Towards a $\mathbb{K}$ool Future

Dorel Lucanu[1], Traian-Florin Şerbănuţă[2], and Grigore Roşu[3]

[1] Alexandru Ioan Cuza University, Iaşi, Romania
[2] University of Bucharest, Romania
[3] University of Illinois at Urbana-Champaign, USA

Abstract. The $\mathbb{K}$ framework was successfully used for defining formal semantics for several practical languages, e.g. C, Java, Java Script, but no language with distributed concurrent objects was defined in $\mathbb{K}$ up to now. In this paper we investigate how the model of asynchronous method calls, using the so-called *futures* for handling the return values, can be added to an existing $\mathbb{K}$ definition using the ideas from the Complete Guide to the Future paper. As the running example we use the $\mathbb{K}$ definition of KOOL, a pedagogical and research language that captures the essence of the object-oriented programming paradigm. This is a first step toward a generic methodology for modularly adding future-based mechanisms to allow asynchronous method calls.

1 Introduction

$\mathbb{K}$ (`www.kframework.org`) is a framework for formally defining the semantics of programming languages. The $\mathbb{K}$ definitions of the programming languages are executable, i.e. they can be used to execute programs written in the defined language, and can be used for program analysis and verification. The $\mathbb{K}$ Framework is scalable: several realistic languages, e.g. Java [3], C [10], Java Script [16], PHP [11], have already been defined in $\mathbb{K}$. The main ingredients of a $\mathbb{K}$ definition are configurations, computations and rules. *Configurations* organise the state in units called cells, which are labeled and can be nested. *Computations* are special nested list structures sequentialising computational tasks, such as fragments of program. $\mathbb{K}$ *(rewrite) rules* make it explicit which parts of the term they read-only, write-only, read-write, or do not care about. This makes $\mathbb{K}$ suitable for defining truly concurrent languages even in the presence of sharing. The only concurrency model described in some languages defined in $\mathbb{K}$ is that described by threads. No language including distributed concurrent objects is defined in $\mathbb{K}$ up to now.

Futures [15,25] are language constructs meant to represent awaited results for asynchronous calls. Roughly speaking, a future is a place holder for a result of an asynchronous concurrent computation. Once this computation is complete the computed result, called *future value*, fills the place holder. An access to an unresolved feature is a blocking operation.

The futures can be *transparent* or *explicit*. For the explicit case, the language includes specific constructs for creating futures and getting the results. On the

other hand the implicit futures are handled by underlying middleware and the syntax of the language remains unchanged. Some languages allow futures to be passed as parameters to other processes; these are called *first class futures*. First class futures are useful for both object-oriented and procedural paradigms since they improve the concurrency patterns and offer more flexibility in design. Futures can be defined directly inside of a language [4,14,8,1,9] or as middleware using a component-based model [23,5,13].

In the *Complete Guide to the Future* [8], Frank de Boer et. al provide the semantics for an object-oriented language including explicit first class futures, defined as an extension of the Creol language [14]. The main features defined there include active multi-threaded objects, asynchronous method calls, and futures. A proof system for proving properties specific to concurrency is provided.

Inspired by [8], in this paper we investigate how the semantics of first class futures can be added, in a generic way, to languages that already have a formal semantics. We consider an object oriented language formally defined in the $\mathbb{K}$ Framework, namely KOOL, and we identify how the configuration is changed, which semantic rules have to be modified, in order to implement implicit futures, and which rules should be added to implement explicit futures. We rely on the modularity of the $\mathbb{K}$ framework and we claim that the number of changed rules is minimal (no rule unrelated to the extension was modified). Moreover, $\mathbb{K}$ definitions being executable, using the $\mathbb{K}$ tool, allow users to effectively test whether the semantics has the desired properties.

The underlying logics for $\mathbb{K}$ definitions are matching logic [17] and reachability logic [20,6]. Using the encoding of Hoare triples into reachability logic [19], we automatically have a translation of the proof system defined in the Complete Guide to the Future into reachability logic. This makes possible the use of provers like that reported in [22] for checking concurrency specific properties.

The paper is structured as follows. Section 2 includes a brief introduction to $\mathbb{K}$. The main ingredients of a $\mathbb{K}$ definition are exemplified using the KOOL programming language, which is a part of the $\mathbb{K}$ tutorial (`http://www.kframework.org/index.php/K_Tutorial`). Section 3 presents KFUTURE, a version of KOOL with asynchronous methods calls modelled using futures. The main changes and new added constructs are briefly presented. Several experiments with the $\mathbb{K}$ tool are reported in Section 4. Finally, Section 5 concludes the paper and discusses future work opportunities.

Acknowledgements. This paper is written in honour of Frank de Boer on the occasion of his 60th birthday and celebrates his exceptional contribution to the object-oriented paradigm. The first author had the privilege to cooperate with Frank and he is deeply grateful to him for that fruitful experience.

The work presented here was partially supported by Romanian Contract 161/15.06.2010, SMIS-CSNR 602-12516 (DAK), which made possible the development of the main first versions of the $\mathbb{K}$ Framework.

2 A Kool Introduction to $\mathbb{K}$

In a nutshell, the $\mathbb{K}$ Framework [21] consists of computations, configurations, and rules. Computations are special sequences of tasks, where a task can be, e.g., a fragment of program that needs to be processed. Configurations are used to describe the program states and are organised as nested pools of cells holding syntactic and semantic information. $\mathbb{K}$ rules distinguish themselves by specifying only what is needed from a configuration, and by clearly identifying what changes, and thus, being more concise, more modular, and more concurrent than regular rewrite rules.

The running example is KOOL [18], a pedagogical and research language that captures the essence of the object-oriented programming paradigm. Among the object-oriented features included in KOOL we find the inheritance and the dynamic method dispatch mechanism. Moreover, KOOL is higher-order, allowing function abstractions to be treated like any other values in the language. The $\mathbb{K}$ definition of KOOL was the starting point for the $\mathbb{K}$ definition of Java [3].

Syntax and Computations. Computations extend syntax with an operation, "$\curvearrowright$", meaning to capture task sequentialization. The basic unit of computation is a *task*, which can either be a fragment of syntax, maybe with holes in it, or a semantic task, such as the recovery of an environment. The computation is abstracted away from the language designer via intuitive program languages syntax annotations like strictness constraints that specify the order of evaluation for its arguments. The decompositions of computations are similar to the use of stacks in abstract machines [12] and to the refocusing techniques for implementing reduction semantics with evaluation contexts [7].

```
Exp ::= Int | Id
      | Exp + Exp [strict]
      | Exp ( Exps ) [strict]
      ...
Decl ::= Type Exp
       | Type Id ( Exp ) Block
       | Id Block
       | Id extends Id Block
```

```
Stmt ::= Decl
       | Block
       | Exp ;
       | if ( Exp ) Block else Block [strict(1)]
       ...
Block ::= { }
        | { Stmts }
Exps ::= List(Exp,",")
Stmts ::= List(Stmt,"")
```

Fig. 1. A fragment of KOOL's syntax

Fig. 1 includes a fragment of the syntax for KOOL, described using BNF notation. The strictness annotations add semantic information to the syntax by specifying the order of evaluation of arguments for the corresponding construct. This is achieved by the means of the heating/cooling rules, which are automatically generated from strictness annotations. The order of evaluation can be left unspecified (if using the "strict" attribute), or specified to happen in a given order using the "seqstrict" attribute. For instance, the "strict" attribute for the

addition operator says that all arguments of addition are evaluated, but in an unspecified order, which is achieved by the following heating/cooling rules:

$$E_1 \ + \ E_2 \rightleftharpoons E_1 \curvearrowright \square \ + \ E_2 \qquad E_1 \ + \ E_2 \rightleftharpoons E_2 \curvearrowright E_1 \ + \ \square$$
$$I_1 \curvearrowright \square \ + \ E_2 \rightleftharpoons I_1 \ + \ E_2 \qquad I_2 \curvearrowright E_1 \ + \ \square \rightleftharpoons E_1 \ + \ I_2$$

When the strict attribute has parameters, only those parameters are evaluated. E.g., for the statement `if` only the first argument (the conditional expression) is evaluated. For seqstrict, no attributes specify the evaluation of all arguments in a left-to-right order, while a list of positions specifies that the given arguments are to be evaluated in the given order, allowing for different order of evaluation to be specified.

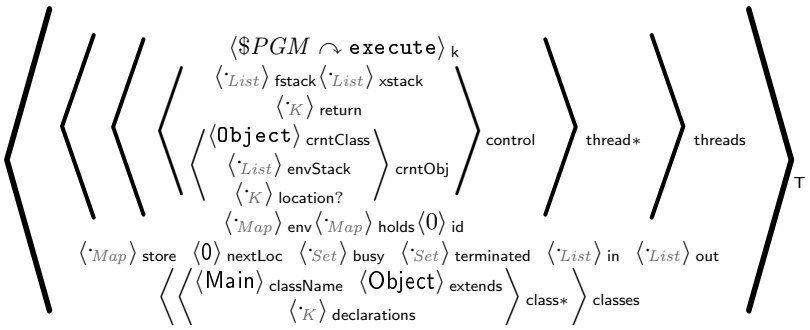


Fig. 2. KOOL Configuration

Configurations. A configuration is a nested multiset of labeled cells, in which each elementary cell can contain either a list, a set, a bag, a map, or a computation. Fig. 2 includes the configuration for the KOOL language. Here is a brief description of the cells (the tree-like structure of the list reflects the nesting structure of the cells):

T – top level cell;
  threads – holds a pool of thread cells;
    thread – holds the sub-configuration of a thread;
      k – holds the nested list of the computations for the thread;
      control – holds the local control state of the thread;
        fstack – holds the function stack;
        xstack – holds the stack of exceptions;
        return – holds the type of the value to be returned by the current method;
        crntObj – holds the description of the current object (this);
          crntClass – holds the name of the current class which the current object belongs to;

envStack – holds the state of the object as a stack of environments (for each ancestor class an environment binding the fields of the class to their current locations);
location – holds the location where the object is stored;
env – holds a map binding each name accessible by the thread to a store location;
holds – include the locks held by the thread;
id – holds a natural number that is the identity of the thread;
store – holds a global (shared by all threads) map binding the allocated locations to some values; the fact that the environment is local but the store is global allows for shared memory while preserving the visibility domain of variables;
nextLoc – holds a natural number indicating the next free location;
busy – holds the locks which have been acquired but not yet released by threads;
terminated – holds the unique identifiers of the threads which already terminated (needed for join);
in – holds the list of the input data (needed for reading statements);
out – holds the list of the output data (needed for writing statements);
classes – the pool of the classes of a KOOL program;
class – holds the description of a class;
classname – holds the name of the class;
extends – holds the name of the parent class;
declarations – holds the declarations for the class fields and methods.

The content specification for the elementary cells has a double role: 1) it specifies the content of that cell in the initial configuration, and 2) it specifies the type (sort) of the information stored in that cell. The dot notation is used for the empty data structures: e.g., $\cdot_{List}$ denotes the empty list, $\cdot_{Set}$ denotes the empty set, and so on. The special variable $\$PGM$ will be replaced in the initial configuration with the program to be executed. The internal command `execute` triggers the execution of the program after its preprocessing to fill the initial configuration.

The star character following the name of a cell specifies the multiplicity of that cell, i.e. a concrete configuration may include zero, one, or more cells of that kind. For instance, a concrete configuration may include several thread cells and/or several class cells.

$\mathbb{K}$ *rules.* The transition relation defining the operational semantics of a language is described by $\mathbb{K}$ rules. For instance, the rule giving the semantics for the addition operator is

$$\frac{I_1 \ + \ I_2}{I_1 +_{Int} I_2}$$

where above the horizontal line is the pattern used for matching and below the horizontal line is the pattern that defines the result term replacing the matched

term. This rule is applied only when the above pattern matches the top of the k cell (we shall see later why). Since the syntax of this operator was defined with the strict attribute, it follows that it is evaluated only after its arguments have been evaluated to the integers I_1 and I_2. The operation $+_{Int}$ effectively adds two integers. The semantics of the statement `if` is given by two rules:

$$\frac{\texttt{if}\ \mathit{true}\ B_1\ \texttt{else}\ \_}{B_1} \qquad \frac{\texttt{if}\ \mathit{false}\ \_\ \texttt{else}\ B_2}{B_2}$$

Recall that the syntax of `if` is strict only in the first argument, so only the condition expression is evaluated first to either *true* or *false*. The two rules corresponds to the two possible values returned by the evaluation of the condition expression.

Although $\mathbb{K}$ rules are in essence rewrite rules, there are several ways in which they differ from a regular rewrite rule. First, *in-place rewriting* (denoted by the horizontal bar) allows one to specify small changes into a bigger context, by underlining the part that needs to change and writing its replacement under the line, instead of repeating the context in both sides of a rewrite rule. For instance, the rule for addition will be applied only when the the pattern I_1 + I_2 matches the top of the computation cell. This enables another optimisation, namely the ability of using anonymous variables (_) for the unused variables in the context (see, e.g., the rule for `if`).

Furthermore, $\mathbb{K}$ allows the use of *cell comprehension* for focusing only on the parts of the cells which are relevant, as in the rule for variable lookup:

$$\left\langle \frac{X{:}Id\ \cdots}{V} \right\rangle_{\mathsf{k}}\ \langle \cdots\ X \mapsto L\ \cdots \rangle_{\mathsf{env}}\ \langle \cdots\ L \mapsto V{:}Val\ \cdots \rangle_{\mathsf{store}}$$

The lookup rule above rule specifies that when a variable X is the first computational task, and X is bound to some location L in the environment, and L is mapped to some value V in the store, then we rewrite X into V.

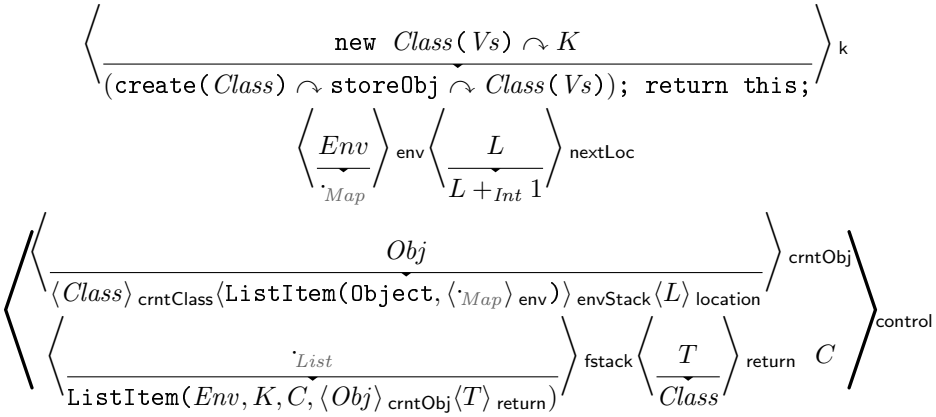
The ellipses at the left/right end of a cell are used to specify that there could be more items in that cell (in the corresponding side) in addition to what is explicitly specified. Note that the variable to be looked up is the first task in the k cell (the cell is closed to the left and open to the right), while the binding of X to L and the mapping of L to V can be anywhere in the env and store cells (these cells are open in both sides).

Finally, the process of *configuration abstraction* allows for mentioning only the relevant cells in a rule, by relying on the static structure of the declared configuration to infer the rest (*configuration concretization*). For instance, without $\mathbb{K}$'s configuration abstraction, the lookup rule above would have to also include the thread and threads cells. Configuration abstraction is crucial for modularity, because it gives the possibility to write definitions in a way that may not require to revisit existing rules when the configuration changes as new (orthogonal) language features are introduced.

Advanced features of the KOOL language semantics

We conclude our brief introduction to $\mathbb{K}$ by showing the semantic rules defining the behavior of some important features of KOOL which would be affected by the introduction of futures in the next section.

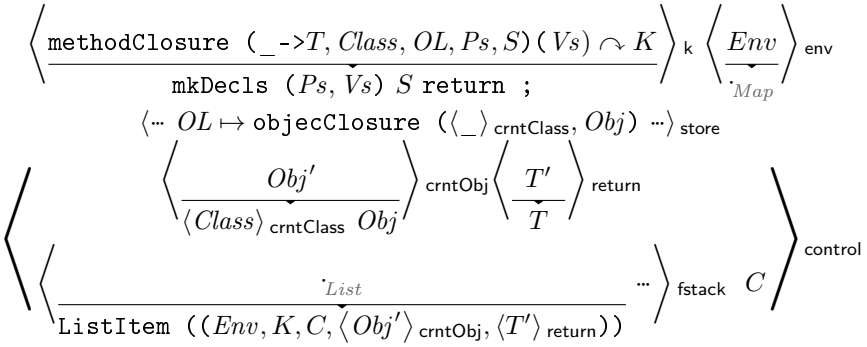
The `new` operator. The rule defining the operator `new` includes a more complex matching part and many local rewrites:



The semantics of `new` consists of two actions: memory allocation for the new object and execution of the corresponding constructor. Then the created object is returned as the result of the new operation. The rule matches a `new` expression on the top of the `k` cell, where the parameters *Vs* are already evaluated due to the strictness, and performs the following changes in the configuration:

- replaces the `new` expression with two actions, memory allocation for the new object (given by the auxiliary operations `create` and `storeObj`) and execution of the corresponding constructor, followed by the instruction returning the created object;
- stores the current environment, computation, control, object, and return type on the function stack;
- initializes the object creation process by emptying the local environment and the current object, and allocating a location in the store where the created object will be eventually stored;
- replaces the return type with the class of the newly created object.

Method Calls. The rule for method calls is somehow similar to that of the `new` operator:



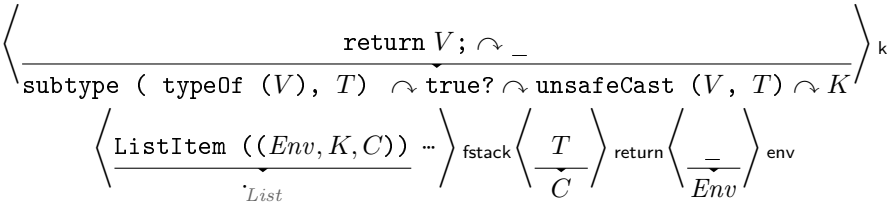
Since the syntax for method calls is strict, the expression describing the method name is evaluated to the corresponding function value. Recall that KOOL is a higher-order language that allows the function abstractions to be treated like any other values. A function value is a closure that includes the method parameters, the body of the method, and the object value. The type held by a method closure is the entire type of the method in order to dynamically upcast values when passed to contexts where values of superclass types are expected. An *object value* consists of an `objectClosure`-wrapped bag containing the current class of the object and the environment stack of the object. The current class of an object will always be one of the classes mapped to an environment in the environment stack of the object. The rule matches a method call on top of the computation cell and performs the following changes in the configuration:

- replaces the method call with the method body followed by a `return;`;
- pushes the current environment, control data, current object and the return type onto the function stack;
- binds the actual arguments to formal parameters using the auxiliary operation `mkDecls`;
- updates the current object and the return type of the current method.

The arguments of the call are evaluated to a list of values Vs due to the strict attribute. The variable K matches the rest of the computation.

The `return` *statement* performs the dual operations:

- pops the environment, control data, current object and the return type from the function stack and stores them into the corresponding cells;
- checks the type of the returned value and passes it to the popped computation; note that its type is cast to that stored in the return cell.



3 KOOL with Futures

This section presents a language design exercise: adding support for futures to an existing object oriented language *executable* definition.

We chose KOOL as the reference object oriented language definition, because it is a relatively small but not trivial language designed for teaching students object oriented concepts and dynamic typing.

An important aspect of the exercise is given by the executability attribute of the definition: as expected, designing executable definitions requires more attention to details; on the other hand, these definitions are testable, making it easier to detect design glitches.

Although we took The Complete Guide to the Future [8] as a starting point for our definitional enterprise, we decided rather early on not supporting certain Creol-specific constructs such as nondeterministic choice and parallel composition, as futures themselves bring a high degree of nondeterminism and concurrency. The new language we obtained is called KFUTURE.

3.1 Syntax

The syntax of KOOL, excepting that for threads – which was removed –, remains unchanged and we only added the same constructs as in [8]:

Exp ::= *Exp* `!` *Id*(*Exps*) [strict(1)]
| *Guard*

Guard ::= `wait`
| *Exp* `?` [strict]

Type ::= `!` *Type*

Id ::= `get`

Stmt ::= `await` *Exp* `;`

The expressions are enriched with asynchronous calls, a future reading operation `get`, and guards used to block/release the objects's tasks. The only added statement `await` is used for releasing tasks and $!T$ is the type of the futures returning values of type T.

3.2 Configuration

The configuration of the new language is represented in Fig. 3. KFUTURE objects are top-level independent agents [24], asynchronously communicating by means of futures.

Being an agent, each object carries its own state – holding fields and their values –, which can be altered only by the object's methods/tasks. Thus, the store cell is now object-local rather than global as in KOOL.

An object manages multiple tasks, each handling a specific future to which it is linked through the futureId cell. To ensure task atomicity, we follow the line in [8] and allow only one task being active at a time. Hence an object's tasks are split into the active-task and a pool of waiting-tasks.

Tasks are similar in essence to KOOL threads: the active-task cell includes almost all cells occurring in a thread cell. However, the location and environment stack envStack of an object are now at the top of the object cell, being shared by all tasks.

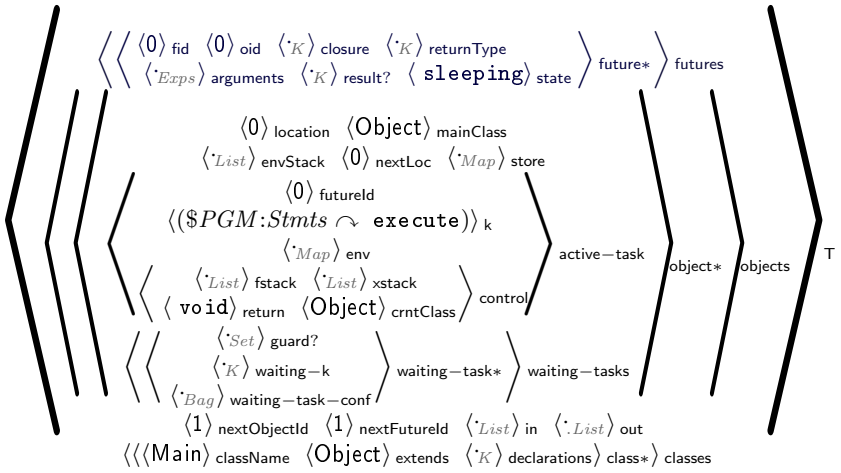


Fig. 3. The KFUTURE configuration

Futures (placed in the futures cell) serve as communication channels between objects. A future method invocation results in the creation of a new future cell containing data identifying the target object (oid), the method (closure) and its arguments. Each future has a state and will eventually produce a result.

To simplify the presentation, we have completely eliminated the threading constructs existing in the KOOL language. From a definitional point of view, this amounted to simply eliminating the extra syntax, cells in the configuration and specific threading rules in the definition. Section 4 shows how Java-like threads can be defined using futures.

In the following, we first revisit the changes required to the KOOL semantics to reflect the new configuration architecture, then describe the semantics of futures as an addition to the existing semantics.

3.3 Objects and methods

As objects are now full citizens of the configuration, several changes to the definition of KOOL are required to reflect that.

Object creation requires a redefinition of the semantics for `new`:

$$\left\langle \frac{\texttt{new}\ Class(Vs)}{\texttt{object}\ (Id, Class, Class)\ .\ \$\mathsf{clinit}(Vs)} \cdots \right\rangle_{\mathsf{k}} \left\langle \frac{Id}{Id +_{Int} 1} \right\rangle_{\mathsf{nextObjectId}}$$

$$\frac{\cdot_{Bag}}{\langle \cdots \langle Id \rangle_{\mathsf{location}} \ \langle Class \rangle_{\mathsf{mainClass}} \ \langle \cdots \langle \text{-1} \rangle_{\mathsf{futureId}} \ \langle \cdot_K \rangle_{\mathsf{k}} \cdots \rangle_{\mathsf{active-task}} \cdots \rangle_{\mathsf{object}}}$$

An instance of the class is created in the cell object and an unique identifier is assigned to it, while the new construct reduces to a method call to the special

method `$clinit` on a reference to the newly created object, a method which: (1) performs basic object allocation and initialisation; (2) calls the constructor method; (3) returns a reference to the object. Hence, `$clinit` basically corresponds to the `create(`*Class*`)` $\curvearrowright$ `storeObj` $\curvearrowright$ *Class*`(`*Vs*`)); return this;` sequence of tasks from the KOOL semantics for `new`, with the difference that now these tasks need to occur within the newly created object.

Object references replace KOOL's object closures. An object reference is a triple $(Id, MainClass, CurrentClass)$, where Id corresponds to the contents of the object's location cell, $MainClass$ corresponds to the contents of the mainClass cell, and $CurrentClass$ corresponds to the current contents of the crntClass cell, as shown by the new semantics for `this`:

$$\left\langle \frac{\texttt{this}}{\texttt{object}\ (Id, MClass, Class)} \cdots \right\rangle_{\mathsf{k}}\ \langle Id \rangle_{\mathsf{location}}\ \langle Class \rangle_{\mathsf{crntClass}}\ \langle MClass \rangle_{\mathsf{mainClass}}$$

The replacement of object closures by object references, made mandatory by the KFUTURE extension, required redefining some of the other KOOL rules, mainly those related to method and field resolution. Note though that these changes are actually simplifications to the existing KOOL semantics, inspiring us to redesign future versions of KOOL to use references instead of closures as object values.

Method calls to foreign objects are desugared into (blocking) future invocations [8]:

$$\left\langle \frac{\texttt{object}\ (Id, BClass, Class)\ .\ Method(Vs)}{(\ \texttt{object}\ (Id, BClass, Class)\ \texttt{!}\ Method(Vs))\ .\ \mathsf{get}(\cdot_{Exps})} \cdots \right\rangle_{\mathsf{k}}\ \langle Id' \rangle_{\mathsf{location}}$$

requires $Id \neq_K Id'$

The condition expressed by the clause **requires** ensures that the called method belongs indeed to a foreign object.

Method calls for the current object remain basically the same as in KOOL. While convenient, this additionally is a proper way to treat self calls, avoiding deadlock (which would occur if handled as foreign object calls) while capturing the direct transfer of control which was a caveat of the workaround solution proposed in [8].

First, the method is looked up in the object's environment stack:

$$\left\langle \frac{\texttt{object}\ (Id, BClass, \_)\ .\ Method(Vs)}{\texttt{lookupThis}\ (\langle ES \rangle_{\mathsf{envStack}}, BClass, Method)(Vs)} \cdots \right\rangle_{\mathsf{k}}\ \langle Id \rangle_{\mathsf{location}}\ \langle ES \rangle_{\mathsf{envStack}}$$

Next, once a method is evaluated to a method closure, application saves the current context before binding the arguments and calling the method:

$$\left\langle \frac{\texttt{methodClosure}(\_\ \texttt{->}\ T, Class, OL, Ps, S)(Vs) \curvearrowright K}{\texttt{mkDecls}\ (Ps, Vs)\ \ S \curvearrowright \texttt{return}\ \texttt{;}} \right\rangle_{\mathsf{k}} \langle OL \rangle_{\mathsf{location}} \left\langle \frac{Env}{\cdot_{Map}} \right\rangle_{\mathsf{env}}$$

$$\left\langle \left\langle \frac{\cdot_{List}}{\texttt{ListItem}\ ((Env, K, C\ \ \langle T' \rangle_{\mathsf{return}}\ \ \langle Class' \rangle_{\mathsf{crntClass}}))} \cdots \right\rangle_{\mathsf{fstack}} \left\langle \frac{Class'}{Class} \right\rangle_{\mathsf{crntClass}} \left\langle \frac{T'}{T} \right\rangle_{\mathsf{return}} C \right\rangle_{\mathsf{control}}$$

The only changes from the corresponding KOOL rule are (1) the fact that we enforce that the method's object location is the same as the current object's location; and (2) since we are inside the same object, the only object-related which needs to be updated/saved/restored is the current class.

The KOOL rule for the `return` statement is preserved unchanged, although an additional rule will be added below to model returning from a future call.

3.4 Futures

Futures model asynchronous method calls as messages exchanged between objects. These exchanges are captured by the future cells in the configuration, which serve as communication channels between objects.

Future method invocations result in opening a channel (future) to the object owning the method, containing a request for executing the method:

$$\left\langle \frac{\texttt{object}\ (OId, Class, \_)\ \texttt{!}\ Method(Vs)}{\texttt{preFuture}\ (F)} \cdots \right\rangle_{\mathsf{k}} \left\langle \frac{F}{F +_{Int} 1} \right\rangle_{\mathsf{nextFutureId}}$$

$$\frac{\cdot_{Bag}}{\langle \cdots \langle F \rangle_{\mathsf{fid}}\ \langle OId \rangle_{\mathsf{oid}}\ \langle\ \texttt{lookupMethod}\ (Class, Method) \rangle_{\mathsf{closure}}\ \langle Vs \rangle_{\mathsf{arguments}}\ \cdots \rangle_{\mathsf{future}}}$$

The future will initially be in the `sleeping` state, waiting to be activated by its corresponding object. The future invocation evaluates to a pre-future reference to the newly created future; this will become a full future reference once the return type of the method is known.

The activation of a `sleeping` *future* occurs when there are no active tasks running for the future's object, and consists in creating a task to initiate the method call, and changing the state of the future to `active` to prevent recurrent activations:

$$\left\langle \cdots \langle F \rangle_{\mathsf{fid}}\ \langle Id \rangle_{\mathsf{oid}}\ \langle Closure \rangle_{\mathsf{closure}}\ \langle Vs \rangle_{\mathsf{arguments}}\ \left\langle \frac{\texttt{sleeping}}{\texttt{active}} \right\rangle_{\mathsf{state}} \cdots \right\rangle_{\mathsf{future}}$$

$$\langle Id \rangle_{\mathsf{location}}\ \ \frac{\langle \cdots \langle \texttt{-1} \rangle_{\mathsf{futureId}}\ \langle \cdot_K \rangle_{\mathsf{k}}\ \cdots \rangle_{\mathsf{active-task}}}{\langle \cdots \langle F \rangle_{\mathsf{futureId}}\ \langle\ \texttt{performCall}\ (Closure, Vs) \rangle_{\mathsf{k}}\ \cdots \rangle_{\mathsf{active-task}}}$$

The auxiliary operation `performCall` does the actual method invocation which is similar to the one in KOOL, only without saving a stack frame (because no context needs to be saved). The contents of the futureId cell links the task to its corresponding future. The value -1 in the futureId cell is used to signal that the object is idle.

Returning from a future call occurs when a `return` statement is encountered and there are no function frames on the function stack. When this happens, we need to set the returned value as the result of the corresponding future and to signal this to the caller by setting the state of the future to complete:

$$\frac{\langle \cdots \langle Id \rangle_{\mathsf{futureId}} \ \langle \ \texttt{return}\ V\ \texttt{;}\ \cdots \rangle_{\mathsf{k}} \ \langle\ \texttt{.List} \rangle_{\mathsf{fstack}}\ \cdots \rangle_{\mathsf{active-task}}}{\langle \cdots \langle \texttt{-1} \rangle_{\mathsf{futureId}} \ \langle \cdot_K \rangle_{\mathsf{k}}\ \cdots \rangle_{\mathsf{active-task}}}$$

$$\left\langle \cdots \langle Id \rangle_{\mathsf{fid}} \ \left\langle \frac{\texttt{active}}{\texttt{completed}} \right\rangle_{\mathsf{state}} \ \frac{\cdot_{Bag}}{\langle V \rangle_{\mathsf{result}}} \ \cdots \right\rangle_{\mathsf{future}}$$

The active-task cell is resetted to indicate there is no current active task running.

Testing whether a future is resolved can be done using the `?` operator:

$$\left\langle \frac{\texttt{future}\ (Id, \_)\ \texttt{?}}{State =_K \ \texttt{completed}} \ \cdots \right\rangle_{\mathsf{k}} \ \langle Id \rangle_{\mathsf{fid}} \ \langle State \rangle_{\mathsf{state}}$$

The semantics of `get`. Get can only be called on future references and blocks until the corresponding future contains a value, with the effect of "returning" that value to the caller:

$$\left\langle \frac{\texttt{future}\ (Id, T)\ \texttt{.}\ \mathsf{get}(\cdot_{Exps})}{\texttt{subtype(}\ \texttt{typeOf}\,(V), T) \curvearrowright \texttt{true?} \curvearrowright \texttt{unsafeCast}\,(V, T)} \ \cdots \right\rangle_{\mathsf{k}} \langle Id \rangle_{\mathsf{fid}} \langle V \rangle_{\mathsf{result}}$$

requires $isExceptionVal(V) \neq_K \mathsf{true}$

The tasks associated to the "returning" value are the same to the ones from the KOOL rule for `return`, because we want to also extend the dynamic type checking aspect of the language over futures.

Another KOOL-related aspect is that of exception handling. Since KOOL gives semantics for exceptional behaviour, this has to be extended in the case of futures. Therefore, uncaught exceptions from a future call need to be propagated.

Exceptions and futures. If there is no exception handler in the exception stack, the exception thrown by the `throw` statement is returned as an exceptional value:

$$\left\langle \frac{\texttt{throw}\ V\ \texttt{;}}{\texttt{return exception}\,(V)\ \texttt{;}} \ \cdots \right\rangle_{\mathsf{k}} \ \langle \cdot_{List} \rangle_{\mathsf{xstack}}$$

When `get` is used on an exceptional value, the exception is thrown again:

$$\left\langle \frac{\texttt{future}\ (Id, \_)\ \texttt{.}\ \mathsf{get}(\cdot_{Exps})}{\texttt{throw}\ V\ \texttt{;}} \ \cdots \right\rangle_{\mathsf{k}} \ \langle Id \rangle_{\mathsf{fid}} \ \langle\ \texttt{exception}\ (V) \rangle_{\mathsf{result}}$$

3.5 Yielding control and rescheduling

As shown above, the semantics of `get` is blocking, which can be counter-productive when there are multiple concurrent asynchronous calls made to the same object.

The `await` *statement* allows one task to yield control until a condition is satisfied:

$$\frac{\texttt{await}\ (E\ \texttt{?})\ \texttt{;}}{\texttt{waiting}\ (\langle\ \texttt{SetItem}\ (E\ \texttt{?})\rangle_{\mathsf{guard}})} \qquad \frac{\texttt{await}\ (\ \texttt{wait})\ \texttt{;}}{\texttt{waiting}\ (\langle\ \texttt{SetItem}\ (\ \texttt{wait})\rangle_{\mathsf{guard}})}$$

To avoid overcomplicating the semantics, we restrict conditions to conjunctions of disjunctions of elements of the *Guard* type (`wait` and *Exp* `?`). There are more rules like the above, handling conjunction and disjunction, and attempting to simplify guards; however, if the condition cannot be reduced to true, the active task will need to block and wait to be rescheduled.

Yielding control. When `waiting` cannot be reduced, the active task is moved to the pool of waiting tasks and the object becomes idle:

$$\frac{\langle \mathit{Task}\ \ \langle\ \texttt{waiting}\ (\langle \mathit{Guards}\rangle_{\mathsf{guard}}) \curvearrowright K\rangle_{\mathsf{k}}\rangle_{\mathsf{active-task}}}{\langle \cdots\ \langle \texttt{-1}\rangle_{\mathsf{futureId}}\ \ \underset{\cdot_{Bag}}{\langle \cdot_K\rangle_{\mathsf{k}}}\ \cdots\rangle_{\mathsf{active-task}}}$$

$$\frac{}{\langle\langle \mathit{Guards}\rangle_{\mathsf{guard}}\ \ \langle K\rangle_{\mathsf{waiting-k}}\ \ \langle \mathit{Task}\rangle_{\mathsf{waiting-task-conf}}\rangle_{\mathsf{waiting-task}}}$$

Note that the guard cell argument of the `waiting` computation task, holding a disjunction of basic guards represented as a set, becomes the guard cell of the newly created waiting-task.

Departing from [8], we chose not to model tasks as a queue, but rather as a bag, to capture any possible scheduling policy.

Simplifying guards. A waiting task's guard is removed if one of the futures it waits upon completes:

$$\underset{\cdot_{Bag}}{\langle \cdots\ \texttt{SetItem}\ (\ \texttt{future}\ (\mathit{Id}, \_)\ \texttt{?})\ \cdots\rangle_{\mathsf{guard}}}\ \ \langle \mathit{Id}\rangle_{\mathsf{fid}}\ \ \langle\ \texttt{completed}\rangle_{\mathsf{state}}$$

The `wait` guard is used to unconditionally yield control; therefore, once the task becomes a waiting task, we can dissolve the guard to allow its reactivation:

$$\left\langle \cdots\ \underset{\cdot_{Bag}}{\langle \cdots\ \texttt{SetItem}\ (\ \texttt{wait})\ \cdots\rangle_{\mathsf{guard}}}\ \cdots \right\rangle_{\mathsf{waiting-task}}$$

Regaining control. If an object is idle and the guard of one of its waiting tasks has dissolved, then that waiting task can be reactivated:

$$\frac{\langle \cdots\ \langle \texttt{-1}\rangle_{\mathsf{futureId}}\ \cdots\rangle_{\mathsf{active-task}}}{\langle \mathit{Task}\ \ \langle K\rangle_{\mathsf{k}}\rangle_{\mathsf{active-task}}} \qquad \frac{\langle\langle K\rangle_{\mathsf{waiting-k}}\ \ \langle \mathit{Task}\rangle_{\mathsf{waiting-task-conf}}\rangle_{\mathsf{waiting-task}}}{\cdot_{Bag}}$$

3.6 Global and Local Future Invariants

In [8] a proof system for proving a set of monitor invariants that describe the release points is presented. A monitor invariant i is a local property of an object

that must hold each time the `await` statement is scheduled. A monitor invariant is proved in the presence of global invariants I, which describe invariant properties of the future objects.

These invariants can be easily expressed as matching logic formulas [17], which can be thought as configuration terms with variables and constraints over these variables. We exhibit this by two simple examples. Let I be the global invariant: for any future z associated to the method `m` of the class `C`, if the state of z is completed then the return value is positive. I is formally expressed by the following matching logic formula, written using the abstraction mechanism:

$$\begin{array}{l}\langle \texttt{C} \rangle_{\mathsf{mainClass}}\ \langle Oid \rangle_{\mathsf{oid}} \langle \cdots\ \texttt{ListItem}(\_, \langle \cdots\ \texttt{m} \mapsto L\ \cdots \rangle_{\mathsf{env}}\ \cdots \rangle_{\mathsf{envStack}} \\ \quad \langle Oid \rangle_{\mathsf{location}} \langle \texttt{lookup}(L) \rangle_{\mathsf{closure}}\ \langle V \rangle_{\mathsf{result}}\ \langle \mathsf{completed} \rangle_{\mathsf{state}}\end{array} \longrightarrow V > 0$$

In the left hand side of the implication we have (the abstraction of) a configuration term, which is a particular matching logic formula: the first line is a pattern matching objects of the class `c` having the method `m` stored at location L, and the second line is a pattern matching futures associated to the method `m` (via location L), and that are completed and have the return value V. The object reference Oid connects the future with its associated object. In the right hand side of the implication is the constraint on V.

Similarly, a monitor invariant saying that "for any instance of the class `C`, its field `fld` must have a nonzero value at any release point" is formally expressed as follows:

$$\langle \texttt{await}\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ \texttt{fld} \mapsto X\ \cdots \rangle_{\mathsf{env}}\ \langle \texttt{C} \rangle_{\mathsf{crntClass}} \longrightarrow X \neq 0$$

Writing the global and monitor invariants as matching logic formulas has the advantage that they are expressed in the same logic used to give semantics for the programming language. This allows the direct use of the semantics for proving the correctness of such properties. In particular, the correctness of invariants can be proved using the symbolic execution and the circular coinduction technique described in [2]. More precisely, that general technique can be combined with the proof system given in [8] to obtain a specialized prover parametric in the language definition. Since matching logic formulas are written at a lower level, by considering the configuration as particular formulas, a richer class of properties can be expressed. On the other hand, the abstraction level used in [8] can be preserved by developing tools that automatically translate higher-level formulas into matching logic formulas following the idea used in the MatchC prover [22].

4 Experiments

A main advantage of the formal semantics defined in $\mathbb{K}$ is that they are directly executable using the $\mathbb{K}$ tool. A first experiment we did was to test if the KOOL programs, used to test the KOOL definition, can be executed with the new semantics. All programs, excepting those including threads, were successfully executed and their executions produced the same outputs with those obtained with the definition of KOOL.

Multithreading defined through futures. Even if the threads were removed from KOOL to define KFUTURE, the concept of threads can be somehow regained at the programming level. For example, one may define a base class `Thread` as follows:

```
class Thread {
  !void id;
  void run() { }  // to be overridden
  void start() {    id = this ! run();   }
  void join() {     await(id ?);   }
}
```

Then, specific threads can be defined by extending the class `Thread` with particular behaviour and overriding the method `run`.

The Thread class enables the concurrent execution of multiple threads. Note, however, that KOOL's globally shared memory is no longer directly available to the programmer, each object now carrying its own memory.

Nevertheless, the objects themselves are still globally shared and that suffices from a programming point of view. Programmers need only to assume a shared-memory model where all object data is hidden and thus only accessible through the interfaces provided by the objects, which is considered good object-oriented programming discipline.

Hence, KOOL with futures brings relative little change to the programming model, while providing certain important benefits at a semantics level: futures allow for all accesses to memory to be clearly sequentialized, enabling better abstraction and reasoning techniques for program analysis and verification.

Future-induced deadlocks. We tested the definition on various examples in order to see if there is a combination of method calls for foreign objects and those for the current object that leads to a deadlock. (Un)Fortunately we found such an example:

```
class A {
 B b;
 void A() { }
 void setB(B b) { this.b = b; }
 void callB() {    b.c();  }
}

class B {
  A a;
  void B(A a) { this.a = a; a.setB(this); }
  void c() { print("It works!"); }
  void callA() { a.callB();  }
}
```

```
class Main {
  void Main() {
    A a = new A();
    B b = new B(a);
    b.callA();
  }
}
```

The `A` object `a` has a reference to the `B` object b, and the object `b` has a reference to `a`. The call of `b.callA()` triggers the call of `a.callB()`, which in turn triggers the call of `b.c()`. The execution blocked on a configuration with two active

futures, the ones for `b.callA()` and `a.callB()`, and a sleeping future , that for `b.c()`. Although not present in Creol, this problem seems to originate in the identification between processes and objects proposed in [8].

5 Conclusion and Future Work

We presented an executable formal semantics, defined using the $\mathbb{K}$ Framework, for an object-oriented programming languages with asynchronous method calls modelled with futures. We used KOOL – an object oriented programming language already defined in $\mathbb{K}$ for teaching and research purposes – and we followed the line from [8] for the definition of the futures. However, there are some important points where the two approaches differ, e.g., the treatment of threads, method calls for the current object, the scheduling of the tasks inside of an object process. A main advantage of using $\mathbb{K}$ Framework is that the formal semantics of the language is directly executable by the $\mathbb{K}$ tool, hence no further encoding of the formal semantics to an executable framework is needed. The designed definition can be tested on programs, analysed, and adjusted according to the desired behaviour. In this way we found several more natural solutions for KFUTURE than those proposed in [8]. We also detected a case when the combination of the method calls for foreign objects with those of the current object can lead to a deadlock.

This exercise to define KFUTURE starting from that of KOOL was also a good test for the modularity of the $\mathbb{K}$ Framework. The configuration of KFUTURE is strongly different from that of KOOL: some cells were removed (e.g. those for threads), some cell have been added (e.g. for objects, futures, auxiliary constructs), and the nesting structure have been substantially changed. In this context, it is expected that the rules of KOOL to be changed in order to accommodate with the new configuration. This did not happen: from the 129 rules of KOOL, 8 have been removed because they give semantics for threads and 13 have been replaced with other 49 that give the semantics to both implicit and explicit futures. Besides the modular aspect of $\mathbb{K}$, this numbers show also that the change is not trivial. Since the semantics is directly executable, all details have to be specifed.

The $\mathbb{K}$ definition of the new language, KFUTURE, can be found on the github repository, `http://github.com/roKmania/KFuture`, and it can be executed with the version 3.5.2 of the $\mathbb{K}$ tool, `https://github.com/kframework/k/releases/tag/v3.5.2`.

This exercise is a first step toward a methodology of how to add asynchronous methods/function calls with futures to an existing programming language defined in $\mathbb{K}$. This methodology could allow to generalise the proof system proposed in [8], for verifying monitor invariant of the release points, to a generic proof system expressed in the terms of matching logic [17] and reachability logic [6].

References

1. E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. *J. Log. Algebr. Program.*, 78(7):491–518, 2009.
2. A. Arusoaie, D. Lucanu, and V. Rusu. A Generic Framework for Symbolic Execution: Theory and Applications. Research Report RR-8189, Inria, Sept. 2015.
3. D. Bogdănaş and G. Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.
4. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 123–134. ACM, 2004.
5. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, 2009.
6. A. Ştefănescu, Ş. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, and G. Roşu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014.
7. O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. RS RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. This report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.
9. C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebr. Meth. Program.*, 83(5-6):360–383, 2014.
10. C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.
11. D. Filaretti and S. Maffeis. An executable formal semantics of PHP. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 567–592. Springer, 2014.
12. D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 2nd edition, 2001.
13. L. Henrio and M. U. Khan. Asynchronous components with futures: Semantics and proofs in Isabelle/HOL. *Electr. Notes Theor. Comput. Sci.*, 264(1):35–53, 2010.
14. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1-2):23–66, 2006.

15. R. H. H. Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
16. D. Park, A. Ştefănescu, and G. Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015.
17. G. Roşu. Matching logic — extended abstract. In *Proceedings of the 26th International Conference on Rewriting Techniques and Applications (RTA'15)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5–21, Dagstuhl, Germany, July 2015.
18. G. Roşu and T. Şerbănuţă. Kool – typed – dynamic. K Tutorial, http://www.kframework.org/index.php/K_Tutorial.
19. G. Roşu and A. Ştefănescu. From Hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2012.
20. G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.
21. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
22. G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012. Also available as technical report `http://hdl.handle.net/2142/33771`.
23. A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 439–453. ACM, 2005.
24. M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, second edition edition, 2009.
25. A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Object-oriented concurrent programming-modelling and programming in an object-oriented concurrent language, ABCL/1. In *Object-oriented concurrent programming*, pages 55–89, 1987.