Towards Formal Verification of Orchestration Computations Using the $\mathbb{K}$ Framework

Musab A. AlTurki(✉) and Omar Alzuhaibi

King Fahd University of Petroleum and Minerals Dhahran,
Dhahran, Saudi Arabia
musab@kfupm.edu.sa, omar.zud@gmail.com

Abstract. Orchestration provides a general model of concurrent computations. A minimal yet expressive theory of orchestration is provided by Orc, in which computations are modeled by site calls and their orchestrations through a few combinators. Using Orc, formal verification of correctness of orchestrations amounts to devising an executable formal semantics of Orc and leveraging existing tool support. Despite its simplicity and elegance, giving formal semantics to Orc capturing precisely its intended behaviors is far from trivial primarily due to the challenges posed by concurrency, timing and the distinction between internal and external actions. This paper presents a semantics-based approach for formally verifying Orc orchestrations using the $\mathbb{K}$ framework. Unlike previously developed operational semantics of Orc, the $\mathbb{K}$ semantics is not directly based on the interleaving semantics given by Orc's SOS specification. Instead, it is based on concurrent rewriting enabled by $\mathbb{K}$. It also utilizes various $\mathbb{K}$ facilities to arrive at a clean, minimal and elegant semantic specification. To demonstrate the usefulness of the proposed approach, we describe a specification for a simple robotics case study and provide initial formal verification results.

Keywords: Formal semantics · Orc · $\mathbb{K}$ framework · Concurrency · Program verification

1 Introduction

Orchestration provides a general model of concurrent computations, although it is more often referred to in the context of service orchestrations describing the composition and management of (web) services. A minimal yet expressive theory of orchestration is provided by the Orc calculus [20,22,21], in which computations are modeled by site calls and their orchestrations through four semantically rich combinators: the "parallel", "sequential", "pruning" and "otherwise" combinators. Orc provides an elegant yet expressive programming model for concurrent and real-time computations. While Orc's simplicity and mathematical elegance enable formal reasoning about its constructs and programs, its programming model is very versatile and easily applicable to a very wide range of programming domains, including web-based programming, business processes, and distributed cyber-physical system applications, as amply demonstrated in [22,21].

N. Bjørner and F. de Boer (Eds.): FM 2015, LNCS 9109, pp. 40–56, 2015.
DOI: 10.1007/978-3-319-19249-9_4

As for other theories and programming models, devising formal semantics for Orc is of fundamental importance for several reasons, including theoretical advancements and refinements to its underlying theory, formal verification of its programs, building formally verifiable implementations, and also for unambiguous documentation. Furthermore, to better satisfy these goals, the semantics has to be *executable*, enabling quick prototyping and simulation of Orc programs through a formally defined interpreter induced by the executable specification. The rewriting logic semantics project [17,18,8,19] has been advocating this approach of formal executable semantics and has proved its value for many programming models and languages, including widely used general-purpose languages like Java [10,11] and C [9].

Giving formal executable semantics to Orc constructs capturing precisely its intended behaviors has been of interest since Orc's inception due mainly to the challenges posed by concurrency, timing and the distinction between internal and external actions. A simple computation in Orc is modeled by a site call, representing a request for a service, and more complex computations can be achieved by combining site calls into expressions using one or more of Orc's four sequential and parallel combinators. A complete formal executable semantics elegantly capturing its semantic subtleties, including its real-time behaviors and transition priorities, was given in rewriting logic [16] and implemented in the Maude tool [1,2]. This semantics is based on the original reference SOS semantic specification of the instantaneous (untimed) semantics of Orc [22].

In what can be considered as a continuation of these efforts, this paper presents a formal, executable semantics of Orc using the $\mathbb{K}$ framework [24,15], which is a derivative of the rewriting logic framework, towards providing a $\mathbb{K}$ -based framework for formally specifying and verifying Orc orchestrations. Unlike previously developed operational semantics of Orc, the $\mathbb{K}$ semantics described here is not directly based on the interleaving semantics given by the reference SOS specification of Orc. Instead, the $\mathbb{K}$ semantics provides the advantage of true concurrency enabled by $\mathbb{K}$, where two (or more) concurrent transitions are allowed to fire even in the presence of (read-access) resource sharing. It also utilizes $\mathbb{K}$'s specialized notations and facilities to arrive at a clean, minimal and elegant semantic specification. Moreover, the semantics is executable in the associated $\mathbb{K}$ tool [6,14], enabling rapid prototyping and formal analysis of Orc programs. Furthermore, the semantics implicitly presents a generic methodology through which concurrency combinators are mapped to threads and computations in $\mathbb{K}$, which can be instantiated to other concurrency calculi. Finally, to demonstrate the usefulness and applicability of the proposed approach, we describe a specification for a simple robotics case study and provide initial formal verification results.

The paper is organized as follows. In Section 2 below, we overview the $\mathbb{K}$ framework and Orc. Then, in Section 3, we present the $\mathbb{K}$ semantics of Orc. This is followed by a discussion of some sample Orc programs in Section 4. The paper concludes in Section 5 with a summary and a discussion of future work.

2 Background

This section presents some preliminaries on the $\mathbb{K}$ framework and the $\mathbb{K}$ tool, and introduces the Orc calculus along with some simple examples.

2.1 The $\mathbb{K}$ Framework

$\mathbb{K}$ [24,25] is a framework for formally defining the syntax and semantics of programming languages. It includes several specialized syntactic notations and semantic innovations that make it easy to write concise and modular definitions of programming languages. $\mathbb{K}$ is based on context-insensitive term rewriting, and builds upon three main concepts inspired by existing semantic frameworks:

- *Computational Structures (or Computations)*: A computation is a task that is represented by a component of the abstract syntax of the language or by an internal structure with a specific semantic purpose. Computations enable a natural mechanism for flattening the (abstract) syntax of a program into a sequence of tasks to be performed.
- *Configurations*: A configuration is a representation of the static state of a program in execution. $\mathbb{K}$ models a configuration as a possibly nested cell structure. Cells are labeled and represent fundamental semantic components, such as environments, stores, threads, locks, stacks, etc., that are needed for defining the semantics.
- *Rules*: Rules give semantics to language constructs. They apply to configurations, or fragments of configurations, to transform them into other configurations. There are two types of rules in $\mathbb{K}$: *structural rules*, which rearrange the structure of a configuration into a behaviorally equivalent configuration, and *computational rules*, which define externally observable transitions across different configurations. This distinction is similar to that of equations and rules in Rewriting Logic [16], and to that of heating/cooling rules and reaction rules in CHAM [3].

To briefly introduce the notations used in $\mathbb{K}$ rules, we present a $\mathbb{K}$ rule used for variable lookup (Fig. 1).

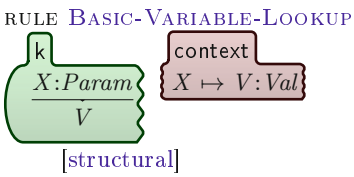

Fig. 1. Variable lookup rule as defined in $\mathbb{K}$

The illustrated rule shows two bubbles, each representing a cell predefined in the configuration. `k` is the computation cell, while `context` is the cell that

holds variable mappings. Each bubble can be smooth or torn from the left, right, or both sides. A both-side-smooth cell means that the matched cell should contain only the content specified in the rule. A right-side-torn cell means that the matching should occur at the beginning of the cell; this allows for matching when more contents are at the end of the matched cell. Similarly, a left-side-torn cell means that the matching should occur at the end of the cell, so that unspecified content can be on left of the specified term. A both-sides-torn cell means that the matching can occur anywhere in the matched cell. Furthermore, Upper-case identifiers such as X and V are variables to be referenced inside the rule only; they can be followed by a colon meaning "of type". Finally, the horizontal line means that the top term rewrites to the bottom term. What this rule does is that it matches a *Param* X at the beginning of a `k` cell, matches the same X in the context cell mapped to a *Value* V, and then rewrites the X in the `k` cell to the value V.

$\mathbb{K}$ combines many of the desirable features of existing semantics frameworks, including expressiveness, modularity, convenient notations, intuitive concepts, conformance to standards, etc. One very useful facility of $\mathbb{K}$ when defining programming languages is the ability to tag rules with built-in attributes, e.g. `strict`, for specifying evaluation strategies, which are essentially notational conveniences for a special category of structural rules (called heating/cooling rules) that rearrange a computation to the desired evaluation strategy. Using attributes, instead of explicitly writing down these rules protects against potential specification errors and avoids going into unwanted non-termination. In general, these attributes constitute a very useful feature of $\mathbb{K}$ that makes defining complex evaluation strategies quite easy and flexible.

Furthermore, $\mathbb{K}$ is unique in that it allows for true concurrency even with shared reads, since rules are treated as transactions. In particular, instances of possibly the same or different computational rules can match overlapping fragments of a configuration and concurrently fire if the overlap is not being rewritten by the rules. Truly concurrent semantics of $\mathbb{K}$ is formally specified by graph rewriting [7]. For more details about the $\mathbb{K}$ framework and its features and semantics, the reader is referred to [24,25].

An implementation of the $\mathbb{K}$ framework is given by the $\mathbb{K}$ tool [6,14], which is based on Maude [4], a high-performance rewriting logic engine. Using the underlying facilities of Maude, the $\mathbb{K}$ tool can interpret and run $\mathbb{K}$ semantic specifications providing a practical mechanism to simulate programs in the language being specified and verify their correctness. In addition, the $\mathbb{K}$ tool includes a state-space search tool and a model checker (based, respectively, on Maude's search and LTL model-checking tools), as well as a deductive program verifier for the targeted language. This allows for dynamic formal verification of Orc programs in our case.

The $\mathbb{K}$ tool can compile definitions into a Maude definition using the `kompile` command. It can then do several operations on the compiled definition using its Maude backend. `krun` can execute programs and display the final configuration. `krun` with the `--search` option displays all different solutions that can be

reached through any non-deterministic choices introduced by the definition. An option `--pattern` can be specified to only display configurations that match a certain pattern. Moreover, `--ltlmc` directly uses Maude's LTL model checker[1].

The $\mathbb{K}$ tool effectively combines the simplicity and suitability of the $\mathbb{K}$ framework to defining programming languages with the power and features of Maude. A fairly recent reference on the $\mathbb{K}$ tool that gently introduces its most commonly useful features can be found in [6].

2.2 The Orc Calculus

Orc [20,22] is a theory for orchestration of services that provides an expressive and elegant programming model for timed, concurrent computations. A *site* in Orc represents a service (computation) provider, which, when called, may produce, or *publish*, at most one value. Site calls are *strict*, i.e., they have a call-by-value semantics. Moreover, different site calls in Orc may occur at different times. For effective programming in Orc, a few *internal* sites are assumed, namely (1) the *if*(b) site, which publishes a signal if b is true and remains silent otherwise, (2) *Clock*, which publishes the current time value, and (3) *Rtimer*(t), which publishes a signal after t time units.

Syntax of Orc. An Orc program $\tilde{d}; f$ is a list of expression definitions $\tilde{d}$ followed by an expression f. An Orc *expression* describes how site calls (and responses) are combined in order to perform a useful computation. The abstract syntax of Orc expressions is shown in Fig. 2. We assume a special site response value **stop**, which may be used to indicate termination of a site call without necessarily publishing a standard Orc value.

$$
\begin{array}{rl}
f, g \in \text{Expression} ::= & \mathbf{0} \mid p(\tilde{p}) \mid f \mid g \mid f >x> g \mid g <x< f \mid f \,;\, g \\
p \in \text{Parameter} ::= & x \mid w \\
& x \in \mathit{Variable} \qquad w \in \mathit{Value} \cup \{\mathbf{stop}\}
\end{array}
$$

Fig. 2. Abstract syntax of Orc expressions

An Orc *expression* can be: (1) the silent expression (**0**), which represents a site that never responds; (2) a parameter or an expression call having an optional list of actual parameters as arguments; or (3) the composition of two expressions by one of four composition operators. These are: (1) the "parallel" combinator, $f \mid g$, which models concurrent execution of independent threads of computation; (2) the "sequential" combinator, $f >x> g$, which executes f, and for each value w published by f creates a fresh instance of g, with x bound to w, and runs that

[1] The latest release of $\mathbb{K}$ 3.5 depends on Maude as well as Java as backends. It is the last version to support the Maude backend. Developments are running on the Java backend to incorporate all of Maude's features.

instance in parallel with the current evaluation of $f >x> g$; (3) the "pruning" combinator, $f <x< g$, which executes f and g concurrently but terminates g once g has published its first value, which is then bound to x in f; finally (4) the "otherwise" combinator, $f \,;\, g$, which attempts to execute f to completion, and then executes g only if f terminates without ever publishing a value.

A variable x occurs *bound* in an expression g when g is the right (resp. left) subexpression of a sequential composition $f >x> g$ (resp. a pruning composition $g <x< f$). If a variable is not bound in either of the two above ways, it is said to be *free*. We use the syntactic sugar $f \gg g$ (resp. $g \ll f$) for sequential composition (resp. pruning composition) when x is *not* free in g. To minimize use of parentheses, we assume the following precedence order (from highest to lowest): $\gg$, $|$, $\ll$, $;$.

To illustrate the informal meaning of the combinators, we list some examples here. Many more examples and larger programs can be found in [22,12,5,13,21].

Example 1. Suppose we want to get the current price of gold, and that we have three sites that provide this service: GoldSeek, GoldPrice, and Kitco. In such a case, we only care about receiving an answer as soon as possible. So, it would make sense to call these three sites in parallel. The expression would be: $(GoldSeek() \mid GoldPrice() \mid Kitco())$. Now, suppose we want the price in a different unit, say Euro/gram instead of USD/Oz. We need only one of these three sites to publish a value. Observe the following Orc expression:

$Converter(x, USD/Oz, EUR/gram) < x < (GoldSeek() \mid GoldPrice() \mid Kitco())$.

The pruning combinator tells the parallel expression to give it only the first value it publishes. As soon as it receives a value, it prunes the whole right-side expression and passes the value to the left side, and binds it to x.

Example 2. Suppose we have a site called *FireAlarm* that when called, remains silent unless a fire has been detected, in which case it publishes the fire's location. That information is sent to the fire department which needs to make a decision to dispatch a fire engine. The fire department calls a site *CalcNearestStation* and gives it the location of the fire to locate the nearest fire station. The response is then passed on to a site *Dispatch* which will dispatch a fire truck from the given station to the given location. The Orc expression would be:

$$FireAlarm() > fireLoc > CalcNearestStation(fireLoc) > station > Dispatch(station, fireLoc)$$

After detailing our semantics of Orc in Section 3, we show the output of executing some sample expressions in Section 4.

Operational Semantics of Orc. The reference semantics of Orc is the informal but detailed semantics of Orc given by Misra and illustrated by many examples in [20]. A structural operational semantics (SOS) for the instantaneous (untimed) behaviors of Orc was also developed by Misra and Cook in [22]. An updated SOS listing that includes rules for the semantics of the *otherwise* combinator and **stop** site responses is given in [2].

The SOS semantics specifies an interleaving semantics of the possible behaviors of an Orc expression as a labeled transition system with four types of actions an Orc expression may take: (1) publishing a value, (2) calling a site, (3) making an unobservable transition τ, and (4) consuming a site response. As discussed by Misra and Cook in [22], the SOS semantics is highly non-deterministic, allowing *internal* transitions within an Orc expression (value publishing, site calls, and τ transitions) and the *external* interaction with sites in the environment (through site return events) to be interleaved in any order. Therefore, a *synchronous semantics* was proposed in [22] by placing further constraints on the application of SOS semantic rules, effectively giving internal transitions higher priority over the external action of consuming a site response.

A timed SOS specification extending the original SOS with timing was also proposed [26]. The timed SOS refines the SOS transition relation into a relation on time-shifted Orc expressions and timed labels of the form (l, t), where t is the amount of time taken by a transition. In this extended relation, a transition step of the form $f \xrightarrow{(l,t)} f'$ states that f may take an action l to evolve to f' in time t, and, if $t \neq 0$, no other transition could have taken place during the t time period. To properly reflect the effects of time elapse, parts of the expression f may also have to be time-shifted by t. The semantics described in [26] abstracted away the non-publishing events as unobservable transitions, which is the level of abstraction we assume in the $\mathbb{K}$ semantics we describe next.

3 $\mathbb{K}$-Semantics of Orc

The semantics of Orc in $\mathbb{K}$ is specified in two modules: (1) the syntax module, which defines the abstract syntax of Orc in a BNF-like style along with any relevant evaluation strategy annotations, and (2) the semantics module, which defines the structure of a configuration and the rules (both structural and computational) that define Orc program behaviors. These modules are explained in some detail in this section. The full $\mathbb{K}$ specification of Orc can be found at (http://www.ccse.kfupm.edu.sa/~musab/orc-k).

3.1 Syntax Module

Orc is based on execution of expressions, which can be simple values or site calls, or more complex compositions of simpler subexpressions using one or more of its combinators. Looking at Fig. 2 showing the abstract syntax of the Orc calculus, the following grammar defined in $\mathbb{K}$ syntax is almost identical (with *Pgm* and *Exp* as syntactic categories for Orc programs and expressions, respectively):

An Orc value, which could be an integer, a string, a boolean, or the `signal` value, is syntactic sugar for a site call that publishes that value and halts.

A site call looks like a function call, having the site name and a list of actual parameters we call *Arguments*. A site, when called, may publish a standard Orc value or a special value `stop`, which indicates termination with no value being published. A site call can result in publishing at most one value.

```
SYNTAX  Pgm ::= ExpDefs Exp

SYNTAX  Arg ::= Val
              | Identifier

SYNTAX  Call ::= ExpId(Params)
              | SiteId(Args) [strict(2)]

SYNTAX  Exp ::= Arg
              | Call
              > Exp > Param > Exp [right]
              > Exp | Exp [right]
              > Exp < Param < Exp [left]
              > Exp ; Exp [left]
```

Fig. 3. Syntax of Orc as defined in $\mathbb{K}$

There are a few semantic elements, which appear in Fig. 3, that $\mathbb{K}$ allows to define within the syntax module. The first is precedence, denoted by the $>$ operator. As mentioned in Section 2.2, the order of precedence of the four combinators from highest to lowest is: the sequential, the parallel, the pruning, and then the otherwise combinator. In addition, we prefer for simpler expressions to be matched before complex ones; so, on top, we put *Arg* and *Call*.

The second semantic element that is defined within the syntax module of $\mathbb{K}$ is `right-` or `left`-associativity. It is important to note that the parallel operator is defined as right-associative, rather than fully-associative because $\mathbb{K}$'s parser does not yet support full associativity. However, this is resolved in the semantics by transforming the tree of parallel composition into a fully-associative soup of threads as discussed in Section 3.2.

The third is strictness. `strict(i)` means that the i^{th} term in the right hand side of the production must be evaluated before the production is matched.

3.2 Semantics Module

This module specifies the semantics of the language using $\mathbb{K}$ rules. Each rule specifies one or more *rewrites*, that take place in different parts of the *configuration*. We first explain the structure of the configuration, followed by key rules.

Configuration. A configuration in $\mathbb{K}$ is a representation of a state consisting of possibly nested cells. Fig. 4 shows the structure of our configuration. A cell `thread` is declared with multiplicity *, i.e., zero, one, or more threads. Enclosed in `thread` is the main cell `k`. `k` is the computation cell where we execute our program. We handle Orc productions from inside the `k` cell.

The `context` cell is for mapping variables to values. The `publish` cell keeps the published values of each thread, and `gPublish` is for globally published values. `props` holds thread management flags. `varReqs` helps manage context sharing. `gVars` holds environment control and synchronization variables. The `in` and `out` cells are respectively the standard input and output streams. And finally, `defs` holds the expressions defined at the beginning of an Orc program.

Each cell is declared with an initial value. The `$PGM` variable, which is the initial value of the `k` cell, tells $\mathbb{K}$ that this is where we want our program to go (after it is parsed). So by default, the initial configuration, shown in Fig. 4, would hold a single thread with the `k` cell holding the whole Orc program as the `Pgm` non-terminal defined in the syntax above.

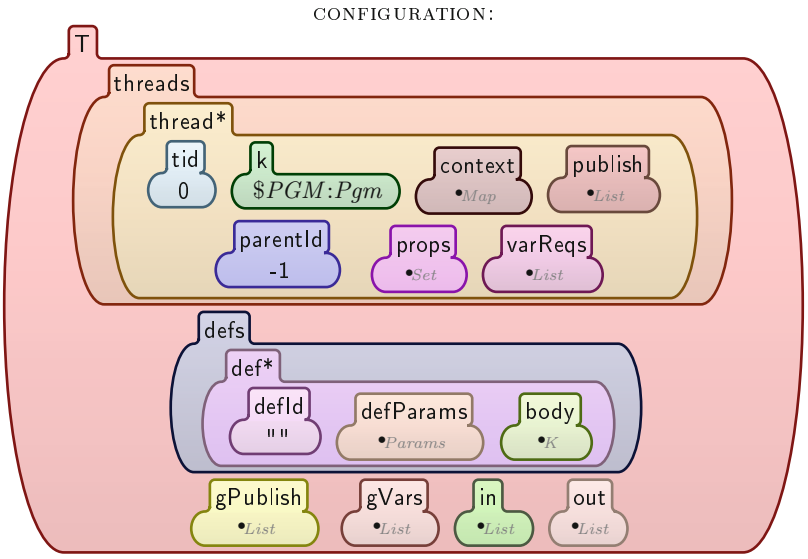

Fig. 4. Structure of the configuration

$\mathbb{K}$ **Rules.** For clarity and convenience, we first illustrate the essence of the rules as transformations in schematic diagrams. Then we show some representative rules exactly as they are defined in $\mathbb{K}$. Our schematic diagrams use the following notations. Each box represents a thread while lines are drawn between boxes to link a parent thread to child threads, where a parent thread appears above its child threads. The positioning of a child thread indicates whether that thread is a left-side child or a right-side child (which is needed by the sequential and pruning compositions). Note that in the specification, this information is maintained through meta thread properties. The center of a box holds the expression the thread is executing. A letter v at the lower right corner of the box represents a value which the thread has published. A letter P at the lower left corner denotes the `publishUp` flag which basically tells the thread to move its published values to its parent thread. Variable mappings such as $x \rightarrow v$ mapping a variable x to a value v are displayed at the bottom of the box. Finally, the symbol $\Rightarrow$ denotes a rewrite.

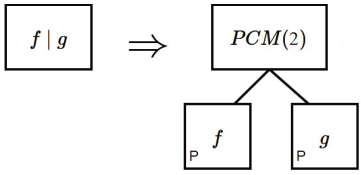

Fig. 5. Transformation rule of the parallel combinator

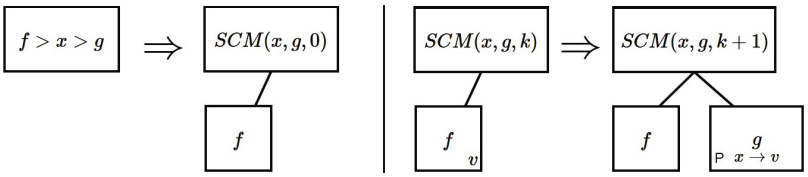

Fig. 6. Transformation rules of the sequential combinator

Combinators. Orc has four combinators, which combine subexpressions according to four distinct patterns of concurrent execution, *parallel*, *sequential*, *pruning* and *otherwise*.

Parallel Combinator. Given an expression $f \mid g$ as shown in Fig. 5, the rule creates a manager thread carrying a meta-function called $PCM(x)$, short for Parallel Composition Manager, where x is the count of sub-threads it is managing. Child threads are created as well for each of the expressions f, and g. This of course extends to any number of subexpressions in the initial expression. For example, $f \mid g \mid h$ will transform to $PCM(3)$ and so on, as each subexpression will be matched in turn.

Sequential Combinator. The first rule of the sequential combinator, shown in Fig. 6, creates a manager called *SCM*, short for Sequential Composition Manager; and it creates one child that will execute f. The manager keeps three pieces of information: x, the parameter through which values are passed to instances of g; g, the right-side expression; and k, a count of active instances of g which is initially 0.

Every time f publishes a value, the second rule in Fig. 6 creates an instance of g with its x parameter mapped to the published value. The new instance will work independently of all of f, the manager, and any other instance that was created before. So in effect, it is working in parallel with the whole composition, as is meant by the informal semantics [20].

Pruning Combinator. The idea of the pruning expression is to pass the first value published by g to f as a variable x defined in the context of f. Regardless, f should start execution anyway. If it needed a value for x to continue its execution,

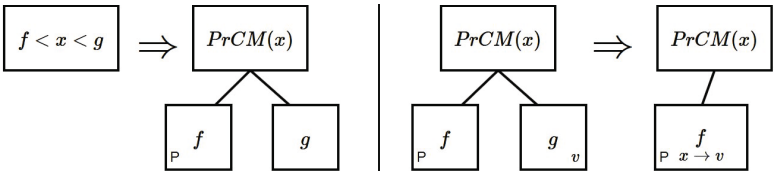

Fig. 7. Transformation rules of the pruning combinator

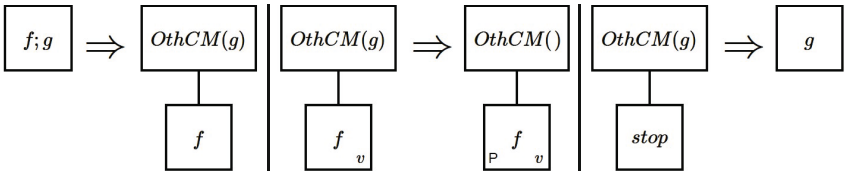

Fig. 8. Transformation rules of the otherwise combinator

it would wait for it. So, the first rule of the pruning combinator creates a manager *PrCM* (short for Pruning Composition Manager), a thread executing f, and another thread executing g. See Fig. 7. The second rule is responsible for passing the published value from g to f and terminating (pruning) g. These two rules are shown in Figures 10 and 11 as they are defined in $\mathbb{K}$.

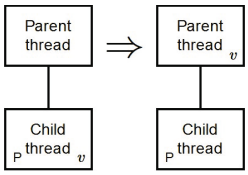

Fig. 9. Transformation rule of publishing values

Otherwise Combinator. The otherwise combinator is implemented in three rules shown in Fig. 8. It starts by creating a manager called *OthCM* (short for Otherwise Composition Manager) and a child thread to execute f. Then if f publishes its first value, g is discarded and f may continue to execute and is given permission to publish. However, if f halts without publishing anything, the third rule applies and the whole otherwise expression is replaced by g. As mentioned in Section 2.2, *stop* is a special value that indicates that an expression has halted.

Publishing and Variable Lookup. Due to the uniform structure of thread hierarchy common in the productions of all four combinators, defining general operations like publishing and variable lookup become compositional.

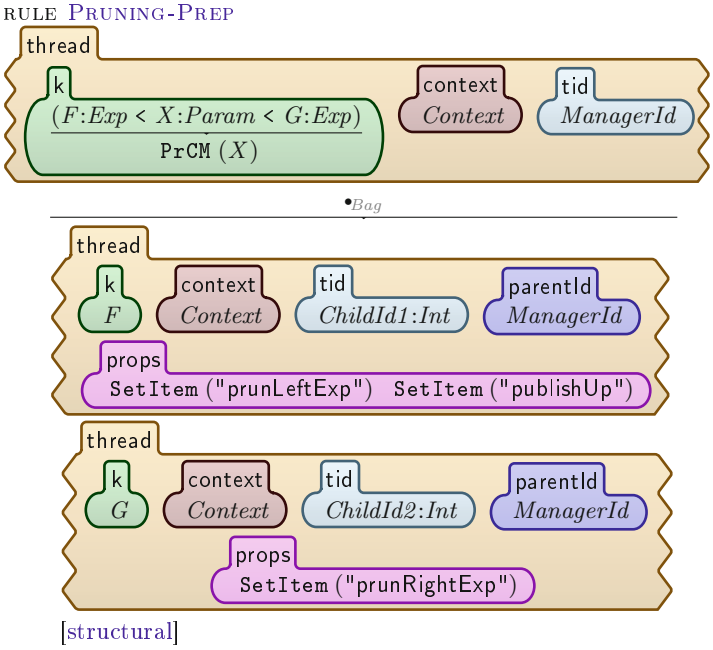

Fig. 10. First $\mathbb{K}$ rule of the Pruning Combinator

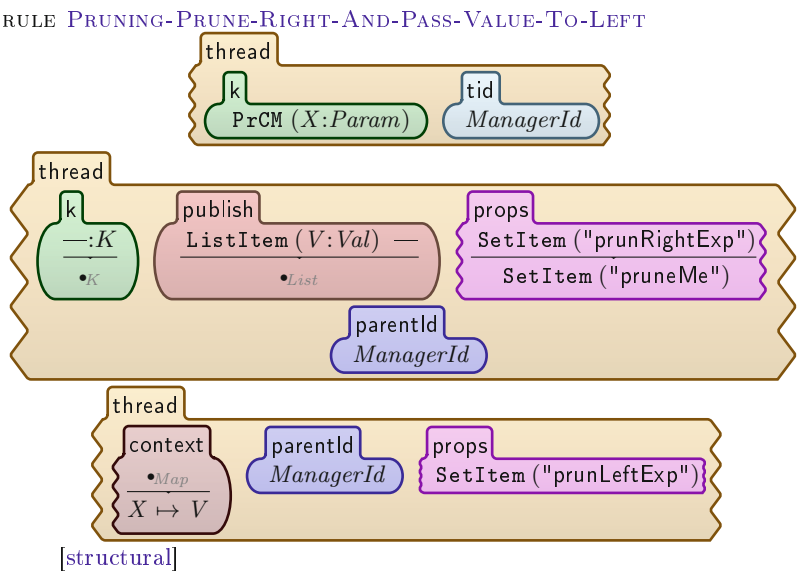

Fig. 11. Second $\mathbb{K}$ rule of the Pruning Combinator

A manager thread expecting values from a certain child simply sets a property in the child called `publishUp` in the cell `props`. As pointed out earlier, in our schematic drawings of the semantics, this property is denoted by a letter P in the lower left corner of the thread box. See Fig. 9. In retrospect, The child receiving the `publishUp` property might be itself a manager of a deeper composition, awaiting values to be published up to it. This behavior creates a channel from the leaves of the thread tree up to the root, which will publish the output of the whole Orc program in the cell `gPublish`. Threads which are given the `publishUp` property are:

- All children of a *Parallel Composition Manager*.
- All right-side instances of a *Sequential Composition Manager*
- The left-side thread of a *Pruning Composition Manager*
- The child of an *Otherwise Composition Manager*.

Such a channel is also evident when variable requests are propagated up the tree, since every thread is allowed to access the context map of any of its ancestors. A variable request, carrying the requester thread's ID, is propagated recursively up the tree, through a specialized cell `varReqs`, until it is resolved or reaches the root in which case it resets.

It is important to note that no manager is allowed to share the context of any of its children with the others, nor is it allowed to access it. Otherwise, some values could be accidentally overwritten if copied from one scope to another.

Synchronization and Time. The semantics of our (discrete) timing model follows the standard semantics of time in rewrite theories implemented in Real-Time Maude [23], in which time is modeled by the set of natural numbers captured by a `clock` cell in the configuration, and the effects of time lapse are modeled by a δ function.

Effectively, the δ function is what advances time in the environment. It is applied to the whole environment, and so it will be applied on all threads, and on the environment's clock to increment it. It will not have an effect on computations of internal sites, but only on timer sites and external sites that are yet to respond. One such site is $\mathit{Rtimer}(t)$, which publishes a signal after t time units. The δ function's effect can be directly seen on *Rtimer* in the following rule:

$\delta(\mathit{Rtimer}(t)) \Rightarrow \mathit{Rtimer}(t-1)$, where $t > 0$.

Therefore, the semantics of the *Rtimer* site, and any timed site, is only realizable through the δ function. When δ successfully runs on the whole environment, it is said to have completed one tick.

4 Formal Analysis of Orc Orchestrations

In this section, we present an example showing the formal analysis that can be done on Orc programs using the $\mathbb{K}$ tool. We defined external Orc sites to simulate a robot moving around a room with obstacles. A layout of the room we will be working with is shown in Fig. 12. We could of course work with a

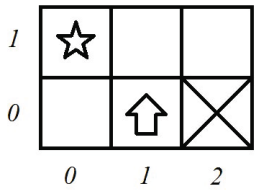

Fig. 12. Initial configuration of the robot environment

more complex environment, but the purpose here is a simple demonstration and a proof of the concept. We first simulate the movement of the robot, and then show an example of formal verification.

Robot Sites' Semantics. Before running any example, we explain our semantics of these robot sites. *MoveFwd* will cause the robot to move a distance of one block in its direction. *turnRight* and *turnLeft* will rotate the robot, while standing on the same block, 90 degrees clockwise and counterclockwise respectively. We also made each of these sites takes a certain amount of time to respond. *MoveFwd* takes three time units while each of *turnRight* and *turnLeft* take one time unit. Hitting an obstacle while trying to move forward will still consume three time units but will turn on a flag called *isBumperHit* which will reset on the next action.

4.1 Simulation

The robot starts at (1,0) facing north. Suppose that we want to move it towards the star at (0,1). The following Orc program will do just that:

MoveFwd() ≫ *TurnLeft*() ≫ *MoveFwd*()

Running `krun` on the expression outputs the final configuration as shown in Fig. 13. Some parts were omitted for space convenience. However, the important parts are the position, direction and the *isBumperHit* flag. We can see that they ended up as expected: the robot is at (1,0) facing west, and the bumper is not hit. Notice also that the clock is at seven time units, the time it takes for two *MoveFwd*'s and one *turnLeft*.

Writing the same program again but this time adding another *MoveFwd* to the end of the sequence makes the expression:

MoveFwd() ≫ *TurnLeft*() ≫ *MoveFwd*() ≫ *MoveFwd*()

Running this will cause the robot to hit the wall. That will turn on the *isBumperHit* flag as in Fig. 13. This time, the clock is at 10 time units, three units more consumed by the additional *MoveFwd*.

4.2 Verification

Here, we show a simplistic example that demonstrates the formal verification capabilities of $\mathbb{K}$. First we introduce an element of nondeterminism. Consider

```
<gVars>
  "BotVars" |->
    "direction" |-> (-1,0)
    "position" |-> (0,1)
    "is_bumper_hit" |-> false
  "clock" |-> 7
</gVars>
```

```
<gVars>
  "BotVars" |->
    "direction" |-> (-1,0)
    "position" |-> (0,1)
    "is_bumper_hit" |-> true
  "clock" |-> 10
</gVars>
```

Fig. 13. selected output of running simulations: example 1 (left), example 2 (right)

the Orc expression *RandomMove*() that is defined as:

MoveFwd() | *TurnLeft*() ≫ *MoveFwd*() | *TurnRight*() ≫ *MoveFwd*()

Executing this expression, the robot should nondeterministically choose between one of the paths separated by the parallel operator. Suppose we need to know whether this program will cause the robot to hit an obstacle or not. Running the program with `krun --search --pattern` and specifying *isBumperHit* → *true* as the pattern will show all configurations where the robot hits. The full command looks like this:

```
krun bot.orc --search --pattern "<gVars>... \"BotVars\" |->
  (M:Map \"is_bumper_hit\" |-> B) </gVars> when B ==K true"
```

The output of that command shows only one solution; it shows a configuration where the position is $(1, 0)$, the initial position, and the direction is east. Obviously, the robot reached there by picking the third choice, *TurnRight*() ≫ *MoveFwd*().

Now consider making two random moves in sequence: *RandomMove*() ≫ *RandomMove*(). Checking for all possible configurations where the robot hits reveals five solutions while checking for when the robot reaches the star at (0,1) shows two solutions. Searching in more complex environments with more complex expressions reveals many more solutions.

We demonstrated the potential of exploiting $\mathbb{K}$'s state search capabilities for purposes of formal verification. Other methods that $\mathbb{K}$ provides such as Maude's LTL model checker and Maude's proof environment are sure to deliver more in-depth verification.

5 Conclusion and Future Developments

In this paper, we have presented a first attempt at devising a formal executable semantics for Orc in the $\mathbb{K}$ framework and how it may be used for verifying Orc programs. The semantics is distinguished from other operational semantics by the fact that it is not directly based on Orc's original interleaving SOS semantics. The semantics takes advantage of concurrent rewriting facilitated by the underlying $\mathbb{K}$ formalism to capture its concurrent semantics and makes use of $\mathbb{K}$'s innovative notation to document the meaning of its various combinators.

Due to subtleties related to timing and transition priorities, faithfully capturing the Orc semantics is a nontrivial challenge for any semantic framework.

We plan to continue extending and refining the semantics so that all such subtleties are appropriately handled. Furthermore, executability of the semantics does not just mean the ability to interpret Orc programs using the semantics specification; it also means that dynamic formal verification, such as model checking, of Orc programs can be performed, which is something that we plan to demonstrate using the $\mathbb{K}$ tool with its Maude model checker. Moreover, an investigation of how the resulting semantics relates to the existing rewriting logic semantics would be an interesting future direction.

Acknowledgments. We thank José Meseguer and Grigore Roşu for their very helpful discussions, suggestions and comments on the work presented here. We also thank the anonymous reviewers for their valuable comments. This work was partially supported by King Fahd University of Petroleum and Minerals through Grant JF121005.

References

1. AlTurki, M.: Rewriting-based Formal Modeling, Analysis and Implementation of Real-Time Distributed Services. PhD thesis, University of Illinois at Urbana-Champaign (August 2011), http://hdl.handle.net/2142/26231
2. AlTurki, M.A., Meseguer, J.: Executable rewriting logic semantics of Orc and formal analysis of Orc programs. Journal of Logical and Algebraic Methods in Programming (to appear, 2015)
3. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comput. Sci. 96(1), 217–248 (1992)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Cook, W.R., Patwardhan, S., Misra, J.: Workflow patterns in Orc. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 82–96. Springer, Heidelberg (2006)
6. Şerbănuţă, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Roşu, G.: The $\mathbb{K}$ primer (version 3.3). In: Hills, M. (ed.) Proceedings of the Second International Workshop on the $\mathbb{K}$ Framework and its Applications ($\mathbb{K}$ 2011), vol. 304, pp. 57–80. Elsevier (2014)
7. Şerbănuţă, T.F., Roşu, G.: A truly concurrent semantics for the $\mathbb{K}$ framework based on graph transformations. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 294–310. Springer, Heidelberg (2012)
8. Şerbănuţă, T.F., Roşu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Information and Computation 207(2), 305–340 (2009); Special issue on Structural Operational Semantics (SOS)
9. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, pp. 533–544. ACM, Philadelphia (2012)
10. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)

11. Farzan, A., Meseguer, J., Roşu, G.: Formal JVM code analysis in JavaFAN. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 132–147. Springer, Heidelberg (2004)
12. Kitchin, D., Powell, E., Misra, J.: Simulation using orchestration. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 2–15. Springer, Heidelberg (2008)
13. Kitchin, D., Quark, A., Misra, J.: Quicksort: Combining concurrency, recursion, and mutable data structures. In: Roscoe, A.W., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare, History of Computing, pp. 229–254. Springer, London (2010)
14. Lazar, D., Arusoaie, A., Şerbănuţă, T.F., Ellison, C., Mereuta, R., Lucanu, D., Roşu, G.: Executing formal semantics with the $\mathbb{K}$ tool. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 267–271. Springer, Heidelberg (2012)
15. Lucanu, D., Şerbănuţă, T.F., Roşu, G.: $\mathbb{K}$ framework distilled. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 31–53. Springer, Heidelberg (2012)
16. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
17. Meseguer, J., Roşu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 1–44. Springer, Heidelberg (2004)
18. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theoretical Computer Science 373(3), 213–237 (2007)
19. Meseguer, J., Roşu, G.: The rewriting logic semantics project: A progress report. In: Owe, O., Steffen, M., Telle, J.A. (eds.) FCT 2011. LNCS, vol. 6914, pp. 1–37. Springer, Heidelberg (2011)
20. Misra, J.: Computation orchestration: A basis for wide-area computing. In: Broy, M. (ed.) Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems. NATO ASI Series, Marktoberdorf, Germany (2004)
21. Misra, J.: Structured concurrent programming. Manuscript, University of Texas at Austin (December 2014), http://www.cs.utexas.edu/users/misra/temporaryFiles.dir/Orc.pdf
22. Misra, J., Cook, W.R.: Computation orchestration. Software and Systems Modeling 6(1), 83–110 (2007)
23. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
24. Roşu, G., Şerbănuţă, T.F.: An overview of the $\mathbb{K}$ semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010); Membrane computing and programming
25. Roşu, G., Şerbănuţă, T.F.: $\mathbb{K}$ overview and SIMPLE case study. In: Hills, M. (ed.) Proceedings of the Second International Workshop on the $\mathbb{K}$ Framework and its Applications ($\mathbb{K}$ 2011). Electronic Notes in Theoretical Computer Science, vol. 304, pp. 3–56. Elsevier (2014)
26. Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: A timed semantics of Orc. Theoretical Computer Science 402(2-3), 234–248 (2008); Trustworthy Global Computing