dining Philosophers scenario, where both philosophers and forks are objects residing on their own processors. Consider, for an example, two separate objects p_1 and p_2 denoting two philosophers, and two separate objects f_1 and f_2 denoting two forks. Assume that p_1 picks f_1, whereas p_2 picks f_2. Then, the following hold whenever p_1 and p_2 want to acquire both forks: $H(p_1) = \{f_1\} = W(p_2)$ and $H(p_2) = \{f_2\} = W(p_1)$. Hence, Definition 21 is satisfied for $D = \{p_1, p_2\}$.

The integration of a deadlock detection mechanism based on Definition 21 on top of the SCOOP formalization in [1] is immediate. As already presented in Section 2, the operational semantics of SCOOP is defined over tuples of shape:

$$\langle p_1 :: St_1 \mid \ldots \mid p_n :: St_n, \sigma \rangle$$

where, p_i and St_i stand for processors and their call stacks, respectively, and σ is the state of the system. Given a processor p' as in (40), the set $H(p')$ corresponds, based on [1], to $\sigma.rq\_locks(p')$. Whenever the top of the instruction stack of a processor p is of shape $lock(\{q_i, \ldots, q_n\})$, we say that the wait set $W(p)$ is the set of processors $\{q_1, \ldots, q_n\}$. Hence, assuming a predefined system configuration $\langle deadlock \rangle$, the SCOOP transition rule corresponding to (40) can be written as:

$$\frac{\begin{array}{c}(\exists D \subseteq \sigma.procs).(\forall p \in D).(\exists p' \in D).(p \neq p') \wedge \\ (aqs := \ldots \mid p :: lock(\{q_i, \ldots\}); St \mid \ldots) \;\wedge\; (\sigma.rq\_locks(p').has(q_i))\end{array}}{\langle aqs, \sigma \rangle \to \langle deadlock \rangle} \tag{41}$$

Note that $\sigma.procs$ in (41) returns the set of processors in the system, whereas aqs stands for the list of these processors and their associated instruction stacks (separated by the associative and commutative operator "|"). We use "…" to represent an arbitrary sequence of processors and processor stacks.

6.2. Testing deadlocks

We implemented (41) and tested the deadlock detection mechanism on top of the formalization in [1] for the Dining Philosophers problem. A case study considering two philosophers sharing two forks can be run by downloading the SCOOP formalization at:

```
https://dl.dropboxusercontent.com/u/1356725/SCOOP-SCP-deadlock.zip
```

and executing the command

```
> maude SCOOP.maude ..\examples\dining-philosophers-rewrite.maude
```

The class implementing the *philosopher* concept is briefly given below:

```
class 'PHILOSOPHER
    create { 'make } (
        attribute {'ANY} 'left : [!,T,'FORK] ;
        attribute {'ANY} 'right : [!,T,'FORK] ;

        procedure { 'ANY } 'make ( 'fl : [!,T,'FORK] ;
                                   'fr : [!,T,'FORK] ; )
            do  ( assign ('left, 'fl) ; assign ('right, 'fr) ; )
            [...]
        end ;

        procedure { 'ANY } 'eat_wrong (nil)
            do ( command ('Current . 'pick_in_turn('left ;)) ; )
            [...]
        end ;

        procedure { 'ANY } 'pick_in_turn ('f : [!,T,'FORK] ; )
            do ( command ('Current . 'pick_two('f ; 'right ;)) ; )
            [...]
        end ;

        procedure { 'ANY } 'pick_two ('fa : [!,T,'FORK] ;
                                      'fb : [!,T,'FORK] ; )
            do ( command ('fa . 'use(nil)) ;  command ('fb . 'use(nil)) ; )
            [...]
        end ;
[...] end
```

It declares two forks – `'left` and `'right` of type `'FORK`, that can be handled by any processor (`T`) and that cannot be `Void` (`!`). Assume two philosophers `p1` and `p2` (of *separate* type `PHILOSOPHER`) and two forks `f1` and `f2` (of *separate* type `FORK`). Moreover, assume that `'left` and `'right` for `p1` correspond to `'f1` and `'f2`. For the case of `p2` they correspond to `'f2` and `'f1`, respectively. Asynchronously, `p1` and `p2` can execute `eat_wrong`, which calls `pick_in_turn(left)`. In the context of `p1`, the actual value of `left` is `f1`, whereas for `p2` is `f2`. Consequently, both resources `f1` and `f2`, respectively, may be locked "at the same time" by `p1` and `p2`, respectively. Note that `pick_in_turn` subsequently calls `pick_two` that, intuitively, should enable the philosophers to use both forks. Thus, if `f1` and `f2`, respectively, are locked by `p1` and `p2`, respectively, the calls `pick_two(f2, f1)` and `pick_two(f1, f2)` corresponding to `p1` and `p2` will (circularly) wait for each other to finish. According to the SCOOP semantics, `pick_two(f1, f2)` is waiting for `p2` to release `f2`, whereas `pick_two(f2, f1)` is waiting for `p1` to release `f1`, as the forks are passed to `pick_two(...)` as *separate* types. In the context of SCOOP, this corresponds to a Coffman deadlock [40].

The entry point of the program implementing the Dining Philosophers example is the function `'make` in the class `APPLICATION`, which asks the two philosophers `p1` and `p2` to adopt a wrong eating strategy as above, possibly leading to a deadlock situation. The flag enabling the deadlock analysis as in (41) is set to "on". This information is specified using a Dining Philosophers configuration (DP-config) specified by `settings('APPLICATION, 'make, deadlock-on)`.

Unfortunately, none of the executions of the Dining Philosophers scenario by simply invoking the Maude `rewrite` command lead to a deadlock situation. Each of our tests displayed the output:

```
rewrite in SYSTEM :
[...]  settings('APPLICATION, 'make, deadlock-on)

|- {0}proc(0) :: nil | {0}proc(1) :: nil | {0}proc(3) :: nil
   {0}proc(5) :: nil | {0}proc(7) :: nil
   {0}proc(9) :: nil | {0}proc(11) :: nil
```

consisting of a list of processors (including the handlers of both the philosophers and the forks) with empty call stacks (`:: nil`). This indicates that every time, the two philosophers proceeded by lifting their forks simultaneously, hence no deadlock was possible.

One possible workaround is to use predefined strategies [41] in order to guide the rewriting of the Maude rules formalizing SCOOP towards a ⟨*deadlock*⟩ system configuration. An example of applying such a strategy for the Dining Philosophers case can be tested by running the command:

```
> maude SCOOP.maude ..\examples\dining-philosophers-strategy.maude
```

The command `srew [...] using init ; parallelismlock ; [...] ; deadlock-on` forces the execution of a `pick_in_turn` approach as above. This determines Maude to first trigger the rule `[init]` in the SCOOP formalization in [1]. This makes all the required initializations of the *bootstrap* processor. Then, one of the processors that managed to *lock* the necessary resources is ("randomly") enabled to proceed to the asynchronous execution of its instruction stack, according to the strategy `parallelism{lock}`. The last step of the strategy calls the rule `[deadlock-on]` implementing the Coffman deadlock detection as in (41).

This time the guided rewriting leads, indeed, to one solution identifying a deadlock. The relevant parts of the corresponding Maude output are as follows:

```
srewrite in SYSTEM :
[...] settings('APPLICATION, 'make, deadlock-on)
using init ; parallelism{lock} ; [...] ; deadlock-on .

Solution 1
result Configuration: deadlock
```

Nevertheless, such an approach requires lots of ingenuity (our strategy has more than 300 rules!) and, moreover, is not automated.

6.3. *Model-checking deadlocks*

In this section we provide an overview on our approach to model-checking deadlocks for SCOOP, using the LTL Maude model-checker [42]. As mentioned in the beginning of the current paper, the idea behind our work is to exploit the unifying flavor of the Maude executable semantics of SCOOP [1]. The latter integrates both the formalization of the language and its concurrency mechanisms, thus enabling using the semantic framework for program analysis purposes, "for free".

One possible way to proceed is by simply running the Maude LTL model-checker on a Dining Philosophers configuration (DP-config) as thoroughly discussed in Section 6.2:

```
red modelCheck(DP-config, [ ] no-deadlock-mck) .
```

Intuitively, `[ ] no-deadlock-mck` is a safety property stating the freedom from deadlocks. The predicate witnessing the absence of deadlocks is defined as:

```
eq < deadlock > |= no-deadlock-mck = false .
eq sys::Configuration |= no-deadlock-mck = true [owise] .
```

Above, `< deadlock >` is the predefined system configuration "signalling" deadlocks in the SCOOP rule (41), whereas *sys::Configuration* is an arbitrary SCOOP system configuration.

Unfortunately, running the LTL model checker led to the state space explosion problem. At a first look, the issue was caused by the size of the SCOOP formalization in [1] which includes all the aspects of a real concurrency model.

As a first step, we reduced this formalization by eliminating the parts that are not relevant in the context of deadlocking; examples include the garbage collection and the exception handling mechanisms.

In addition, we provided a simplified, abstract semantics of SCOOP based on aliasing. This idea stems from the fact that SCOOP processors are known from object references, that may be aliased. Therefore, the SCOOP semantics can be simplified by retaining within the corresponding transition rules only the information important for aliasing. Consider, for instance, the rules (2)–(4) specifying "**if**" instructions in Section 2. The abstract transition rule omits the evaluation of the conditional and computes the aliasing information similarly to the semantics of **then ... else ... end** in (10), in Section 3. The abstraction collects the aliases resulted after the execution of both "**if**" and "**else**" branches:

$$\frac{\cdot}{\begin{array}{l}\langle p :: \textbf{if}\ e\ \textbf{then}\ St_1\ \textbf{else}\ St_2\ \textbf{end}\,;St,\ \sigma\,, alias_{old}\rangle \rightarrow \\ \qquad\qquad \langle p :: St,\ \sigma\,, alias_{new}\rangle\end{array}} \tag{42}$$

Observe that the SCOOP system configurations in (1) are enriched with a new component *alias_* consisting of a set of alias expressions. Above, $alias_{old}$ is the aliasing before the execution of the "**if**" instruction, whereas, intuitively, $alias_{new}$ stands for $alias_{old} \gg St_1 \cup alias_{old} \gg St_2$.

As a second step, we analyzed the implementation in [1] from a more engineering perspective, and identified a series of design decisions that either slowed down considerably the rewriting or made the search space grow unnecessarily large.

After running some experiments, we understood that the parallelism rule

$$\frac{\langle p_1 :: St_1,\, \sigma\rangle \to \langle p_1' :: St_1',\, \sigma'\rangle}{\langle p_1 :: St_1 \mid p_2 :: St_2,\, \sigma\rangle \to \langle p_1' :: St_1' \mid p_2 :: St_2,\, \sigma'\rangle} \tag{43}$$

in [1] was increasing the rewriting time. Though elegant from the formalization perspective, the usage of this rule was not efficient. Therefore, we eliminated it from the SCOOP semantics and made the remaining rules apply directly, by matching at top. For instance, the abstract rule (42) formalizing "**if**" instructions in the context of one processor p becomes:

$$\frac{\cdot}{\begin{array}{c}\langle p :: \mathbf{if}\ e\ \mathbf{then}\ St_1\ \mathbf{else}\ St_2\ \mathbf{end}\,;St \mid aqs,\, \sigma\,, alias_{old}\rangle \to \\ \langle p :: St \mid aqs,\, \sigma\,, alias_{new}\rangle\end{array}} \tag{44}$$

for an arbitrary list *aqs* of processors and their instruction stacks. For Dining Philosophers, for example, this modification reduced the rewriting time from around 10 s to less than 1 s.

Recall that SCOOP processors communicate via channels. The implementation in [1] creates *fresh* channels (as in (2), for instance) parameterized by natural indexes. This was inefficient for model-checking purposes, as the state space contained many identical states up-to channel naming.

The implementation of the above observations enabled us to successfully identify a deadlock situation in a Dining Philosophers scenario, by using the Maude LTL model-checker. The new (reduced) formalization of SCOOP can be found at:

`https://dl.dropboxusercontent.com/u/1356725/SCOOP-SCP.zip.`

This approach successfully identifies a SCOOP deadlock configuration:

```
  [...]
  proc(7) ::
  lock(({proc(3)} U {proc(5)})) ;
  command('fa . 'use(nil)) ; command('fb . 'use(nil)) ;
  [...]
  proc(9) ::
  lock(({proc(3)} U {proc(5)})) ;
  command('fa . 'use(nil)) ; command('fb . 'use(nil)) ;
  [...]

state
  regions(
      proc(3) --> {ref(4)} #  proc(5) --> {ref(6)} #
          proc(7) --> {ref(8)} # proc(9) --> {ref(10)} #
  [...]
  )
  heap
    objects
        'f1 --> ref(4) # 'f2 --> ref(6) #
               'p1 --> ref(8) # 'p2 --> ref(10) #
    [...]
  end
```

At a closer look, the information within the SCOOP deadlock state above identifies two processors `proc(7)` and `proc(9)` that cannot process the `lock(({proc(3)} U {proc(5)}))` statement at the top of their corresponding instruction stacks. Observe, based on the content of `regions` above, that `proc(7)` is the processor handling the reference `ref(8)` which corresponds to the philosopher object `'p1`. Similarly, `proc(9)` corresponds to the philosopher `'p2`, whereas `proc(3)` and `proc(5)` are associated to the fork objects `'f1` and `'f2`, respectively. Hence, it must be the case that each of the philosophers asynchronously picked (*i.e.*, locked) one of the two forks by executing `pick_in_turn(...)`. This way, locking both fork resources (`lock(({proc(3)} U {proc(5)}))`) by any of the philosophers, in order to execute `pick_two(...)`, is impossible according to the SCOOP semantics of locks.

It is worth mentioning that our approach can introduce spurious results, due to the over-approximating nature of the alias calculus. Nevertheless, since the over-approximations are sound, it is guaranteed that no false-negatives are identified.

Unfortunately, model-checking deadlocks in a scenario with five philosophers, for instance, takes unacceptably long time. Further improvements may be obtained by following the same recipe of collapsing semantically equivalent states, from the deadlocking perspective. A major source of redundancy is represented by the so-called *regions* in [1] that, intuitively, manage all the objects handled by the same processor. Their elimination from the SCOOP abstract state could improve the overall time performance by enabling the model-checker to make less identifications.

An additional observation is that most of the rewrite rules formalizing SCOOP are conditional and have rewrites in their conditions. This makes their execution expensive and inefficient for model-checking. A reduction semantics of SCOOP [33,43] is therefore worth investigating as future work.

7. Conclusions and pointers to future work

The focus of this paper is on building a toolbox for the analysis of SCOOP programs, with emphasis on an alias analyzer and a deadlock detector. The naturalness of our approach consists of exploiting the recent formalization of SCOOP in [1], that is executable and implemented in Maude [6]. This provides a unifying framework that can be used not only to reason about the SCOOP model and its design as in [1], but also to analyze SCOOP programs via Maude's analysis tools.

Of particular interest for the aliasing tool is the calculus introduced in [19], which abstracts the aliasing information in terms of explicit access paths referred to as "alias expressions". We provide an extension of this calculus from finite alias relations to infinite ones corresponding to loops and recursive calls. Moreover, we devise an associated RL-based executable specification using $\mathbb{K}$ notation [36]. In Theorem 20 we show that the RL-based machinery implements an algorithm that can terminate with a sound over-approximation of "may aliasing", in non-concurrent settings. This is achieved based on the sound (finitely representable) over-approximation of alias expressions in terms of regular expressions, as in Lemma 15. The integration of the alias calculus on top of the Maude formalization of SCOOP [1] is straightforward, based on the aforementioned executable specification of the calculus. Executions of SCOOP programs can be simulated by simply exploiting the Maude rewriting capabilities, hence the computation of the corresponding aliasing information is immediate.

One limitation of our approach is that the termination of the aforementioned RL-machinery holds up to non-qualified (mutually) recursive calls. Handling qualified (mutually) recursive calls is left to further investigation.

We agree that it could be worth presenting our analysis as an abstract interpretation (AI) [44]. A modeling exploiting the machinery of AI (based on abstract domains, abstraction and concretization functions, Galois connections, fixed-points, *etc.*) is an interesting topic left for future investigation.

An immediate direction for future work is to identify interesting (industrial) case studies to be analyzed using the framework developed in this paper. We are also interested in devising heuristics comparing the efficiency and the precision (*e.g.*, the number of false positives introduced by the alias approximations) between our approach and other aliasing techniques.

Another research direction is to derive alias-based abstractions for analyzing concurrent programs. We foresee possible connections with the work in [45] on concurrent Kleene algebra formalizing choice, iteration, sequential and concurrent composition of programs. The corresponding definitions exploit abstractions of programs in terms of traces of events that can depend on each other. Thus, obvious challenges in this respect include: (i) defining notions of dependence for all the program constructs in this paper, (ii) relating the concurrent Kleene operators to the semantics of the SCOOP concurrency model and (iii) checking whether fixed-points approximating the aliasing information can be identified via fixed-point theorems.

Furthermore, it would be worth investigating whether the graph-based model of alias relations introduced in [19] can be exploited in order to derive finite $\mathbb{K}$ specifications of the extended alias calculus. In case of a positive answer, the general aim is to study whether this type of representation increases the speed of the reasoning mechanism, and why not – its accuracy. With the same purpose, we refer to a possible integration with the technique in [46] that handles point-to graphs via a stack-based algorithm for fixed-point computations.

Related to deadlock detection, the second contribution of this paper, we provided a formalization based on sets of acquired resources and sets of handlers processors wait to lock in their attempt to execute asynchronously. This definition corresponds to Coffman deadlocks [40] in the context of SCOOP, occurring whenever there is a set of processors reserving each other circularly. We introduced the equivalent SCOOP semantic rule and discussed the results of using Maude in order to analyze deadlocks in the context of a Dining Philosophers scenario. On the one hand, the SCOOP semantics in [1] is very large, as it incorporates all the aspects of a real concurrency model. On the other hand, the formalization in [1] was tailored for reasoning about the SCOOP model, and not necessarily about SCOOP applications, hence it includes design decisions (such as index-based naming of communication channels) that make the state space grow unacceptably large for model-checking purposes. As a workaround for the model-checking issue, we presented the idea behind building an abstract semantics of SCOOP based on aliases, together with a series of implementation improvements that eventually enabled the Maude LTL model-checker to correctly identify deadlocks. A survey on abstracting techniques on top of Maude executable semantics is provided in [5].

We leave investigating the relationship between the original SCOOP semantics and the aliasing-based one as future work. At this point we observe that the abstract alias semantics introduces false positives (w.r.t. deadlocking as well) due to the over-approximating nature of the alias calculus.

As a clear direction for future work we consider designing and analyzing deadlock situations for more SCOOP applications. Based on the experience so far, this would help better understand and observe the SCOOP state space, thus providing hints for further improvements in the context of model-checking, for instance. As we could already see, major advances in this regard are obtained when semantically equivalent states are identified and collapsed within the same equivalence classes. This was the case of the indexed-based communication channels in Section 6.3. Similar redundant states are introduced by the so-called "regions" in SCOOP "administrating" objects handled by the same processor. Nevertheless, as the rewrite rules defining the semantics of SCOOP are conditional and have rewrites in their conditions, model-checking can still be expensive. A reduction semantics [33,43] of SCOOP is therefore worth investigating.

Acknowledgements

We are grateful for valuable comments to the anonymous reviewers of SCP, FTSCS'14, Măriuca Asăvoae, Alexander Kogtenkov, José Meseguer, Benjamin Morandi and Sergey Velder.

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389.

References

[1] B. Morandi, M. Schill, S. Nanz, B. Meyer, Prototyping a concurrency model, in: J. Carmona, M.T. Lazarescu, M. Pietkiewicz-Koutny (Eds.), 13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8–10 July, 2013, IEEE Computer Society, 2013, pp. 170–179.

[2] B. Meyer, Eiffel: The Language, Prentice-Hall, 1991.

[3] F.A. Torshizi, J.S. Ostroff, R.F. Paige, M. Chechik, The SCOOP concurrency model in Java-like languages, in: P.H. Welch, H.W. Roebbers, J.F. Broenink, F.R.M. Barnes, C.G. Ritson, A.T. Sampson, G.S. Stiles, B. Vinter (Eds.), The Thirty-Second Communicating Process Architectures Conference, CPA 2009, organised under the auspices of WoTUG, Eindhoven, The Netherlands, 1–6 November 2009, in: Concurr. Syst. Eng. Ser., vol. 67, IOS Press, 2009, pp. 1–6.

[4] A. Rusakov, J. Shin, B. Meyer, Simple concurrency for robotics with the Roboscoop framework, in: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14–18, 2014, 2014, pp. 1563–1569.

[5] J. Meseguer, G. Rosu, The rewriting logic semantics project: a progress report, Inf. Comput. 231 (2013) 38–69, http://dx.doi.org/10.1016/j.ic.2013.08.004.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude – a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic, Springer-Verlag, Berlin, Heidelberg, 2007.

[7] W. Landi, B.G. Ryder, Pointer-induced aliasing: a problem classification, in: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '91, ACM, New York, NY, USA, 1991, pp. 93–103, http://doi.acm.org/10.1145/99583.99599.

[8] W. Landi, Undecidability of static analysis, ACM Lett. Program. Lang. Syst. 1 (4) (1992) 323–337, http://doi.acm.org/10.1145/161494.161501.

[9] E.M. Myers, A precise inter-procedural data flow algorithm, in: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '81, ACM, New York, NY, USA, 1981, pp. 219–230, http://doi.acm.org/10.1145/567532.567556.

[10] M. Hind, M.G. Burke, P.R. Carini, J. Choi, Interprocedural pointer alias analysis, ACM Trans. Program. Lang. Syst. 21 (4) (1999) 848–894, http://doi.acm.org/10.1145/325478.325519.

[11] A. Diwan, K.S. McKinley, J.E.B. Moss, Type-based alias analysis, in: J.W. Davidson, K.D. Cooper, A.M. Berman (Eds.), Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, PLDI, Montreal, Canada, June 17–19, 1998, ACM, 1998, pp. 106–117, http://doi.acm.org/10.1145/277650.277670.

[12] M. Burke, P. Carini, J.-D. Choi, M. Hind, Flow-insensitive interprocedural alias analysis in the presence of pointers, in: K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing, in: Lect. Notes Comput. Sci., vol. 892, Springer, Berlin, Heidelberg, 1995, pp. 234–250.

[13] J.-D. Choi, M. Burke, P. Carini, Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects, in: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '93, ACM, New York, NY, USA, 1993, pp. 232–245, http://doi.acm.org/10.1145/158511.158639.

[14] M. Emami, R. Ghiya, L.J. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, ACM, New York, NY, USA, 1994, pp. 242–256, http://doi.acm.org/10.1145/178243.178264.

[15] R.P. Wilson, M.S. Lam, Efficient context-sensitive pointer analysis for C programs, in: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI '95, ACM, New York, NY, USA, 1995, pp. 1–12, http://doi.acm.org/10.1145/207110.207111.

[16] A. Miné, Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics, in: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems, LCTES '06, ACM, New York, NY, USA, 2006, pp. 54–63, http://doi.acm.org/10.1145/1134650.1134659.

[17] E. Albert, P. Arenas, S. Genaim, G. Puebla, Field-sensitive value analysis by field-insensitive analysis, in: A. Cavalcanti, D. Dams (Eds.), Proceedings of the FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2–6, 2009, in: Lect. Notes Comput. Sci., vol. 5850, Springer, 2009, pp. 370–386.

[18] M. Hind, Pointer analysis: haven't we solved this problem yet?, in: J. Field, G. Snelting (Eds.), Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18–19, 2001, ACM, 2001, pp. 54–61, http://doi.acm.org/10.1145/379605.379665.

[19] A. Kogtenkov, B. Meyer, S. Velder, Alias calculus, change calculus and frame inference, Sci. Comput. Program. 97 (2015) 163–172, http://dx.doi.org/10.1016/j.scico.2013.11.006.

[20] J.R. Larus, P.N. Hilfinger, Detecting conflicts between structure accesses, in: R.L. Wexelblat (Ed.), Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, PLDI, Atlanta, Georgia, USA, June 22–24, 1988, ACM, 1988, pp. 21–34, http://doi.acm.org/10.1145/53990.53993.

[21] I.M. Asavoae, Abstract semantics for alias analysis in K, Electron. Notes Theor. Comput. Sci. 304 (2014) 97–110, http://dx.doi.org/10.1016/j.entcs.2014.05.005.

[22] M. Hills, P. Klint, J.J. Vinju, Program analysis scenarios in Rascal, in: F. Durán (Ed.), Rewriting Logic and Its Applications – 9th International Workshop, WRLA 2012, Held as a Satellite Event of ETAPS, Tallinn, Estonia, March 24–25, 2012, in: Lect. Notes Comput. Sci., vol. 7571, Springer, 2012, pp. 10–30, Revised Selected Papers.

[23] C. Shih, J.A. Stankovic, Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems, Tech. rep., Amherst, MA, USA, 1990.

[24] G.R. Andrews, G.M. Levin, On-the-fly deadlock prevention, in: Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC '82, ACM, New York, NY, USA, 1982, pp. 165–172.

[25] T. Minoura, Deadlock avoidance revisited, J. ACM 29 (4) (1982) 1023–1048.

[26] K.M. Chandy, J. Misra, L.M. Haas, Distributed deadlock detection, ACM Trans. Comput. Syst. 1 (2) (1983) 144–156, http://doi.acm.org/10.1145/357360.357365.

[27] D.Z. Badal, M.T. Gehl, On deadlock detection in distributed computing systems, in: Proceedings IEEE INFOCOM 83, San Diego, CA, USA, April 18–21, 1983, IEEE, 1983, pp. 36–45.

[28] E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, P.Y.H. Wong, Deadlock analysis of concurrent objects: theory and practice, in: Integrated Formal Methods, 10th International Conference. Proceedings, IFM 2013, Turku, Finland, June 10–14, 2013, 2013, pp. 394–411.

[29] A. Heußner, C.M. Poskitt, C. Corrodi, B. Morandi, Towards practical graph-based verification for an object-oriented concurrency model, in: Proc. Graphs as Models, GaM 2015, in: Discret. Math. Theor. Comput. Sci., vol. 181, 2015, pp. 32–47.

[30] A.H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, M. Zimakova, Modelling and analysis using GROOVE, Int. J. Softw. Tools Technol. Transf. 14 (1) (2012) 15–40.

[31] C. Ellison, G. Rosu, An executable formal semantics of C with applications, in: J. Field, M. Hicks (Eds.), Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22–28, 2012, ACM, 2012, pp. 533–544.

[32] C. Ellison, A formal semantics of C with applications, Ph.D. thesis, University of Illinois, July 2012.

[33] M. AlTurki, J. Meseguer, Reduction semantics and formal analysis of Orc programs, Electron. Notes Theor. Comput. Sci. 200 (3) (2008) 25–41, http://dx.doi.org/10.1016/j.entcs.2008.04.091.

[34] J. Misra, W.R. Cook, Computation orchestration, Softw. Syst. Model. 6 (1) (2007) 83–110.

[35] G. Caltais, Expression-based aliasing for OO-languages, in: C. Artho, P.C. Ölveczky (Eds.), Formal Techniques for Safety-Critical Systems – Third International Workshop, FTSCS 2014, Luxembourg, November 6–7, 2014, in: Commun. Comput. Inf. Sci., vol. 476, Springer, 2014, pp. 47–61, Revised Selected Papers.

[36] G. Rosu, T. Serbanuta, K overview and SIMPLE case study, Electron. Notes Theor. Comput. Sci. 304 (2014) 3–56, http://dx.doi.org/10.1016/j.entcs.2014.05.002.

[37] M.O. Rabin, D. Scott, Finite automata and their decision problems, IBM J. Res. Dev. 3 (2) (1959) 114–125, http://dx.doi.org/10.1147/rd.32.0114.

[38] V. Rusu, D. Lucanu, T. Serbanuta, A. Arusoaie, A. Stefanescu, G. Rosu, Language definitions as rewrite theories, J. Log. Algebraic Methods Program. 85 (1) (2016) 98–120, http://dx.doi.org/10.1016/j.jlamp.2015.09.001.

[39] P. Nienaltowski, Practical Framework for Contract-based Concurrent Object-oriented Programming, ETH, http://books.google.ch/books?id=ZDcEkgAACAAJ, 2007.

[40] E.G. Coffman, M. Elphick, A. Shoshani, System deadlocks, ACM Comput. Surv. 3 (2) (1971) 67–78.

[41] N. Martí-Oliet, J. Meseguer, A. Verdejo, Towards a strategy language for Maude, Electron. Notes Theor. Comput. Sci. 117 (2005) 417–441.

[42] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, Electron. Notes Theor. Comput. Sci. 71 (2002) 162–187, http://dx.doi.org/10.1016/S1571-0661(05)82534-4.

[43] M. Felleisen, D.P. Friedman, A reduction semantics for imperative higher-order languages, in: J.W. de Bakker, A.J. Nijman, P.C. Treleaven (Eds.), PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Proceedings, Eindhoven, The Netherlands, June 15–19, 1987, in: Lect. Notes Comput. Sci., vol. 259, Springer, 1987, pp. 206–223.

[44] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, J. Log. Program. 13 (2&3) (1992) 103–179, http://dx.doi.org/10.1016/0743-1066(92)90030-7.

[45] C.A.R. Hoare, B. Möller, G. Struth, I. Wehrman, Concurrent Kleene algebra, in: M. Bravetti, G. Zavattaro (Eds.), CONCUR 2009 – Concurrency Theory, 20th International Conference. Proceedings, CONCUR 2009, Bologna, Italy, September 1–4, 2009, in: Lect. Notes Comput. Sci., vol. 5710, Springer, 2009, pp. 399–414.

[46] D.R. Chase, M.N. Wegman, F.K. Zadeck, Analysis of pointers and structures, in: B.N. Fischer (Ed.), Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20–22, 1990, ACM, 1990, pp. 296–310, http://doi.acm.org/10.1145/93542.93585, 1990.