A FORMAL EXECUTABLE SEMANTICS OF ORC USING THE $\mathbb{K}$ FRAMEWORK

BY

OMAR ALZUHAIBI

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

FEBRUARY 22, 2016

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **OMAR ALZUHAIBI** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Deanship of Graduate Studies in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

[illegible]

Dr. Musab A. Alturki (Advisor)

[illegible]

Dr. Moataz A. Ahmed
(Member)

[illegible]

Dr. Mohammad R. Alshayeb
(Member)

[illegible]

Dr. Abdulaziz M. Alkhoraidly
Department Chairman

[illegible]

Dr. Salam A. Zummo
Dean of Graduate Studies

عمادة الدراسات العليا
جامعة الملك فهد للبترول والمعادن
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DEANSHIP OF GRADUATE STUDIES

2/3/16

Date

To all that have been pushing

Acknowledgment

I have many more to acknowledge than mentioned here; but these are the most obvious, the most directly invloved, and the most deserving:

A great advisor that has fulfilled and surpassed his role and whose meticulousness and appreciation of quality are very encouraging; a true mentor given the right circumstances... and the right apprentice.

An Experienced committee with insightful thoughts—well, mostly about tables; so much so that I barely managed to resist the urge to make this very page into a table!

The $\mathbb{K}$ people, Grigore Roșu, Dwight Guth, Traian Florin Șerbănuță, Andrei Ștefănescu, Radu Mereuță, Cosmin Rădoi and others for their great help and support, and most of all, for the outstanding effort put in developing, honing and perfecting the $\mathbb{K}$ framework, tool, its tutorials and supporting projects. Did I mention Dwight Guth?

Contents

List of Figures

List of Tables

List of $\mathbb{K}$ Rules

Abstract

- **Name:** Omar Alzuhaibi
- **Thesis Title:** A Formal Executable Semantics of Orc using the $\mathbb{K}$ Framework
- **Major:** Computer Science
- **Degree Date:** January 2016

Orc is a programming model for expressing orchestrated distributed and concurrent computations. It abstracts computations through calling *sites* and expresses concurrency through four concise constructors. Our goal is twofold: (1) to devise a formal semantics of Orc that elegantly captures its intended semantics and (2) to formally verify the correctness of programs written in Orc. In order to achieve that, we wrote a complete formal semantics of Orc using the $\mathbb{K}$ framework, which in turn enables the execution and verification of Orc programs. This thesis presents in detail how the $\mathbb{K}$ framework's various facilities were utilized to arrive at a clean, minimal, precise and elegant semantic specification; informally compares our specification to previously developed operational semantics of Orc; and demonstrates how all of that enables the execution and verification of Orc programs through a case study and a few distinctive examples.

Arabic Abstract

ملخّص الرسالة

- الاسم: عمر زهير الذهيبي
- عنوان الرسالة: وصف رياضيّ للغة أُرخ بمحيط $\mathbb{K}$
- التخصّص: علوم حاسب
- تاريخ التخرّج: يناير ٢٠١٦

أُرخ لغة تعبير رياضية تمكّن من وصف عمليات حوسبة متوازية (كما في الحوسبة متعددة المهام)، و هي اختصار كلمة أرخصطرة (أُوركَسترا)، سُمّيت بذلك تنويها عن عمل قائد الأرخصطرة في تنظيم و توزيع المهام على عازفيه و التنسيق بينهم. تخفي لغة التعبير هذه تفاصيل عمليات الحوسبة المعروفة في لغات البرمجة خلف ستار أنيق لا يحتوي إلا على أدوات بسيطة هي القدرة على نداء الدوالّ و الجمع بينها بأربع منسقات أرخصطرية بسيطة و جامعة. هدفنا من هذا البحث تمكين التأكّد الآليّ من صحة البرامج المكتوبة بلغة أُرخ، فوجب لذلك كتابة وصف رياضيّ (دلاليّ) كامل للغة أُرخ، و لعمل ذلك اخترنا محيط عمل يسمى $\mathbb{K}$ يمكّن من تعريف لغات ثم تشغيل برامج بتلك اللغات، و يمكّن كذلك من تأكّدٍ من صحتها. يعرض هذا البحث بدقّة كيف استخدمنا مُختلف مميزات $\mathbb{K}$ لنصل إلى وصف رياضيّ بسيط و أنيق للغة أُرخ، و كيف يُقارَن وصفُنا بما سبقه، و كيف يمكّننا هذا من تشغيل برامج بلغة أُرخ و التأكدِ من صحّتها من خلال عدّة أمثلة مشروحة.

Chapter 1

Introduction

The past few decades have witnessed a tremendous growth in cloud-based web applications on one hand, and in parallel algorithms on a totally different hand. These two directions of growth might not be correlated but they are definitely converging towards our vision of future internet use, i.e., cloud services using parallel algorithms to utilize other cloud services.

Nowadays, many computer applications face a difficult problem; that is, how to obtain sufficiently safe operation. Critical systems such as medical instrumentation, aircraft flight control and nuclear reactor safety have extreme safety requirements. Traditionally, software testing methods such as fault tree analysis (FTA) were used for safety analysis. However, such methods were subjective and dependent on the software engineer; and even at their best, they could not ensure the extreme safety requirements that today's complex safety-critical systems demand [YJC09]. Here lies the indispensability of formal program verification. It provides a very rigorous demonstration that the program satisfies the given specification in every execution, the specification itself being an explicit description of the intentions and requirements of the program.

In the early days of formal verification, it used to be done manually exactly like mathematical proofs. But with the (1) prevalence of concurrent and distributed systems and with (2) increasing demand for safe and reliable operation in different practical fields like the health sector, big machines and space missions, the need for formal verification is bigger than it has ever been. Add to that (3) the increasing complexity of systems that nowadays most are intrinsically concurrent. All these factors increase the demand for automatic, instead of manual, verification. And this is one side of this thesis' work; it is an effort to provide a method to verify these systems' correctness automatically.

Another side to this work is revealed by the general lack of formal semantics for programming languages, models and paradigms. Even though many formalisms have been proposed to describe programming language semantics, most programming languages, even the most widely used ones, have neither been formally defined nor analyzed until very recently. This has created a gap between language designers, language implementers, compiler developers, and programmers. This is why until now we find very subtle differences in the interpretation of a C program for example from one compiler to the other. On our end, by providing a formal semantics for an elegant programming model such as Orc, we do our part in mending that gap.

The third side to this work is that it provides an executable semantics; and that is the part that allows formal verification to be automated. An executable semantics means that we can use specialized tools to simulate programs and run different verification techniques on them that are carried out automatically.

To sum up, this work:

- addresses the rising complexity of distributed systems and the need for safe operation
- by providing a formal semantics for such systems
- that is also an executable semantics
- which enables automatic formal verification on these systems.

The notion of composing web applications out of individual web services called and managed concurrently is called *service composition*. Out of many service composition systems, we focus on one that is formal, abstract, and expressive, namely *Orc* [Mis04, MC07, Mis14]. Orc is named after the theory of orchestration of services being a way of composing them into a well-synchronized web application just like an orchestra composes a piece of music. Orc is capable of describing concurrent systems in a minimal way. This abstractness helps simplify the process of formal verification by shifting the focus from all computational details into just the concurrency computations that we wish to analyze. So, this work tries to enable automatic verification for systems described in Orc.

1.1 Problem Description

Our concern is with the correctness of concurrent systems, service orchestrations in particular described through Orc. Our goal is to enable formal analysis of service orchestrations. But that itself requires a formal semantics able to capture every detail in an orchestration as abstractly and minimally as possible. We need a semantics that:

- reflects the simplicity and expressiveness of Orc
- is executable
- can be used for automatic formal verification

- is defined with modularity

All those factors will help us arrive at a minimal definition of Orc that faithfully captures its semantics while also being executable and on top of that easy to validate manually.

1.2 Approach

Because our goal is to verify properties related to correctness formally, we would always need a formal description of the system under question. Therefore, we chose a formal description language that is specifically designed to describe distributed and concurrent systems. We chose *Orc* [Mis04, MC07, Mis14], a programming model that can elegantly express such systems through what is called *service orchestration*. Orc uses only a few combinators to express concurrency, while abstracting away all other computations through services. This abstract precise formalism gives us concurrent systems written as Orc programs which we can formally verify certain properties about.

To do the verification automatically, we need not only a formal semantics for Orc, but an executable formal semantics. We also need it written in a way abstract enough to directly relate to the informal semantics of Orc, hide technical details and minimize human error; yet expressive enough to capture all of Orc's features. Taking all that into consideration, we use a framework-and-tool called 𝕂 [Rc10, LcR12].

We use 𝕂 to, first, write a formal semantics of Orc's service orchestration calculus; and second, execute and formally automatically verify sample Orc programs. Our 𝕂 semantics capture Orc's concurrent computations through an intuitive definition with which 𝕂 can understand Orc programs. Applying that to Orc extends its power beyond expressiveness towards meeting extreme safety requirements.

1.3 Contribution

This work contributes a formal executable semantics written in 𝕂. Utilizing 𝕂, it presents a definition that is much more elegant than any that have been developed before it. It allows the execution of Orc programs and enables formal verification on them. Moreover, The new semantics carries the promise of true concurrency from 𝕂, thus by enabling the simulation and verification of certain Orc programs in unprecedented accuracy. It allows for very expressive formal analysis of concurrent computations. This was not provided through other combinations of formalisms of concurrent computation and formal analysis.

On top of the 𝕂 semantics being simple enough to allow manual validation, this work also provides a simplistic test suite that aims to build confidence in the correctness of the 𝕂 definition itself.

The work also reviews the latest work related to Orc as well as other similar frameworks, verification done on these frameworks, and other semantics defined in $\mathbb{K}$.

1.4 Outline

The outline of this thesis is as follows. First, in Chapter 2, we give a comprehensive background covering all the basic concepts and terminology required to comprehend this work. Then, in Chapter 3, we walk through the design of our formal semantics of Orc and how we formed modular rewrite-based rules. And in Chapter 4, we show how those rules took form as a full-fledged specification written and structured in the $\mathbb{K}$ tool, and we overview the validity tests. In Chapter 5, we discuss the results of executing sample Orc programs through our $\mathbb{K}$ semantics and show examples of formal analysis. Chapter 6 reviews previous work related to service orchestration and program verification, and compares this work to it. Finally, we conclude with Chapter 7 with a recap of the thesis and a discussion of limitations of this work and how it can further be developed.

Chapter 2

Background

In this chapter, we cover all the basic information needed to understand the work explained in Chapter 4. We first explain the concept of Service Orchestration in general followed by an in-depth explanation of Orc, its syntax and its semantics. In Section 2.2, we introduce the $\mathbb{K}$ framework and its tool and we compare them to other semantic-definition frameworks, formalisms and tools.

2.1 Service Orchestration

This section gives a background on service orchestration by explaining the Orc calculus, its syntax and semantics with examples.

To understand this term, we simply define the terms *service* and *orchestration*. In general, services are units of discrete isolated tasks. In the context of service orchestration, they usually refer to software tasks. The term web-services is often used in cloud computing referring to software tasks that are available over the internet.

The term *Orchestration* is named after the process that a musical orchestra conductor undertakes when managing and synchronizing the individual members. Computation orchestration works the same way. Individual services are managed and coordinated by a central orchestrator which is also a service. Likewise, a controlled service can itself be an orchestrator of a sub-group of services.

That is, in general, what Service Orchestration means. A more specific definition would require bringing an implementation of the concept to the discussion; and that is what we do in this section by viewing Service Orchestration through the perspective of Orc[1].

[1]Throughout this thesis, the term Orc will refer to Orc the calculus [MC07], not its child

2.1.1 Orc

Orc [Mis04, MC07] is a theory for orchestration of services that provides an elegant expressive programming model for timed, concurrent computations. Orc has special terminology that relates to general concepts of service orchestration. Let us go through these terms in a brief overview of Orc.

Orc, short for orchestration, is a process calculus for concurrent computations. The building block of an Orc expression is the *site* call. A site in Orc represents a service. Site calls are joined together using one or more of Orc's concurrency combinators to make Orc expressions. An expression that has completed it execution is said to have *halted*. Based on [MC07], we give the informal semantics of Orc in a somewhat structured manner centered around its main concepts:

- Values: A value, on its own, is a valid Orc expression. When the expression is executed, it publishes that value, and then halts. An Orc value could be:
  - The **signal** value which indicates the termination of some expression evaluation, but carries no information.
  - **stop**, which indicates termination of a site call without publishing.
  - A number, such as `0`, `2.718`, or `-14`
  - A boolean, `true` or `false`.
  - a character string, such as `"I am not an orc"`.
- Sites: When a site is called, it may return, *publish*, at most one value. A called site may not always respond, or may respond after a unplanned delay. A site call that never responds is called *silent*. In that regard, sites are either external or internal.
  - External sites are those that may have a delayed response.
  - Internal sites need zero time to respond, but they can also never respond. Internal sites facilitate Orc with basic functions. Here are some of Orc's internal sites.
    - **0**, or **zero**, the internal silent site which never responds.
    - $let(x)$, publishes the value x.
    - $if(b)$, publishes a signal if b is true and remains silent otherwise.
    - *Clock*, publishes the current time value.
    - $Rtimer(t)$, publishes a signal after t time units.
- Combinators:
  - The *parallel* or the *symmetric parallel* combinator ($|$), written as: $f \mid g$. f and g are executed concurrently independently from each other. The expression $f \mid g$ publishes all values published by f and g in timed order. The expression halts when both f and g halt.

the full-fledged programming language [KQCM09].

  - The *sequential* or *spawning* combinator, written as: f >x> g. f is executed first. For each output x, published by f, an instance $g(x)$ is spawned to be executed in parallel with f >x> g. The sequential combinator may be written as $f \gg g$, when x is not bound in g. The execution halts when f and all instances of g halt.
  - The *asymmetric parallel* or *pruning* combinator, written as: f <x< g. f and g start execution concurrently. The variable x may or may not be bound in f but f will start executing nonetheless.

    As soon as g publishes a value v, g halts immediately, and x is bound to v in f. If g halts without publishing a value, all occurrences of x in f are replaced by **stop**. A site call with an argument **stop** halts immediately.

    Since f begins executing before x is bound to a value, execution of f may encounter a site call with an argument x, or even x as its own expression. Execution of such expressions blocks; nothing happens until x is bound to a value or **stop**; and then the expression resumes execution. A blocked expression is not considered halted, since it might become unblocked in the future.

    When f has halted, and g has either published or halted, the expression f <x< g halts. The pruning combinator may be written as $f \ll g$, when x is not bound in f.
  - The *otherwise* combinator, written as: f **;** g. f is executed first. If f published a value then the expression f **;** g publishes that value and halts. However, if f halted without publishing, then the expression is replaced by g.

  The precedence order of these combinators from higher to lower is:

  - Sequential ($\gg$).
  - Parallel (|).
  - Pruning ($\ll$).
  - Otherwise (**;**).

- Expressions: Finally, an Orc expression is any which the syntax allows. Orc's abstract syntax is shown in Figure 2.1. From that definition, we can see that an expression can be either of:

  - A site call.
  - A parameter.
  - An expression call having an optional list of actual parameters as arguments.
  - A composition of two or more subexpression through the combinators.

$$
\begin{array}{rcl}
x, y & \in & \textit{Variable} \\
k & \in & \textit{Handle} \\
V & \in & \textit{Value}
\end{array}
$$

$$
\begin{array}{rclcl}
v & \in & \textit{Orc value} & ::= & V \\
w & \in & \textit{Response} & ::= & v \mid \textbf{stop} \\
p & \in & \textit{Parameter} & ::= & w \mid x
\end{array}
$$

$$
\begin{array}{rclcll}
f, g & \in & \textit{Expression} & ::= & p & \text{(Parameter)} \\
 & & & \mid & p(\bar{p}) & \text{(Site or Function Call)} \\
 & & & \mid & k? & \text{(Site Call In Progress)} \\
 & & & \mid & f > x > g & \text{(Sequential Combinator)} \\
 & & & \mid & f \mid g & \text{(Parallel Combinator)} \\
 & & & \mid & f < x < g & \text{(Pruning Combinator)} \\
 & & & \mid & f \; \textbf{;} \; g & \text{(Otherwise Combinator)}
\end{array}
$$

Figure 2.1: Abstract syntax of Orc expressions.

Time in Orc

On top of mentioned main features, Orc also accounts for time. The original semantics by Misra in [Mis04] did not model time. However, later in [MC07], timer sites like $Rtimer(t)$ were introduced, and in [WKCM08], the SOS specification was extended to include time.

Formally, the passage of time can be expressed in different ways, but informally it is very basic. And like all other computation details, time is also abstracted behind site calls. The way it is done is that certain sites, whether internal or external, are timed. A simple example is the aforementioned $Rtimer(t)$ site which takes t time units to respond. Another example would be a collection of sites that control a machine. Every site that, for example, requests motion from the machine or naturally imposes mechanical delay would take time; and so those sites would be timed. This can be seen in our simulation in Chapter 5 where we model such sites that control a virtual robot's movements.

Orc Examples

To illustrate the informal meaning of the combinators, we list some examples here. More demonstrative examples can be found in [MC07, KPM08, Mis14]. Larger programs can be found in [CPM06] where Orc was used to describe workflows over internal and remote services; and in [KQM10], an excellent demonstration of the power of Orc through an implementation of the quicksort algorithm.

Example 2.1. Suppose we want to get the current prices of three stocks: *Microsoft*, *Google*, and *Apple*; and that there is a site that provides such service called *StockPrice*. The three pieces of information are independent from each other, and we want to receive them as soon as possible not preferring a certain order. Therefore, it would make sense to request all three prices in parallel, instructing each called site to send a value as soon as possible. The Orc expression would be:

$$StockPrice(Google) \mid StockPrice(Microsoft) \mid StockPrice(Apple)$$

and the output would be a time-ordered list of the three values, ordered by the time they were received.

Example 2.2. Now, suppose you own some of these stocks and you wish to know your unrealized gain/loss, i.e., how much you gain/lose if you sell now, which depends on two pieces of information: the current price of the stock, and how much you own of it. The first example takes care of the former. For the latter, assume that we simply have a site called $MyUnrealizedGainLoss(Stock, Price)$ that has access to your account and can calculate the unrealized gain/loss. The following Orc expression gives the answer:

$$\begin{aligned}&\mathit{StockPrice}(\mathit{Google}) \mid \mathit{StockPrice}(\mathit{Microsoft}) \mid \mathit{StockPrice}(\mathit{Apple})\\ &\quad >(\mathit{Stock}, \mathit{Price})>\\ &\qquad \mathit{MyUnrealizedGainLoss}(\mathit{Stock}, \mathit{Price})\end{aligned}$$

The site *StockPrice* returns a tuple containing two values: the stock and its price. Each response from *StockPrice* creates a new instance of *MyUnrealizedGainLoss* passing it the arguments $(\mathit{Stock}, \mathit{Price})$. These instances run independently in parallel and act exactly like the three subexpressions of the first example.

Example 2.3. Suppose we want to get the current gold price, and that we have three sites that provide this service: GoldSeek, GoldPrice, and Kitco. This is just like Example 2.1 where we don't prefer a certain order of response. The expression would be:

$$(\mathit{Gold\text{-}Live}() \mid \mathit{GoldPrice}() \mid \mathit{Kitco}())$$

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:

$$\mathit{USDtoEuro}(x)\ <x<\ (\mathit{Gold\text{-}Live}() \mid \mathit{GoldPrice}() \mid \mathit{Kitco}())$$

The pruning combinator tells its right-side operand, 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.

Example 2.4. Suppose we have a site called $FireAlarm$ that when called, will hang. It would only respond when a fire has been detected giving its location. That information is sent to the fire department who need 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.

$$\begin{aligned}&FireAlarm()\\ &\quad >\mathit{fireLocation}>\\ &\qquad \mathit{CalcNearestStation}(\mathit{fireLocation})\\ &\qquad\quad >\mathit{station}>\\ &\qquad\qquad \mathit{Dispatch}(\mathit{station}, \mathit{fireLocation})\end{aligned}$$

Example 2.5. The standard programming idiom of the two-branch conditional **if** b **then** f **else** g can be written in Orc as the expression:

$$\mathit{if}(b) \gg f \ \mid\ \mathit{if}(\neg b) \gg g$$

Given the behavior of the internal site *if*, only one of the expressions f and g is executed, depending on the truth value of b.

In the next few examples, we introduce the pivotal role of the site *Rtimer* which is the building block of Orc's timed computations.

Example 2.6. The Orc expression below specifies a timeout t on the call to a site M:

$$\mathit{let}(x) <x< (\, M() \mid \mathit{Rtimer}(t) \gg \mathit{let}(\mathit{signal})\,)$$

Upon executing the expression, both sites M and *Rtimer* are called. If M publishes a value w before t time units, then w is the value published by the expression. But if M publishes w in exactly t time unites, then either w or *signal* is published. Otherwise, *signal* is published.

Example 2.7. Another *Rtimer* example is the following Orc expression declaration, which defines an expression that recursively publishes a signal every t time units, indefinitely.

$$\mathit{Metronome}(t) \triangleq \mathit{let}(\mathit{signal}) \;\mid\; \mathit{Rtimer}(t) \gg \mathit{Metronome}(t)$$

The expression named *Metronome* can be used to repeatedly initiate an instance of a task every t time units. For example, the expression $\mathit{Metronome}(10) \gg \mathit{UpdateLocation}()$ calls on the task of updating the current location of a mobile user every ten time units.

2.2 Formal Specification and Verification

Formal specification is the modeling of systems mathematically through certain standard notations. Different approaches for how to capture semantics formally make different frameworks. Some of these frameworks can be used for automatic program verification. But what is program verification? It is a mechanism by which certain premises in the execution of a program are mathematically proven. In order to prove anything about a program, it has to be written in a formal mathematical language; in other words, it has to be formally specified. Once formally specified, a program can undergo mathematical and logical operations with the goal of proving certain properties about it; that process is called Formal Verification.

Formal verification can be done manually, but with larger and more complex programs, the need for automatic program verification becomes more imminent. Today, we have different tools that apply different verification techniques given of course a formal specification of the program of interest.

In this section, we review the most prominent semantic frameworks, and the formal specification tools built upon them, ending with the $\mathbb{K}$ framework which we explain in detail in light of the other frameworks. We start with frameworks that are based on Structural Operational Semantics (SOS) [Plo81]; then ones based on Rewriting Logic [Mes92]; and then we review a few more tools that have different approaches but are useful to view in contrast.

2.2.1 SOS-based Frameworks

Structural operational semantics (SOS) [Plo81] is a framework used to give operational semantics to programming and specification languages. SOS generates a labeled transition system, where the states are specific terms and the transitions between states are represented by transition rules. The concept of rules of SOS gave rise to many theories of formalization, each trying to improve and expand the general framework, like for instance Modular SOS which was introduced to deal with the non-modularity of SOS. However, with our focus on concurrency, the important point is that none of these theories gives a "true concurrency" semantics. They only allow "interleaving semantics" for concurrency. What this means is that no two rules can be applied on a certain term at the same time. There has to be an order with which they apply.

Semantics given by SOS-based frameworks cannot, in general, be executed even though there are specific cases of executable semantics [MR07]. That is why researchers were not motivated to build tools on them. Another point is such semantics cannot be used for automatic verification either [Ro5]. Yet, we include them in the review building up to $\mathbb{K}$ for two reasons: first, because they are very well-known; and second, because $\mathbb{K}$ uses their basic concepts which need to be explained.

Small-step SOS

Small-step semantics, also called transition (or reduction) semantics, is the first introduced variant of SOS, given in the same paper by Poltkin [Plo81]. A rule in Small-step SOS describes a single computational step. So it is easy to trace the computations and find errors. It gives interleaving semantics for concurrency.

Big-step SOS

Big-step SOS, introduced by Kahn in [Kah87] by the name "natural semantics", also called relational or evaluation semantics, is another well-known approach to operational semantics. To make sense out of the name, big-step rules are written in a way that describes evaluation in a single step that can only end at a terminal state, while a small-step rule describes only how to transition to the next state. Big-step is the closest of all operational semantics to denotational semantics [Rc10], in the sense that it maps the construct to a final value. Big-step rules are like schemes; variables have to be instantiated in order to be applied. This means that it makes the defined system difficult to trace and debug, and thus make it more difficult to execute.

Modularity in SOS

Neither of the two aforementioned SOS frameworks (small-step and big-step) is modular. Several attempts were made to write modular semantics using SOS, the first being Modular SOS (MSOS), introduced by Mosses [Mos04]. It modified SOS such that only the needed attributes are selected from a state, and that on every transition, the non-syntactic components of a state are moved into the labels of that transition. This would generate a labeled transition system. Whereas That is a syntactic approach to modifying SOS, a semantic one is made by Jaskelioff in [JGH11] who also reviews other interesting approaches.

2.2.2 The Chemical Abstract Machine (CHAM) [BB92]

CHAM views a state in a distributed system as a chemical *solution* with *molecules* floating. It understands concurrent transitions as *reactions* that can occur simultaneously in different parts of the solution. This is a unique paradigm for distributed systems, and it works for defining languages and systems with concurrency. However, it cannot handle complex features of languages such as threads and thread synchronization and control features.

2.2.3 Rewriting Logic [Mes92]

Rewriting logic was proposed by Meseguer [Mes92] as a logical formalism. Its design generalized both equational logic and term rewriting. In contrast with plain term rewriting, Rewriting Logic was made to be more suited and optimized for language semantics. In other words, it made it possible to naturally specify languages and systems.

Rewriting logic is a general semantic framework in which languages and systems can be naturally specified.

The generality and flexibility of rewriting logic make it suitable for specifying both deterministic computations (algebraically using equations) and non-deterministic computations (using rewrite rules) within a uniform model.

Here is where it falls short for our purposes. First, rewriting logic's power comes from it being a general-purpose computational framework. However, it is not focused towards defining language semantics, which makes defining language constructs, precedence, evaluation strategies and so on less intuitive. The second point is that it cannot satisfy our objective of true concurrency. It is true that one rule can be applied on two terms at the same time, because the framework deals with instances of rules, so that two or more instances can run at the same time. However, in such a case where two threads are trying to read from a shared store, the two rule instances would have to interleave the operation.

Maude [CDE^{+}07]

Maude is the software tool implementing rewriting logic. Maude is built in a way that makes defining systems as general as possible. It can be used to formally and modularly define almost any system. It can define formalisms; it was used to define MSOS [MB04]. And [cRM09] shows how it can define CHAM and Reduction Semantics with Evaluation Contexts. An example of generality is that equations and rules can be conditional and can have extra variables in the right-hand side of a rewrite, and they can have side conditions. Another example is that functions can take not just other functions as arguments, but entire modules as arguments. This shows the true generality and extensibility of Maude. A unique feature of Maude is its efficient built-in support for model checking. It supports reachability analysis, invariant verification and LTL model checking.

Maude has been applied in a wide range of applications. It has also been used to give formal semantics to, and provide formal analysis for, several real-time programming languages and software modeling languages [CDE$^+$07].

However, because Maude is so general, defining a very specific system almost from scratch is rather tedious. This is why, Maude being so extensible, a lot of extensions have been added to it; and this is exactly where $\mathbb{K}$ comes in. $\mathbb{K}$ is built on Maude and it utilizes its strengths while at the same time specializing in defining and analyzing programming languages.

2.2.4 The $\mathbb{K}$ Framework

$\mathbb{K}$ [Rc10, Rc14] 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 [Mes92], and to that of heating/cooling rules and reaction rules in CHAM [BB92].

Concepts Inspired by Other Frameworks

The $\mathbb{K}$ framework carries concepts in its design inspired by other semantic frameworks. Following is an overview of them that is summarized in Table 2.1.

Concepts used in 𝕂 from Small Step. The concept of rewriting used in 𝕂, in its essence, can be seen as small-step in the sense that a rewrite specifies a transition to the next configuration. However, a 𝕂 rewrite can combine multiple smaller rewrites, making it more general than a small-step rule.

Concepts used in 𝕂 from Big Step. Terminal states are either acceptable or error states. SOS does not define which are which, neither does Rewriting Logic. The way 𝕂 does it is through its syntax definition. Combined with its rules, a closure is achieved through one or many transitions. Such closure has the same concept as a big-step relation.

Concepts used in 𝕂 from MSOS. 𝕂 uses MSOS's approach to increase modularity through labeling information. It is apparent in the structure of a 𝕂 configuration as cell names.

Concepts used in 𝕂 from CHAM. Heating/Cooling rules: One very useful concept of CHAM that was adopted by 𝕂 is Heating/Cooling rules. These rules come in pairs: a heating rule and a cooling rule. Heating refers to taking apart the different components of a statement and then evaluating each separately. Cooling refers to bringing these components into the statement to evaluate it. This is very useful for specifying which computations should be brought to the top to be evaluated first. It adds a layer of control for rules being applied on a program's syntax tree. You can also subtly conceive how the concepts of big-step and small-step apply here.

Concepts used in 𝕂 from RL. 𝕂 is based mostly on Rewriting Logic. It uses rewrite rules. However, 𝕂 can capture a concurrency feature that Rewriting Logic cannot, that is concurrency with shared reads. And regarding concurrent term rewriting in general, where Rewriting Logic needs multiple interleaved rewrites, 𝕂 can capture the same in one concurrent rewrite.

Overview of 𝕂 Rules

To briefly introduce the notations used in 𝕂 rules, we present a 𝕂 rule used for variable lookup (Figure 2.2).

- 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.

Framework	Concepts
Small-step	Rewrites are more a general version of small-step.
Big-step	Transitive closure is a big-step relation.
MSOS	$\mathbb{K}$ has labeled information as cell names.
Reduction Semantics	Where it splits/plugs expressions into context, $\mathbb{K}$ flattens them into computations.
Rewriting Logic	1) Where RL splits sentences into equations and rules, $\mathbb{K}$ rules are split into structural and computational. 2) $\mathbb{K}$ rules are rewrites. 3) Like RL, $\mathbb{K}$ is based on context-insensitive term rewriting.
CHAM	1) CHAM's solutions are $\mathbb{K}$ configurations. 2) Heating/Cooling is how $\mathbb{K}$ moves computations to the top and evaluates them. 3) Heating/Cooling rules vs. Reaction rules in CHAM is similar to $\mathbb{K}$'s structural vs. computational rules.

Table 2.1: Summary table of all concepts used in $\mathbb{K}$ categorized by the frameworks that inspired them.

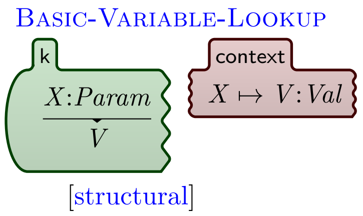


Figure 2.2: Variable lookup rule as defined in $\mathbb{K}$.

  - 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.

Main Features of $\mathbb{K}$

$\mathbb{K}$ 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.

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 [cR12]. For more details about the $\mathbb{K}$ framework and its features and semantics, the reader is referred to [Rc10, Rc14].

The $\mathbb{K}$ Tool

An implementation of the $\mathbb{K}$ framework is given by the $\mathbb{K}$ tool [cAL+14, LAc+12], which is based on Maude [CDE+07], 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 [2].

The $\mathbb{K}$ tool leverages the powerful and generic formal verification tools implemented in Maude by translating $\mathbb{K}$ specifications into rewrite theories, whose corresponding Maude modules are then executed and analyzed using the Maude rewrite engine. This is how it is built on top of Maude. As explained earlier, it makes it a lot easier and more intuitive to write many kinds of specifications like configurations, syntax rules, evaluation strategies, and heating/cooling rule pairs. At the same time, it makes use of Maude's strengths, such as its multi-set and context-insensitive rewriting.

Where Maude is very general that it can be used to define any system, $\mathbb{K}$ is very simple and is perfect for defining programming languages. Moreover, $\mathbb{K}$ has added a lot more features as explained earlier, not to mention it is much faster and lighter on resources when compared to Maude.

Advantages over Maude in defining configurations

The following example shows the simplicity of the $\mathbb{K}$ tool over Maude, particularly in configuration definition. Declaring a configuration in $\mathbb{K}$ defines several things at the same time, whereas in Maude, it would be necessary to define the following things separately:

- First, to define an algebraic signature for configurations in general.
- Second, to tell the engine how to initialize the configuration without any extra instructions.
- Third, to give a basis for specifying concrete rewrite rules.

The $\mathbb{K}$ tool's implementation of defining configurations uses XML-style cells (a nested cell structure). It allows for custom initialization of the configuration. It even allows for connecting any certain cell of the configuration to the standard I/O stream. Moreover, the $\mathbb{K}$ tool allows for defining purely syntactic, substitution-based semantics. For example, say we want to define the semantics of the pruning operator of Orc, which is a substitution-based semantics. Then, we would only need a one-cell configuration initialized with the main program.

Overall, we showed that defining a language in $\mathbb{K}$ is much simpler than, yet as expressive as, in Maude.

[2] 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.

Advantages over SOS in modularity

$\mathbb{K}$ applies a mechanism called *Configuration Abstraction*. Basically, it allows us to add new features to the configuration without the need to revisit all the rules to change the structure of the configuration. This makes $\mathbb{K}$ modular.

Final words about $\mathbb{K}$

In short, $\mathbb{K}$ is the best choice for the following four reasons:

- $\mathbb{K}$'s semantic specification is simple, expressive and concise.
- The infrastructure for defining semantics is predefined in $\mathbb{K}$.
- Simpler rules; means simpler and more efficient matching.
- True concurrency.

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 [cAL+14].

2.2.5 Formal Verification techniques

Here we briefly talk about a specific formal verification method, i.e., Model Checking, and the temporal logics used in model checking. In fact, this subsection should be titled "Model Checking Logics". Given a state model and certain formal properties, checking that these properties are met in all states generated by the model is the technique called Model Checking. In our case, an Orc program would be translated to a state model by $\mathbb{K}$ and then checked. The checked formal properties are written in a certain logic and they represent some behavior within the model. Different logics differ in their expressiveness and the ability to apply automatically. These two properties are reversely proportional. The logics which are mostly used and have tools based on them are LTL, CTL, CTL* and μ-calculus, ordered from the least to the most expressive.

Maude [CDE+07], $\mathbb{K}$ and SPIN [spi] use LTL for formal verification; NuMSV [num] uses CTL; ARC, the checker of the AltaRica project [alt], uses CTL*; and mCRL2 [mcr] uses the most expressive temporal logic μ-calculus.

A completely different tool that uses Higher Order Logic called Isabelle/HOL [NPW02, isa] is worth mentioning here. Isabelle/HOL is a theorem prover used as a specification and verification system, while Isabelle (just Isabelle) is a generic system for implementing logical formalisms.

A Note Comparing Formal Verification Tools. It is important to add that these different tools have different approaches and different goals which make them incomparable. Maude, along with Maude-based $\mathbb{K}$, is a generic framework which can define systems and even other frameworks as mentioned earlier, whereas mCRL2 is a different formalism with a different objective and a different underlying formal system. For example mCRL2 and NuMSV are mostly used for the verification of industrial designs, so the tools used depend on the system being verified. In some instances the logic is modified to bring it up to the level of the system to be defined. For example, Isabelle/HOL originally used predicate logic which has relations, and then later, higher order functions were added that take other functions as arguments. This combination turned out to be quite useful because it made it possible to define complex structures. In conclusion, these tools cannot be directly comparedto each other because they have completely different approaches and thus different uses.

A Note on Comparing Different Temporal Logics. The same can be said about comparing different temporal logics. LTL, the logic used by $\mathbb{K}$'s model checker, is for specifying properties, while the more expressive logics such as CTL* and μ-calculus are for specifying whole systems. The suitable logic for our application not only depends on that, but even on the property to be defined. In LTL, for example, although temporal properties (that show progress) can be specified, they can't be defined on specific execution paths. However, on the other hand, LTL has a useful property not found in comparable logics, that is the efficiency of deciding the properties. The reason LTL is widely used is its extreme efficiency. Predicate logic is incomparable in that regard because it's intractable.

Chapter 3

Designing a Formal Semantics of Orc

This chapter presents the core of our formal semantics of Orc. We will go through the main semantic features of Orc explained in the background and show how we captured those features in a concise set of semantic rules. Each section of this chapter explains a single independent part of the semantics. This shows the large-scale modularity with which this design was conceived. This later reflects on the implementation of the semantics in the $\mathbb{K}$ tool detailed in Chapter 4. The small-scale modularity is shown in individual rules and in the assumed constructs used in those rules. It is shown in how each rule is specialized, i.e, it has a limited domain on which it acts so that rules don't overlap.

We illustrate these rules as transformations in schematic diagrams shown in Figure 3.1 to Figure 3.5. These schematics use the following notations:

- Each box represents a thread.
- 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 our formal rules, 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 is a flag meaning that the thread is allowed to move its published values to its parent thread.

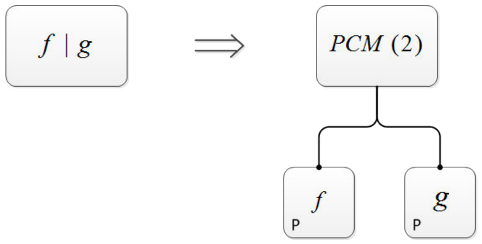


Figure 3.1: Transformation schematic of the parallel combinator.

- Variable mappings such as $x \to v$ mapping a variable x to a value v are displayed at the bottom of the box.
- Finally and most significantly, the symbol $\Rightarrow$ denotes a rewrite, the transformation from one state to another.

This chapter focuses on the parts of Orc that needed the most careful design. We divide up these parts into sections: first the four combinators, then publishing, then time.

3.1 Combinators

Orc is based on the execution of expressions, and simple expression can be made into more complex ones using one or more of its combinators. So, let us start with the design of combinators' semantic rules. Notice that we do not care about what the expressions being combined reduce to. We want our rules to be abstract enough to handle all expressions regardless of their complexity, i.e., whether they are simple like values and site calls, or are more complex combinations.

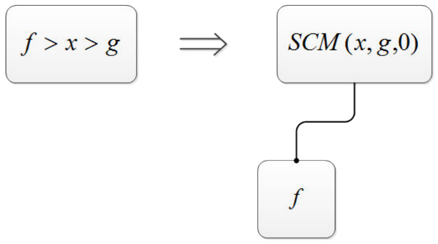
Orc has four combinators, which combine subexpressions according to four distinct patterns of concurrent execution, *parallel*, *sequential*, *pruning* and *otherwise*.

3.1.1 Parallel Combinator

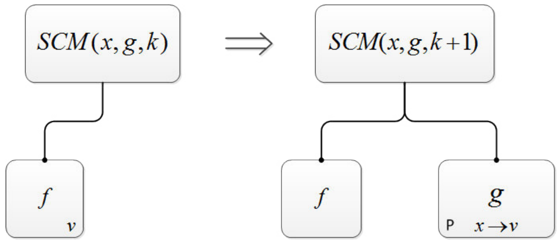
Given an expression $f \mid g$ as shown in Figure 3.1, the rule creates a manager thread carrying a meta-function called $PCM(x)$, short for Parallel Composition Manager, where x is the count of sub-threads it is managing. Child threads are created as well for each of the expressions f, and g. This of 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.

3.1.2 Sequential Combinator

The first rule of the sequential combinator, shown in Figure 3.2a, creates a manager called *SCM*, short for Sequential Composition Manager; and it creates

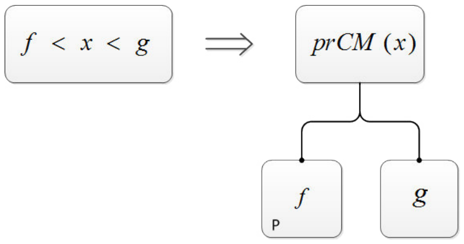


(a) Prep.

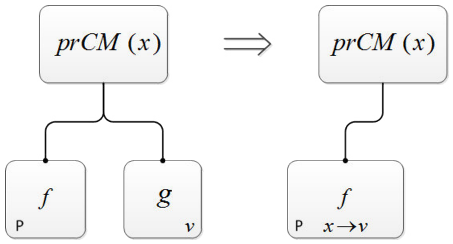


(b) Spawn.

Figure 3.2: Transformation schematics of the sequential combinator



(a) Prep.



(b) Prune.

Figure 3.3: Transformation schematics of the pruning combinator.

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

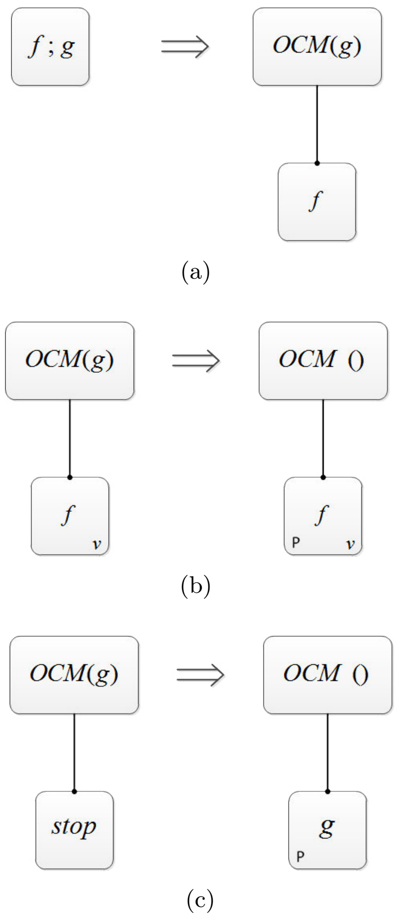


Figure 3.4: Transformation schematics of the otherwise combinator.

expression; and k, a count of active instances of g that is initially 0.

Every time f publishes a value, the second rule, shown in Figure 3.2b, creates an instance of g with its x parameter bound 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 of [Mis04].

3.1.3 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, it would wait for it. So, the first rule of the pruning combinator, shown in Figure 3.3a, creates a manager *PrCM* (short for Pruning Composition Manager), a thread executing

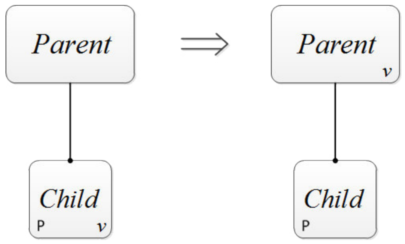


Figure 3.5: Transformation schematic of publishing values.

f, and another thread executing g.

The second rule, Figure 3.3b, is responsible for passing the published value from g to f and terminating (pruning) g.

3.1.4 Otherwise Combinator

The otherwise combinator is first processed as in Figure 3.4a by creating a manager called *OthCM* (short for Otherwise Composition Manager) and a child thread to execute f. Then, Figure 3.4b tells that if f publishes its first value, g is discarded and f may continue to execute and is given permission to publish. However, in Figure 3.4c, if f halts without publishing anything, then it is discarded and replaced by g. As mentioned in Section 2.1.1, *stop* is a special value that indicates that an expression has halted.

3.2 From Abstract Managers to Publishing

3.2.1 Abstraction of Manager Threads

In the previous section, we showed the design of the semantic rules of the combinators through schematics. Note the common structure of thread hierarchy in all of those rules. We made it such that any parent thread is a manager of one of four types, one type for each combinator. We also made it such that each parent is responsible for creating and deleting child threads, and for managing any values they publish. From the schematics, you can see that some arrangements allow for a child to pass a published value up to its parent marked by giving the child the publishing flag P. Child threads that are given the P flag are:

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

This scheme is particularly useful in making rules as abstract as possible for publishing.

3.2.2 Publishing

Since every manager controls when it receives publishes from its children, and since all such cases are abstracted through the publishing flag P, the publishing rule becomes straightforward as shown in Figure 3.5. It says that if a child thread has the P flag and has published a value, then it should send that value to its parent. This can happen recursively until the value reaches the topmost thread in the hierarchy, i.e., the root thread which represents the whole Orc program. Whatever is published by the root thread is considered published by the program.

About Variable Lookup. This scheme also helps us in making simple rules for variable lookup and helps us in defining scopes. However, because that issue is technical and depends on the implementation environment, we deferred its explanation to Chapter 4.

3.3 Time

As mentioned in subsubsection 2.1.1, Orc accounts for the passage of time through timer sites. This means that we could write a specification that is simply limited to handling only untimed sites as a partial definition of Orc; and indeed we have. After that, we extended the specification by defining timed sites and devising rules to model time. The untimed semantics of Orc does not depend on the time extension. This also shows modularity in the definition.

Another point is that in our definition, time is logical (discrete), not dense.

The semantics of our discrete timing model follows the standard semantics of time in rewrite theories implemented in Real-Time Maude [ÖM07], where (1) time is modeled by the set of natural numbers held in a certain environment variable, say *Clock*, and (2) the effects of time elapsing are modeled by a function called δ.

The way it applies to the definition we described in this chapter thus far can be put simply as follows. When δ is applied on the environment, *Clock* advances by one time unit, an event called a *tick*, while at the same time any timed site will be time-shifted by one unit as well. The time lapse through the environment is synchronous.

To simply formalize this, we use Orc's $\mathit{Rtimer}(t)$ site, which publishes a *signal* after t time units. The δ function's effect can be directly seen on *Rtimer* in the following rewrite rule:

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

Needless to say, adding this to our definition will not affect the processing of a completely untimed Orc program. Such a program will be processed from beginning to end in 0 time units.

Synchronization. Another semantic element that we adopted is that threads synchronize before a time tick occurs. The tick rule is designed carefully so that it does not conflict with other actions that an Orc expression may take. To be specific, When a thread has a site call that hasn't been processed, time should not elapse until that site is called. Similarly, If a called site is ready to publish, then time must also not elapse until that value is published.

CHAPTER 4

𝕂 Semantics of Orc

This chapter shows in detail the semantics of Orc as defined in the 𝕂 tool. The definition is specified in multiple modules. These modules are: (1) the syntax module, (2) the main module, and (3) a cluster of semantics modules. These modules are explained in detail in this chapter. Table 4.1 shows the names of all the modules along with a brief description of each.

Although this chapter explains the definition in depth, it leaves out a few rules that are too technical and carry little semantic significance. Modules from which such rules were omitted—and only such modules—are fully delineated in Appendix A.

How to Read This Chapter. Each of the following sections will explain one module showing key rules and explaining its mechanics, its role and how it completes the picture. To do so, when explaining some modules, we will naturally refer to other modules, which is why we tried to list the sections in order of significance. Moreover, we tried as much as possible to order the modules themselves and their contents such that the simpler rules come first so as to provide gentle progression through the semantics. Therefore, it is important that this chapter is read in order because explaining some of the later rules depends on understanding details of earlier ones.

Another thing we will see as we go through the modules is how each performs its role in a way independent from yet harmonious with others. We will also see the modularity and abstractness of the rules that we intended in the design, in Chapter 3, and how it translated into a simple and elegant implementation in the 𝕂 tool.

```
rule [Sequential-Prep]:
  ⟨thread⟩ ...
    ⟨k⟩ (F:Exp > X:Param > G:Exp) ⇒ seqCompMgr(X, G, 0) ⟨/
        k⟩
    ⟨context⟩ Context ⟨/context⟩
    ⟨tid⟩ MgrId ⟨/tid⟩
  ...⟨/thread⟩
  (.Bag ⇒ ⟨thread⟩ ...
    ⟨k⟩ F ⟨/k⟩
    ⟨context⟩ Context ⟨/context⟩
    ⟨tid⟩ !NewId:Int ⟨/tid⟩
    ⟨parentId⟩ MgrId ⟨/parentId⟩
    ⟨props⟩ SetItem("seqLeftExp") ⟨/props⟩
  ...⟨/thread⟩)
 [structural]
```

Figure 4.1: The sequential-prep rule as defined in the 𝕂 tool.

Specification vs. Implementation. We sometimes refer to the work of this chapter as an *implementation*, even though it is far from the notion of a low-level application optimized for practicality; it is a specification, a formal specification. However, we call it an implementation because it is executable and because in a sense *implementation* completes *design*. Here and in the rest of the thesis, we will be using the two words, *specification* and *implementation*, interchangeably.

Representation of Rules. Rules are written in 𝕂 in plain ASCII text. Figure 4.1 shows one of the rules for the sequential combinator exactly as defined in plain text. We choose however to show the rules in a different representation—the one explained in Section 2.2.4—where each cell is shown as a bubble. The mentioned rule is shown in this chapter as Rule 4.5.1. We chose this because it is more readable and it clearly shows the nested cell structure. We explain it in Table 4.2.

4.1 Syntax Module

The syntax module contains syntactic productions from Orc's abstract syntax shown in Figure 2.1. In addition, it defines sorts (types) to make defining entities in rules simpler and more convenient. It uses a format similar to BNF, but has a few more syntactic and semantic elements defined, which we will explain shortly.

Orc is based on the 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 Figure 2.1 showing the abstract syntax of the Orc calculus, the following grammar defined in 𝕂 syntax is almost identical (with *Pgm* and *Exp* as syntactic categories for Orc programs and expressions, respectively):

Module Name	Description
`ORC-SYNTAX`	Defines the abstract syntax of Orc in a BNF-like style along with precedence, evaluation strategies and other useful annotations.
`ORC`	This is the main module where we import all the other modules and define our *configuration*.
Core modules	The core semantics modules specify the semantics of Orc using $\mathbb{K}$ rules. Each rule specifies one or more *rewrites*, that take place in different parts of the *configuration*.
`ORC-OPS`	Orc's four operators or combinators.
`ORC-SITECALL`	Manages site calls.
`ORC-EXPDEF`	Expression definition and calling.
`ORC-TIME`	Defines a discrete time model for timed sites.
`ORC-PREDICATES`	Contains functions that check if a certain type of thread exists in the current configuration.
`ORC-PUB`	Manages value publishing.
`ORC-VARLOOKUP`	Defines variable lookup mechanics and scope.
Sites modules	These give semantics to sites that serve specific purposes.
`ORC-ISITES`	Defines Orc internal sites like *let* and *if*.
`ORC-TSITES`	Defines Orc timer sites like *Rtimer*.
`ORC-MATH`	Defines some mathematical sites.
`ORC-ROBOT`	Defines sites for the virtual robot environment used in the test case of Section 5.1.
`ORC-LTL`	Provides functions necessary to compose LTL formulas used in model checking.

Table 4.1: Table of all modules implemented in our definition.

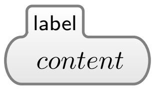
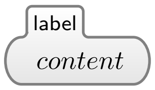
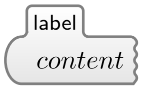
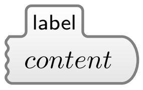
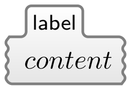
Notation	Meaning
	A cell has a label and content.
$\frac{\textit{Match this}}{\textit{Rewrite to that}}$	The horizontal bar represents a rewrite, the central operation of any rewriting-based semantics. Anything above the bar is matched and rewritten to what is below the bar.
Matching:	When matching a cell the label is always matched exactly. However, regarding its content, different shapes are used that affect how the matching is done.
	Match a cell having exactly this content.
	Match a cell having this content at its beginning.
	Match a cell having this content at its end.
	Match a cell having this content anywhere in it.
Content:	
Variable:*Sort*	This notation shows a variable and its sort. Sometimes, the sort is omitted because $\mathbb{K}$ can infer it.
—:*Sort*	This means any variable of the sort specified can be matched. In this case as well, the sort can be omitted when $\mathbb{K}$ can infer it.
•	The dot refers to empty content. It is applied to any sort to indicate an empty content of that sort. For example, the empty set is $\bullet_{Set}$; so are $\bullet_{List}$, $\bullet_{Map}$ and $\bullet_{Bag}$. Most other (non-group) sorts used in the definition are subsorts of the K sort and so the dot is applied to them like $\bullet_{K}$.

Table 4.2: Explaining $\mathbb{K}$'s bubble cell notation.

$$\begin{array}{ll}\textsc{syntax} & \mathit{Pgm} ::= \mathit{Exp} \\ & \quad\;\; \mid\ \mathit{ExpDefs}\ \ \mathit{Exp}\end{array}$$

$$\begin{array}{ll}\textsc{syntax} & \mathit{Exp} ::= \mathit{Arg} \\ & \quad > \ \mathit{Exp}\ \mathtt{>}\ \mathit{Param}\ \mathtt{>}\ \mathit{Exp}\ [\text{right}] \\ & \quad\ \mid\ \mathit{Exp} \gg \mathit{Exp}\ [\text{right}] \\ & \quad > \ \mathit{Exp}\ \mathtt{|}\ \mathit{Exp}\ [\text{right}] \\ & \quad > \ \mathit{Exp}\ \mathtt{<}\ \mathit{Param}\ \mathtt{<}\ \mathit{Exp}\ [\text{left}] \\ & \quad\ \mid\ \mathit{Exp} \ll \mathit{Exp}\ [\text{left}] \\ & \quad > \ \mathit{Exp}\ \mathtt{;}\ \mathit{Exp}\ [\text{left}]\end{array}$$

4.1.1 Expression Definitions

Defined expressions, if any, are placed before the main Orc expression. The syntax for defining expressions is as follows:

$$\textsc{syntax}\quad \mathit{ExpDefs} ::= \mathit{List}\{\mathit{ExpDef}, \text{“”}\}$$

$$\textsc{syntax}\quad \mathit{ExpDef} ::= \mathit{Decl}\ \mathtt{:=}\ \mathit{Exp}$$

$$\textsc{syntax}\quad \mathit{Decl} ::= \mathit{ExpId}(\mathit{Params})$$

$$\textsc{syntax}\quad \mathit{ExpId} ::= \mathit{Id}$$

4.1.2 Parameters and Arguments

Here we define what should arguments of site calls and expression calls be. Parameters, on the other hand, are the ones an expression is defined with. It should be mentioned that *Arguments* are what is called Actual Parameters in Orc's literature, while our *Parameters* are what they called Formal Parameters.

The following syntax specifies that an argument can be an Orc value, a tuple of values, an identifier, or a call (site or expression); and a parameter can only be an identifier. *Id* is $\mathbb{K}$'s builtin syntactic category for a general identifier string.

$$\begin{array}{ll}\textsc{syntax} & \mathit{Arg} ::= \mathit{Val} \\ & \qquad\ \mid\ \mathit{Tuple} \\ & \qquad\ \mid\ \mathit{Id} \\ & \qquad\ \mid\ \mathit{Call}\end{array}$$

$$\textsc{syntax}\quad \mathit{Param} ::= \mathit{Id}$$

4.1.3 Calls and Handles

A *Call* can be a site call or an expression call. Once a site is called, as explained in Section 4.10, it becomes a *Handle*. The four categories under it are also explained in that section. Site identifiers, *SiteId* are divided into *ISiteId* for internal sites and *TSiteId* for timer sites.

SYNTAX $\mathit{Call} ::= \mathit{SiteCall}$
$\quad\mid \mathit{Handle}$
$\quad\mid \mathit{ExpCall}$

SYNTAX $\mathit{SiteCall} ::= \mathit{SiteId}(\mathit{Args})$ [strict(2)]

SYNTAX $\mathit{ExpCall} ::= \mathit{ExpId}(\mathit{Args})$

SYNTAX $\mathit{Handle} ::= \mathit{FreeHandle}$
$\quad\mid \mathit{PubHandle}$
$\quad\mid \mathit{SilentHandle}$
$\quad\mid \mathit{TimedHandle}$

SYNTAX $\mathit{FreeHandle} ::=$ `handle` $(\mathit{SiteCall})$

SYNTAX $\mathit{PubHandle} ::=$ `pubHandle` (Val)

SYNTAX $\mathit{SilentHandle} ::=$ `silentHandle` $(\mathit{SiteCall})$

SYNTAX $\mathit{TimedHandle} ::=$ `timedHandle` $(\mathit{Int}, \mathit{SiteCall}, \mathit{Arg})$
$\quad\mid$ `timedHandle` $(\mathit{Int}, \mathit{SiteCall})$

SYNTAX $\mathit{SiteId} ::= \mathit{ISiteId}$
$\quad\mid \mathit{TSiteId}$

Internal or Meta Functions. Functions such as `pubHandle` and `silentHandle`, and later seen `prllCompMgr` and `seqCompMgr`, are specific to the definition in $\mathbb{K}$ and are not part of Orc. They are used in rules.

4.1.4 Values

Orc values are defined as described in Section 2.1.1.

SYNTAX $\mathit{Val} ::= \mathit{Int}$
$\quad\mid \mathit{Float}$
$\quad\mid \mathit{Bool}$
$\quad\mid \mathit{String}$
$\quad\mid$ `signal`
$\quad\mid$ `stop`

4.1.5 Manager Functions

Finally, we define the manager functions that were introduced in Section 3.1 for the combinators, a function for each composition.

SYNTAX *Exp* ::= `prllCompMgr` (*Int*)
SYNTAX *Exp* ::= `seqCompMgr` (*Param*, *Exp*, *Int*)
SYNTAX *Exp* ::= `prunCompMgr` (*Param*)
SYNTAX *Exp* ::= `othrCompMgr` (*Exp*)

4.1.6 Semantic Elements in Syntax

A few semantic elements appear in the defined syntax.

- The first is precedence, denoted by the $>$ operator, seen in the *Exp* production. As mentioned in Section 2.1.1, 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*.
- The second semantic element that is defined within the syntax module of $\mathbb{K}$ is `right`- or `left`-associativity.
- 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.

Associativity of the Parallel Operator. It is important to note that the parallel operator is defined as right-associative, even though it is in fact fully-associative. However, because that option is not provided in $\mathbb{K}$, we use rules to work around it; Section 4.4 details how this is resolved by transforming the tree of parallel composition into a fully-associative soup of threads.

4.2 Main Module

When $\mathbb{K}$ simulates a program, it generates a state-transition system where every state is represented by a configuration. The main module is where we define the structure of the configuration, shown in Figure 4.2, as a nested-cell structure. Each cell holds certain information about the state of the program. These cells and their contents are then used to define semantics in $\mathbb{K}$ rules as we will see in the coming modules. The following overviews the cells of our configuration and the role of each:

- Cell `T` is the topmost cell that holds the whole configuration. It is needed for technical convenience.

- Cell `threads` holds all the threads in the environment.
- Cell `thread` represents a single thread in the configuration. It is declared with multiplicity '*', which means a configuration can have zero, one, or more threads.
- Enclosed in `thread` is the cell `k`. `k` is the computation cell where we execute our program. It is where the program resides after it's parsed. We handle different Orc constructs from inside the `k` cell.
- Cell `context` is for mapping variables to values.
- Cell `publish` keeps the published values of each thread, while `gPublish` is for globally published values.
- Cell `props` holds thread management flags.
- Cell `varReqs` helps manage context sharing.
- Cell `gVars` holds global variables for environment control.
- Cells `in` and `out` are respectively the standard input and output streams.
- And finally, cell `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 Figure 4.2, would hold a single thread with the `k` cell holding the entire parsed Orc program as the *Pgm* non-terminal defined in the syntax above.

Modularity in Defining a Configuration. What is convenient about defining a configuration this way is that you can add new cells to the configuration, if needed, as you progress through defining your language. Suppose you want to add a feature to your already-defined language that needs certain information that you hadn't captured. In that case, you would add a cell to the configuration whose role is to hold that information; and then, without the need to change any of the current rules, you would make rules that target and manipulate that cell according to certain other predicates in the environment.

4.3 Combinators Module

The combinators or operators module called `ORC-OPS` defines how Orc's four combinators should be handled. It is the most significant semantics module; therefore it was designed and implemented with great care.

Orc has four combinators that combine subexpressions according to four distinct patterns of concurrent execution, *parallel*, *sequential*, *pruning* and *otherwise*. We chose to explain each in its own section, even though they are all part

CONFIGURATION:

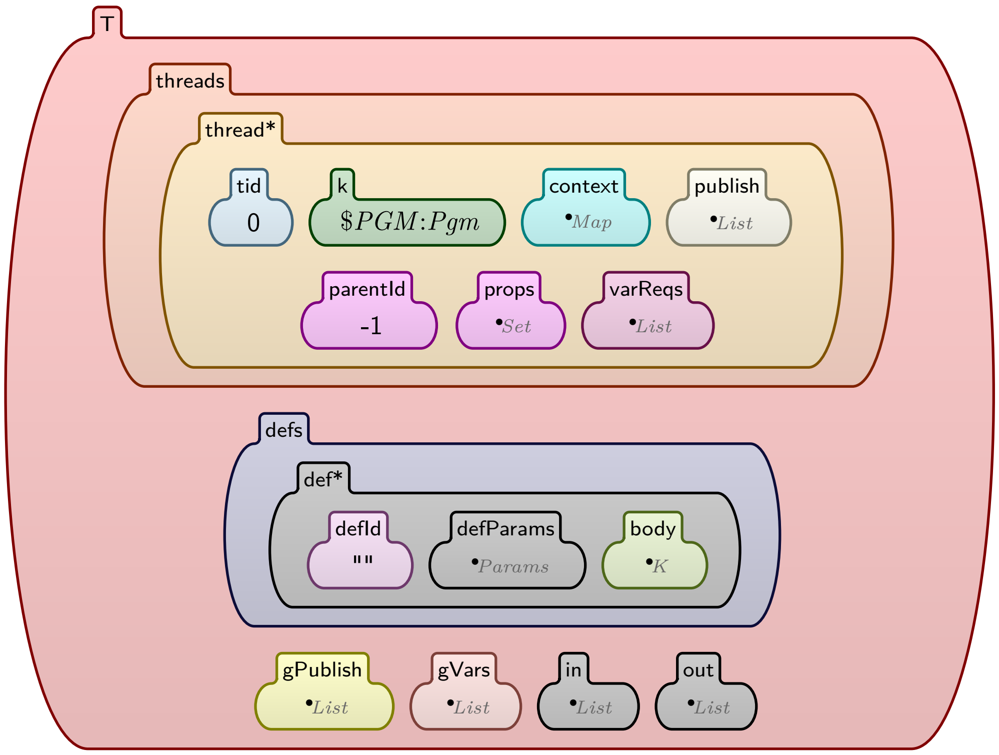


Figure 4.2: Structure of the configuration.

of the same module, because each takes as much space to explain as a whole module and even more, and because we would like to separate the rules for each combinator.

The next four sections will explain the implementation of these four combinators. We chose to start with the combinators because they are the core part of the semantics, and because their rules hold many concept and techniques that are essential to understand before diving into the rest of the definition. To make these concepts and techniques easily accessible and identifiable, we tried as much as possible to list them under properly titled sub-headings, so that even if a reader jumped forward in the chapter and faced an unexplained concept, or forgot its meaning, then they can quickly jump back and skim through sub-headings to find it.

4.4 Parallel Combinator

To process parallel expressions, we start with the simplest and the most general, i.e., $f \mid g$. It is handled through Rule 4.4.1, which creates a manager thread carrying an internal function called `prllCompMgr`(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. In $\mathbb{K}$, any new thread will run immediately and all threads in the environment automatically run concurrently. So it suffices to create a thread to tell $\mathbb{K}$ to run it in parallel with other threads. In this case, f and g will run in parallel, which is what we intended.

Now what remains is to generalize that to extend it to any number of subexpressions in the initial expression. For example, $f \mid g \mid h$ should rewrite to a managing thread with `prllCompMgr`(3) and create three child threads, and so on. To achieve that, we made Rule 4.4.2. To understand its role, consider the parallel expression:

$$\mathit{E1} \mid \mathit{E2} \mid \mathit{E3} \mid \mathit{E4}$$

Recall that in the syntax module, we declared the parallel operator right-associative. This means that our expression will be parsed effectively like the following:

$$\mathit{E1} \mid (\mathit{E2} \mid (\mathit{E3} \mid (\mathit{E4})))$$

In this case, Rule 4.4.1 will apply to that expression where F is substituted by $\mathit{E1}$ and G by $(\mathit{E2} \mid (\mathit{E3} \mid (\mathit{E4})))$. So now we need to simplify or flatten this compound G expression, and we do so using Rule 4.4.2. This rule applies recursively on the compound expression, creating a new child thread every time and increasing the count of managed children by one. This rule effectively makes the parallel operator fully associative.

Thread Properties

The `props` cell is used throughout the specification to carry information about threads that is vital for rules to communicate so that each carries out its role. In this case, Rule 4.4.1 and Rule 4.4.2 give the following two properties to each child thread they create:

- The `publishUp` property which means that a thread is allowed to pass any value found in its `publish` cell to its parent, whether it published it itself or received it from a child. A certain rule, Rule 4.11.3, is responsible for passing values up the thread tree from threads carrying this property. This is an example of communication between rules using the `props` cell. This property is given to multiple kinds of threads as detailed in Section 4.11.1.
- The `prllChild` property is one of many properties that tag a thread by its type. It is used to ensure that the matched thread is indeed a parallel child. Even though this specific property is not needed because the rules already match the parent having the `prllCompMgr` function, the properties similar to this one in other combinators are indeed necessary as will be seen. On top of that, such properties are useful for debugging purposes and when simply viewing simulation results.

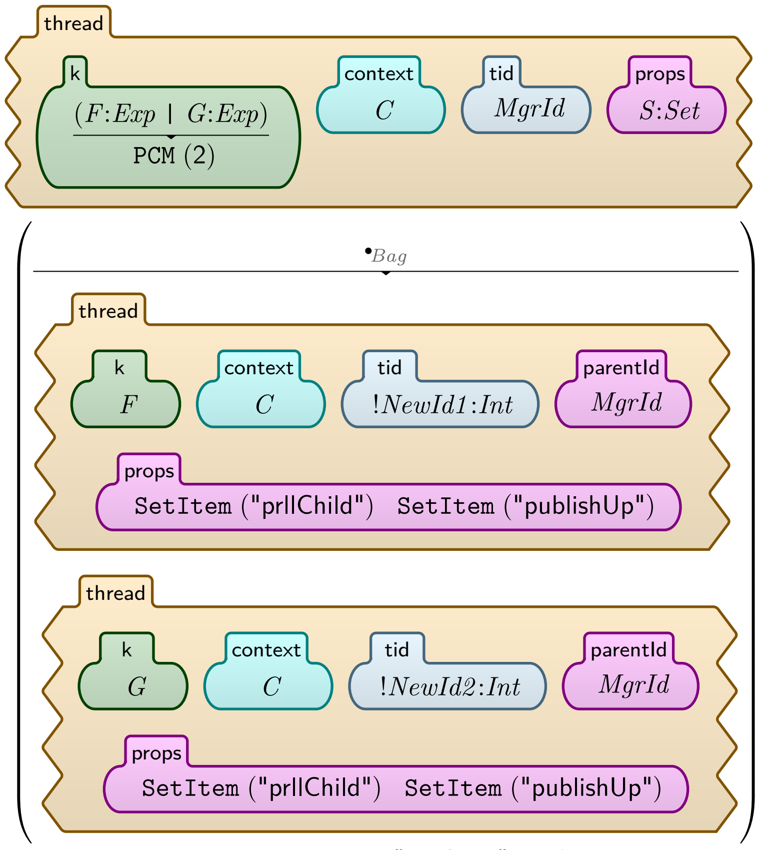
Cell Shapes on the Rewritten Side

We explained in Table 4.2 under matching the meaning of each shape. Those meanings apply only when the shapes are at the matching side of a rewrite. At the rewritten side, a both-side-smooth cell will make cell have the exact same content as specified whereas a both-side-torn cell tells $\mathbb{K}$ to fill unspecified content of the cell with the default values specified in the configuration. Always, for the sake of modularity, we use the latter. There never is a case where incomplete cells would help anywhere, at least in our definition.

Creating New Threads

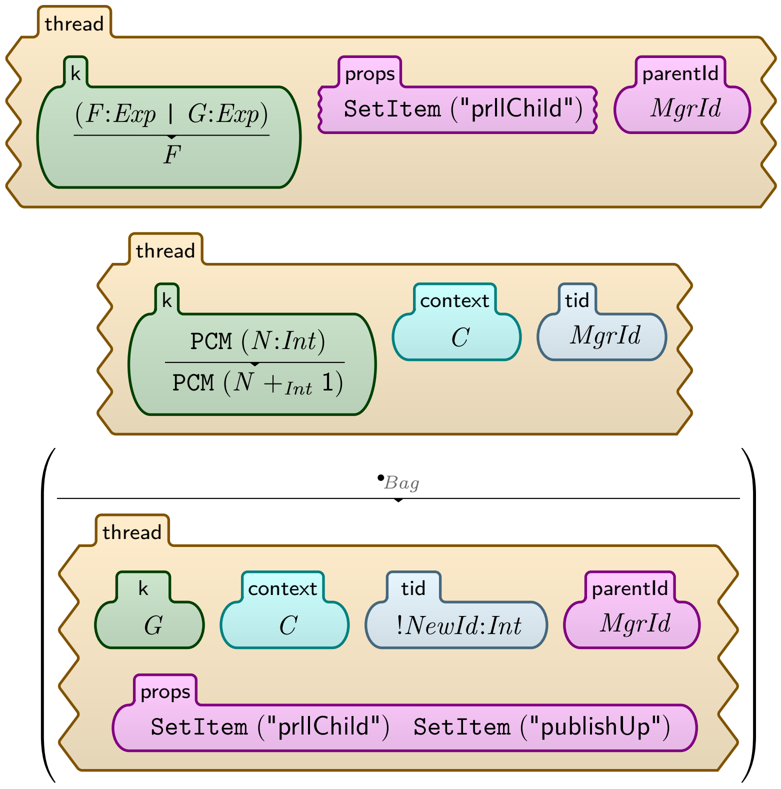
Every time a new thread is created, the following points are done:

- A both-side-torn cell is used to let $\mathbb{K}$ fill in the rest of the content as we just explained.
- The `context` map, which carries variable mappings, is copied from the parent to the newly created child, because a child shares the scope of its parent.
- The new thread is assigned a new thread ID (`tid`) using $\mathbb{K}$'s `!` operator.
- The parent's thread ID is copied into the `parentId` cell of the child thread to link child to parent.



requires $\neg_{Bool}$"prllChild" in S

Rule 4.4.1: Parallel prep.



Rule 4.4.2: Parallel expression flattening.

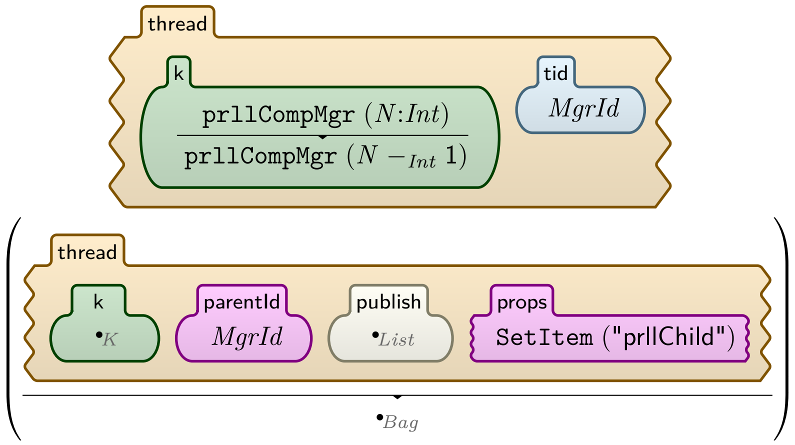
Halted Thread Cleanup

After the created threads finish execution, they need to be erased. Generally, we consider a thread to have finished execution (halted) if it has no more commands to execute, and it has nothing to publish. So we check its `k` and `publish` cells, and if they are empty, we erase the thread.

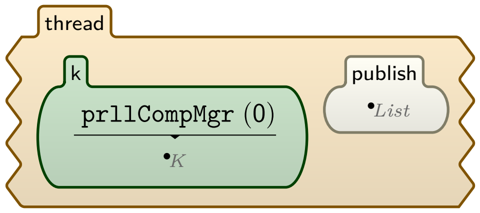
In the case of threads whose manager keeps count of, like the parallel composition manager does, we need to decrement the count as we erase them. This is the work of Rule 4.4.3; it kills a child thread that has halted and decrements the count of managed children. Once all managed threads are killed, Rule 4.4.4 is responsible for halting the manager by simply erasing the content of its `k` cell. These two rules constitute the cleanup for the parallel composition. Each of the four combinators has similar cleanup rules that will be explained.

Modularity in Thread Management

We should point out here that in the grander scheme of our design, no thread is responsible for erasing itself, but rather every manager is responsible for cleaning up its own children. This will become clear throughout this section as we go through every combinator's cleanup rules. Also, every parent is a manager, which means that the only threads that are not managers are the leaves of the thread tree. Therefore, this design adds modularity. This is especially obvious considering that each of the managed threads can itself be—and would most probably be—a manager of other threads.



Rule 4.4.3: Parallel cleanup 1.



Rule 4.4.4: Parallel cleanup 2.

4.5 Sequential Combinator

Processing a sequential expression $f >x> g$ is done exactly as the design shown in Figure 3.2 intended. Rule 4.5.1 is the first step. It prepares the expression by rewriting it into a manager thread carrying the internal function `seqCompMgr` in its `k` cell. It also creates a child that will execute f. The manager keeps three pieces of information that are the three arguments of `seqCompMgr` in order:

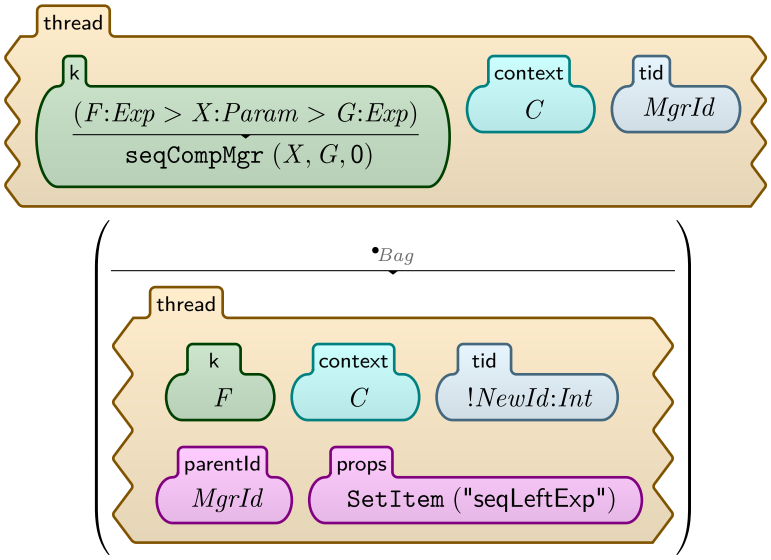
- X, the parameter through which values are passed to instances of G.
- G, the right-side expression.
- The third argument, a count of active instances of G which is initially 0.

The first two are used in the spawning rule, explained shortly, while the third is necessary for defining cleanup rules. Let us have a close look at the created child. It carries its parent's context map as well as its thread ID, and a property declaring its type a `seqLeftExp`, a left-side child of a sequential expression. This is similar to children of the parallel expression we just explained in Section 4.4, except that this child does not have the `publishUp` property. That is because any value that this child publishes should be handled directly and exclusively by its parent, the `seqCompMgr`; and it does that in Rule 4.5.2.

The Spawning Rule

Rule 4.5.2, the spawning rule, is the key rule of the sequential combinator, and here is how it works. It fires as soon as the left-side child publishes a value. It detects the published value by matching the content of the child's `publish` cell with a type *Val*. Keep in mind that this rule applies every time F publishes a value. When it fires, the following happens:

- The rule creates a thread and places in its `k` cell the expression G which the manager has been carrying for this purpose.
- The manager's count of managed instances of G is incremented by one.
- The created child is given the `publishUp` property.



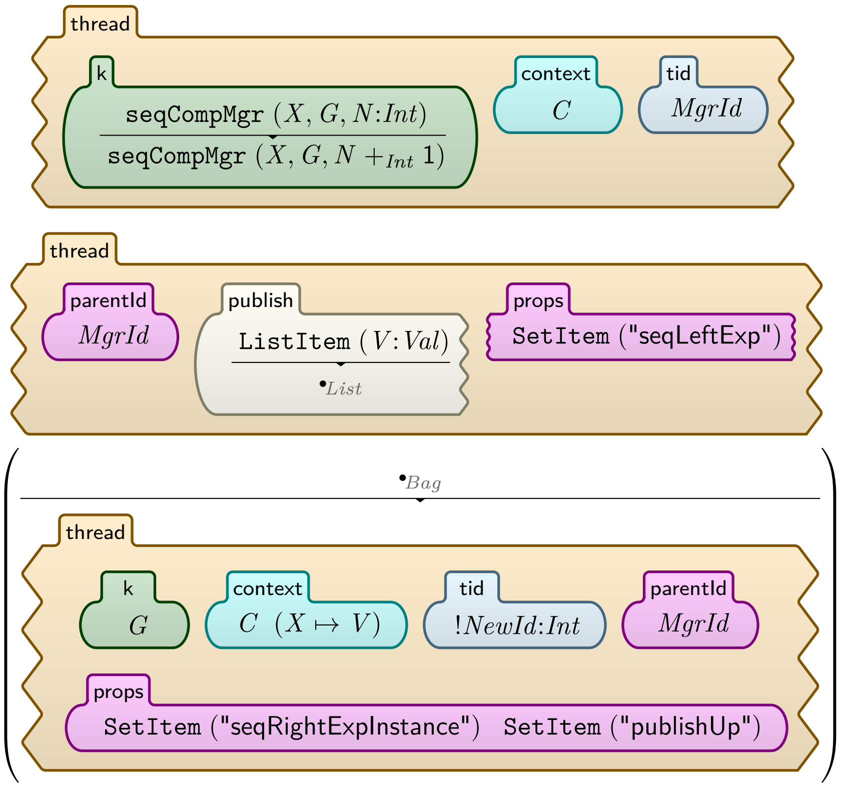
Rule 4.5.1: Sequential prep.

- The created child is tagged by its type: `seqRightExpInstance`. This is necessary for cleanup rules.
- Most notably, the parent's context map is copied to the created child but with adding a mapping $X \mapsto V$, binding the variable X in G to the value V that was published by F.

Matching the `publish` Cell. Note that the `publish` cell has only its right side torn, which means that the content is matched at the beginning of the cell. That is because we need to match only one value and remove it from the list, not caring about the rest of the cell. If the rest of the cell contains any other values, they will be processed in the same way in future iterations of this rule. This is how, for every value published by F, the rule will apply.

Handling the $\gg$ Operator

As explained in Section 2.1.1, the $\gg$ operator is used like in $f \gg g$ when no variable is to be bound in g. This is effectively a sequential operation where g is followed by f which could be directly modeled in $\mathbb{K}$ as so. However, for the reasons of generality, modularity, and faithfulness to the original semantics, we chose to treat it as the syntactic sugar it is, and expand it into its more general form. We do so by adding a dummy variable so that the expression becomes



Rule 4.5.2: Sequential spawning.

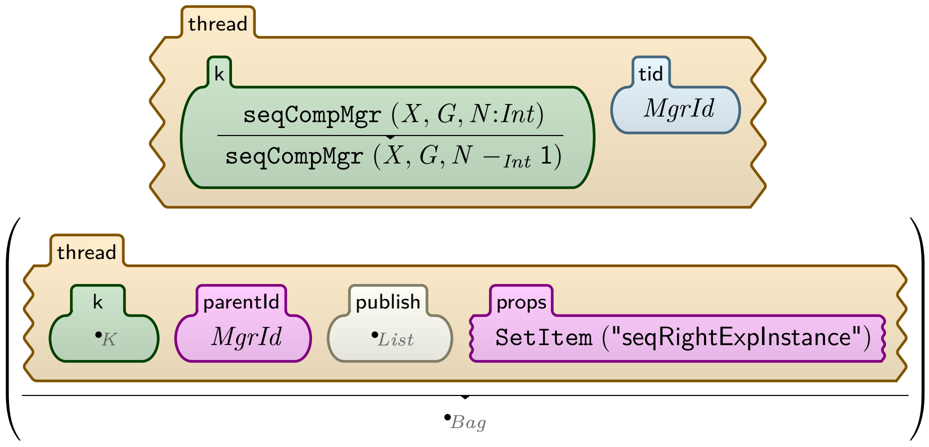
f >*dummy_var*> g. Rule 4.5.3 does just that though it needs a $\mathbb{K}$ built-in function to define the string `dummy_var` as an identifier.

$$\frac{F \gg G}{F > \texttt{String2Id}\,(\texttt{"dummy\_var"}) > G}$$

Rule 4.5.3: Sequential, de-sugaring of the $\gg$ operator.

Cleanup: Right-side Instances

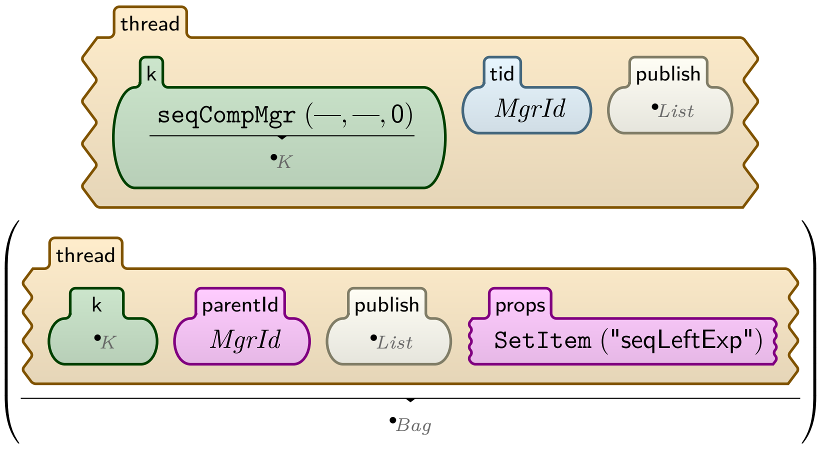
Cleaning up created threads is done differently for each of the two types of threads: the left-side child, and the right-side instance. We start with the latter, which will continue to be treated like parallel threads. Rule 4.5.4 matches a child thread tagged with `seqRightExpInstance` that has halted and has nothing to more to publish, erases it and decrements the count of managed instances from the manager. This is similar to Rule 4.4.3, the parallel child cleanup rule. In fact it is so similar that these two can be generalized into one rule with a bit of work—if the count of children could be isolated and the two properties unified—, but that would add unnecessary complexity and will make the rules look less uniform and harder to read and understand.



Rule 4.5.4: Sequential cleanup 1.

Cleanup: Manager and Left-side Child

After all right-side instances had halted and are cleaned up, we proceed to clean up the left-side child and the manager using Rule 4.5.5. This rule matches two things: A manager thread that has `0` managed children and left-side child that has halted. Both of these must have empty `publish` cells, as is the case in all cleanup actions. Once the match is done, the rule then erases the manager function thus by halting the manager thread; and it kills the left-side child. Note that this rule may apply under a slightly different scenario, that is if the left-side child halts without publishing any values, which is a correct application.



Rule 4.5.5: Sequential cleanup 2.

4.6 Pruning Combinator

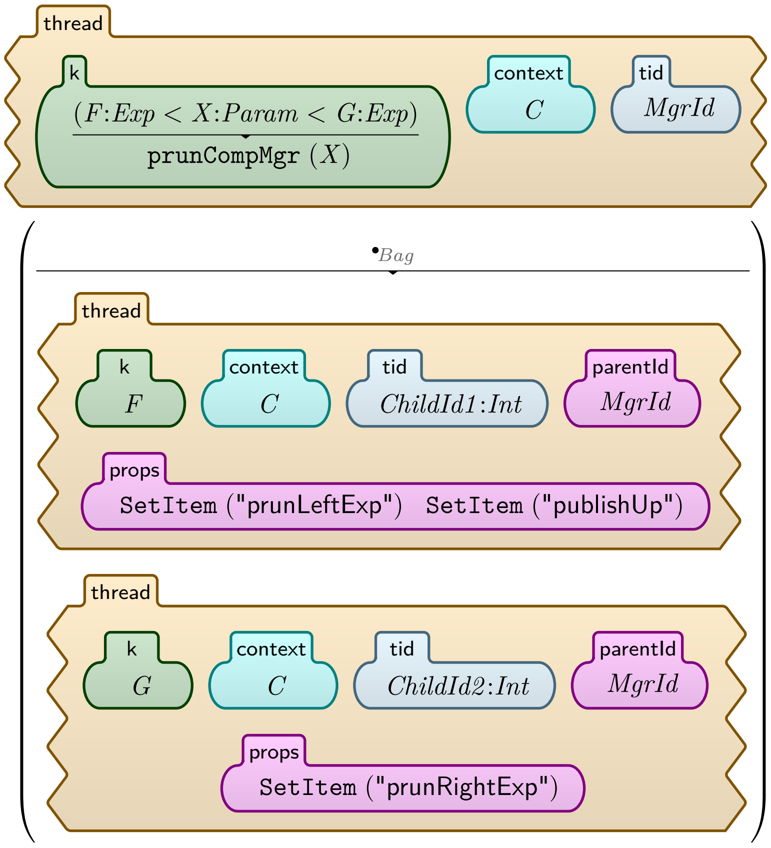
Just like with the parallel and sequential combinators, we process a pruning expression $F <X< G$ first through a prep rule, that is Rule 4.6.1. It rewrites the expression to a manager thread and creates two child threads, a left-side child executing F and a right-side child executing G. That is because according to the semantics, both should start execution concurrently even if F needs a value from G.

The Pruning Operation

As soon as G publishes a value comes the role of Rule 4.6.2. This is the key rule here that does the actual pruning, and here is how it works. It matches three threads, the manager and its two children, which are linked to the parent through its thread ID.

Right-side Child. In the right-side child, Rule 4.6.2 does the following:

- It terminates *any* computation (—:K) in the `k` cell by rewriting it to *nothing* ($\bullet_K$).
- It takes the first value from the `publish` cell and erases the rest.

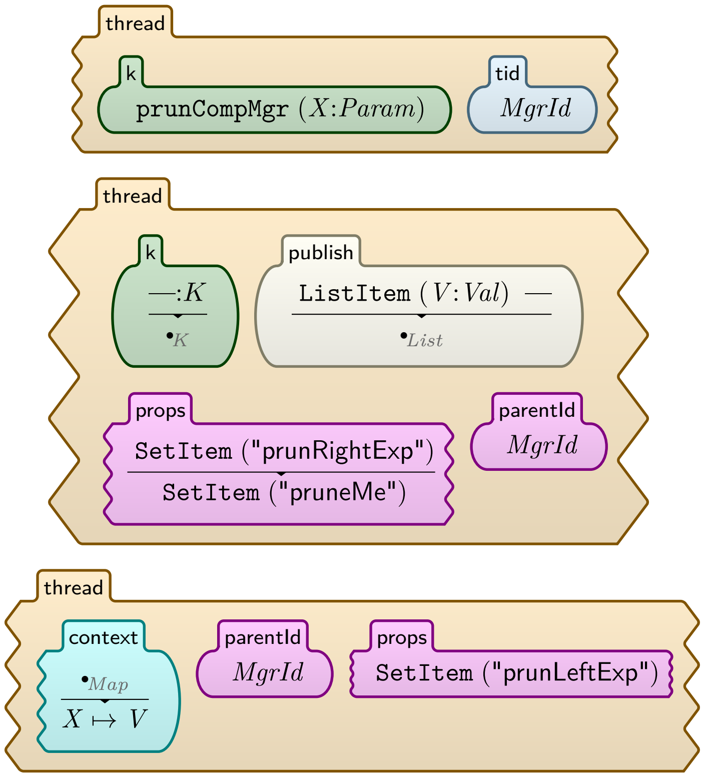


Rule 4.6.1: Pruning prep.

- It changes the property from `prunRightExp` to `pruneMe` so that the cleanup rule processes it later.

Matching the `publish` Cell. Note that the `publish` cell has both sides smooth, which means that matching is done on the whole cell. Note also that it matches a $V\!:\!Val$ at the beginning of the cell followed by $\mathbb{K}$'s — operator which means *anything*. This is to make sure that, if the cell contains any other published values, the rewrite to *nothing* ($\bullet_{List}$) happens to the whole cell.

Left-side Child. At the same time, the rule (Rule 4.6.2) maps X to V in the left-side child, where V is the value published by the right-side child.



Rule 4.6.2: Pruning prune.

De-sugaring the $\ll$ Operator

Similarly to the de-sugaring of the sequential $\gg$ operator, we use Rule 4.6.3 to expand $F \ll G$ to F *<dummy_var<* G

$$\frac{F \ll G}{F < \texttt{String2Id}\,(\texttt{"dummy\_var"}) < G}$$

Rule 4.6.3: Pruning, de-sugaring of the ≪ operator.

Pruning Cleanup

Pruning has two cleanup rules to handle the two different cases where a pruning expression can halt:

- The first case is that *G* publishes and *F* halts. This is handled by Rule 4.6.4 which matches the left-side child having halted and not had any published values left in its `publish` cell; and matches the right-side child with the property `pruneMe`, which is given by Rule 4.6.2 to indicate that *G* has published.
- The second case is when *F* and *G* have both halted without *G* having published anything. In this case, the right-side child would still have the property `prunRightExp`, but its `k` cell should be empty. And this is what Rule 4.6.5 matches.

Both Rule 4.6.4 and Rule 4.6.5 kill both children and halt the manager to mark the pruning expression itself halted.

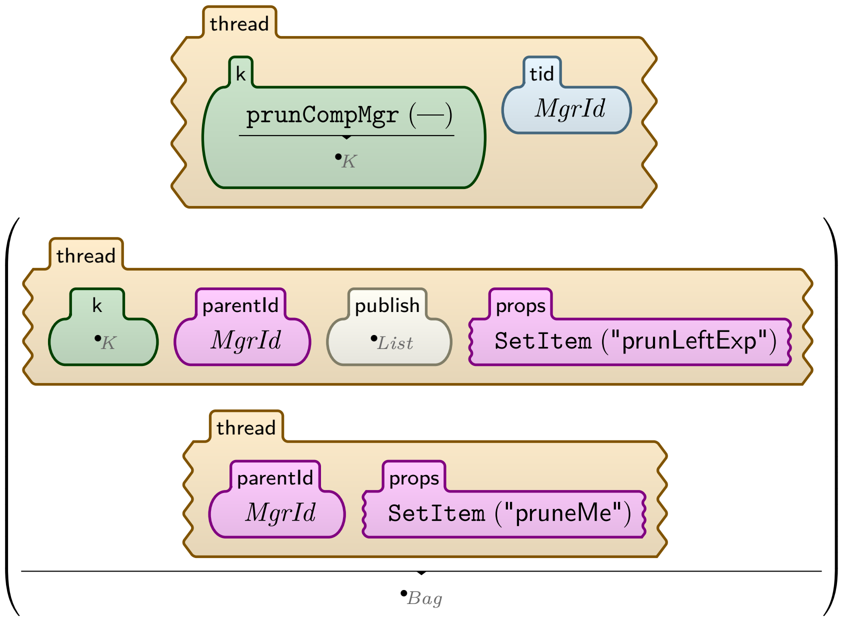
4.7 Otherwise Combinator

Rule 4.7.1 processes the otherwise expression $F; G$ by rewriting it into a manager function called `othrCompMgr`, and creating a child thread executing *F* and carrying the property `othrLeftExp`. *G* is stored as an argument of `othrCompMgr`. Now we have two cases to deal with; either *F* publishes or halts without publishing:

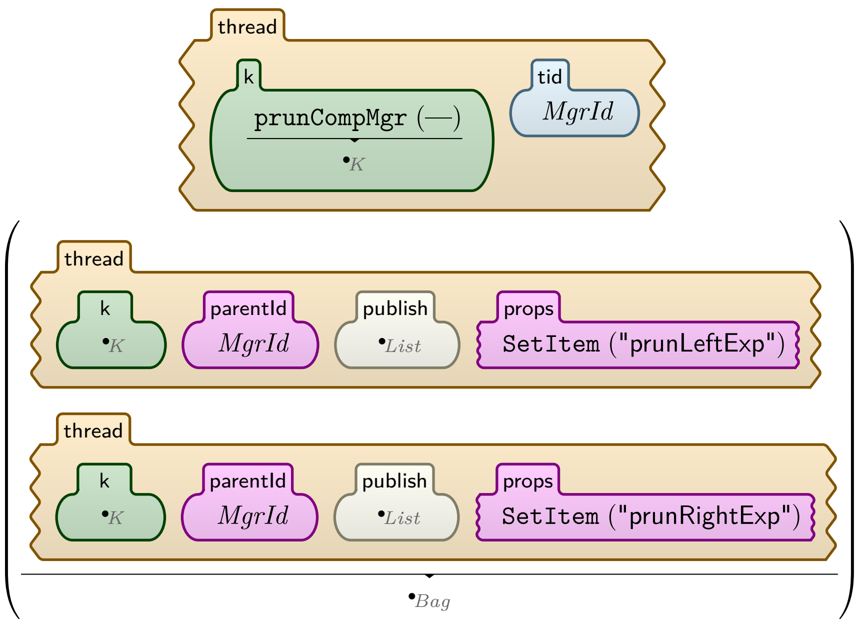
- If *F* publishes a value, then Rule 4.7.2 applies. It discards *G* from the manager and gives *F* the `publishUp` property. Now *F* will continue its execution normally and pass published values towards its parent.
- If *F* halts without publishing anything, then Rule 4.7.3 applies. It kills the left-side child, creates a child executing *G* and gives it the `publishUp` property, effectively replacing *F* by *G*.

Otherwise Halting and Cleanup

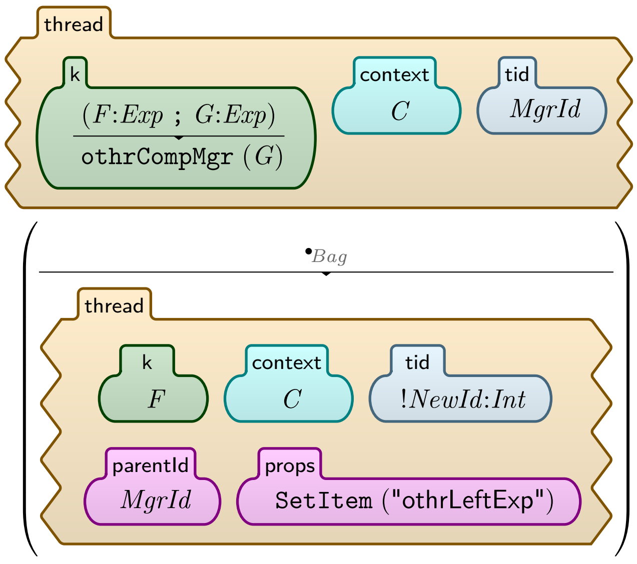
Now whether Rule 4.7.2 or Rule 4.7.3 applied, we will end up with similar configurations because of the symmetry of the two rules. This makes Rule 4.7.4, the cleanup rule, apply in both cases once the child halts. The rule will then kill the child and halt the manager.



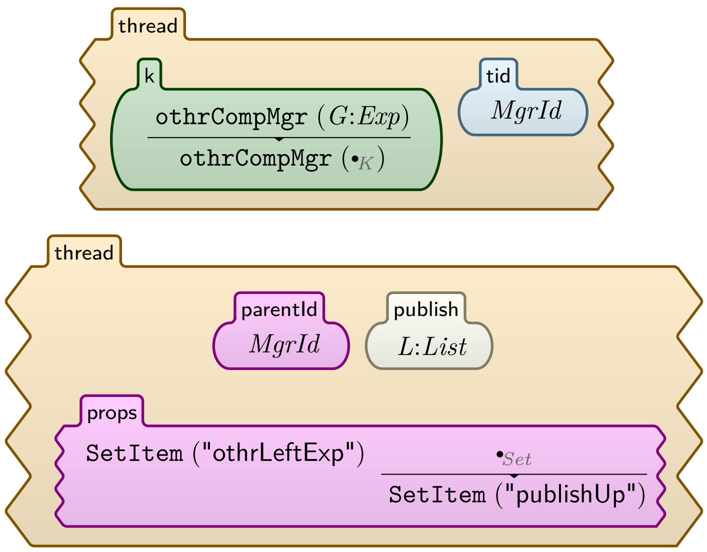
Rule 4.6.4: Pruning cleanup 1.



Rule 4.6.5: Pruning cleanup 2.

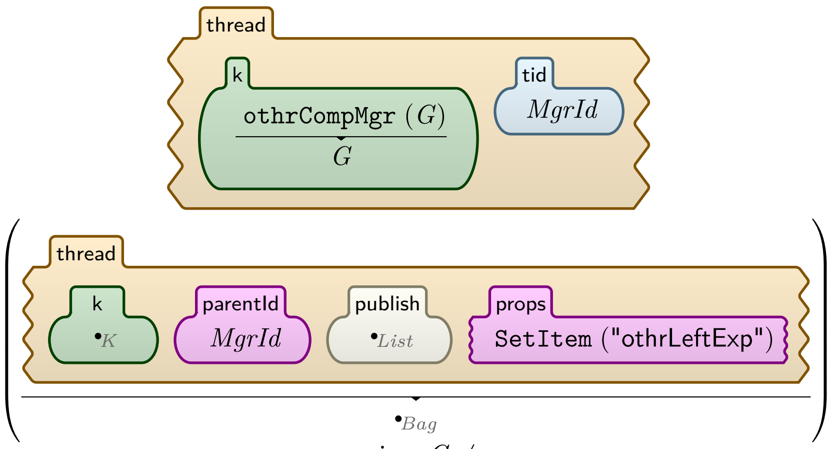


Rule 4.7.1: Otherwise prep.



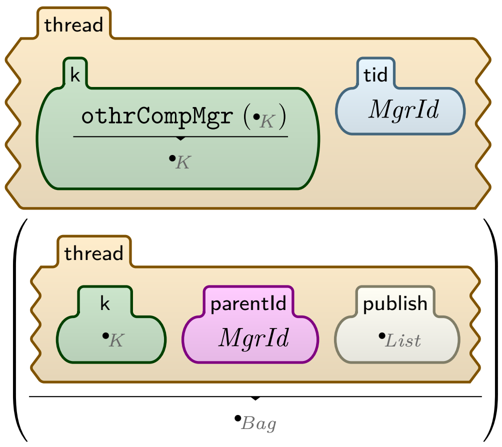
requires $L \neq_K \bullet_{List} \wedge_{Bool} G \neq_K \bullet_K$

Rule 4.7.2: Otherwise left published.



requires $G \neq_K \bullet_K$

Rule 4.7.3: Otherwise left halted without publishing.



Rule 4.7.4: Otherwise cleanup.

4.8 Recap

Before moving on to the next module, we would like take a pause and look back. In Section 3.1, we showed the design of the semantic rules of the combinators through schematics; and in Section 3.2, we pointed out how the uniform structure of thread hierarchy was common in the rules of all four combinators. We also showed that the rules are oblivious to how complex the expressions processed are. And this was reconfirmed by the rules explained so far in this chapter where each thread is managed by its parent. This kind of abstraction made it possible to handle publish-ups and halts modularly and will make—as will be seen later in this chapter—defining general operations like publishing and variable lookup straightforward.

What About the Root Thread? We keep saying that every thread has a manager which makes the rules modular and uniform; but what about the root thread? How is it managed? Well, the root thread, the topmost node, is the whole program. If it halts then that's the end of the program's execution. If it is stuck trying to resolve a variable, then the program is stuck, and is rightfully so. If it needs to publish or pass on published values then that is the output of the whole program; and that is called root publishing and it is handled by Rule 4.11.4.

Matching Contents of `k` Cells. In almost all the rules, `k` cells are both-side-smooth, meaning they are matched entirely. That is because if a `k` cell had a complex expression, then it will be expanded by parsing and combinators' prep rules.

4.9 Synchronization

What we mean by synchronization is controlling the order of certain events; in particular, controlling the order at which certain rules apply. We synchronize threads on the following actions:

- Site calling (Rule 4.10.1).
- Publishing (Rule 4.11.2).
- Time ticking (Rule 4.13.4).

And these actions are done in that order. So no site is allowed to publish before all site calls in the environment have been made. Likewise, time is not allowed to tick unless all publishes currently in the environment have been done.

This is implemented through the rules that apply these actions. Each of these rules has a side condition that guarantees the synchronization. The conditions use predicate functions explained in Section 4.14.

4.10 Site-Call Management

Here we explain how the module `ORC-SITECALL` works. A site call is replaced by a handle as Rule 4.10.1 does; and with that, we consider the call to have been made. In the syntax explained in Section 4.1, we defined four types of handles:

- *FreeHandle*: an unprocessed handle. This needs a rule exclusive to the site called in order to process it.
- *PubHandle*: A processed handle that represents a site publishing the value v.
- *SilentHandle*: A processed handle for a site that should remain silent, and will not undergo further processing by any rule.
- *TimedHandle*: A processed handle for a site that should remain silent, and will not undergo further processing by any rule.

From there on, individual rules that define individual sites, such as `if` and `Rtimer`, rewrite the handles into the appropriate type of handle. Those rules can be viewed in the `ORC-ISITES` module and the `ORC-TSITES` module.

For example, see Rule 4.10.2 defining the behavior of the site *if* in case its argument is evaluated to *true*. Recall that in Orc, *if*(*true*) publishes a signal, but *if*(*false*) remains silent. So when a site publishes, its handle is rewritten to a `pubHandle` which is then processed in the `ORC-PUB` module explained in Section 4.11. To process *if*(*false*), we rewrite it into a different type of handle, `silentHandle`, in Rule 4.10.3.

$$\frac{SC{:}\mathit{SiteCall}}{\texttt{handle}\ (SC)}$$

Rule 4.10.1: Site calling.

$$\frac{\texttt{handle}\ (\ \texttt{if}(\mathsf{true}))}{\texttt{pubHandle}\ (\ \texttt{signal})}$$

Rule 4.10.2: *if*(*true*).

Why the Silent Handle? If a site is supposed to remain silent, why do we rewrite its handle into a *SilentHandle* instead of just ignoring it and leaving it as a *FreeHandle*? The answer is because all site calls need to be processed in order for certain other rules to apply. And in our model is no other way of knowing whether a handle has been processed and is supposed to remain silent, or hasn't been processed at all. This approach is discussed in Section 7.2.4.

4.11 Publishing

The way publishing is implemented is through the `publish` cell. The act of publishing a value, in our semantics, is moving that value from inside a `pubHandle` in the `k` cell, into the `publish` cell. This is exactly what Rule 4.11.2 does. In Orc, only sites publish values, and we stay true to that in our semantics. Even an expression that is just a single value v, which we call a *lone Val*, is actually syntactic sugar for the site call *let*(v). Rule 4.11.1 processes that by simply rewriting the lone *Val* v into *let*(v).

Taking that and the rules we have seen so far in this chapter into account, we can see that any Orc expression will boil down to a group of site calls each in its own thread. Once any of these sites publishes a value, it will be carried by a `pubHandle`.

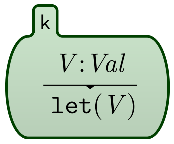
After that, we only need to process `pubHandle`'s; and we do so in Rule 4.11.2 just as explained in the beginning of this section.

Using Predicates in Publishing. The side condition of Rule 4.11.2 checks that two functions return *false*. We gave these functions a special name, predicates, and they are defined in the `ORC-PREDICATES` module which is explained

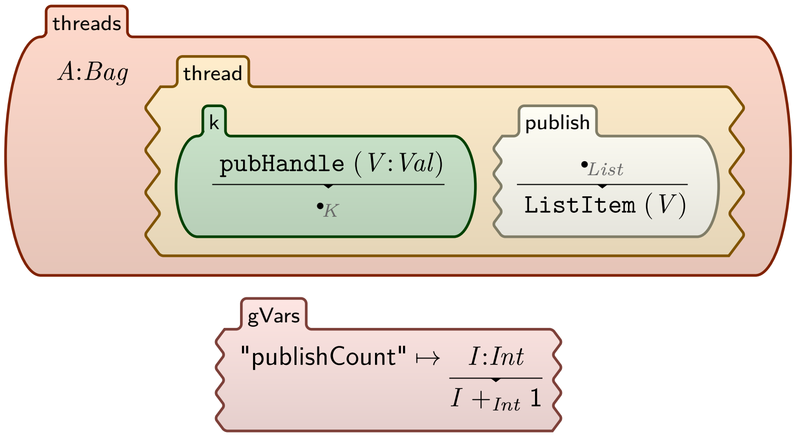
$$\frac{\texttt{handle}\ (\ \texttt{if}(\mathsf{false}))}{\texttt{silentHandle}\ (\ \texttt{if}(\mathsf{false}))}$$

Rule 4.10.3: *if*(*false*).

in Section 4.14. These predicates are given as an argument the whole bag of threads excepting the one thread being matched where the publish is being made. The side condition ensures that no publish is made unless the environment is empty of *SiteCalls* and of *FreeHandles*, both of which were just explained in Section 4.10. This is to enforce the priority of site calling over publishing.



Rule 4.11.1: Lone *Val* de-sugaring.



requires $\neg_{Bool}($ `anySiteCall` $(A)) \wedge_{Bool} \neg_{Bool}($ `anyFreeHandle` $(A))$
[transition]

Rule 4.11.2: Publishing.

The transition Attribute. The transition attribute decorating a rule marks it as an observable transition from one configuration to the next. Very few rules are given this attribute because it adds exponential complexity to the computations made by $\mathbb{K}$, evident especially when using state search. All other rules are given by default the structural attribute marking them as unobservable transitions, which means that even if they were designed to produce nondeterministic behavior, it will not be explored by state searching. In Chapter 5, we run examples that explore nondeterministic behavior. The choice of which rules are given this attribute is discussed in Section 7.1.1.

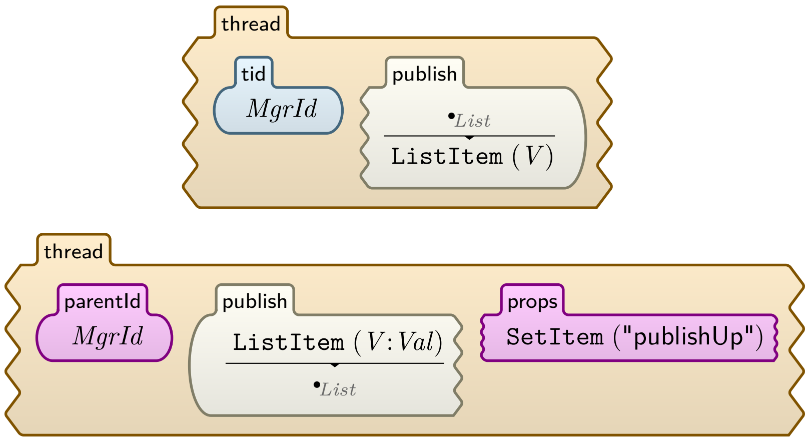
4.11.1 Publishing to Parents

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 in Chapter 3, this property is denoted by a letter P in the lower left corner of the thread box as in Figure 3.5. To complete the picture, notice that the child receiving the `publishUp` property might itself be a manager of a deeper composition, awaiting values to be published up to it. This behavior creates a channel from the leaves of a thread tree up to its 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*
- Any child of an *Otherwise Composition Manager*.

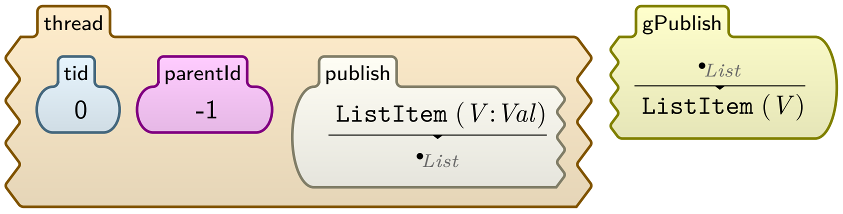
Rule 4.11.3 is responsible for publishing values to parents and Rule 4.11.4 handles that for the root thread publishing to the `gPublish` cell.



Rule 4.11.3: Published value propagation.

4.12 Variable Lookup

When a variable name is encountered in a computation, i.e. in the `k` cell, it is resolved by Rule 4.12.1. This rule checks the `context` cell for the variable name and, if found mapped to a value, it substitutes the variable in the computation with the value. However, if it is not found, Rule 4.12.2 creates what we call a variable request.



Rule 4.11.4: Publishing at root.

4.12.1 Variable Requests

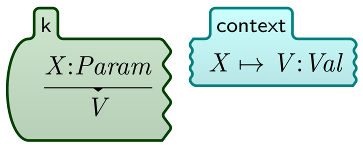
Similar to the propagation of published values up a thread tree is that of variable lookup inquiries. A child thread shares the scope of its parent, but not vice versa. Therefore, every thread is allowed to access the context map of any of its ancestors. An inquiry about a variable name, represented by the `varRequest` function, carrying the requester thread's ID, is created by Rule 4.12.2 and then propagated recursively up the tree by Rule 4.12.3, through a specialized cell `varReqs`. The request keeps propagating up the tree until it is resolved by Rule 4.12.5 or until it reaches the root in which case Rule 4.12.4 resets it.

Condition on Reset. Notice the side condition of Rule 4.12.4. The first part checks that the variable is not found in the current context which is a condition common with Rule 4.12.2 and Rule 4.12.3. The second part, that is our focus here, is specific to the reset action and holds the key to understanding the underlying mechanism. That is the condition: $\mathit{CurrPcount} >_{\mathit{Int}} \mathit{Pcount}$. *CurrP-count* is being retrieved from inside the `gVars` cell from a global environment variable called `publishCount` which keeps the number of publishes made since the program started. The publishing rule, Rule 4.11.2, updates that variable with every publish. Now, the function `varReq` keeps, as an argument, the number of publishes made when it was created. The condition ensures that at least one publish has been made after the request was created. This is important because if no publish has been made, then nothing is gained from resetting the request, because no other action may produce a value for the requested variable. Another reason for this condition is that it will prevent an infinite loop of the request propagating until the root, resetting, getting recreated, propagating again and so on.

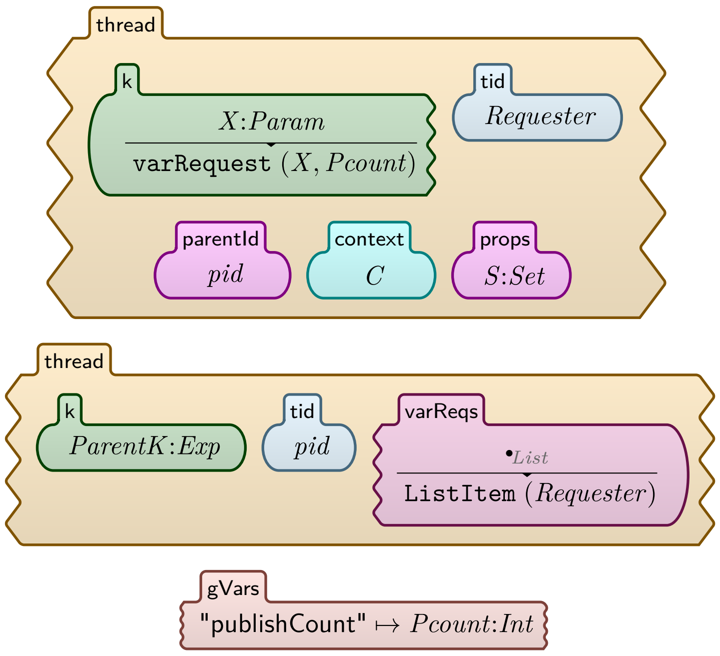
Role of `varReqs` Cell. This cell is where each thread keeps requests that reached it from its descendants. It is a list of thread IDs of the requester children. When a request is pushed up to a parent, like in Rule 4.12.3, the item concerning the requester is deleted from the child and created at the parent. This happens until eventually an ancestor who has the needed variable gets the requester's ID in its `varReqs` cell; and that makes Rule 4.12.5 apply resolving the request.

Scope. 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.

Expected Variables. Both Rule 4.12.2 and Rule 4.12.3 do not even try to request a variable in case it is expected to be provided by a managing parent. Such is the case when a left-side child of a pruning composition is requesting the variable that the manager is supposed to receive from the right-side child. That case is checked by the side conditions of those two rules.

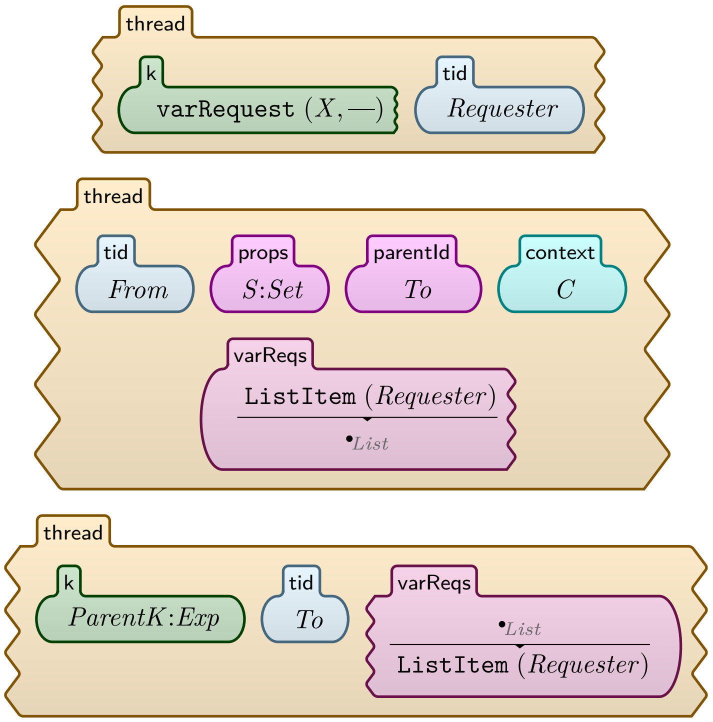


Rule 4.12.1: Basic variable lookup.



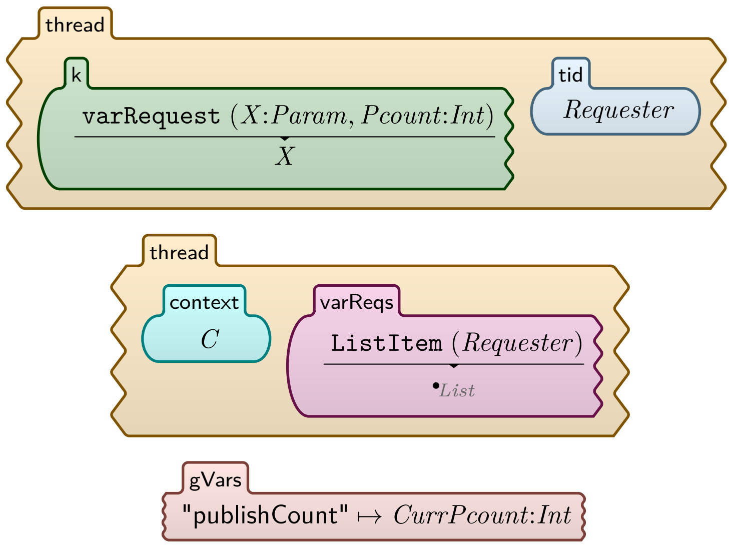
requires $\neg_{Bool} X$ `in keys` $(C) \wedge_{Bool}$
$(ParentK \neq_K$ `prunCompMgr` $(X) \vee_{Bool}$ `"prunRightExp" in` $S)$

Rule 4.12.2: Variable request creation.



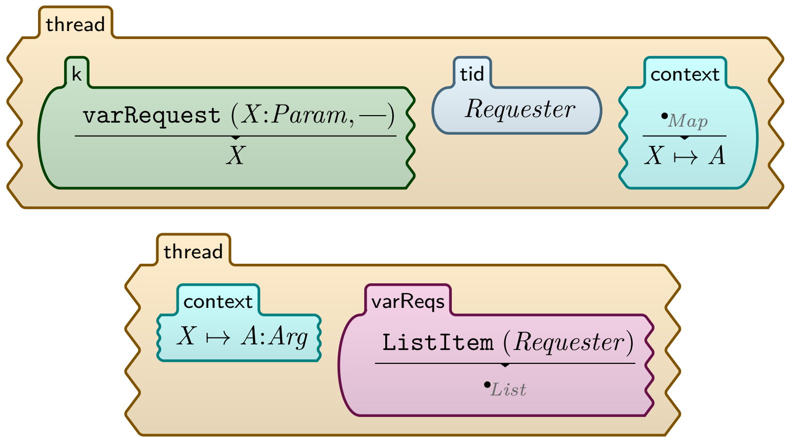
requires $\neg_{Bool} X$ `in` `keys` $(C) \wedge_{Bool}$
$(ParentK \neq_K$ `prunCompMgr` $(X) \vee_{Bool}$ `"prunRightExp"` `in` $S)$

Rule 4.12.3: Variable request propagation.



requires $\neg_{Bool} X$ `in keys` $(C) \wedge_{Bool} CurrPcount >_{Int} Pcount$

Rule 4.12.4: Variable request reset.



Rule 4.12.5: Variable request resolution.

4.13 Time

Continuing on from Section 3.3, we now delve into the subject of timing. Let us point out the distinction between different kinds of Orc sites. Orc has *internal*, *external*, and *timer* sites. Internal sites are those that run locally and need zero time to respond, but do not involve time. External sites run on an outside server and may respond at any given time or may not respond at all. Timer sites run locally like internal sites, but they involve time. Let us elaborate.

Timer Sites. Timer sites—*Clock*, *Atimer* and *Rtimer*—are used for computations involving time. Since these are local sites, they respond at the exact moment they are expected to. So, *Rtimer*(*0*) responds immediately. *Atimer* and *Rtimer* are essentially the same except that the former takes an absolute value of time as an argument and the latter takes a relative value of time.[MC07]

- *Clock* publishes the current time.
- *Rtimer*(t) publishes a signal after t time units.
- *Atimer*(t) publishes a signal at time t.

4.13.1 The δ Function

Here we give a simple informal description of the 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 *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:

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

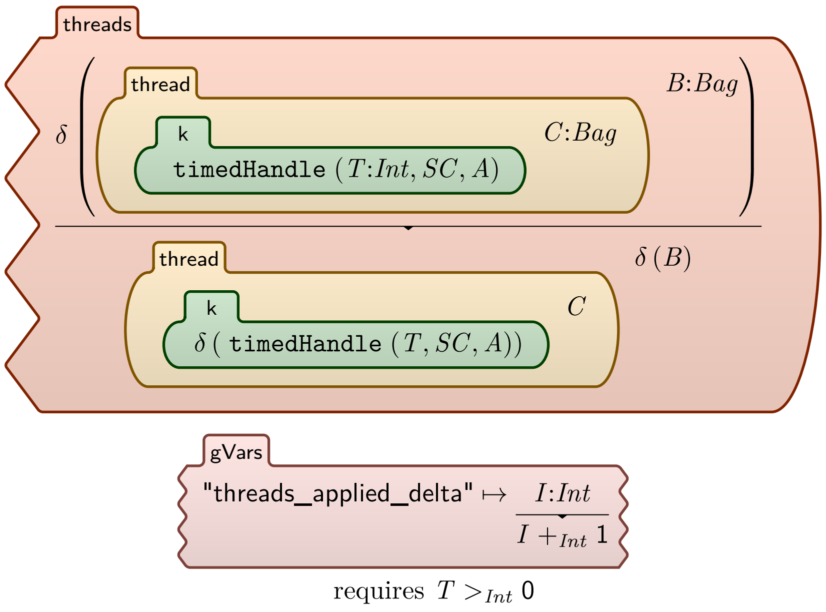
Therefore, the semantics of the *Rtimer* site, and any timed site, is only realizable through the δ function. When δ successfully runs on the whole environment, it is said to have completed one tick.

4.13.2 Implemented Time Modules

We implemented two different modules for time. The first applied δ sequentially which is one reason why we implemented a second time module that has the potential to apply δ concurrently. We called them `ORC-TIMESEQ` and `ORC-TIME` respectively. Since the latter is the preferred module, we explain it thoroughly in this section, but start with an overview of the former.

Sequential Time Module

The first of the two time modules implemented is called `ORC-TIMESEQ`; it tries to follow the model more literally than the other. Here we explain key rules whereas the whole module can be found in Appendix A.4.2. This module applies δ to the whole environment and then processes threads one by one. When it finds a thread with a timed site, it applies δ to it without processing it, which is the role of Rule 4.13.1. After that, Rule 4.13.2 executes δ on the thread by decrementing the time value held in its `timeHandle`. Both these rules keep count of threads that applied δ. The count is used to determine if all timed threads applied δ, which is when Rule 4.13.3 ticks the environment's clock.

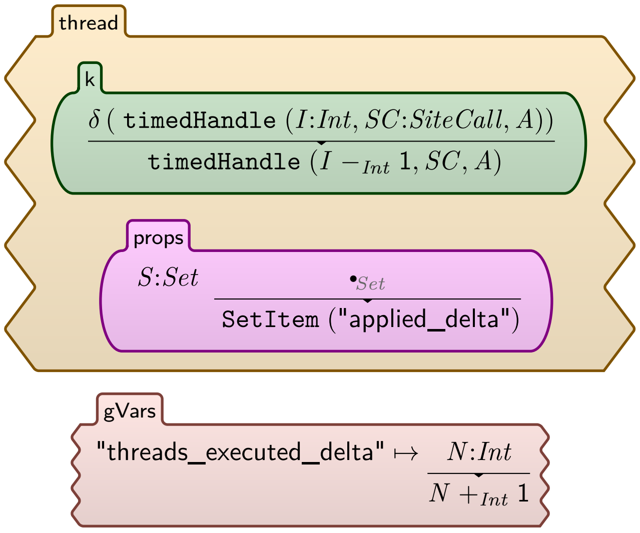

threads
thread
k
δ
timedHandle $(T{:}Int, SC, A)$
$C{:}Bag$
$B{:}Bag$
thread
k
$\delta\,($ timedHandle $(T, SC, A))$
C
$\delta\,(B)$
gVars
"threads_applied_delta" $\mapsto$ $\frac{I{:}Int}{I +_{Int} 1}$
requires $T >_{Int} 0$

Rule 4.13.1: Sequential δ applied to a single thread.

Concurrent Time Module

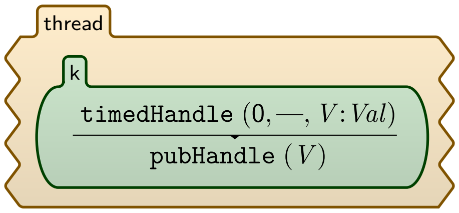
This module named `ORC-TIME` is a simpler and more straightforward module for time. On top of that, it is closer to the followed model in that the clock tick occurs before delta is applied, not after. It also makes perfect use of the predicates (Section 4.14). Observe the use of predicates in the side condition of Rule 4.13.4, the first rule of this module, to guarantee the priority explained in Section 4.9. This rule ticks the clock and turns on a flag called `time.ticked`. This flag enables Rule 4.13.5 to process `timeHandle`'s. This can apply concurrently on all such threads even though multiple instances of this rule will all have to read the `gVars` cell, because $\mathbb{K}$ allows for concurrency with shared reads. After all timed threads are processed, Rule 4.13.6 turns the `time.ticked` flag back off.

This whole operation repeats as long as `timeHandle`'s are in the environment. Once the timer on one of these handles reaches zero, Rule 4.13.7 rewrites it into


thread
k
δ (timedHandle (I:Int, SC:SiteCall, A))
timedHandle (I −Int 1, SC, A)
props
S:Set
•Set
SetItem ("applied_delta")
gVars
"threads_executed_delta" ↦
N:Int
N +Int 1

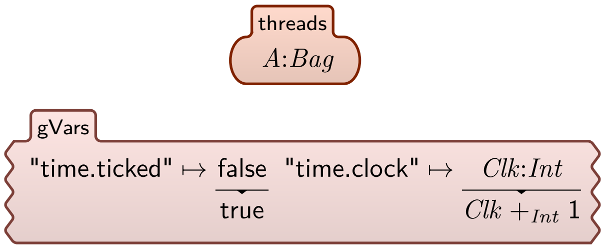
requires $I >_{Int} 0 \wedge_{Bool} \neg_{Bool}$"applied_delta" `in` S

Rule 4.13.2: Sequential δ processed.



Rule 4.13.3: Clock tick after sequential δ is applied.

a `pubHandle` where it can be further processed by the `ORC-PUB` module.



$$\begin{array}{c}\text{requires } \mathit{Clk} <_{\mathit{Int}} \mathit{TL} \\ \wedge_{\mathit{Bool}} \neg_{\mathit{Bool}} (\ \texttt{anySiteCall}\ (A)) \\ \wedge_{\mathit{Bool}} \neg_{\mathit{Bool}} (\ \texttt{anyFreeHandle}\ (A)) \\ \wedge_{\mathit{Bool}} \neg_{\mathit{Bool}} (\ \texttt{anyPubHandle}\ (A)) \\ \wedge_{\mathit{Bool}}\ \texttt{anyTimedHandle}\ (A) \\ \wedge_{\mathit{Bool}} \neg_{\mathit{Bool}} (\ \texttt{anyAppliedDelta}\ (A))\end{array}$$

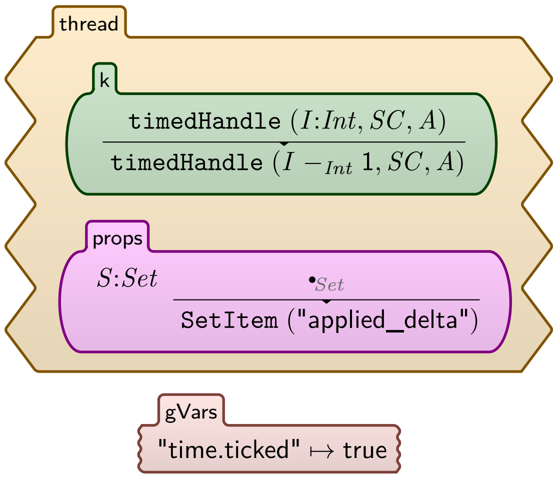
Rule 4.13.4: Clock Tick.

4.14 Predicate Functions

The predicate functions, defined in the `ORC-PREDICATES` module, are meta functions used in side conditions of certain rules to ensure the correct order of applying those rules. They do so by checking the environment for threads carrying certain features. Following is a list of these functions, their rules, and what each of them is checking in the environment:

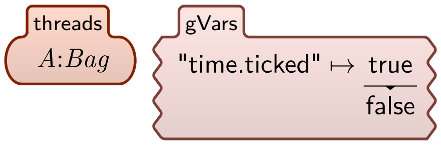
- Rule 4.14.1, `anySiteCall`: Any thread that has a site call that hasn't been made.
- Rule 4.14.2, `anyFreeHandle`: Any thread that has an unprocessed handle.
- Rule 4.14.3, `anyPubHandle`: Any thread that has a handle ready to publish a value.
- Rule 4.14.4, `anyTimedHandle`: Any thread that has a timed handle.
- Rule 4.14.5, `allAppliedDelta`: All threads have applied δ.
- Rule 4.14.6, `anyAppliedDelta`: Any thread that has the `applied_delta` flag on and is not reset.

Note that these rules come in pairs of two, one that returns *true*, for instance, if the thread is found and another that returns *false* otherwise. The 'otherwise' counterparts were omitted here but are shown in Appendix A.9.


thread
k
timedHandle (I:Int, SC, A)
timedHandle (I −Int 1, SC, A)
props
S:Set
•Set
SetItem ("applied_delta")
gVars
"time.ticked" ↦ true

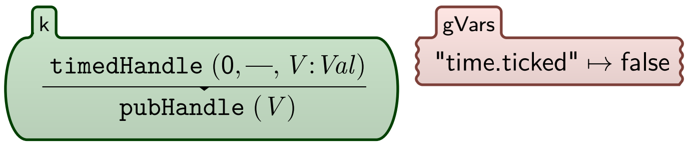
requires $I >_{Int} 0 \wedge_{Bool} \neg_{Bool}($"applied_delta" in $S)$

Rule 4.13.5: Apply δ.

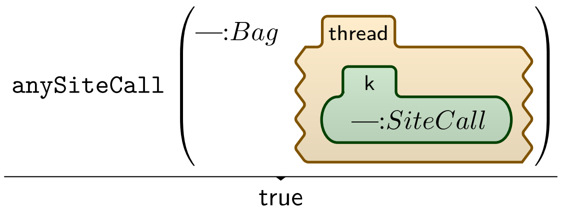


requires allAppliedDelta (A)

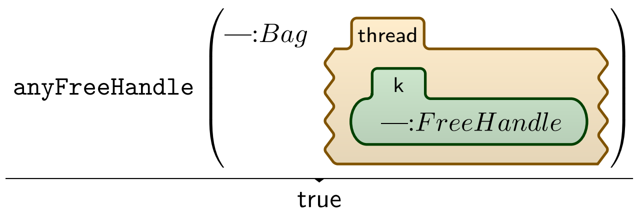
Rule 4.13.6: δ applied.



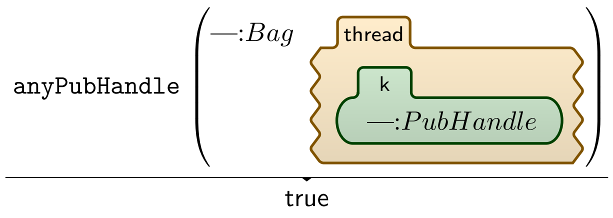
Rule 4.13.7: Timed handle done.



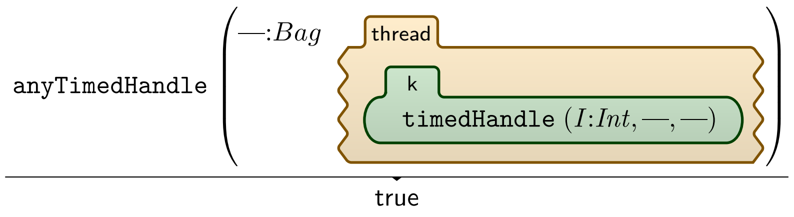
Rule 4.14.1: Match any thread that has a site call that hasn't been made.



Rule 4.14.2: Match any thread that has an unprocessed handle.

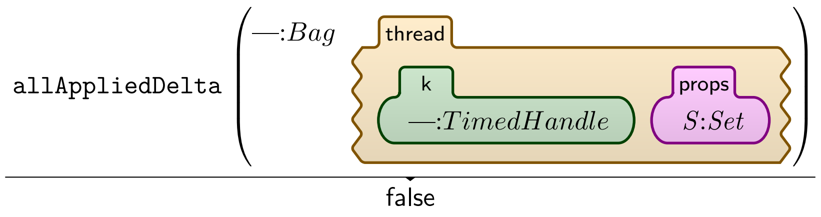


Rule 4.14.3: Match any thread that has a handle ready to respond with a value.



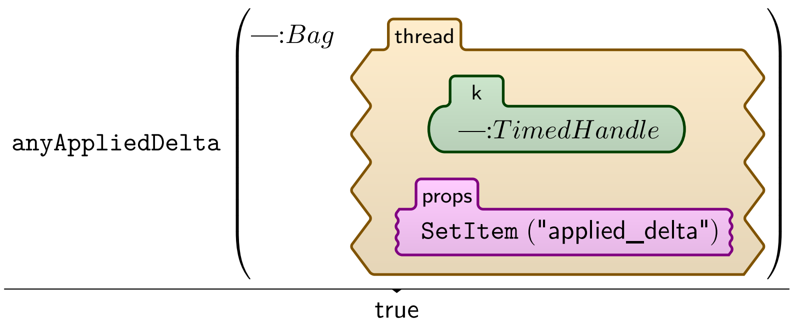
requires $I >_{Int} 0$

Rule 4.14.4: Match any thread that has a timed handle.



requires $\neg_{Bool}$"applied_delta" in S

Rule 4.14.5: Match any thread that hasn't yet applied δ.



Rule 4.14.6: Match any thread that has the `applied_delta` flag on and is not reset.

4.15 Expression Definition and Expression Calls

4.15.1 Expression Definitions

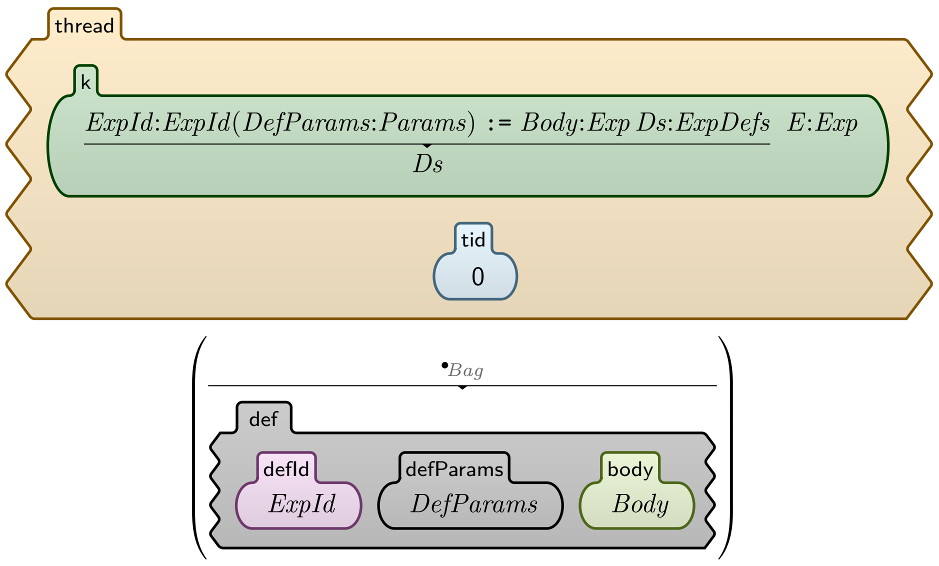
Orc allows for defining expressions in a program before the main Orc expression. Expression Definitions are parsed into the *ExpDefs* syntactic category seen in the syntax in Section 4.1. That is defined as a list of expressions definitions, each of which is expanded further in the syntax. Rule 4.15.1 takes that whole production of a single expression definition, creates for it a new `def` cell, and places each part of it into the appropriate cell inside that `def`. That rule applies recursively, once for each definition, until the list of expression definitions is empty. That is when Rule 4.15.2 discards the empty list and allows the main Orc expression to be on the top of the computation so that other rules can do their work.

4.15.2 Expression Calling

When an expression call is encountered, the identifier, that is the expression name, is matched with the ones stored in the `defs` cell. If it is not found then the program gets stuck. If it is, then the defined body substitutes the call while the arguments substitute the parameters in that body. The substitution is done using 𝕂's builtin substitution module. The substitution is done through four rules that are too technical to list here, but can be found in Appendix A.3.

4.16 Testing and Validation

Having the goal of formal verification to ensure soundness for safety-critical applications, ideally, we would like to formally prove the correctness of our 𝕂 specification. However, formal validation is not possible because there is no formal semantics to validate ours against. We are basing our work directly


thread
k
ExpId:ExpId(DefParams:Params) := Body:Exp Ds:ExpDefs   E:Exp
Ds
tid
0
•Bag
def
defId
ExpId
defParams
DefParams
body
Body

Rule 4.15.1: Expression Definition Prep.

$$\underbrace{\bullet_{ExpDefs}}_{E}\ \ E{:}Exp$$

Rule 4.15.2: Expression Definition End.

on an informal semantics. That being said, it is still in our interest to build sufficient confidence in the correctness of our work. We do so through running tests which are sample Orc programs that test different elements of Orc, and different arrangements of orchestrations. Other works using $\mathbb{K}$, some of which we reviewed in Section 6.3 have solely depended on this approach to build confidence in the correctness of their semantics; these are the semantics of C [ER12b][gita], Java [BR15][gitb], RISC Assembly [As4], Python [gitc] and Verilog [MKMR10].

Table 4.3 is a summary of all test programs we have run. For the full details of these programs, see Appendix B. Most of them are basic examples with the aim of testing individual features, and some test subtle principles and delicate mechanics. Chapter 5 demonstrates and discusses the results of running selected key examples in the $\mathbb{K}$ tool.

	Orc Features										
Test Examples	Parallel	Sequential	Pruning	Otherwise	LTL Model Checking	Expressions	Recursion	Time	Synchronization	Variable Lookup	Scope
B.1.1	√										
B.1.2	√	√									
B.1.3	√	√									
B.1.4	√	√									
B.1.5	√	√									
B.1.6	√	√									
B.1.7	√		√								
B.1.8	√		√								
B.1.9	√	√	√								
B.1.10			√								
B.1.11			√								
B.1.12			√								
B.1.13			√								√
B.1.14				√							
B.1.15				√							
B.2.1			√		√						
B.2.2	√				√						
B.2.3	√	√			√						
B.3.1						√					
B.3.2						√					
B.3.3						√					
B.3.4– B.3.9						√					√
B.3.10						√	√				
B.4.1								√			
B.4.2								√			
B.4.3								√			
B.4.4								√	√		
B.5.1–B.5.14								√	√	√	
B.5.18								√	√	√	
B.5.19						√	√	√	√	√	
B.5.20						√	√	√	√	√	
B.5.21						√	√	√	√	√	
B.5.22					√	√	√	√	√	√	
B.5.23					√	√	√	√	√	√	
B.6.1–B.8.13					√					√	√

Table 4.3: Summary of test programs in Appendix B and what features of Orc they test.

CHAPTER 5

Executing the Semantics

In this chapter, we show the results of executing the $\mathbb{K}$ semantics on sample Orc programs as well as running verification techniques on them, namely state space search and model checking. We first go through a case study that envelopes all elements of Orc and showcases the power of the $\mathbb{K}$ tool. Then, we discuss various examples that address technical and subtle points in our definition.

5.1 Case Study: Robot Movement

In this section, we present a case study 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 two-dimensional room with walls and possibly obstacles. The room consists of six tiles, 2 rows by 3 columns, where a crossed out tile represents an obstacle. The results of the executed programs in this case study will be shown as terminal output as well as visualized. We could of course work with a more complex environment whether through a bigger room or through controlling more than one robot; but the purpose here is a simple demonstration and a proof of the concept. Other reasons are discussed in Section 7.2.2.

5.1.1 Semantics of Robot Sites

Table 5.1 explains the semantics of the robot sites that we created for this case study. We also made each of these sites take a certain amount of time to respond; this was made as simulation of the time needed to make the physical movement. The amount time each site consumes is shown in the table as well.

Site	Time	Semantics
stepFwd()	3	Causes the robot to move a distance of one tile in the direction it is facing.
rotateRight()	1	Rotates the robot 90 degrees clockwise.
rotateLeft()	1	Rotates the robot 90 degrees counterclockwise.
scan()	1	Scans the tile directly in front of the robot for obstacles.
mapInit(<Length,Width>)	0	Initializes a room with the given dimensions in tiles.
init(Position, Direction)	0	Places a robot in the given position facing the given direction.
setObstacles(Locations)	0	Places obstacles in the given location on the map.
`is_bumper_hit`	–	Not a site, but a flag that turns on when the robot bumps into a wall or an obstacle. It turns off just as the robot attempts the next move.

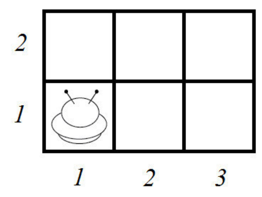
Table 5.1: Semantics of Orc sites that control the robot and the time needed by each site.

Hitting an obstacle while trying to move forward will still consume the time needed for the full movement, but will turn on a flag called *isBumperHit* which will reset on the next action.

Moreover, locks have been implemented in sites that move the robot. This is a simulation of a motor responsible for moving the robot. Calling a site that causes movement will first try to reserve the motor. After the movement is done, the motor is released.

5.1.2 Running the Programs

Let us now start the simulation with simple examples and increase the complexity as we go through the rest. Our aim is not to increase complexity simply through bigger examples, but to reach just enough complexity such that nondeterminism is introduced, and to show the results of running formal verification tools on a such a simple environment.



(a)

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

(b)

Figure 5.1: Result of Example 5.1.1.

Example 5.1.1. The following program will initialize the robot environment and place a robot in the default position <1,1> facing the default direction which is north. The result of running this program is shown in Figure 5.1.

```
  bot.mapInit() >> bot.init()
```

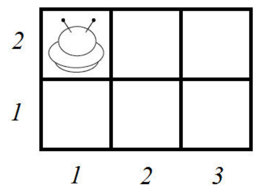
Example 5.1.2. In this example, we simply move the robot one step forward. The result is shown in Figure 5.2.

```
  bot.mapInit() >> bot.init() >> bot.stepFwd()
```

Example 5.1.3. Now let us place an obstacle in front of the robot and command it to move forward. Running this program will cause the robot to bump into the obstacle, resulting in a state shown in Figure 5.3.

```
  bot.mapInit() >> bot.init() >> bot.setObstacles(<1,2>) >>
      bot.stepFwd()
```

Notice that the robot has consumed the time needed for the movement even though it wasn't successful in moving to the adjacent tile because of the obstacle. Also notice that the `is_bumper_hit` flag is `true`. Here, we can try out LTL model checking for the same program by running it using the `--ltlmc` option and giving a LTL formula like so:



(a)

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

(b)

Figure 5.2: Result of Example 5.1.2.

```
krun program.orc --ltlmc "<>Ltl gVarEqTo(\"is_bumper_hit
    \",true)"
```

The LTL formula starts with a LTL operator `<>Ltl` which means *eventually*. The function `gVarEqTo` is defined in our `ORC-LTLMC` module and it obviously takes two arguments, a flag in the cell `gVars` and a value to check. The function retrieves the flag specified and passes it to 𝕂's LTL model checker which then checks if that flag is *eventually* going to be `true`. In this case, the flag is eventually going to be true in any case, and so the command returns `true`. If however, the model checker detected nondeterminism which leads to multiple execution paths, and it found that in one path, the flag does not eventually become `true`, then it displays that path to the user as a counter example to prove that the formula does not hold.

Example 5.1.4. In this example, we analyze a slightly larger program. In it, two Orc expressions are defined. The first one, called `ChangeLane`, is the key component that introduces an element of nondeterminism. Curiously enough, it tries to perform two movements in parallel. The movements effectively aim to strafe the robot to the right and to the left respectively. But because only one of these two movements can reserve the robot's motor, only one of them will be executed every time this expression is called. The question of "Which one?" is what creates nondeterministic behavior.

The next defined expression in the program is `SmartStep`. It uses the site

```
<gVars>
  "BotVars" |->
    "direction" |-> (0,1)
    "position" |-> (1,1)
    "is_bumper_hit" |-> true
  "clock" |-> 3
</gVars>
```

Figure 5.3: Result of Example 5.1.3.

`if` as explained in Example 2.5. Its behavior is that it checks the tile ahead for obstacles; if it does not find one, it steps into that tile; if it finds one, it tries to avoid it by calling `ChangeLane` which we just explained.

The main expression sets up the environment to look like in Figure 5.4a, then simply calls `SmartStep`.

```
ChangeLane(b) :=
(
  (bot.rotateRight(b) >> bot.stepFwd(b) >> bot.rotateLeft(
      b))
  | (bot.rotateLeft(b) >> bot.stepFwd(b) >> bot.
      rotateRight(b))
)

SmartStep(b) :=
bot.scan(b) > isBlocked >
  (
    if(isBlocked) >> ChangeLane(b)
    | ifNot(isBlocked) >> bot.stepFwd(b)
  )

bot.mapInit() >>
bot.setObstacles(<2,2>) >>
bot.init(<2,1>,<0,1>) > x > SmartStep(x)
```

The expected behavior from running this program is that the robot ends up in either of the positions shown in Figure 5.4b. But what if we want to explore all the different execution paths that this program may follow. Here comes 𝕂's state search tool. Using the option `--search` in the command gives us one final configuration for each possible execution path. `search` can also display configurations at a certain depth if needed instead of exploring the whole tree. A `pattern` can be provided to the `search` command to specify what kind of configuration we are looking for. Here, we give a general pattern that says we're interested in the contents of the `gVars` cell. We run the program with the following command:

```
krun program.orc --search --pattern "<gVars> M:Map </gVars
   >"
```

Running this command will result in the output shown in Figure 5.4c. It displays two solutions, each representing a possible final configuration. Solution 1 shows the robot having switched to the left lane, while solution 2 shows it at the right lane.

Next, we use LTL model checking to verify that the robot never bumps into an obstacle or a wall. We use the following command:

```
krun program.orc --ltlmc "[]Ltl gVarEqTo(\"is_bumper_hit
   \",false)"
```

We use the LTL operator `[]Ltl` which means *always*. This returns `true` which means that along all execution paths, the flag `is_bumper_hit` is always `false` (during the whole execution).

In the next example, we use the same program but with a different setting.

Example 5.1.5. In this example, we use the same program as in Example 5.1.4, but we change the setting such that the robot is forced to bump into something. The initial setting is shown in Figure 5.5a. Like the previous example, the robot will nondeterministically choose between switching lanes to its right and to its left. However, this time, the robot will, in one of two cases, hit its bumper into the obstacle to its left. The program is shown below without repeating the defined expressions `ChangeLane` and `SmartStep`.

```
bot.mapInit() >>
bot.setObstacles(<1,1>,<2,2>) >>
bot.init(<2,1>,<0,1>) > x > SmartStep(x)
```

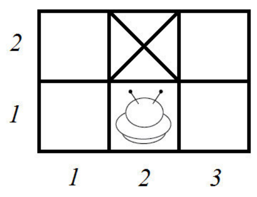
We run the program with the same command as Example 5.1.4:

```
krun program.orc --search --pattern "<gVars> M:Map </gVars
   >"
```

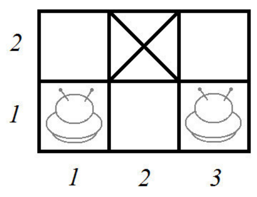
Running this command will result in the output shown in Figure 5.5b which has two solutions. Solution 1 shows the robot in its initial position, while solution 2 shows it at the right lane. We have already seen solution 2 in Example 5.1.4 and we expect it, but solution 1 doesn't provide enough information as to whether the robot hit its bumper during its movement or not. We know it performed some actions because the `clock` is at 6 time units, but the `is_bumper_hit` flag is `false`. That is because we programmed it such that it resets before performing any new move. So we will use LTL model checking to verify whether it bumped or not. We give the following command:

```
krun program.orc --ltlmc "[]Ltl gVarEqTo(\"is_bumper_hit
   \",false)"
```

Now if it has never hit, this command should return `true`, yet instead, it returns a trace of configurations that represent an execution path in which the



(a)



(b)

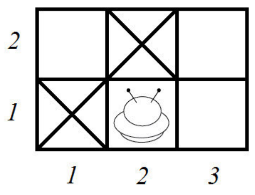
```
Solution 1:
<gVars>
  "BotVars" |->
    "direction" |-> (0,1)
    "position" |-> (1,1)
    "is_bumper_hit" |->
        false
  "clock" |-> 6
</gVars>
```

```
Solution 2:
<gVars>
  "BotVars" |->
    "direction" |-> (0,1)
    "position" |-> (3,1)
    "is_bumper_hit" |->
        false
  "clock" |-> 6
</gVars>
```

(c)

Figure 5.4: Result of Example 5.1.4.

robot has indeed hit its bumper. The trace it shows ends at the configuration of interest where we find that `is_bumper_hit` is indeed `true`. The whole trace is too lengthy to list here. So Figure 5.5c shows only that mid-execution configuration of interest—the relevant part of it.



(a)

```
Solution 1:
<gVars>
  "BotVars" |->
    "direction" |-> (0,1)
    "position" |-> (2,1)
    "is_bumper_hit" |->
        false
  "clock" |-> 6
</gVars>
```

```
Solution 2:
<gVars>
  "BotVars" |->
    "direction" |-> (0,1)
    "position" |-> (3,1)
    "is_bumper_hit" |->
        false
  "clock" |-> 6
</gVars>
```

(b)

```
<gVars>
  "BotVars" |->
    "direction" |-> (-1,0)
    "position" |-> (2,2)
    "is_bumper_hit" |-> true
  "clock" |-> 5
</gVars>
```

(c)

Figure 5.5: Result of Example 5.1.5.

5.2 Various Key Programs

This section shows unique Orc programs that test certain technicalities and features.

Example 5.2.1. Variable Scope
Here is an orchestration made out of three pruning combinators and several calls to the site `Add`. This tests variable lookup and scope sharing and ensures that no incorrect scope sharing is occurring. Notice how the parentheses are

arranged such that the parse tree would be like shown in the code's comments. The topmost node in the tree is `prunMgr(f1)`, i.e., the thread managing the pruning expression whose center variable is `f1`. Notice the positions of the other two managers in the tree. `prunMgr(f1)` will simply take the result from its right-side child adding 1 and 1, and will bind `f1` to it in its left-side child `prunMgr(f3)`. Now, `prunMgr(f2)` is trying to resolve the variable `f1` in order to call the site adding `1` to `f1`. It does not have `f1` in its context, so it will request it from its parent which indeed has `f1` in its context mapped to `2`. After `prunMgr(f2)` resolves the variable `f1` and after its right-side child publishes, it binds `f2` to the published value `3` in its left-side child which is adding `f1` and `f2`. Now it is `prunMgr(f3)` who will finally bind its variable to the value published by its right-side `5`. Its left-side child requires `f2`, so it will request it from its parent, but because `f2` is not in its scope, that variable request will never be resolved and the program will get stuck right there.

This example demonstrates how our variable lookup system works and how the request-from-parent system ensures each thread's scope remains correct.

This is also available in the appendix as Example B.1.13.

```
(Add(f2,f3) < f3 < (Add(f1,f2) < f2 < Add(f1,1))) < f1 <
    Add(1,1)

/* Here is how it is structured

                             prunMgr(f1)
                             /         \
                        prunMgr(f3)    1+1
                       /         \
  This f2 --------> f2+f3        prunMgr(f2)
  is stuck.                     /          \
  It should not              f1+f2         1+f1
  see the value
  provided by
  prunMgr(f2).

*/
```

Example 5.2.2. Factorial

This Orc program defines the factorial function and helps us observe its equivalent in Orc calculus. It uses a few mathematical sites that we defined in the semantics, particularly in the `ORC-MATH` module. Notice the calls of `if` in parallel. As we pointed out in Example 2.5, this is how **if** b **then** f **else** g is expressed in Orc. The main point here however is demonstrating the use of recursion.

This is also available in the appendix as Example B.3.10.

```
// Define factorial
Factorial(x) := (((if(r) < r < Equals(x,0)) >> 1) | ((if(r
    ) < r < Gr(x,0)) >> (Mul(a,x) < a < (Factorial(b) < b <
```

```
        Sub(x,1)))))

  // Print factorial(5)
  Factorial(5) > f > print(f)
```

Example 5.2.3. This example is taken directly from Misra's work [MC07] where it demonstrated the use of timer sites. It is used here to test timer sites; but more particularly, it test a subtle point in the semantics which is that publishing has priority over passing time. So if two threads are running in parallel, one that is ready to publish and the other is just waiting for time to pass, then time should never pass until the publish is done. Likewise, site calling has priority over passing time, and this example tests that as well.

To understand how this is tested, notice the sites `count.inc()` which increments a certain count, and `count.read()` which reads the value of that count. The program starts by publishing three numbers. These could be any numbers. The point is to call `count.inc()` three times because, for each publish, the sequential combinator $\gg$ will instantiate a thread that will increment the count. Each of these three instances will do its work and then publish a `signal` which in turn calls the site `zero` which does nothing.

In parallel with all of that runs a thread with `Rtimer(1)`. Its job is to simply publish a `signal` once one time unit has passed, after which the site `count.read()` will be called and would publish the current value of the count.

The final value of the count should be equal to the number of publishes done at the very beginning, which is three, exactly three—well, nothing could stress that more than this famous passage:

> "First shalt thou take out the Holy Pin, then shalt thou count to three, no more, no less. Three shall be the number thou shalt count, and the number of the counting shall be three. Four shalt thou not count, neither count thou two, excepting that thou then proceed to three. Five is right out. Once the number three, being the third number, be reached, then lobbest thou thy Holy Hand Grenade of Antioch towards thy foe, who being naughty in my sight, shall snuff it." — *Monty Python and the Holy Grail (1975)*

Thankfully we got that out of the way as it was most assuredly not intended. Now the goal of the test is to assure that time does not pass before all three publishes are done, nor even before the calls to `count.inc()` are made. This can be tested with LTL model checking using the command `krun --ltlmc "<>Ltl isGPublished(3)"` which means that this must *eventually* publish `3`.

This is also available in the appendix as Example B.4.4.

```
(1|1|1) >> count.inc() >> zero() | Rtimer(1) >> count.read
    ()
```

5.3 Conclusion

This chapter demonstrated the potential of exploiting $\mathbb{K}$'s state search capabilities and the power of its LTL model checking for purposes of formal verification. It also showed and discussed a few key examples so that we can see some technicalities of our implementation. For more examples, see Appendix B; it has many more examples with results detailed, some of which involve state searching and model checking.

Chapter 6

Related Work

In this chapter, we cover related works done on Orc as well as on other service composition frameworks regarding modeling and verification as well as composition techniques. We also review selected semantic definitions that were done using $\mathbb{K}$.

6.1 Service Composition

Service orchestration, explained in Section 2.1.1, is but one specific approach to applying service composition, which is simply composing services to form other services. In this section, we review other approaches to service composition pointing out which are pure formalisms. Then we discuss some verification techniques done on those formalisms.

6.1.1 Approaches to Service Composition

Service composition, as the name suggests, is when one or more services are needed for a bigger task and thus are utilized to compose a bigger service. Service composition is an active research field with many publications addressing different problems. For example, the problem of which services to select is well-covered in the literature. Works such as [OMF14] focus on the best estimation of Quality of Service (QoS). Other works focus on how to prioritize services given their QoS, some on general service composition [DBS15, PSFRC14, PSFRC14], and some specifically on service orchestration [RBJ12].

A more relevant area of research is the different approaches to service composition. Some of these approaches are based on Artificial Intelligence techniques [ZSPZ15, VBS15, WYM15, WZ13, FS14, ZGCZ14]. Some other approaches are based on Petri-nets [CBC12, TJZ11, VECDM09, CL08].

In [cF14], Țuțu and Fiadeiro developed a relational variant of logic programming based on Horn-clause logic programming to give a service orchestration semantics. They introduced a new categorical model of service orchestrations they called *orchestration scheme*.

Aside from orchestration, one well-known formalism is called *service choreography*. A significant work that advanced formal methods research about service choreography is that of Su et al. in [SBFZ08] who formalized a theory for service choreographies. This is significant because until then, service choreography had not had a well-defined set of rules and concepts [cF14].

Also worthy of mention here is the Service-Centered Calculus (SCC) [BBC+06].

Recent surveys on different service composition approaches can be found in [JSO14, SQV+14, GMM15].

Service-based computations

In [GSS13], Gabarró et al. propose a way to analyze the behavior of service-based computations "in a realistic way". They argue that in under realistic conditions, like under mobile network stress, the demand for a particular service can fluctuate—that a service can be attractive and acquire more users if it has a high QoS which in turn will put it under stress with which it will fail to deliver its QoS. They use game theory's Nash equilibria to model and analyze this behavior. As a continuation of [GSS13], Gabarró et al. in [GGS14] and later in [SGK15], where they finally composed a rather amusing title for their work, went on to build uncertainty profiles for QoS fluctuations, also using Nash equilibria, and using Orc to model uncertain cloud environments.

Others used BPEL-WS as a service composition framework and a service behavioral description language to analyze the computational complexity of description-based compositions [NKL11].

BPEL4WS [JEA+07]

For the sake of coverage, we present an overview of BPEL4WS, a prominent service orchestration framework, and then we contrast it with Orc. BPEL4WS (Business Process Execution Language for Web Services) builds on two main features inspired by two frameworks: the feature of directed graphs taken from IBM's WSFL (Web Services Flow Language); and the feature of a block structured language taken from Microsoft's XLANG (Web Services for Business Process Design). The main concept by which BPEL would be understood is the division of processes into two types: executable and abstract. BPEL supports the modeling of the following two types.

- *Abstract processes* which are not executable. An abstract process is a business protocol that specifies the behavior of messaging between different peers without revealing the internal behavior of any of them.

- *Executable processes* which specify four things:
  - the order in which activities constituting the process are executed,
  - the peers that the process involves,
  - the messages those peers exchange,
  - and exception handling.

In BPEL4WS, A process consists of activities which are either primitive or structured. Observe the likeness to Orc's sites and combinators in these activities. Primitive activities comprise the following:

- **invoke:** invokes a web service.
- **receive:** waits for a message from an external source.
- **reply:** replies to an external source.
- **wait:** waits for a given amount of time.
- **assign:** copies data from one place to another.
- **throw:** throws errors and exceptions.
- **terminate:** terminates the entire service instance.
- **empty:** does nothing.

Structured activities are used to combine simple simple structures into more complex ones, just like Orc's combinators. Structured activities comprise the following:

- **sequence:** puts execution in a sequential order.
- **switch:** used for conditional routing.
- **while:** the while loop.
- **pick:** used for race conditions decided by timing or external triggers.
- **flow:** used for parallel routing.
- **scope:** groups activities under a certain exception handler.

How BPEL Compares to Orc. In contrast with Orc, an executable process can be seen as a site call, whereas an abstract process can be seen as an expression call. Note that expressions in Orc could contain site calls, so abstract processes might contain executable processes, which is totally conformant with BPEL's definition of abstract processes. However in Orc, the concept of an abstract process is even more abstract than in BPEL in that Orc allows for seamless restructuring and evaluation of the contents of an abstract process using its unique concurrency combinators.

Moreover, all activities defined in BPEL, primitive or structured, are included in Orc, either explicitly as local (built-in) sites, or implicitly as the inner-workings of the concurrency combinators.

We can see that Orc is simpler and more intuitive than BPEL4WS.

6.1.2 Verification of Service Composition Frameworks

Using timed automata-based approaches for verification of service compositions has been done many times in recent years. For instance, it was used in verification of service choreographies [DPC$^+$06, ECDVM11]. In another work, a new model called the Enhanced Stacked Automata Model (ESAM) has been used to express and verify BPEL4WS service compositions [NEKN15]. The verification was of properties like dead transition, deadlock, liveness and reachability, and was done through the SPIN tool [spi] that we mentioned in Section 2.2.5.

6.2 Orc-related

This section discusses the work most significantly related to ours. First we preview different formal semantics of Orc, and then we review in more detail efforts with the same goal as ours, i.e., to formally analyze and verify Orc programs specifically. We then compare them with this thesis' work summarizing the comparison in a nice table.

6.2.1 Formal Semantics of Orc

Apart from the SOS semantics of Orc given in [MC07], its timed SOS extension given in [WKCM08], and its later transaction-executing extension, Ora [Kit13], several denotational formalizations of Orc's semantics have been developed [HMM05, KCM06, RKB$^+$08, WKCM07, LZH10]. In [DNMT12], De Nicola et al. introduced a new formalism combining concepts and primitives from two calculi: Orc and Klaim [BBDN$^+$03]. These denotational formalizations are not of much interest to us because they do not describe operational behavior, and thus cannot be executed. The most comprehensive up-to-date work on Orc is the yet-to-be-published book by Misra [Mis14], which gives a thorough treatment of the Orc semantics, the Orc language, and its powerful features for structured concurrent programming.

Token semantics of Orc

In [Thy11], Thywissen presents what is called a token semantics. It is based on SOS with two modifications: (1) environments are carried in tokens, and (2) values are carried in tokens, keeping the program's text reserved instead of reduced to a value. This makes the semantics free of rewriting.

Concurrent Semantics of Orc

A relevant work was done by Perrin et al. who in [PJM13] introduced the idea behind two semantics they constructed for Orc: one they called an instrumented semantics which extracts as much information as possible in a single execution; and the other is a concurrent semantics which describes all possible behaviors of the program. Their plan is to use these two to design a debugger for Orc that can replay execution scenarios and provide tools for root cause analysis and race conditions. The semantics were published later in [PJM15], but the debugger is yet to be published.

6.2.2 Verification of Orc

Dong et al., 2014 [DLSZ14]

The work presented by Dong et al. in this paper is very relevant to ours. They wrote a complete semantics of Orc with the aim of formal verification using two different approaches. The first was a Timed Automata written in Computational Tree Logic (CTL) and verified using UPPAAL [BDL04, BDL$^+$06, upp]. The second was Constraint Logic Programming (CLP) and was verified using CLP(R) [JMSY92]. Both specifications were based on the timed operational semantics of Orc provided in [WKCM08]. For the first approach, they gave a proof of the weak bi-simulation relation between their Timed Automata and the target semantics. For the second approach, they made a proof sketch of the weak bi-simulation between any finite state Orc program and their corresponding CLP program. Unfortunately, none of the two approaches has the capacity to preserve Orc's true concurrency, nor can they translate Orc programs automatically to their respective analyzable languages.

Moreover, they did formal verification on a certain Orc program checking for the following properties: "Will the program terminate on all inputs?" and "Is it deadlock-free?". However, verification using UPPAAL in this way has a major flaw in that it can only verify an Orc expression with a fixed number of threads, thus by limiting recursion.

The CTL part of this work follows what Dong et al. had proposed in [DLSZ06], an automata approach, i.e., writing Orc expressions as networks of timed automata, which would allow the use of UPPAAL as a model checking tool for timed automata models.

Defining Orc in Maude

In his PhD work [AlT11], and later in [AM15], Alturki translated the Structural Operational Semantics (SOS) of Orc into the corresponding rewrite theory, which he wrote in Maude as a complete specification of Orc. He then used Maude as a formal analysis tool for Orc programs. To avoid the inefficiency in execution inherent in the rules necessary for such a semantics in Rewriting Logic, he further developed an equivalent Reduction Rewriting Semantics that minimized the number and complexity of the needed rewrite rules. He then wrote Object-based semantics which improves on the previous two by introducing object-level concurrency which laid the foundation for a distributed deployment of Orc's rewriting specification. Despite being similar to our semantics in that both are rewriting based, the semantics of [AM15] considered a lower level abstraction where site and expression calls and site returns, in addition to publishing values, were all observable actions.

6.2.3 Summary and Comparison

Table 6.1 and Table 6.2 compare different definitions of Orc; namely Whermann's timed SOS [WKCM08], Dong's Timed Automata [DLSZ06, DLSZ14], Dong's Constraint Logic Programming [DLSZ14], Alturki's Rewriting Logic definition in Maude [AlT11], and finally, our $\mathbb{K}$ definition.

Table 6.1 starts by showing which features of Orc were implemented in each definition. It ignores certain technical features because we are comparing primarily theoretical definitions. For example, we don't mention variable lookup and scope. The TA definition has no specification of scope at all. In it, value passing is done primitively by giving unique names to variables. However, that is not part of the Orc calculus, rather it is a technical issue that could be overlooked.

Another important issue is how Orc programs are processed. In our work, we defined an executable semantics of Orc. With that, the $\mathbb{K}$ tool enables us to feed it raw Orc programs without any translation. The TA and CLP approaches do not have that. They have an extra layer of manual work they need to do in order to analyze an Orc program using their respective tools. And that also acts as the core reason why they have lower scalability.

6.3 Work done in $\mathbb{K}$

A Complete Java Semantics. After years of work, Bogdănaș and Roșu unveil the complete executable semantics of Java written in $\mathbb{K}$ in [BR15]. The paper discusses many details of Java that required complex rules to be written in $\mathbb{K}$. To validate their work, they developed their own test suite consisting of 840 Java programs. Even though they developed the test suite in tandem with their semantics, it is general enough that it can be used by any Java analysis project. Their $\mathbb{K}$ semantics passed all the tests.

Similar to Java's semantics, some real-life languages have been formalized in $\mathbb{K}$ to develop analysis and verification tools for, most notably, C11 [ER12a, HER15] and JavaScript ES5 [PSR15].

Additionally, several other partial semantics have been developed:

- Python [Dwi13].
- Scheme [MHR07].

Feature	Wh	TA	CLP	MOrc	𝕂Orc
	Definition				
Captured Features:					
Expression definition	●	●	●	●	●
Recursion	●	○	●	●	●
Time	●	●	●	●	●
Synchronization	●	○	○	●	●
Non-determinism	●	○	●	●	●
Tool support	○	◐	◐	●	●
Model construction of Orc programs:					
Systematic	●	●	●	●	●
Automatic	○	○	○	●	●
Model executable	○	◐	◐	●	●
Enables model checking	●	●	●	●	●

Table 6.1: Comparison of different definitions of Orc; Wh:Whermann SOS [WKCM08], TA: Dong's Timed Automata [DLSZ14, DLSZ06], CLP: Dong's Constraint Logic Programming [DLSZ14], MOrc: Alturki's Maude definition [AlT11], 𝕂Orc: This thesis' definition.
●: Fully Implemented ◐: Partially Implemented ○: Not Implemented

	Definition			
Feature	TA	CLP	MOrc	𝕂Orc
Constructed Orc models:				
Scalability	○	◐	●	●
Model checking capability	◐	●	●	●
Definition:				
Expressiveness	○	◐	●	●
Conciseness	○	○	◐	●
Maintainability	○	◐	●	●
Human-readability	●	○	◐	●
Comprehensiveness	◐	●	●	●
Abstractness	●	●	●	●
Modularity	◐	○	●	●
Closeness to abstract syntax	○	○	◐	●
Closeness to original SOS	○	○	●	●

Table 6.2: Comparing the strength of certain qualities of the different definitions of Orc in relation to ours; TA: Dong's Timed Automata [DLSZ14][DLSZ06], CLP: Dong's Constraint Logic Programming [DLSZ14], MOrc: Alturki's Maude definition [AlT11], 𝕂Orc: This thesis' definition.
○: Significantly Less ◐: Relatively Less ●: Almost Equal

- Haskell [Laz12].
- Verilog [MKMR10].
- A RISC Assembly [As4].
- LLVM assembly [EL12].
- An automobile OS [ZCO14].

Chapter 7

Conclusion

In this thesis, we presented a complete formal executable semantics for Orc constructed in the 𝕂 framework in what is the first time where 𝕂 is used to define either of what Orc is: a calculus for concurrent operations, and a service composition system. We also demonstrated, through a case study and a number of examples, how its executability may be used not just for interpreting Orc programs, but also for dynamic formal verification, such as model checking.

This semantics is distinguished from other operational semantics by the fact that it is not directly based on Orc's interleaving SOS semantics. It takes advantage of concurrent rewriting facilitated by the underlying 𝕂 formalism to capture its concurrent semantics; and it makes use of 𝕂's innovative notations to document the meaning of its various combinators.

We now discuss some of the limitations of this work and the future work that is needed to perfect this semantics and to utilize it for formal verification.

7.1 Limitations

7.1.1 Observable Transitions and State Space Explosion

When a program is simulated by 𝕂, it goes through a number of transitions dictated by the transition rules that applied. Rules can be structural or transitional (computational) as mentioned in Section 2.2.4. When a transition rule applies, we call that an observable transition, and it creates a new configuration. That makes the change made by that rule observable and traceable. We can see the trace of configurations using the state search tool in 𝕂.

We make most rules structural, because transitions make the simulation and analysis considerably slower. The rules that we choose to make into transitions are usually the ones that introduce nondeterminism, i.e., the ones that have the potential to fork the program into multiple paths of execution. We did that with the rules that move the robot, which is how we were able to produce multiple execution paths from the same program in Example 5.1.4.

Another rule that we made transitional was the publishing rule. That is to introduce nondeterminism in the question of which site of many called in parallel will publish first. However, nothing dictates that they can't publish at the same time, except when a pruning manager would take only one published value out of multiple. In that case, the only way to guarantee that all solutions would be explored is by making publishing a transition. However, in some places, this was undesirable because it would give permutations of all concurrent publishes and that exponentially increases the number of solutions. We got more than 1000 solutions in an example with a few sets of two to three concurrent publishes.

In this subject is a limitation and potential for future development to alleviate this problem. It is a well known problem called state space explosion. A possible future direction is using techniques such as equational abstraction or partial order reductions.

7.1.2 Threats to Validity

Untested Cases

Some parts of our test set shown in Appendix B was not designed with a tester's mentality and is prone to miss some cases. For example, most of our tests end by publishing a value. So the focus has been on publishing rather than on halting. In particular, nothing tests specifically Rule 4.6.5, because no test focuses on that specific part of an expression halting—although it is tested more broadly like in the factorial of Example 5.2.2. However, focusing all of the tests into publishing is very relevant to the general design, especially that not much of the semantics is dedicated to silent site calls. Testing halting will naturally receive a lot of attention when real time is introduced and site calls will then follow a time-to-live model as mentioned in Section 7.2.4. When sites timing out is the focus, halting will be the primary test subject.

So, such cases are natural and are certainly not unexpected.

That being said, this generally poses a threat to the validity of our semantics. On the other hand, what abates this threat is the underlying design that makes the whole specification highly modular; and the bigger the definition, the more valuable the modularity.

Human Error

Human error is always a threat. Even though $\mathbb{K}$ minimizes human error through its concise and expressive notation—which is made clear by the amount of detail it can express through a handful of rules as seen from Chapter 4—, it does not eliminate it. Throughout the period this semantics was being developed, some modules were completely rewritten from scratch. Central modules like publishing, time, the combinators, site-handling, etc., have all been rewritten at least once and modified several times over. The work seemed never ending and that is always the case with such projects. After all, the goal from this project and the whole of formal methods is only to reduce human error to naught. Let us keep trying then!

7.2 Future Work

Here we discuss the areas of our work which we perceived to have most potential for future development.

7.2.1 A Formal Comparison

Having our semantics not depend on any other formal semantics, it would be interesting to study formally how it relates to the existing SOS semantics or rewriting logic semantics.

7.2.2 Extending the Case Study

In the case study discussed in Section 5.1, we simulated only one robot moving around a small room. Currently, the module defining sites for robot simulation, the `ORC-ROBOT` module, allows for bigger rooms and for more than one robot to roam the environment; it even allows for them to collide. This is an interesting future direction for the case study, especially once state search is optimized for this specific application as mentioned in Section 7.1.1.

7.2.3 Making a Test Suite

The current set of test examples given in Appendix B could be made into a test suite but it needs to be better organized, better categorized and better documented. It needs an automatic testing script as well. And most importantly, it needs to cover more cases. For example, subsubsection 7.1.2 mentions one area where it needs to expand.

7.2.4 Real Time

Adopting real (dense) time into our definition in place of logical discrete time will greatly aid in precisely capturing the behavior of external sites, which is what Orc intended. To better explain the value of real time, consider the syntactic category, *SilentHandle*, which we created to express site calls that will remain silent, like *if*(*false*). Now this is an internal site so it should take zero time to respond, and that is suitable for immediately rewriting it into a *SilentHandle*. But for an external site, we can never know that it will remain silent forever. So with real time, a call to an external site should be tied to a certain time-out value after which a site call is considered silent, but after a certain time, it is called again and so on. This is much more precise in capturing the behavior of calling a site that is forever silent.

Bibliography

[alt] ARC | AltaRica Project. `http://altarica.labri.fr/wp/`.

[AlT11] Musab AlTurki. *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`.

[AM15] Musab A. AlTurki and José Meseguer. Executable rewriting logic semantics of Orc and formal analysis of Orc programs. *Journal of Logical and Algebraic Methods in Programming*, 84(4):505–533, July 2015.

[As4] Mihail Asăvoae. $\mathbb{K}$ semantics for assembly languages: A case study. *Electronic Notes in Theoretical Computer Science*, 304:111–125, 2014.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.

[BBC$^+$06] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, et al. *SCC: a service centered calculus*. Springer, 2006.

[BBDN$^+$03] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gianluigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. In Corrado Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer Berlin Heidelberg, 2003.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[BDL+06] Glenn Behrmann, Alexandre David, Kim G. Larsen, John Hakansson, Paul Petterson, Wang Yi, and Monique Hendriks. UPPAAL 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[BR15] Denis Bogdănaș and Grigore Roșu. 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.

[cAL+14] Traian Florin Șerbănuță, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roșu. The 𝕂 primer (version 3.3). In Mark Hills, editor, *Proceedings of the Second International Workshop on the 𝕂 Framework and its Applications (𝕂 2011)*, volume 304, pages 57 – 80. Elsevier, 2014.

[CBC12] Sofiane Chemaa, Faycal Bachtarzi, and Allaoua Chaoui. A High-level Petri Net Based Approach for Modeling and Composition of Web Services. *Procedia Computer Science*, 9:469–478, 2012.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, Secaucus, NJ, USA, 2007.

[cF14] Ionut Țuțu and José Luiz Fiadeiro. Service-oriented logic programming. *Logical Methods in Computer Science*, 2014.

[CL08] Y Chi and H Lee. A formal modeling platform for composing web services. *Expert Systems with Applications*, 34(2):1500–1507, February 2008.

[CPM06] William Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in Orc. In Paolo Ciancarini and Herbert Wiklicky, editors, *Coordination Models and Languages*, volume 4038 of *Lecture Notes in Computer Science*, pages 82–96. Springer Berlin / Heidelberg, 2006.

[cR12] Traian Florin Șerbănuță and Grigore Roșu. A truly concurrent semantics for the 𝕂 framework based on graph transformations. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 7562 of *Lecture Notes in Computer Science*, pages 294–310. Springer Berlin Heidelberg, 2012.

[cRM09] Traian Florin Șerbănuță, Grigore Roșu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305 – 340, 2009. Special issue on Structural Operational Semantics (SOS).

[DBS15] Deivamani Mallayya, Baskaran Ramachandran, and Suganya Viswanathan. An Automatic Web Service Composition Framework Using QoS-Based Web Service Ranking Algorithm. *The Scientific World Journal*, 2015, 2015.

[DLSZ06] Jin Song Dong, Yang Liu, Jun Sun, and Xian Zhang. Verification of computation orchestration via timed automata. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*, pages 226–245. Springer, 2006.

[DLSZ14] Jin Song Dong, Yang Liu, Jun Sun, and Xian Zhang. Towards verification of computation orchestration. *Formal Aspects of Computing*, 26(4):729–759, 2014.

[DNMT12] Rocco De Nicola, Andrea Margheri, and Francesco Tiezzi. Orchestrating tuple-based languages. In *Trustworthy Global Computing*, pages 160–178. Springer, 2012.

[DPC$^+$06] Gregorio Diaz, Juan-JosÃľ Pardo, MarÃŋa-Emilia Cambronero, ValentÃŋn Valero, and Fernando Cuartero. Verification of Web Services with Timed Automata. *Electronic Notes in Theoretical Computer Science*, 157(2):19–34, May 2006.

[Dwi13] Dwight Guth. *A formal semantics of Python 3.3.* PhD thesis, University of Illinois at Urbana-Champaign, 2013.

[ECDVM11] M. Emilia Cambronero, Gregorio DÃŋaz, ValentÃŋn Valero, and Enrique MartÃŋnez. Validation and verification of Web services choreographies by using timed automata. *The Journal of Logic and Algebraic Programming*, 80(1):25–49, January 2011.

[EL12] C Ellison and D Lazar. K definition of the llvm assembly language. 2012.

[ER12a] Chucky Ellison and Grigore Roșu. Defining the undefinedness of C. 2012.

[ER12b] Chucky Ellison and Grigore Roșu. An executable formal semantics of C with applications. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 533–544. ACM, 2012.

[FS14] Yong-Yi FanJiang and Yang Syu. Semantic-based automatic service composition with functional and non-functional requirements in design time: A genetic algorithm approach. *Information and Software Technology*, 56(3):352–373, March 2014.

[GGS14] Joaquim Gabarró, Alina Garcia, and Maria Serna. Computational Aspects of Uncertainty Profiles and Angel-Daemon Games. *Theory of Computing Systems*, 54(1):83–110, January 2014.

[gita] $\mathbb{K}$ framework — C semantics, GitHub. `https://github.com/kframework/c-semantics`.

[gitb] $\mathbb{K}$ framework — Java semantics, GitHub. `https://github.com/kframework/java-semantics`.

[gitc] $\mathbb{K}$ framework — Python semantics, GitHub. `https://github.com/kframework/python-semantics`.

[GMM15] V. Gabrel, M. Manouvrier, and C. Murat. Web services composition: Complexity and models. *Discrete Applied Mathematics*, 196:100–114, December 2015.

[GSS13] Joaquim. Gabarró, Maria Serna, and Alan Stewart. Analysing Web-Orchestrations Under Stress Using Uncertainty Profiles. *The Computer Journal*, July 2013.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roșu. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 336–345. ACM, 2015.

[HMM05] Tony Hoare, Galen Menzel, and Jayadev Misra. A tree semantics of an orchestration language. *Engineering Theories of Software Intensive Systems*, pages 331–350, 2005.

[isa] Isabelle/HOL. A Proof Assistant for Higher-Order Logic. `https://www21.in.tum.de/~nipkow/LNCS2283/`.

[JEA$^+$07] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11:11, 2007.

[JGH11] Mauro Jaskelioff, Neil Ghani, and Graham Hutton. Modularity and implementation of mathematical operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(5):75 – 95, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, May 1992.

[JSO14] Amin Jula, Elankovan Sundararajan, and Zalinda Othman. Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, 41(8):3809–3824, June 2014.

[Kah87] G. Kahn. Natural semantics. In FranzJ. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer Berlin Heidelberg, 1987.

[KCM06] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In *CONCUR 2006*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006.

[Kit13] David Wilson Kitchin. *Orchestration and atomicity.* PhD thesis, University of Texas at Austin, 2013.

[KPM08] David Kitchin, Evan Powell, and Jayadev Misra. Simulation using orchestration. In José Meseguer and Grigore Roșu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 2–15. Springer Berlin / Heidelberg, 2008.

[KQCM09] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The Orc programming language. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *Formal techniques for Distributed Systems; Proc. of FMOODS/FORTE*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009.

[KQM10] David Kitchin, Adrian Quark, and Jayadev Misra. Quicksort: Combining concurrency, recursion, and mutable data structures. In A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 229–254. Springer London, 2010.

[LAc$^+$12] David Lazar, Andrei Arusoaie, Traian Florin Șerbănuță, Chucky Ellison, Radu Mereuta, Dorel Lucanu, and Grigore Roșu. Executing formal semantics with the $\mathbb{K}$ tool. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 267–271. Springer Berlin / Heidelberg, 2012.

[Laz12] D Lazar. K definition of haskell'98. 2012.

[LcR12] Dorel Lucanu, Traian Florin Șerbănuță, and Grigore Roșu. $\mathbb{K}$ framework distilled. In Francisco Durán, editor, *Rewriting Logic and Its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer Berlin / Heidelberg, 2012.

[LZH10] Qin Li, Huibiao Zhu, and Jifeng He. A denotational semantical model for Orc language. In *Theoretical Aspects of ComputingâĂŞICTAC 2010*, pages 106–120. Springer, 2010.

[MB04] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. *Algebraic Methodology and Software Technology*, pages 364–378, 2004.

[MC07] Jayadev Misra and WilliamR. Cook. Computation orchestration. *Software & Systems Modeling*, 6(1):83–110, 2007.

[mcr] mCRL2 201409.1. `http://www.mcrl2.org/`.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155, 1992.

[MHR07] Patrick Meredith, Mark Hills, and Grigore Roșu. An executable rewriting logic semantics of k-scheme. In *Workshop on Scheme and Functional Programming*, volume 1, page 10, 2007.

[Mis04] Jayadev Misra. Computation orchestration: A basis for wide-area computing. In Manfred Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktoberdorf, Germany, 2004.

[Mis14] Jayadev Misra. Structured concurrent programming. Manuscript, University of Texas at Austin, December 2014. `http://www.cs.utexas.edu/users/misra/temporaryFiles.dir/Orc.pdf`.

[MKMR10] Patrick Meredith, Michael Katelman, José Meseguer, and Grigore Roșu. A formal executable semantics of verilog. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 179–188. IEEE, 2010.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[MR07] José Meseguer and Grigore Roșu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[NEKN15] Danapaquiame Nagamouttou, Ilavarasan Egambaram, Muthumanickam Krishnan, and Poonkuzhali Narasingam. A verification strategy for web services composition using enhanced stacked automata model. *SpringerPlus*, 4(1), December 2015.

[NKL11] Wonhong Nam, Hyunyoung Kil, and Dongwon Lee. On the computational complexity of behavioral description-based web service composition. *Theoretical Computer Science*, 412(48):6736–6749, November 2011.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[num] NuSMV: a new symbolic model checker. `http://nusmv.fbk.eu/`.

[ÖM07] PeterCsaba Ölveczky and José Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[OMF14] Marc Oriol, Jordi Marco, and Xavier Franch. Quality models for web services: A systematic mapping. *Information and Software Technology*, 56(10):1167–1182, October 2014.

[PJM13] Matthieu Perrin, Claude Jard, and Achour Mostefaoui. Construction d'une sémantique concurrente par instrumentation d'une sémantique opérationelle structurelle. In *MSR 2013-Modélisation des Systèmes Réactifs*, 2013.

[PJM15] Matthieu Perrin, Claude Jard, and Achour Mostefaoui. Tracking Causal Dependencies in Web Services Orchestrations Defined in ORC. *arXiv preprint arXiv:1505.06299*, 2015.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[PSFRC14] José Antonio Parejo, Sergio Segura, Pablo Fernandez, and Antonio Ruiz-Cortés. QoS-aware web services composition using GRASP with Path Relinking. *Expert Systems with Applications*, 41(9):4211–4223, July 2014.

[PSR15] Daejun Park, Andrei Stefănescu, and Grigore Roșu. KJS: a complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–356. ACM, 2015.

[RBJ12] Sidney Rosario, Albert Benveniste, and Claude Jard. Flexible probabilistic qos management of orchestrations. *Web Service Composition and New Frameworks in Designing Semantics: Innovations: Innovations*, page 195, 2012.

[Rc10] Grigore Roșu and Traian Florin șerbănuță. An overview of the $\mathbb{K}$ semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010. Membrane computing and programming.

[Rc14] Grigore Roșu and Traian Florin Șerbănuță. $\mathbb{K}$ overview and SIMPLE case study. In Mark Hills, editor, *Proceedings of the Second International Workshop on the $\mathbb{K}$ Framework and its Applications ($\mathbb{K}$ 2011)*, volume 304 of *Electronic Notes in Theoretical Computer Science*, pages 3 – 56. Elsevier, 2014.

[RKB$^+$08] Sidney Rosario, David Kitchin, Albert Benveniste, William Cook, Stefan Haar, and Claude Jard. Event structure semantics of Orc. In *WS-FM 2007*, volume 4937 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008.

[Ro5] Grigore Roșu. From Rewriting Logic, to Programming Language Semantics, to Program Verification. In *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 598–616, Cham, Switzerland, 2015. Springer.

[SBFZ08] Jianwen Su, Tevfik Bultan, Xiang Fu, and Xiangpeng Zhao. Towards a theory of web service choreographies. In Marlon Dumas and Reiko Heckel, editors, *Web Services and Formal Methods*, volume 4937 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.

[SGK15] Alan Stewart, Joaquim Gabarró, and Anthony Keenan. Uncertainty in the cloud: An angel-daemon approach to modelling performance. In SÃľbastien Destercke and Thierry Denoeux, editors, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 9161 of *Lecture Notes in Computer Science*, pages 141–150. Springer International Publishing, 2015.

[spi] Spin - Formal Verification. `http://spinroot.com/spin/whatispin.html`.

[SQV$^+$14] Quan Z. Sheng, Xiaoqiang Qiao, Athanasios V. Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decadeâĂŹs overview. *Information Sciences*, 280:218–238, October 2014.

[Thy11] John A. Thywissen. Token Semantics of Orc. *Unpublished working draftâĂŞNot for distributionâĂŞSeptember*, 27(13):48, 2011.

[TJZ11] Xianfei Tang, Changjun Jiang, and Mengchu Zhou. Automatic Web service composition based on Horn clauses and Petri nets. *Expert Systems with Applications*, 38(10):13024–13031, September 2011.

[upp] UPPAAL. `http://www.uppaal.org/`.

[VBS15] Klemo Vladimir, Ivan Budiselić, and Siniša Srbljić. Consumerized and peer-tutored service composition. *Expert Systems with Applications*, 42(3):1028–1038, February 2015.

[VECDM09] Valentín Valero, M. Emilia Cambronero, Gregorio Díaz, and Hermenegilda Macià. A Petri net approach for the design and analysis of Web Services Choreographies. *The Journal of Logic and Algebraic Programming*, 78(5):359–380, May 2009.

[WKCM07] Ian Wehrman, David Kitchin, William R Cook, and Jayadev Misra. *Properties of the timed operational and denotational semantics of Orc*. 2007.

[WKCM08] Ian Wehrman, David Kitchin, William R. Cook, and Jayadev Misra. A timed semantics of Orc. *Theoretical Computer Science*, 402(2 - 3):234 – 248, 2008. Trustworthy Global Computing.

[WYM15] Dandan Wang, Yang Yang, and Zhenqiang Mi. A genetic-based approach to web service composition in geo-distributed cloud environment. *Computers & Electrical Engineering*, 43:129–141, April 2015.

[WZ13] Quanwang Wu and Qingsheng Zhu. Transactional and QoS-aware dynamic service composition based on ant colony optimization. *Future Generation Computer Systems*, 29(5):1112–1119, July 2013.

[YJC09] Junbeom Yoo, Eunkyoung Jee, and Sungdeok Cha. Formal modeling and verification of safety-critical software. *Software, IEEE*, 26(3):42–49, 2009.

[ZCO14] Min Zhang, Yunja Choi, and Kazuhiro Ogata. A formal semantics of the osek/vdx standard in $\mathbb{K}$ framework and its applications. 2014.

[ZGCZ14] Guobing Zou, Yanglan Gan, Yixin Chen, and Bofeng Zhang. Dynamic composition of Web services using efficient planners in large-scale service repository. *Knowledge-Based Systems*, 62:98–112, May 2014.

[ZSPZ15] Xin Zhao, Liwei Shen, Xin Peng, and Wenyun Zhao. Toward SLA-constrained service composition: An approach based on a fuzzy linguistic preference model and an evolutionary algorithm. *Information Sciences*, 316:370–396, September 2015.

Vita

Personal Information

- Name: Omar Zuhair Alzuhaibi
- Citizenship: Saudi Arabia
- Born: 1985, Jeddah, Saudi Arabia
- Email: *omarzud@gmail.com*

Education

- B.S., Electrical Engineering, 2008, King Fahd Univ. of Petroleum and Minerals, Dhahran, Saudi Arabia.
- M.S., Computer Science, 2016, King Fahd Univ. of Petroleum and Minerals, Dhahran, Saudi Arabia.

Publications

- AlTurki, Musab A., and Omar Alzuhaibi. "Towards Formal Verification of Orchestration Computations Using the $\mathbb{K}$ Framework." *FM 2015: Formal Methods*. Springer International Publishing, 2015. 40-56.
  Paper was presented by the author of this thesis at FM 2015, Oslo, Norway; and it included partial work from this thesis.

Research Interests

Formal Semantics and Syntactic Analysis, especially for natural language.

Appendix A

Select Modules of the $\mathbb{K}$-Orc Definition

This appendix shows only those modules of the $\mathbb{K}$-Orc definition that contain rules which were omitted from Chapter 4. As mentioned, these rules were omitted because they are too technical and carry too little semantic significance to be included and explained in Chapter 4.

A.1 Syntax Module

MODULE ORC-SYNTAX

A.1.1 The Orc Calculus

"The Orc process calculus is based on the execution of *expressions*. As an expression executes, it may interact with external services (called *sites*), and may *publish* values during its execution. An expression may also *halt*, explicitly indicating that its execution is complete.

Simple expressions are built up into more complex ones by using *combinators* and by defining *functions*."[Kit13]

A.1.2 Expressions

Looking at Figure 2.1 showing the grammar of the Orc calculus, the following grammar defined in $\mathbb{K}$ syntax is almost identical. There are a few extra semantic elements that $\mathbb{K}$ allows to define within the syntax module. The first is precedence, denoted by the >operator. As is clearly stated in the Orc paper, the order of precedence of the four combinators from higher to lower is: Sequential, Parallel, Pruning, Otherwise. In addition, we prefer for simpler expressions to be matched before complex ones; so, we put on top, `Arg` and `Call`.

The second semantic element that is defined within the syntax module of $\mathbb{K}$ is `right-` or `left`-associativity, and strictness. It is important to note that for the parallel operator, being fully-associative, defining it here as right-associative is to help in parsing. In other words, it is defined this way only for technical reasons that become clear when we define the rules later.

`strict` means that the terms in the right hand side of the production must be evaluated before the production is matched. strict can also take integer arguments denoting, by order from left to right which non-terminals are strict.

$$
\begin{array}{ll}
\textsc{syntax} & \mathit{Pgm} ::= \mathit{Exp} \\
& \quad\mid\ \mathit{ExpDefs}\ \ \mathit{Exp}
\end{array}
$$

$$
\begin{array}{ll}
\textsc{syntax} & \mathit{Exp} ::= \mathit{Arg} \\
& \quad > \ \mathit{Exp}\ \texttt{>}\ \mathit{Param}\ \texttt{>}\ \mathit{Exp}\ [\text{right}] \\
& \quad\ \mid\ \mathit{Exp} \gg \mathit{Exp}\ [\text{right}] \\
& \quad > \ \mathit{Exp}\ \texttt{|}\ \mathit{Exp}\ [\text{right}] \\
& \quad > \ \mathit{Exp}\ \texttt{<}\ \mathit{Param}\ \texttt{<}\ \mathit{Exp}\ [\text{left}] \\
& \quad\ \mid\ \mathit{Exp} \gg \mathit{Exp}\ [\text{left}] \\
& \quad > \ \mathit{Exp}\ \texttt{;}\ \mathit{Exp}\ [\text{left}]
\end{array}
$$

$$
\textsc{syntax} \quad \mathit{ExpDefs} ::= \mathit{List}\{\mathit{ExpDef}, \text{“”}\}
$$

$$
\textsc{syntax} \quad \mathit{ExpDef} ::= \mathit{Decl}\ \texttt{:=}\ \mathit{Exp}
$$

$$
\textsc{syntax} \quad \mathit{Decl} ::= \mathit{ExpId}(\mathit{Params})
$$

$$
\textsc{syntax} \quad \mathit{ExpId} ::= \mathit{Id}
$$

A.1.3 Parameters and Arguments

Arguments are what is called Actual Parameters in the Orc grammar, while our *Parameters* are what they called Formal Parameters.

$$
\begin{array}{ll}
\textsc{syntax} & \mathit{Arg} ::= \mathit{Val} \\
& \quad\mid\ \texttt{clock} \\
& \quad\mid\ \mathit{Id} \\
& \quad\mid\ \mathit{Tuple} \\
& \quad\mid\ \mathit{Call}
\end{array}
$$

$$
\textsc{syntax} \quad \mathit{Param} ::= \mathit{Id}
$$

A.1.4 Site Calls and Expression Calls

SYNTAX $\mathit{Call} ::= \mathit{SiteCall}$
$\mid \mathit{Handle}$
$\mid \mathit{ExpCall}$

SYNTAX $\mathit{SiteCall} ::= \mathit{SiteId}(\mathit{Args})$ [strict(2)]

SYNTAX $\mathit{ExpCall} ::= \mathit{ExpId}(\mathit{Args})$

SYNTAX $\mathit{Handle} ::= \mathit{FreeHandle}$
$\mid \mathit{PubHandle}$
$\mid \mathit{SilentHandle}$
$\mid \mathit{TimedHandle}$

SYNTAX $\mathit{FreeHandle} ::=$ `handle` $(\mathit{SiteCall})$

SYNTAX $\mathit{PubHandle} ::=$ `pubHandle` (Val)

SYNTAX $\mathit{SilentHandle} ::=$ `silentHandle` $(\mathit{SiteCall})$

SYNTAX $\mathit{TimedHandle} ::=$ `timedHandle` $(\mathit{Int}, \mathit{SiteCall}, \mathit{Arg})$
$\mid$ `timedHandle` $(\mathit{Int}, \mathit{SiteCall})$

SYNTAX $\mathit{SiteId} ::= \mathit{ISiteId}$

SYNTAX $\mathit{SiteId} ::= \mathit{TSiteId}$

A.1.5 Values

SYNTAX $\mathit{Val} ::= \mathit{Int}$
$\mid \mathit{Float}$
$\mid \mathit{Bool}$
$\mid \mathit{String}$
$\mid$ `signal`
$\mid$ `stop`

A.1.6 KResult

To tell $\mathbb{K}$ what sorts are accepted as final evaluations of a `k` cell, we declare them as the $\mathbb{K}$ built-in syntactic category `KResult`. Note that *Val* is not accepted as a final evaluation because it is syntactic sugar for a call to $let(v)$ where v is of sort *Val*. However, `silentHandle` is accepted because otherwise the program will get stuck on sites that never respond.

SYNTAX $\mathit{KResult} ::= \mathit{SilentHandle}$

A.1.7 Secondary Syntax Productions

Variable is also a $\mathbb{K}$ built-in syntactic category that is necessary when using the substitution module.

SYNTAX $\mathit{Variable} ::= \mathit{Id}$

Sites/Expressions may publish a tuple of values. So, we declare lists to handle that in rules ahead.

SYNTAX $\mathit{Exps} ::= \mathit{List}\{\mathit{Exp}, \text{“, ”}\}$

SYNTAX $\mathit{Args} ::= \mathit{List}\{\mathit{Arg}, \text{“, ”}\}$ [strict]

SYNTAX $\mathit{Params} ::= \mathit{List}\{\mathit{Param}, \text{“, ”}\}$

SYNTAX $\mathit{Vals} ::= \mathit{List}\{\mathit{Val}, \text{“, ”}\}$

SYNTAX $\mathit{Ids} ::= \mathit{List}\{\mathit{Id}, \text{“, ”}\}$

SYNTAX $\mathit{Tuple} ::= $ `<` Vals `>` [strict]

Here we declare the parentheses used for grouping as brackets.

SYNTAX $\mathit{Exp} ::= (\mathit{Exp})$ [bracket]

Thread managing functions

SYNTAX $\mathit{KItem} ::=$ `killed` (K)

SYNTAX $\mathit{Exp} ::=$ `prllCompMgr` (Int)

SYNTAX $\mathit{Exp} ::=$ `seqCompMgr` $(\mathit{Param}, \mathit{Exp}, \mathit{Int})$

SYNTAX $\mathit{Exp} ::=$ `prunCompMgr` (Param)

SYNTAX $\mathit{Exp} ::=$ `othrCompMgr` (Exp)

Convert Orc Tuple to KList

SYNTAX $\mathit{List} ::=$ `tuple2Klist` (Tuple) [function]

$$\frac{\texttt{tuple2Klist}\,(\,\texttt{<}\ V\!:\!\mathit{Val}\,\texttt{,}\ \mathit{Vs}\!:\!\mathit{Vals}\ \texttt{>})}{\texttt{ListItem}\,(\,V\,)\ \ \texttt{tuple2Klist}\,(\,\texttt{<}\ \mathit{Vs}\ \texttt{>})}$$

[anywhere]

$$\frac{\texttt{tuple2Klist}\,(\,\texttt{<}\,\bullet_{Vals}\,\texttt{>})}{\bullet_{List}}$$

[anywhere]

END MODULE

A.2 Main Orc Module

MODULE ORC

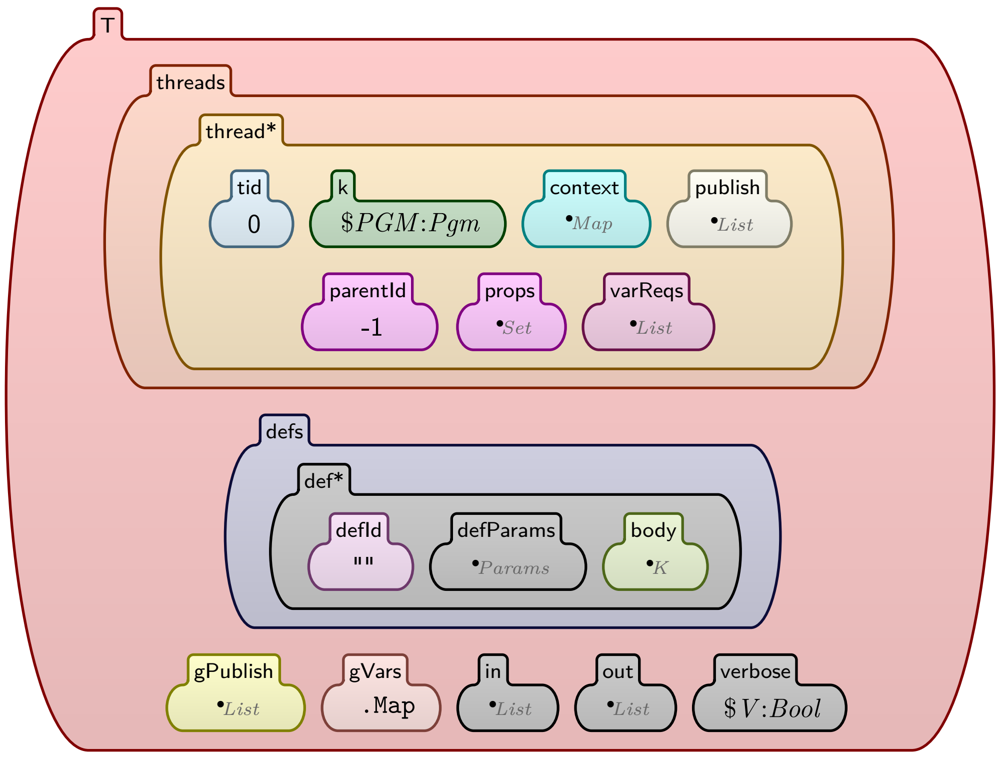
A.2.1 Configuration

A configuration in 𝕂 is simply a state. Here we define the structure of our configuration. We call each XML-like element declared below a *cell*. We declare a cell `thread` with multiplicity *, that is zero, one, or more. 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 as we will see later.

- The `context` cell is for mapping variables to values.
- The `publish` cell is for keeping 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.
- `defs` holds the expressions defined at the beginning of the Orc program.

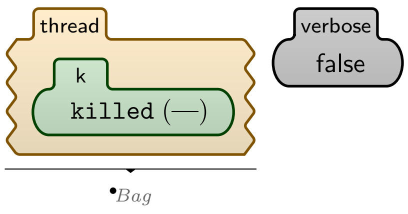
Each cell is declared with an initial value. The `$PGM` variable tells 𝕂 that this is where we want our program to go. So by default, the first state would hold a single thread with the `k` cell holding the whole Orc program as the *Pgm* non-terminal defined in the syntax module.

CONFIGURATION:



An orphan rule whose only purpose is to delete threads marked as 'killed'. Only a few rules in the whole definition kill threads, but we chose to centralize deletion of threads at this rule because it only deletes threads if 'verbose' is false. Sometimes we need to see all the threads that were created throughout the execution, so we make 'verbose' true.

CLEANUP-KILLED



[structural]

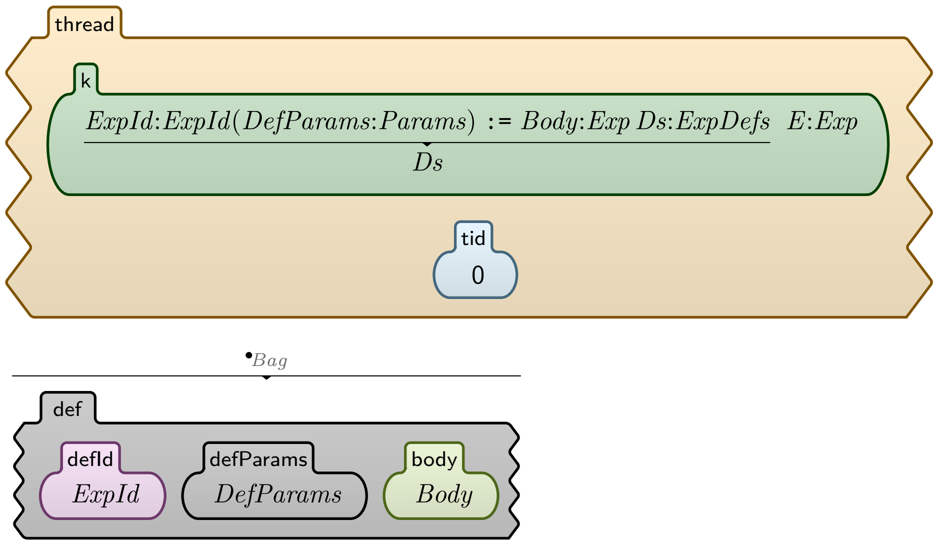
END MODULE

A.3 Expression Definitions and Calls

MODULE ORC-EXPDEF

Translate expression definitions to `def` cells.

EXPRESSION-DEFINITION-PREP-1



[structural]

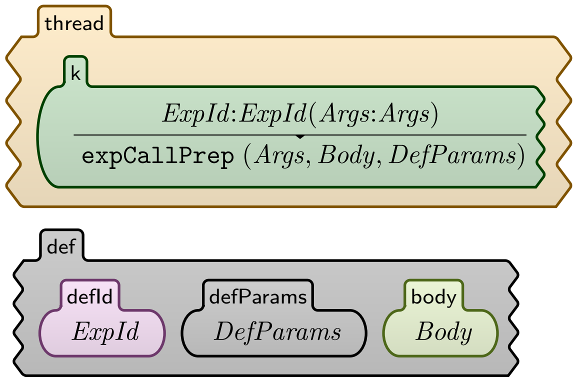
EXPRESSION-DEFINITION-PREP-2

$$\underbrace{\bullet_{ExpDefs}\ E{:}Exp}_{E}$$

[structural]

Replace an expression call with the predefined `exp` body and populate the `expCall` cell

EXPRESSION-CALL-PREP-1-CREATE



[structural]

Define functions needed for expression calls.

SYNTAX $Exp ::= \texttt{expCallPrep}\ (Args, Exp, Params)$ [function]

SYNTAX $Arg ::= \texttt{Param2Arg}\ (Param)$ [function]

PARAM-TO-ARG

$$\frac{\texttt{Param2Arg}\ (P{:}Id)}{P}$$

[anywhere, function]

A rule that uses substitution for expression calls. This runs recursively substituting each parameter in the body with the corresponding argument.

EXPRESSION-CALL-PREP-2-SUBSTITUTE

thread

k

$$\frac{\texttt{expCallPrep}\ (A{:}Arg\texttt{,}\ As{:}Args, Body, P{:}Param\texttt{,}\ Ps{:}Params)}{\texttt{expCallPrep}\ (As, Body[A\ \texttt{/}\ \texttt{Param2Arg}\ (P)], Ps)}$$

This rule ends the recursion.

EXPRESSION-CALL-PREP-3-END

thread

k

$$\frac{\texttt{expCallPrep}\ (\bullet_{Args}, Body, \bullet_{Params})}{(Body)}$$

END MODULE

A.4 Time

The semantics of our time model are inspired by those of real-time Maude described in [ÖM07]. We implemented two time modules, one that sequentially applies delta, and another that can do it concurrently. Only one of these two can be active at a time even though we list both of them here.

A.4.1 Concurrent Time Module

MODULE ORC-TIME

Tick-Clock

threads
$A{:}Bag$

gVars
$\texttt{"time.ticked"} \mapsto \dfrac{\mathsf{false}}{\mathsf{true}} \quad \texttt{"time.clock"} \mapsto \dfrac{Clk{:}Int}{Clk +_{Int} 1} \quad \texttt{"time.limit"} \mapsto TL{:}Int$

requires $Clk <_{Int} TL$
$\wedge_{Bool} \neg_{Bool}(\ \texttt{anySiteCall}\ (A))$
$\wedge_{Bool} \neg_{Bool}(\ \texttt{anyFreeHandle}\ (A))$
$\wedge_{Bool} \neg_{Bool}(\ \texttt{anyPubHandle}\ (A))$
$\wedge_{Bool}\ \texttt{anyTimedHandle}\ (A)$
$\wedge_{Bool} \neg_{Bool}(\ \texttt{anyAppliedDelta}\ (A))$
[transition]

Apply-Delta

thread

k
$\dfrac{\texttt{timedHandle}\ (I{:}Int, SC, A)}{\texttt{timedHandle}\ (I -_{Int} 1, SC, A)}$

props
$S{:}Set \quad \dfrac{\bullet_{Set}}{\texttt{SetItem}\ (\texttt{"applied\_delta"})}$

gVars
$\texttt{"time.ticked"} \mapsto \mathsf{true}$

requires $I >_{Int} 0 \wedge_{Bool} \neg_{Bool}(\texttt{"applied\_delta"}\ \texttt{in}\ S)$
[structural]

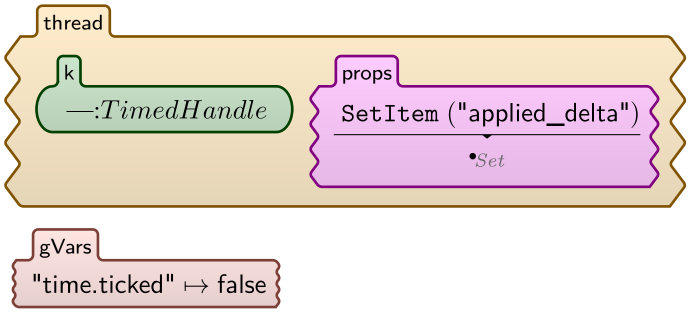
Delta-Done

threads
$A{:}Bag$

gVars
$\texttt{"time.ticked"} \mapsto \dfrac{\mathsf{true}}{\mathsf{false}}$

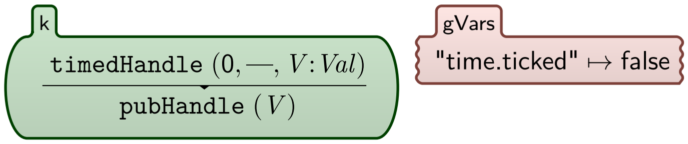
requires $\texttt{allAppliedDelta}\ (A)$
[structural]

Reset-TimedHandles



[structural]

TimedHandle-Outro



[structural]

END MODULE

A.4.2 Sequential Time Module

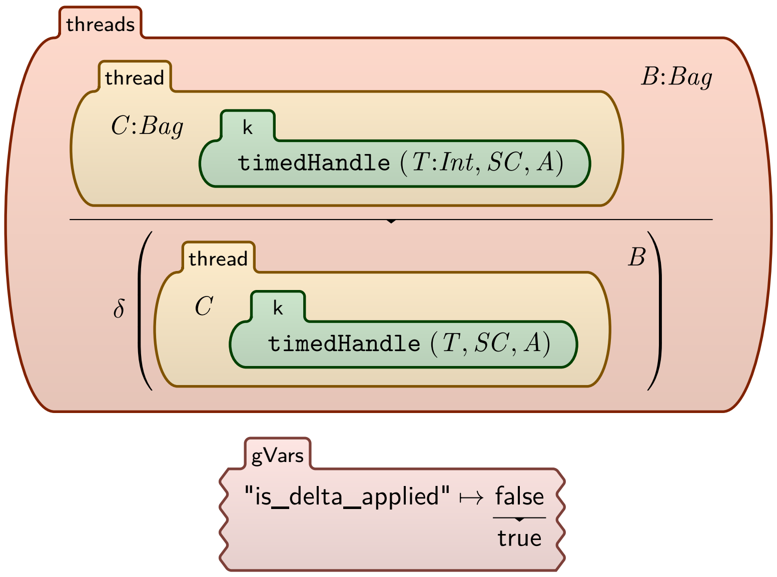
MODULE ORC-TIMESEQ

The semantics of our time model are inspired by those of real-time Maude described in [ÖM07].
The δ function applies on the whole environment and then moved down the thread tree being applied on individual expressions.

SYNTAX $Bag ::= \delta\,(Bag)$

SYNTAX $Exp ::= \delta\,(Exp)$
Apply delta to the whole bag of threads if it has at least one timed thread.

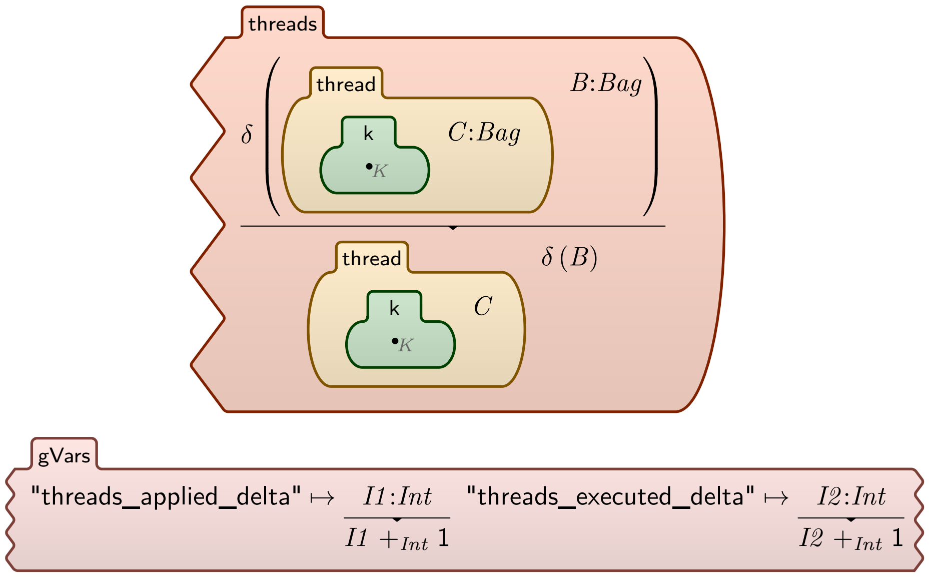
SYNTAX $KItem ::=$ `bag` (Bag)
Apply-Delta-Globally



requires $T >_{Int} 0$
[structural]

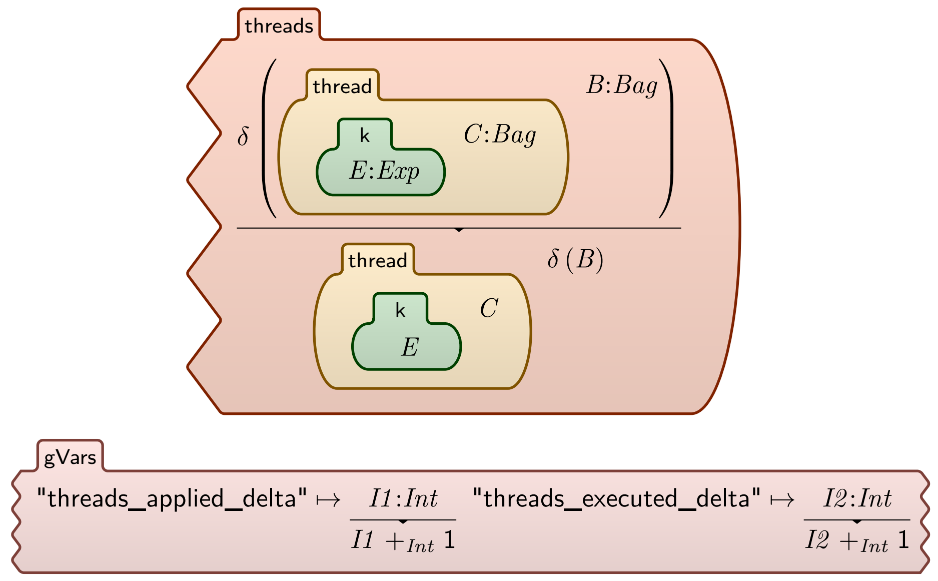
After delta is applied to the whole bag of threads, the following two rules run repeatedly on all threads. Exactly one of these two rules will match each thread. The first is for untimed threads, and the second is for timed threads.

Delta-On-Thread-1-Skip-1


threads
δ
thread
k
•K
C:Bag
B:Bag
thread
k
•K
C
δ (B)
gVars
"threads_applied_delta" ↦ I1:Int
I1 +Int 1
"threads_executed_delta" ↦ I2:Int
I2 +Int 1

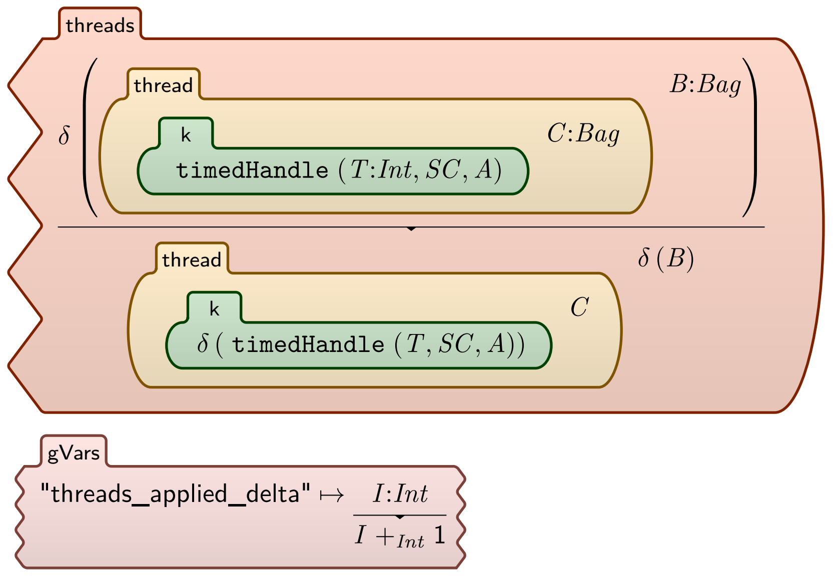
[structural]

Delta-On-Thread-1-Skip-2


threads
δ
thread
k
$E{:}Exp$
$C{:}Bag$
$B{:}Bag$
thread
k
E
C
$\delta\,(B)$
gVars
"threads_applied_delta" $\mapsto$ $I1{:}Int$ / $I1 +_{Int} 1$
"threads_executed_delta" $\mapsto$ $I2{:}Int$ / $I2 +_{Int} 1$

requires $isTimedHandle(E) \neq_K$ true
[structural]

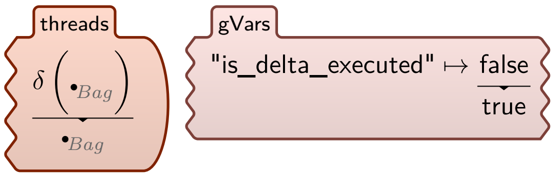
Delta-On-Thread-2-Hold


threads
δ
thread
k
timedHandle $(T{:}Int, SC, A)$
$C{:}Bag$
$B{:}Bag$
thread
k
δ (timedHandle (T, SC, A))
C
$\delta\,(B)$
gVars
"threads_applied_delta" $\mapsto$ $I{:}Int$ / $I +_{Int} 1$

requires $T >_{Int} 0$
[structural]

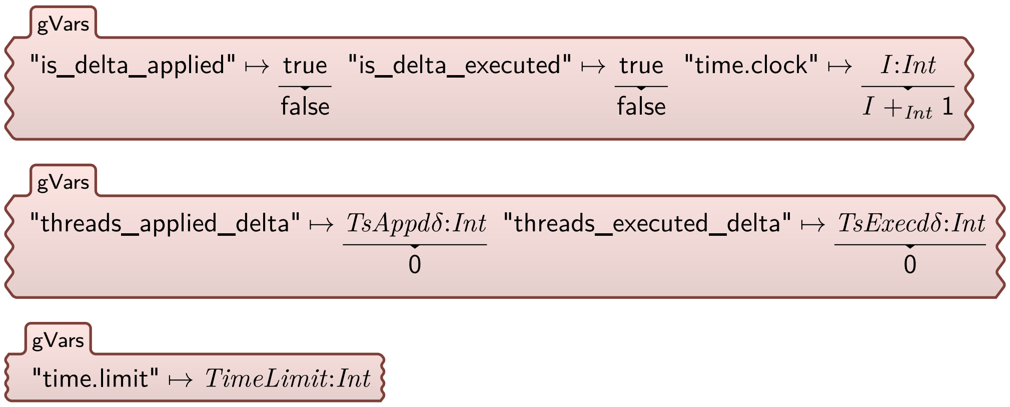
After δ has finished running on the whole environment, perform a clock tick, i.e., advance the clock by one time unit.

MARK-DELTA-DONE


threads
$\delta\left(\bullet_{Bag}\right)$ / $\bullet_{Bag}$
gVars
"is_delta_executed" $\mapsto$ false / true

[structural]

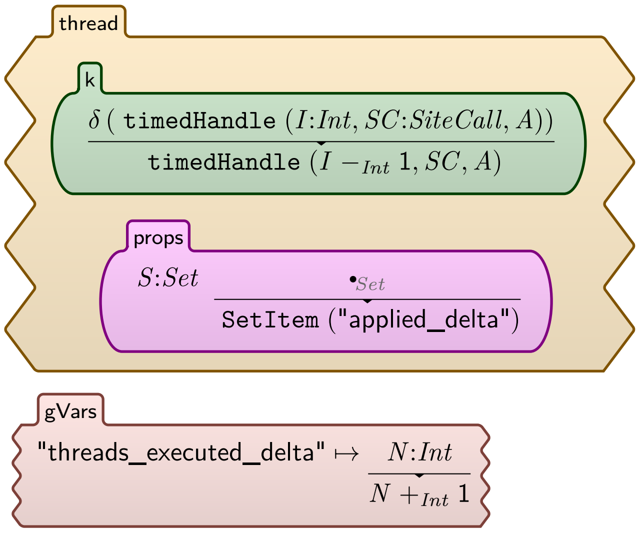
TICK-CLOCK


gVars
"is_delta_applied" $\mapsto$ true / false
"is_delta_executed" $\mapsto$ true / false
"time.clock" $\mapsto$ $I{:}Int$ / $I +_{Int} 1$
gVars
"threads_applied_delta" $\mapsto$ $TsAppd\delta{:}Int$ / 0
"threads_executed_delta" $\mapsto$ $TsExecd\delta{:}Int$ / 0
gVars
"time.limit" $\mapsto$ $TimeLimit{:}Int$

requires $TsAppd\delta ==_{Int} TsExecd\delta \wedge_{Bool} I <_{Int} TimeLimit$
[transition]

δ applying on handles of timed sites.

TIME-HANDLE-ONE-TICK


thread
k
δ (timedHandle $(I{:}Int, SC{:}SiteCall, A)$) / timedHandle $(I -_{Int} 1, SC, A)$
props
$S{:}Set$
$\bullet_{Set}$ / SetItem ("applied_delta")
gVars
"threads_executed_delta" $\mapsto$ $N{:}Int$ / $N +_{Int} 1$

requires $I >_{Int} 0 \wedge_{Bool} \neg_{Bool}$"applied_delta" `in` S
[structural]

TIME-HANDLE-OUTRO

thread

k

$$\frac{\texttt{timedHandle}\,(0, \text{—}, V{:}Val)}{\texttt{pubHandle}\,(V)}$$

[structural]

END MODULE

A.5 Local Site Identifiers

MODULE ORC-ISITES

Orc has some built-in sites. We call these local sites. Here, we syntactically declare the Identifiers (names) of these local sites and give semantics to each. We divide them into *Internal*, and *Timed* sites. This division is not based on any particular syntactic or semantic difference; It only provides better organization.

A.5.1 Internal Sites

Signal `Signal` Publishes the `signal` value immediately. It is the same as `if(true)`

SYNTAX $ISiteId ::=$ `Signal`

SITE-SIGNAL

$$\frac{\texttt{handle}\,(\,\texttt{Signal}(\text{—}))}{\texttt{pubHandle}\,(\,\texttt{signal})}$$

[anywhere]

Stop The expression `stop`, when executed, halts immediately. And any site call with an argument `stop` halts as well. A site halting simply means that its execution is complete.

VAL-STOP-1

$$\frac{A{:}Arg\,\texttt{, stop}}{\texttt{stop}}$$

[anywhere]

Val-stop-2

$$\frac{\texttt{stop}\,\texttt{,}\ A{:}\mathit{Arg}}{\texttt{stop}}$$

[anywhere]

Val-stop-3

$$\frac{\text{—}{:}\mathit{SiteId}(\ \texttt{stop})}{\bullet_{K}}$$

[anywhere]

let publishes a tuple consisting of the values of its arguments.

syntax *ISiteId* ::= `let`

Site-Let

$$\frac{\texttt{handle}\ (\ \texttt{let}(V{:}\mathit{Val}))}{\texttt{pubHandle}\ (\ V\ )}$$

[structural]

print prints text to the console, then publishes `signal`, then halts.

syntax *ISiteId* ::= `print`

Site-Print-1

$$\boxed{\texttt{k}:\ \texttt{handle}\ (\ \texttt{print}(\frac{V{:}\mathit{Val}\texttt{,}\ \mathit{Vs}{:}\mathit{Vals}}{\mathit{Vs}}))\ \cdots}\quad \boxed{\texttt{out}:\ \cdots\ \frac{\bullet_{List}}{\texttt{ListItem}\ (\ V\ )}}$$

[structural]

Site-Print-2

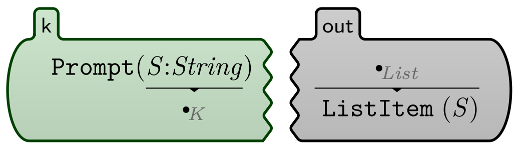
$$\frac{\texttt{handle}\ (\ \texttt{print}(\bullet_{Args}))}{\texttt{pubHandle}\ (\ \texttt{signal})}$$

[macro]

Prompt The `Prompt` site asks the user for text input. `Prompt("username:")` presents a prompt with the text `username:`, receives a line of input, publishes that line of input as a string, and then halts.

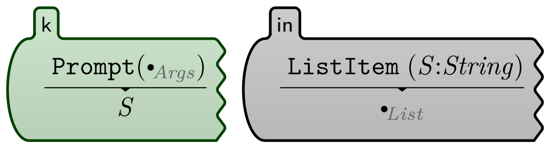
syntax *ISiteId* ::= `Prompt`

Site-Prompt-1



[structural]

SITE-PROMPT-2



[structural]

`if(b)` where `b` is boolean, publishes a signal if `b` is true and it remains silent (i.e., does not respond) if `b` is false.

SYNTAX *ISiteId* ::= `if`

SITE-IF-TRUE

$$\frac{\texttt{handle}\ (\ \texttt{if}(\mathsf{true}))}{\texttt{pubHandle}\ (\ \texttt{signal})}$$

[macro]

SITE-IF-FALSE

$$\frac{\texttt{handle}\ (\ \texttt{if}(\mathsf{false}))}{\texttt{silentHandle}\ (\ \texttt{if}(\mathsf{false}))}$$

[macro]

We make a site `ifNot` for convenience.

SYNTAX *ISiteId* ::= `ifNot`

SITE-IFNOT

$$\frac{\texttt{handle}\ (\ \texttt{ifNot}(B{:}\mathit{Bool}))}{\texttt{handle}\ (\ \texttt{if}(\neg_{\mathit{Bool}} B))}$$

[macro]

zero should just be silent.

SYNTAX *ISiteId* ::= `zero`

$$\frac{\texttt{handle}\ (\ \texttt{zero}(\bullet_{\mathit{Args}}))}{\texttt{silentHandle}\ (\ \texttt{zero}(\bullet_{\mathit{Args}}))}$$

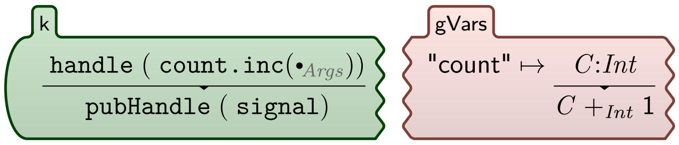
[macro]

Define sites that exclusively use the `count` global variable. `count.inc` increments the count while `count.read` publishes the current count.

SYNTAX $\mathit{ISiteId} ::=$ `count.inc`

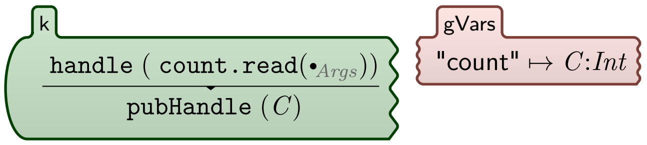
SYNTAX $\mathit{ISiteId} ::=$ `count.read`

COUNT.INC



[structural]

COUNT.READ



[structural]

END MODULE

A.5.2 Timer Sites

MODULE ORC-TSITES

> "The timer sites—*Clock*, *Atimer* and *Rtimer*—are used for computations involving time. Time is measured locally by the server on which the computation is performed. Since the timer is a local site, the client experiences no network delay in calling the timer or receiving a response from it; this means that the signal from the timer can be delivered at exactly the right moment. With $t = 0$, *Rtimer* responds immediately. Sites *Atimer* and *Rtimer* differ only in having absolute and relative values of time as their arguments, respectively."[MC07]

Clock publishes the current time. Even though it is a timer site, it does not affect the time, so we define it as an internal site instead, so that it doesn't receive the "timed" property thus by not requiring its own δ rule.

SYNTAX $\mathit{ISiteId} ::=$ `Clock`

VAL-CLOCK

$$\frac{\texttt{clock}}{\texttt{Clock}(\bullet_{Args})}$$

[macro]

SITE-CLOCK

k: $\dfrac{\texttt{handle}\ (\ \texttt{Clock}(\bullet_{Args}))}{\texttt{pubHandle}\ (T)}$ … gVars: … "time.clock" $\mapsto T{:}Int$ …

[structural]

δ applying on handles of timed sites.

The semantics of the Rtimer and Atimer sites are only realizable through the δ function.

Rtimer(t) publishes a signal after t time units.

SYNTAX $\quad TSiteId ::=$ `Rtimer`

RTIMER

k: $\dfrac{\texttt{handle}\ (\ \texttt{Rtimer}(I{:}Int))}{\texttt{timedHandle}\ (I,\ \texttt{Rtimer}(0),\ \texttt{signal})}$

[structural]

Atimer(t) publishes a signal at time t.

We write *Atimer* in terms of *Rtimer*

SYNTAX $\quad TSiteId ::=$ `Atimer`

ATIMER

k: $\dfrac{\texttt{handle}\ (\ \texttt{Atimer}(T{:}Int))}{\texttt{timedHandle}\ (T -_{Int} Clk,\ \texttt{Atimer}(0),\ \texttt{signal})}$ … gVars: … "time.clock" $\mapsto Clk{:}Int$ …

requires $T >_{Int} Clk$

[structural]

END MODULE

A.6 Math Sites

MODULE ORC_MATH

SYNTAX $\mathit{ISiteId} ::= \texttt{Add}$

$$\frac{\texttt{handle}\ (\ \texttt{Add}(\mathit{I1}{:}\mathit{Int}\,,\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\mathit{I1} +_{\mathit{Int}} \mathit{I2})}$$

[structural]

SYNTAX $\mathit{ISiteId} ::= \texttt{Incr}$

$$\frac{\texttt{handle}\ (\ \texttt{Incr}(I{:}\mathit{Int}))}{\texttt{pubHandle}\ (I +_{\mathit{Int}} 1)}$$

[structural]

SYNTAX $\mathit{ISiteId} ::= \texttt{Decr}$

$$\frac{\texttt{handle}\ (\ \texttt{Decr}(I{:}\mathit{Int}))}{\texttt{pubHandle}\ (I -_{\mathit{Int}} 1)}$$

[structural]

SYNTAX $\mathit{ISiteId} ::= \texttt{Sum}$

$$\frac{\texttt{handle}\ (\ \texttt{Sum}(\mathit{I1}{:}\mathit{Int}\,,\ \mathit{I2}{:}\mathit{Int}\,,\ \mathit{As}{:}\mathit{Args}))}{\texttt{handle}\ (\ \texttt{Sum}(\mathit{I1} +_{\mathit{Int}} \mathit{I2}\,,\ \mathit{As}))}$$

[macro()]

$$\frac{\texttt{handle}\ (\ \texttt{Sum}(I{:}\mathit{Int}\,,\ \bullet_{\mathit{Args}}))}{\texttt{pubHandle}\ (I)}$$

[structural]

SYNTAX $\mathit{ISiteId} ::= \texttt{Sqrt}$

$$\frac{\texttt{handle}\ (\ \texttt{Sqrt}(\mathit{I1}{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{rootFloat}\ (\mathit{I1}, 2))}$$

[structural]

SYNTAX $\mathit{ISiteId} ::=$ `Root`

$$\frac{\texttt{handle}\ (\ \texttt{Root}(\mathit{I1}{:}\mathit{Float}\,,\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\ \texttt{rootFloat}\ (\mathit{I1}\,,\mathit{I2}))}$$

[structural]

SYNTAX $\mathit{ISiteId} ::=$ `Mul`

$$\frac{\texttt{handle}\ (\ \texttt{Mul}(\mathit{I1}{:}\mathit{Int}\,,\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\mathit{I1}\ *_{\mathit{Int}}\ \mathit{I2})}$$

[structural]

SYNTAX $\mathit{ISiteId} ::=$ `Sub`

$$\frac{\texttt{handle}\ (\ \texttt{Sub}(\mathit{I1}{:}\mathit{Int}\,,\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\mathit{I1}\ -_{\mathit{Int}}\ \mathit{I2})}$$

[structural]

SYNTAX $\mathit{ISiteId} ::=$ `Div`

$$\frac{\texttt{handle}\ (\ \texttt{Div}(\mathit{I1}{:}\mathit{Int}\,,\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\mathit{I1}\ \div_{\mathit{Int}}\ \mathit{I2})}$$

requires $\mathit{I2} \neq_{K} \mathsf{0}$
[structural]

SYNTAX $\mathit{ISiteId} ::=$ `Mod`

$$\frac{\texttt{handle}\ (\ \texttt{Mod}(\mathit{I1}{:}\mathit{Int}\,,\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\mathit{I1}\ \%_{\mathit{Int}}\ \mathit{I2})}$$

requires $\mathit{I2} \neq_{K} \mathsf{0}$
[structural]

SYNTAX $\mathit{ISiteId} ::=$ `Floor`

$$\frac{\texttt{handle}\ (\ \texttt{Floor}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{floorFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId}$ ::= `Ceil`

$$\frac{\texttt{handle}\ (\ \texttt{Ceil}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{ceilFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId}$ ::= `Round`

$$\frac{\texttt{handle}\ (\ \texttt{Round}(\,V{:}\mathit{Float}\,,\ I1{:}\mathit{Int}\,,\ I2{:}\mathit{Int}))}{\texttt{pubHandle}\ (\ \texttt{roundFloat}\ (\,V, I1, I2))}$$
[structural]

SYNTAX $\mathit{ISiteId}$ ::= `Abs`

$$\frac{\texttt{handle}\ (\ \texttt{Abs}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{absFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId}$ ::= `Exp`

$$\frac{\texttt{handle}\ (\ \texttt{Exp}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{expFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId}$ ::= `LogFloat`

$$\frac{\texttt{handle}\ (\ \texttt{LogFloat}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{logFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId}$ ::= `SinFloat`

$$\frac{\texttt{handle}\ (\ \texttt{SinFloat}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{sinFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId} ::=$ `CosFloat`

$$\frac{\texttt{handle}\ (\ \texttt{CosFloat}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{cosFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId} ::=$ `TanFloat`

$$\frac{\texttt{handle}\ (\ \texttt{TanFloat}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{tanFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId} ::=$ `AsinFloat`

$$\frac{\texttt{handle}\ (\ \texttt{AsinFloat}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{asinFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId} ::=$ `AcosFloat`

$$\frac{\texttt{handle}\ (\ \texttt{AcosFloat}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{acosFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId} ::=$ `AtanFloat`

$$\frac{\texttt{handle}\ (\ \texttt{AtanFloat}(\,V{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{atanFloat}\ (\,V))}$$
[structural]

SYNTAX $\mathit{ISiteId} ::=$ `Atan2Float`

$$\frac{\texttt{handle}\ (\ \texttt{Atan2Float}(\mathit{V1}{:}\mathit{Float},\ \mathit{V2}{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{atan2Float}\ (\mathit{V1},\ \mathit{V2}))}$$

[structural]

SYNTAX *ISiteId* ::= `MaxFloat`

$$\frac{\texttt{handle}\ (\ \texttt{MaxFloat}(\mathit{V1}{:}\mathit{Float},\ \mathit{V2}{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{maxFloat}\ (\mathit{V1},\ \mathit{V2}))}$$

[structural]

SYNTAX *ISiteId* ::= `MinFloat`

$$\frac{\texttt{handle}\ (\ \texttt{MinFloat}(\mathit{V1}{:}\mathit{Float},\ \mathit{V2}{:}\mathit{Float}))}{\texttt{pubHandle}\ (\ \texttt{minFloat}\ (\mathit{V1},\ \mathit{V2}))}$$

[structural]

SYNTAX *ISiteId* ::= `Equals`

$$\frac{\texttt{handle}\ (\ \texttt{Equals}(\mathit{I1}{:}\mathit{Int},\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\mathit{I1} =_{K} \mathit{I2})}$$

[structural]

SYNTAX *ISiteId* ::= `Gr`

$$\frac{\texttt{handle}\ (\ \texttt{Gr}(\mathit{I1}{:}\mathit{Int},\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\mathit{I1} >_{\mathit{Int}} \mathit{I2})}$$

[structural]

SYNTAX *ISiteId* ::= `GrEq`

$$\frac{\texttt{handle}\ (\ \texttt{GrEq}(\mathit{I1}{:}\mathit{Int},\ \mathit{I2}{:}\mathit{Int}))}{\texttt{pubHandle}\ (\mathit{I1} \geq_{\mathit{Int}} \mathit{I2})}$$

[structural]

SYNTAX *ISiteId* ::= `Ls`

$$\frac{\texttt{handle}\ (\ \texttt{Ls}(I1{:}Int\texttt{,}\ I2{:}Int))}{\texttt{pubHandle}\ (I1 <_{Int} I2)}$$

[structural]

SYNTAX *ISiteId* ::= `LsEq`

$$\frac{\texttt{handle}\ (\ \texttt{LsEq}(I1{:}Int\texttt{,}\ I2{:}Int))}{\texttt{pubHandle}\ (I1 \leq_{Int} I2)}$$

[structural]

END MODULE

A.7 Bot Environment

MODULE ORC-ROBOT

Define a special category of sites that deals with locks

SYNTAX *Arg* ::= `bot_lock`

SYNTAX *TSiteId* ::= *BotTSite*

A.7.1 Internal Bot Sites

SYNTAX *ISiteId* ::= `bot.mapInit`

MAPINIT-1

k

$$\frac{\texttt{handle}\ (\ \texttt{bot.mapInit}(\bullet_{Args}))}{\texttt{pubHandle}\ (\ \texttt{signal})}$$

gVars

$$\frac{\bullet_{Map}}{(\texttt{"bot.map\_size"} \mapsto \texttt{ListItem}\ (5)\ \ \texttt{ListItem}\ (5))\ \ (\texttt{"bot.map\_obstacles"} \mapsto \bullet_{List})}$$

[structural]

MAPINIT-2

k

$$\frac{\texttt{handle (bot.mapInit(}T{:}\mathit{Tuple}\texttt{))}}{\texttt{pubHandle (signal)}}$$

gVars

$$\frac{\bullet_{\mathit{Map}}}{(\text{"bot.map\_size"} \mapsto \texttt{tuple2Klist}\ (T))\ \ (\text{"bot.map\_obstacles"} \mapsto \bullet_{\mathit{List}})}$$

[structural]

SYNTAX $\mathit{ISiteId} ::= \texttt{bot.init}$

BOT.INIT-1

k

$$\frac{\texttt{handle (bot.init(}\bullet_{\mathit{Args}}\texttt{))}}{\texttt{handle (bot.init(}\text{"Default"}\texttt{))}}$$

[macro]

BOT.INIT-2

k
$$\frac{\texttt{handle (bot.init(}S{:}\mathit{String}\texttt{))}}{\texttt{pubHandle (}S\texttt{)}}$$

gVars
$$\frac{\bullet_{\mathit{Map}}}{\texttt{(botStr (}S\texttt{, "position")} \mapsto \texttt{ListItem (1) ListItem (1))}}$$

gVars
$$\frac{\bullet_{\mathit{Map}}}{\texttt{(botStr (}S\texttt{, "direction")} \mapsto \texttt{ListItem (0) ListItem (1))}}$$

gVars
$$\frac{\bullet_{\mathit{Map}}}{\texttt{(botStr (}S\texttt{, "bot\_lock")} \mapsto \texttt{false)}}$$

gVars
$$\frac{\bullet_{\mathit{Map}}}{\texttt{(botStr (}S\texttt{, "is\_bumper\_hit")} \mapsto \texttt{false)}}$$

gVars
$$\frac{\bullet_{\mathit{Map}}}{\texttt{(botStr (}S\texttt{, "wall\_indicator")} \mapsto \texttt{ListItem (-1) ListItem (-1))}}$$

gVars
$$\frac{\bullet_{\mathit{Map}}}{\texttt{(botStr (}S\texttt{, "block\_indicator")} \mapsto \texttt{ListItem (-1) ListItem (-1))}}$$

[structural]

BOT.INIT-3

k
$$\frac{\texttt{handle (bot.init(}T1{:}\mathit{Tuple}\texttt{, }T2{:}\mathit{Tuple}\texttt{))}}{\texttt{handle (bot.init("Default", }T1\texttt{, }T2\texttt{))}}$$

[macro]

BOT.INIT-4

k

$$\frac{\texttt{handle}\,(\,\texttt{bot.init}(S{:}\mathit{String},\ T1{:}\mathit{Tuple},\ T2{:}\mathit{Tuple}))}{\texttt{pubHandle}\,(S)}$$

gVars

$$\frac{\bullet_{\mathit{Map}}}{(\,\texttt{botStr}\,(S, \text{"position"}) \mapsto \texttt{tuple2Klist}\,(T1))}$$

gVars

$$\frac{\bullet_{\mathit{Map}}}{(\,\texttt{botStr}\,(S, \text{"direction"}) \mapsto \texttt{tuple2Klist}\,(T2))}$$

gVars

$$\frac{\bullet_{\mathit{Map}}}{(\,\texttt{botStr}\,(S, \text{"bot\_lock"}) \mapsto \text{false})}$$

gVars

$$\frac{\bullet_{\mathit{Map}}}{(\,\texttt{botStr}\,(S, \text{"is\_bumper\_hit"}) \mapsto \text{false})}$$

gVars

$$\frac{\bullet_{\mathit{Map}}}{(\,\texttt{botStr}\,(S, \text{"wall\_indicator"}) \mapsto \texttt{ListItem}\,(\text{-1})\ \ \texttt{ListItem}\,(\text{-1}))}$$

gVars

$$\frac{\bullet_{\mathit{Map}}}{(\,\texttt{botStr}\,(S, \text{"block\_indicator"}) \mapsto \texttt{ListItem}\,(\text{-1})\ \ \texttt{ListItem}\,(\text{-1}))}$$

[structural]

SYNTAX *ISiteId* ::= `bot.setPosition`

BOT.SETPOSITION

k

$$\frac{\texttt{handle}\,(\,\texttt{bot.setPosition}(S{:}\mathit{String},\ T{:}\mathit{Tuple}))}{\texttt{pubHandle}\,(\,\texttt{signal})}$$

gVars

`botStr` $(S, \texttt{"position"}) \mapsto \dfrac{\text{—}}{\texttt{tuple2Klist}\ (T)}$

[structural]

SYNTAX *ISiteId* ::= `bot.setDirection`

BOT.SETDIRECTION

k

$\dfrac{\texttt{handle}\ (\ \texttt{bot.setDirection}(S{:}String\texttt{,}\ T{:}Tuple))}{\texttt{pubHandle}\ (\ \texttt{signal})}$

gVars

`botStr` $(S, \texttt{"direction"}) \mapsto \dfrac{\text{—}}{\texttt{tuple2Klist}\ (T)}$

[structural]

SYNTAX *ISiteId* ::= `bot.setObstacles`

SYNTAX *KItem* ::= `addObstacle` $(Tuple)$

BOT.SETOBSTACLES

k

$\dfrac{\texttt{handle}\ (\ \texttt{bot.setObstacles}(As{:}Args))}{\texttt{handle}\ (\ \texttt{bot.addObstacles}(As))}$

gVars

$\texttt{"bot.map\_obstacles"} \mapsto \dfrac{L{:}List}{\bullet_{List}}$

[structural]

SYNTAX *ISiteId* ::= `bot.addObstacles`

BOT.ADDOBSTACLES-1

k

$\dfrac{\texttt{handle}\ (\ \texttt{bot.addObstacles}(T{:}Tuple\texttt{,}\ As{:}Args))}{\texttt{addObstacle}\ (T) \curvearrowright \texttt{handle}\ (\ \texttt{bot.addObstacles}(As))}$

[structural]

BOT.ADDOBSTACLES-2

k
$$\frac{\texttt{handle}\ (\ \texttt{bot.addObstacles}(\bullet_{Args}))}{\texttt{pubHandle}\ (\ \texttt{signal})}$$

[structural]

BOT.ADDOBSTACLES-3

k
$$\frac{\texttt{addObstacle}\ (T{:}Tuple)}{\bullet_{K}}$$

gVars
$$\text{"bot.map\_obstacles"} \mapsto L{:}List\ \frac{\bullet_{List}}{\texttt{ListItem}\ (\ \texttt{tuple2Klist}\ (T))}$$

[structural]

SYNTAX $KItem$ ::= `botMoved` ($String$, $Bool$) [function]

BOTMOVED

k
$$\frac{\texttt{botMoved}\ (S, \text{—})}{\texttt{signal}}$$

gVars
$$\left(Lock{:}String \mapsto \frac{\text{true}}{\text{false}} \right)$$

gVars
$$\left(Wall{:}String \mapsto \frac{\text{—}{:}List}{\texttt{ListItem}\ (\text{-}1)\ \ \texttt{ListItem}\ (\text{-}1)} \right)$$

gVars
$$\left(Block{:}String \mapsto \frac{\text{—}{:}List}{\texttt{ListItem}\ (\text{-}1)\ \ \texttt{ListItem}\ (\text{-}1)} \right)$$

requires $Lock$ `==String` `botStr` $(S,$ "bot_lock"$)$
$\wedge_{Bool}$ $Wall$ `==String` `botStr` $(S,$ "wall_indicator"$)$
$\wedge_{Bool}$ $Block$ `==String` `botStr` $(S,$ "block_indicator"$)$
[structural]

SYNTAX $String$ ::= `botStr` $(String, String)$ [function]

BOTSTR

$$\frac{\texttt{botStr}\ (S1, S2)}{\text{"bot."} +_{String} S1 +_{String} \text{"."} +_{String} S2}$$

[anywhere]

A.7.2 Timed Bot Sites

Define scan site

SYNTAX $BotTSite$ ::= `bot.scan`

BOT.SCAN-1

k

$$\frac{\texttt{handle}\ (\ \texttt{bot.scan}(\bullet_{Args}))}{\texttt{handle}\ (\ \texttt{bot.scan}(\text{"Default"}))}$$

[macro]

BOT.SCAN-2

k

$$\frac{\texttt{handle}\ (\ \texttt{bot.scan}(S{:}String))}{\texttt{timedHandle}\ (1,\ \texttt{bot.scan}(S),\ \texttt{bot\_lock})}$$

[structural]

BOT.SCAN-3

k

$$\frac{\texttt{timedHandle}\ (0,\ \texttt{bot.scan}(S),\ \texttt{bot\_lock})}{\texttt{timedHandle}\ (0,\ \texttt{bot.scan}(S),\ Condition)}$$

gVars

("bot.map_size" $\mapsto$ $Dims{:}List$)

gVars

("bot.map_obstacles" $\mapsto$ $Obs{:}List$)

gVars
$(PosStr \mapsto Pos{:}List)$

gVars
$(DirStr \mapsto Dir{:}List)$

requires
$PosStr$==Stringbot`Str`$(S,$"position"$)$
$\wedge_{Bool} DirStr$ ==String botStr $(S,$"direction"$)$
$\wedge_{Bool} Condition$ ==Bool (checkWall (vSub (vAdd $(Pos, Dir), Dims))$
$\vee_{Bool}$ checkBlock (vAdd $(Pos, Dir), Obs))$
[structural]

Define step forward site

SYNTAX $BotTSite$ ::= `bot.stepFwd`

SYNTAX $Bool$ ::= `checkWall` $(List)$ [function]

SYNTAX $Bool$ ::= `checkBlock` $(List, List)$ [function]

BOT.STEPFWD-1

k
$$\frac{\texttt{handle (bot.stepFwd(}\bullet_{Args}\texttt{))}}{\texttt{handle (bot.stepFwd("Default"))}}$$
[macro]

BOT-STEPFWD-2

k
$$\frac{\texttt{handle (bot.stepFwd(}S{:}String\texttt{))}}{\texttt{timedHandle (3, bot.stepFwd(}S\texttt{), bot\_lock)}}$$

gVars
$$\left(Lock \mapsto \frac{\mathsf{false}}{\mathsf{true}}\right) \left(Bumper \mapsto \frac{—}{\mathsf{false}}\right)$$

gVars
$(PosStr \mapsto Pos{:}List)$

gVars
$(DirStr \mapsto Dir{:}List)$

gVars

$$\left(\mathit{Wall} \mapsto \frac{\text{—}:\mathit{List}}{\texttt{vSub}\ (\ \texttt{vAdd}\ (\mathit{Pos}, \mathit{Dir}), \mathit{Dims})} \right)$$

gVars

$$\left(\mathit{Block} \mapsto \frac{\text{—}:\mathit{List}}{\texttt{vAdd}\ (\mathit{Pos}, \mathit{Dir})} \right)$$

gVars

("bot.map_obstacles" $\mapsto$ Obs:List)

gVars

("bot.map_size" $\mapsto$ Dims:List)

requires Lock `==String` `botStr` (S, "bot_lock")
$\wedge_{\mathit{Bool}}\mathit{Bumper}$ `==String` `botStr` (S, "is_bumper_hit")
$\wedge_{\mathit{Bool}}\mathit{PosStr}$ `==String` `botStr` (S, "position")
$\wedge_{\mathit{Bool}}\mathit{DirStr}$ `==String` `botStr` (S, "direction")
$\wedge_{\mathit{Bool}}\mathit{Wall}$ `==String` `botStr` (S, "wall_indicator")
$\wedge_{\mathit{Bool}}\mathit{Block}$ `==String` `botStr` (S, "block_indicator")
[transition]

CHECKWALL

$$\frac{\texttt{checkWall}\ (\mathit{WI})}{1\ \texttt{in}\ \mathit{WI}}$$

[anywhere, function]

CHECKBLOCK

$$\frac{\texttt{checkBlock}\ (\mathit{BI}, \mathit{Obs})}{\mathit{BI}\ \texttt{in}\ \mathit{Obs}}$$

[anywhere, function]

Stepped forward and not hit a wall or block
STEP-DONE-NO-HIT

k

$$\frac{\texttt{timedHandle}\ (0,\ \texttt{bot.stepFwd}(S{:}\mathit{String}), \text{—})}{\texttt{botMoved}\ (S, \text{true})}$$

gVars

$$\left(\mathit{PosStr} \mapsto \frac{\mathit{Pos}}{\texttt{vAdd}\,(\mathit{Pos}, \mathit{Dir})} \right)$$

gVars

$(\mathit{DirStr} \mapsto \mathit{Dir})$

gVars

$(\text{"bot.map\_obstacles"} \mapsto \mathit{Obs}{:}\mathit{List})$

gVars

$(\mathit{Wall} \mapsto \mathit{WL}{:}\mathit{List})$

gVars

$(\mathit{Block} \mapsto \mathit{BI}{:}\mathit{List})$

requires $(\neg_{\mathit{Bool}} 1 \ \texttt{in}\ \mathit{WL}$
$\wedge_{\mathit{Bool}} \neg_{\mathit{Bool}} \mathit{BI}\ \texttt{in}\ \mathit{Obs}$
$\wedge_{\mathit{Bool}} (\mathit{PosStr}\ \texttt{==String}\ \texttt{botStr}\ (S, \text{"position"}))$
$\wedge_{\mathit{Bool}} (\mathit{DirStr}\ \texttt{==String}\ \texttt{botStr}\ (S, \text{"direction"}))$
$\wedge_{\mathit{Bool}} (\mathit{Wall}\ \texttt{==String}\ \texttt{botStr}\ (S, \text{"wall\_indicator"}))$
$\wedge_{\mathit{Bool}} (\mathit{Block}\ \texttt{==String}\ \texttt{botStr}\ (S, \text{"block\_indicator"}))$
[structural]

Tried to step forward but hit a wall or block
NO-STEP-CAUSE-HIT

k

$$\frac{\texttt{timedHandle}\,(0,\ \texttt{bot.stepFwd}(S{:}\mathit{String}), —)}{\texttt{botMoved}\,(S, \text{true})}$$

gVars

$$\left(\mathit{Bumper} \mapsto \frac{—}{\text{true}} \right)$$

gVars

$\text{"bot.map\_obstacles"} \mapsto \mathit{Obs}{:}\mathit{List}$

gVars

$\mathit{Wall} \mapsto \mathit{WL}{:}\mathit{List}$

gVars

$\mathit{Block} \mapsto \mathit{BI}{:}\mathit{List}$

requires $(1\ \mathtt{in}\ \mathit{WL} \lor_{\mathit{Bool}} \mathit{BI}\ \mathtt{in}\ \mathit{Obs})$
$\land_{\mathit{Bool}}(\mathit{Bumper}\ \mathtt{==String}\ \mathtt{botStr}\ (S, \text{"is\_bumper\_hit"}))$
$\land_{\mathit{Bool}}(\mathit{Wall}\ \mathtt{==String}\ \mathtt{botStr}\ (S, \text{"wall\_indicator"}))$
$\land_{\mathit{Bool}}(\mathit{Block}\ \mathtt{==String}\ \mathtt{botStr}\ (S, \text{"block\_indicator"}))$
[structural]

Rotate right (clockwise)

SYNTAX $\mathit{BotTSite} ::=$ `bot.rotateRight`

BOT-ROTATERIGHT-1

k

$$\frac{\mathtt{handle}\ (\ \mathtt{bot.rotateRight}(\bullet_{\mathit{Args}}))}{\mathtt{handle}\ (\ \mathtt{bot.rotateRight}(\text{"Default"}))}$$

[macro]

BOT-ROTATERIGHT-2

k

$$\frac{\mathtt{handle}\ (\ \mathtt{bot.rotateRight}(S{:}\mathit{String}))}{\mathtt{timedHandle}\ (1,\ \mathtt{bot.rotateRight}(S),\ \mathtt{bot\_lock})}$$

gVars

$$\mathit{Lock} \mapsto \frac{\mathsf{false}}{\mathsf{true}}$$

requires $\mathit{Lock}\ \mathtt{==String}\ \mathtt{botStr}\ (S, \text{"bot\_lock"})$
[transition]

BOT-ROTATERIGHT-3

k

$$\frac{\mathtt{timedHandle}\ (0,\ \mathtt{bot.rotateRight}(S{:}\mathit{String}), \text{—})}{\mathtt{botMoved}\ (S, \mathsf{true})}$$

gVars

$$\left(\mathit{DirStr} \mapsto \frac{\mathit{Direction}}{\mathtt{cw}\ (\mathit{Direction})}\right) \left(\mathit{Bumper} \mapsto \frac{\text{—}}{\mathsf{false}}\right)$$

requires $(\mathit{Bumper}\ \mathtt{==String}\ \mathtt{botStr}\ (S, \text{"is\_bumper\_hit"}))$
$\land_{\mathit{Bool}}(\mathit{DirStr}\ \mathtt{==String}\ \mathtt{botStr}\ (S, \text{"direction"}))$
[structural]

Rotate left (counter-clockwise)

SYNTAX $\quad BotTSite$ ::= `bot.rotateLeft`

BOT-ROTATELEFT-1

k
$$\frac{\texttt{handle}\ (\ \texttt{bot.rotateLeft}(\bullet_{Args}))}{\texttt{handle}\ (\ \texttt{bot.rotateLeft}(\text{"Default"}))}$$

[macro]

BOT-ROTATELEFT-2

k
$$\frac{\texttt{handle}\ (\ \texttt{bot.rotateLeft}(S{:}String))}{\texttt{timedHandle}\ (1,\ \texttt{bot.rotateLeft}(S),\ \texttt{bot\_lock})}$$

gVars
$$Lock \mapsto \frac{\text{false}}{\text{true}}$$

requires $Lock$ ==`String` `botStr` (S, "bot_lock")
[transition]

BOT-ROTATELEFT-3

k
$$\frac{\texttt{timedHandle}\ (0,\ \texttt{bot.rotateLeft}(S{:}String), \text{—})}{\texttt{botMoved}\ (S, \text{true})}$$

gVars
$$\left(DirStr \mapsto \frac{Direction}{\texttt{ccw}\ (Direction)}\right) \quad \left(Bumper \mapsto \frac{\text{—}}{\text{false}}\right)$$

requires ($Bumper$ ==`String` `botStr` (S, "is_bumper_hit"))
$\wedge_{Bool}$($DirStr$ ==`String` `botStr` (S, "direction"))
[structural]

Generic rule to stop execution if lock on robot

BOT-LOCKED

k
$$\frac{\texttt{handle}\ (Site{:}BotTSite(S{:}String))}{\texttt{silentHandle}\ (Site(S))}$$

gVars
$Lock \mapsto$ true

requires $Lock$ ==`String` `botStr` (S, "bot_lock")
[structural]

Functions to process rotating the robot

SYNTAX $\quad List$ ::= `cw` ($List$) [function]

$$\frac{\texttt{cw}\,(\,\texttt{ListItem}\,(\text{-}1)\ \ \texttt{ListItem}\,(0))}{\texttt{ListItem}\,(0)\ \ \texttt{ListItem}\,(1)}$$
[anywhere, function]

$$\frac{\texttt{cw}\,(\,\texttt{ListItem}\,(0)\ \ \texttt{ListItem}\,(\text{-}1))}{\texttt{ListItem}\,(\text{-}1)\ \ \texttt{ListItem}\,(0)}$$
[anywhere, function]

$$\frac{\texttt{cw}\,(\,\texttt{ListItem}\,(1)\ \ \texttt{ListItem}\,(0))}{\texttt{ListItem}\,(0)\ \ \texttt{ListItem}\,(\text{-}1)}$$
[anywhere, function]

$$\frac{\texttt{cw}\,(\,\texttt{ListItem}\,(0)\ \ \texttt{ListItem}\,(1))}{\texttt{ListItem}\,(1)\ \ \texttt{ListItem}\,(0)}$$
[anywhere, function]

SYNTAX $\mathit{List} ::= \texttt{ccw}\,(\mathit{List})$ [function]

$$\frac{\texttt{ccw}\,(\,\texttt{ListItem}\,(0)\ \ \texttt{ListItem}\,(1))}{\texttt{ListItem}\,(\text{-}1)\ \ \texttt{ListItem}\,(0)}$$
[anywhere, function]

$$\frac{\texttt{ccw}\,(\,\texttt{ListItem}\,(\text{-}1)\ \ \texttt{ListItem}\,(0))}{\texttt{ListItem}\,(0)\ \ \texttt{ListItem}\,(\text{-}1)}$$
[anywhere, function]

$$\frac{\texttt{ccw}\,(\,\texttt{ListItem}\,(0)\ \ \texttt{ListItem}\,(\text{-}1))}{\texttt{ListItem}\,(1)\ \ \texttt{ListItem}\,(0)}$$
[anywhere, function]

$$\frac{\texttt{ccw}\,(\,\texttt{ListItem}\,(1)\ \ \texttt{ListItem}\,(0))}{\texttt{ListItem}\,(0)\ \ \texttt{ListItem}\,(1)}$$
[anywhere, function]

END MODULE

A.8 LTL Model Checking

MODULE ORC-LTL

An input program can be a LTL Formula (this is needed for parsing purposes)

SYNTAX *Pgm* ::= *LtlFormula*

We extend Orc expressions with labeled expressions to be able to specify inside an Orc program, using LTL formulas, when an expressions is reached.

SYNTAX *Exp* ::= *Id* : *Exp*

The imported module `MODEL-CHECKER-HOOKS` is a 𝕂 interface to the Maude module defining the syntax for the model-checker. In addition to this interface, we have to define the atomic propositions. Here we define a small set of such propositions. The semantics for these propositions will be given in the next module.

SYNTAX *Prop* ::= *Id*
| `gVarEqTo` (*String*, *Val*)
| `gVarEqTo` (*String*, *Tuple*)
| `gVarEqTo` (*String*, *List*)
| `isGPublished` (*Int*)

This module combines the semantics of Orc with an interface to the model-checker, given by the module `LTL-HOOKS`. The states of the transition system that will be be model-checked are given by the configurations of Orc programs, which are of sort `Bag`.

Semantics of the labeled expressions:

LTL-LABELED-EXP-1

$$\frac{L{:}Id\ :\ E{:}Exp}{E}$$

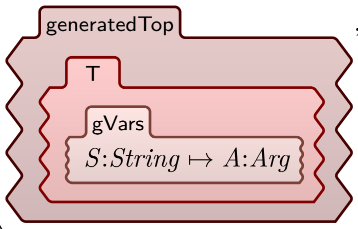
[transition]

In order to give semantics to the proposition `gVarEqTo`, we use auxiliary functions like `gVarGet` that returns a certain value from inside a given configuration:

SYNTAX *Arg* ::= `gVarGet` (*Bag*, *String*) [function]

LTL-GVARGET

`gVarGet` (



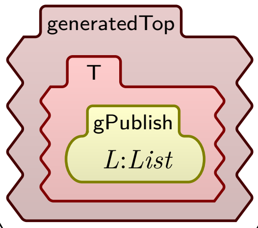
, S)

A

SYNTAX *List* ::= `gPublishGet` (*Bag*) [function]

LTL-gPublishGet

`gPublishGet` (



)

L

We are ready now to give the semantics for atomic propositions:

LTL-gVarEqTo

B:*Bag* $\models_{Ltl}$ `gVarEqTo` (S:*String*, V:*Val*)

---

true

requires `gVarGet` (B, S) $=_K$ V
[anywhere, ltl]

LTL-gVarEqTo

`gVarEqTo` (S:*String*, T:*Tuple*)

`gVarEqTo` (S, `tuple2Klist` (T))
[macro]

LTL-gVarEqTo

B:*Bag* $\models_{Ltl}$ `gVarEqTo` (S:*String*, L:*List*)

---

true

requires `gVarGet` (B, S) $=_K$ L
[anywhere, ltl]

LTL-isGPublished

B:*Bag* $\models_{Ltl}$ `isGPublished` (I:*Int*)

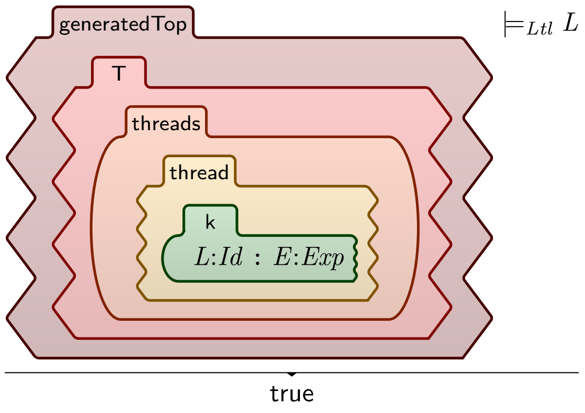
---

true

requires I `in` `gPublishGet` (B)

[anywhere, ltl]

LTL-Label-evaluation



[anywhere]

END MODULE

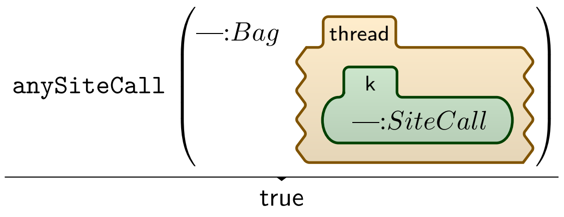
A.9 Predicates

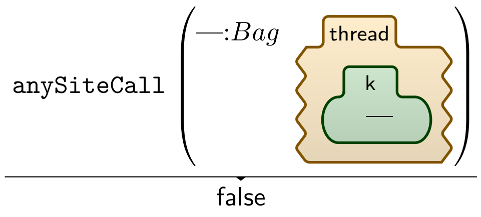
MODULE ORC-PREDICATES

These functions are used in side conditions of other rules to find if a certain type of thread exists in the current configuration.

Is there any thread that has a site call that hasn't been made?

SYNTAX $Bool ::=$ `anySiteCall` (Bag) [function]

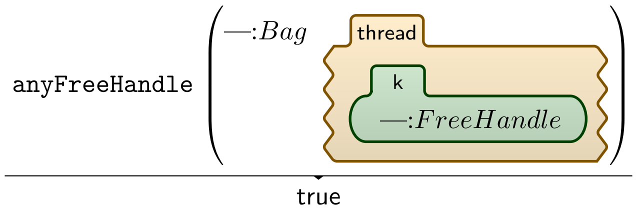


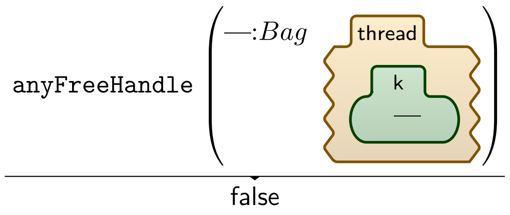


[owise]

Is there any thread that has an unprocessed handle?

SYNTAX *Bool* ::= `anyFreeHandle` (*Bag*) [function]

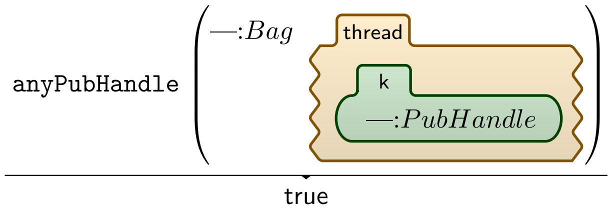


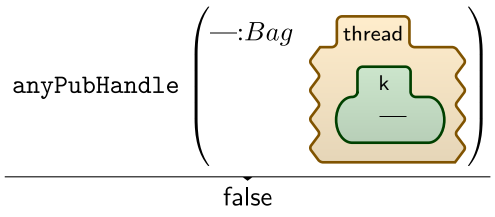


[owise]

Is there any thread that has a handle ready to publish a value?

SYNTAX *Bool* ::= `anyPubHandle` (*Bag*) [function]

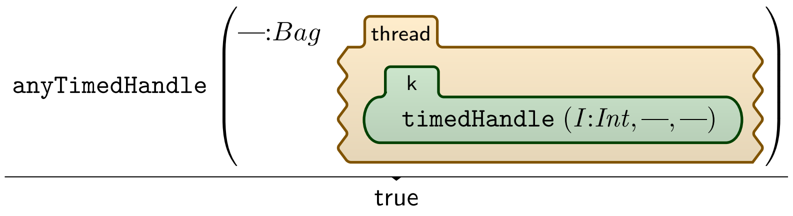




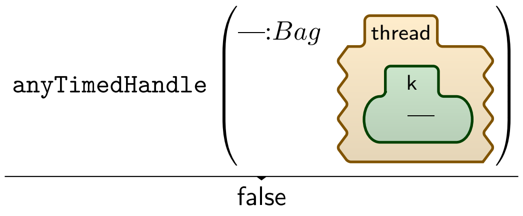
[owise]

Is there any thread that has a timed handle

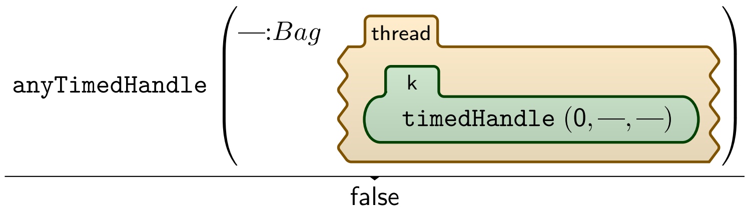
SYNTAX $Bool$ ::= `anyTimedHandle` (Bag) [function]



requires $I >_{Int} 0$

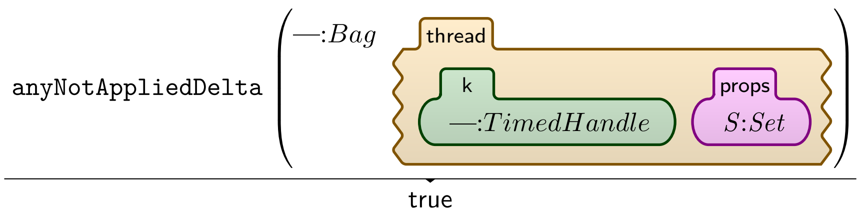


[owise]

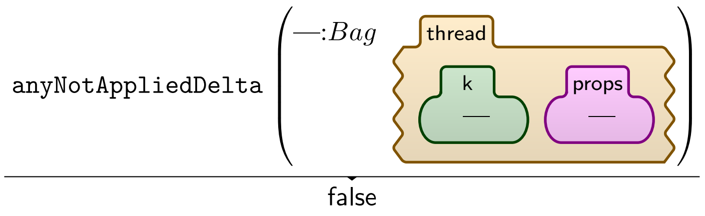


Is there any thread that hasn't yet applied delta

SYNTAX $Bool$ ::= `anyNotAppliedDelta` (Bag) [function]



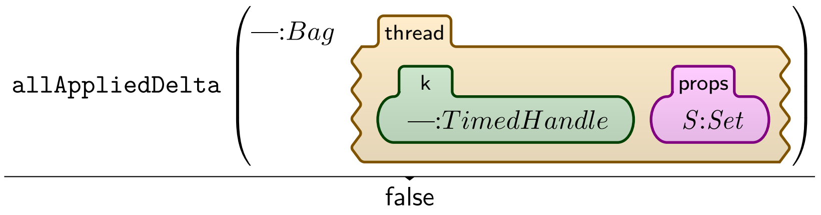
requires $\neg_{Bool}$"applied_delta" `in` S



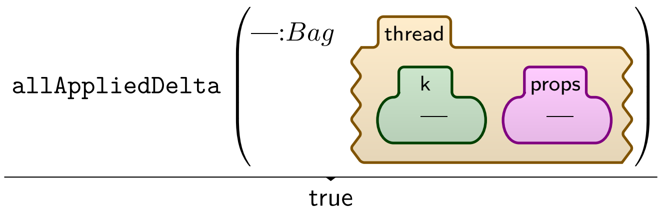
[owise]

This is equivalent to `notBool anyNotAppliedDelta`

SYNTAX $\quad$ *Bool* ::= `allAppliedDelta` (*Bag*) [function]



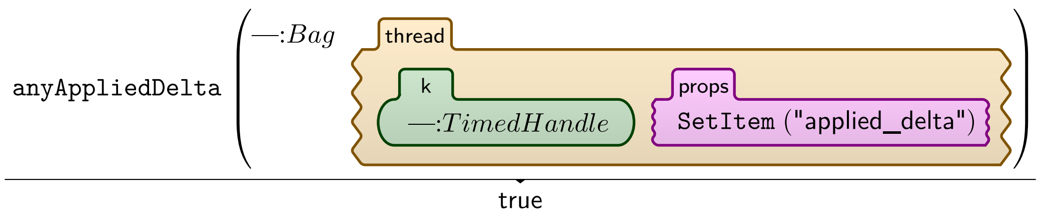
requires $\neg_{Bool}$"applied_delta" `in` S

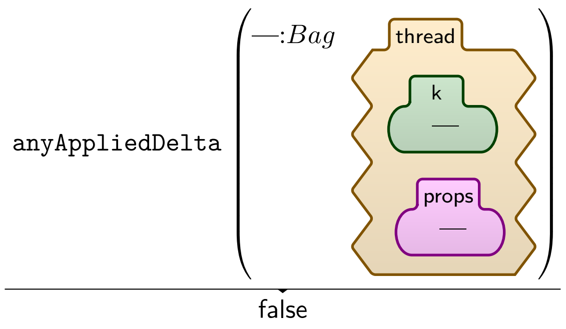


[owise]

Is there any thread that has the `applied_delta` flag on and is not reset?

SYNTAX $\quad$ *Bool* ::= `anyAppliedDelta` (*Bag*) [function]





[owise]

END MODULE

APPENDIX B

Testing Examples

This appendix contains all test examples used for validation. It contains 86 examples. A summary of it can be found in Table 4.3. Each example has three main elements.

- What it tests.
- The commands used to execute it.
- The expected output.
- The Orc program itself.
- Some discussion as needed.

All of these examples have been executed using the shown commands and their results matched the expected output shown.

B.1 Combinators

Example B.1.1. Parallel
Tests basic parallel expression.

```
Add(0,1) | Add(0,2) | Add(0,3)
```

Command	Expected output
`krun`	Publishes 1, 2 and 3.

Example B.1.2. Sequential
Tests basic sequential expression.

```
(Add(0,3)) > x > Add(x,4) > y > print(y)
```

Command	Expected output
`krun`	Prints 7 and publishes `signal`.

Example B.1.3. Parallel and Sequential
Tests parallel spawning of threads.

```
(Add(0,1) | Add(0,2)) > x > Add(x,3)
```

Command	Expected output
`krun`	Publishes 4 and 5.

Example B.1.4. Parallel and Sequential
Tests passing parameters in a nested sequential composition.

```
(Add(0,1) | Add(0,2)) > x > (Add(x,1) > y > Add(x,y))
```

Command	Expected output
`krun`	Publishes 3 and 5.

Example B.1.5. Parallel and Sequential
On top of basic testing of parallel spawning, it tests precedence of sequential over parallel.

```
(Add(0,1) | Add(0,2)) > x > Add(x,1) | print(100)
```

Command	Expected output
`krun`	Prints 100 and publishes 2, 3 and a `signal`.

Example B.1.6. Parallel and Sequential
A bigger example but basically tests the same things. It imitates the first few iterations of a Fibonacci function.

Command	Expected output
`krun`	Publishes 5 and 8.

```
(Add(0,1) | Add(0,2)) > x > (Add(x,1) > y > (Add(x,y) > z
   > Add(y,z)))
```

Example B.1.7. Parallel and Pruning
Tests basic pruning of a parallel expression.

Command	Expected output
`krun --search --pattern "<gPublish> L:List </gPublish>"`	Two solutions for `L`: `11` and `21`.

```
Add(x,1) < x < (Add(0,20) | Add(0,10))
```

Example B.1.8. Parallel and Pruning
Tests basic pruning of a parallel expression, and the precedence of parallel over pruning.

Command	Expected output
`krun --search --pattern "<gPublish> L:List </gPublish>"`	Four solutions for `L`: `11`, `21`, `31` and `41`.

```
Add(x,1) < x < Add(0,10) | Add(0,20) | Add(0,30) | Add
   (0,40)
```

Example B.1.9. Parallel, Sequential and Pruning
Tests parallel spawning and pruning as well as precedence.

```
print(y) < y < (Add(0,1) | Add(0,2)) > x > Add(x,3)
```

Example B.1.10. Pruning
Tests basic pruning.

Command	Expected output
`krun`	Publishes 2.

Command	Expected output
`krun`	Prints either `4` or `5`.
`krun --search --pattern "<out> L:List </out>"`	Two solutions for `L`: `4` and `5`.
`krun --ltlmc "<>Ltl isPrinted(4)"`	`true`

```
Add(x,1) < x < Add(0,1)
```

Example B.1.11. Pruning
Tests basic pruning, but more importantly tests the variable lookup mechanism that requests variable from parents.

Command	Expected output
`krun`	Publishes `5`.

```
(Add(f1,f2) < f2 < Add(f1,1)) < f1 < Add(1,1)
```

Example B.1.12. Pruning
Similar to Example B.1.6, but uses pruning instead of sequential. More importantly, it tests variable lookup and scope sharing.

Command	Expected output
`krun`	Publishes `8`.

```
((Add(f2,f3) < f3 < Add(f1,f2)) < f2 < Add(f1,1)) < f1 <
    Add(1,1)
```

Example B.1.13. Pruning
This example is almost the same as Example B.1.12 except that its brackets have been rearranged. This tests variable lookup and scope sharing and ensures that no incorrect scope sharing is occurring because in this case, the program should in fact not find a value and should get stuck. See the details in the code's comments.

Command	Expected output
`krun`	Gets stuck.

```
(Add(f2,f3) < f3 < (Add(f1,f2) < f2 < Add(f1,1))) < f1 <
    Add(1,1)

/* Here is how it is structured

                              prunMgr(f1)
                              /          \
                         prunMgr(f3)    1+1
                        /         \
  This f2 --------> f2+f3         prunMgr(f2)
  is stuck.                      /          \
  It should not               f1+f2          1+f1
  see the value
  provided by
  prunMgr(f2).

*/
```

Example B.1.14. Otherwise

```
print("Success!") ; print("Failure!")
```

Example B.1.15. Otherwise

```
stop ; print("Success!")
```

B.2 LTL Model Checking

Here are basic examples that use only constants and the four operators to check if `ltlmc` functionality is working.

Example B.2.1.

Command	Expected output
`krun --ltlmc "<>Ltl isGPublished(3)"`	`true`.
`krun --ltlmc "[]Ltl isGPublished(3)"`	`true`.
`krun --ltlmc "gVarEqTo(\"clock\",0)"`	`true`.

```
Add(1,r) < r < 2
```

Example B.2.2.

Command	Expected output
`krun --ltlmc "<>Ltl isGPublished(3)"`	`true`.
`krun --ltlmc "[]Ltl isGPublished(3)"`	Gives a counter example showing mid-execution configuration where `3` hasn't been published yet.
`krun --ltlmc "gVarEqTo(\"clock\",0)"`	`true`.

```
1 | 2 | 3
```

Example B.2.3.

Command	Expected output
`krun --ltlmc "<>Ltl isGPublished(3)"`	`true`.
`krun --ltlmc "<>Ltl []Ltl isGPublished(3)"`	`true`.
`krun --ltlmc "[]Ltl <>Ltl isGPublished(3)"`	`true`.

```
1 | 2 >> 3
```

B.3 Expression Definition and Calling

Example B.3.1. Basic Definition and Call
Tests basic expression definition and call.

Command	Expected output
`krun`	Prints `6` and publishes `signal`.

```
SumAndPrint(a,b,c) := Sum(a,b,c) > x > print(x)
SumAndPrint(1,2,3)
```

Example B.3.2. Nested Definition and Call
Tests:

- Nested expression definition and call.
- Scope of variables, i.e, vars with the same name don't mix up.

Command	Expected output
`krun`	Prints `5` and publishes `signal`.

```
Sum3(x,y,z) := Add(x,a) < a < Add(y,z)
Sum2(x,y) := Sum3(x,y,0)
Sum2(2,3) > x > print(x)
```

Example B.3.3. Nested Definition and Call
Tests:

- Nested expression definition and call.
- Scope of variables, i.e, vars with the same name don't mix up.

Command	Expected output
`krun`	Prints `10` and publishes `signal`.

```
Sum3(x,y,z) := Add(x,a) < a < Add(y,z)
Sum2(a1,a2) := Sum3(a1,a2,0)
(Sum2(4,f1) < f1 < Sum3(1,2,3)) > x > print(x)
```

This example is more complex than Example B.3.2 because it has one more layer in the expression calling stack. To debug this, let us take the topmost pruning expression. It has two child threads. Call them *left* and *right*. After one step of execution, we'll have two threads:

```
  left:
  tid=1
  Add(x,a) < a < Add(y,z)
  a1 -> 4
  a2 -> f1
  x -> a1
  y -> a2
  z -> 0

  right:
  tid=2
  Add(x,a) < a < Add(y,z)
  x -> 1
  y -> 2
  z -> 3
```

left publishes `6` up. Now in the next execution step, *right* (2) gives `f1 -> 6` to left (1), but if *left* copied context from right or manager, then it overwrites things. Next step, here is what we have:

```
  left:
  tid=1
  Add(x,a) < a < Add(y,z)
  a1 -> 4
  a2 -> f1
  x -> a1
  y -> a2
  z -> 0
  f1 -> 6

  right:
  tid=2
  Add(x,a) < a < Add(y,z)
  x -> 1
  y -> 2
  z -> 3
```

Example B.3.4. Nested Definition and Call

Tests:

- Nested expression definition and call.

- Calling inside a nested pruning expression.
- Scope of variables by calling with identical variable names.

Command	Expected output
`krun`	Prints `15` and publishes `signal`.

```
Sum3(x,y,z) := Add(x,a) < a < Add(y,z)
print(x) < x < (Sum3(4,5,x) < x < Sum3(1,2,3))
```

This example tests that the two `x`'s are not confused. So it should print `15`, because if they were confused, it would print `13`.

Example B.3.5. Nested Definition and Call
Tests:

- Nested expression definition and call.
- Calling inside a nested pruning expression.
- Scope of variables by calling with identical variable names.

Command	Expected output
`krun`	Prints `15` and publishes `signal`.

```
Sum3(x,y,z) := Add(x,a) < a < Add(y,z)
print(f2) < f2 < (Sum3(4,5,x) < x < Sum3(1,2,3))
```

Similarly to the previous example, this should print `15` and not `13`.

Example B.3.6. Nested Definition and Call
Tests:

- Nested expression definition and call.
- Calling inside a nested pruning expression.
- Scope of variables by calling with identical variable names.

Command	Expected output
`krun`	Prints `15` and publishes `signal`.

```
Sum3(x,y,z) := Add(x,a) < a < Add(y,z)
print(x) < x < (Sum3(4,5,f1) < f1 < Sum3(1,2,3))
```

Similar to the previous example, but with increased complexity.

Example B.3.7. Nested Definition and Call
Tests:

- Correct parsing and precedence among operators.
- Nested expression definition and call.
- Calling inside a nested pruning expression.
- Scope of variables by calling with identical variable names.

Command	Expected output
`krun`	Prints `15` and publishes `signal`.

```
Sum3(x,y,z) := Add(x,a) < a < Add(y,z)
print(x) < x < Sum3(4,5,f1) < f1 < Sum3(1,2,3)
```

Similar to the previous example, but with no parentheses.

Example B.3.8. Nested Definition and Call
Tests:

- Nested expression definition and call.
- Calling inside a nested pruning expression.
- Scope of variables by calling with identical variable names.
- Value passing through sequential and pruning expressions.

Command	Expected output
`krun`	Prints `15` and publishes `signal`.

```
Sum3(x,y,z) := Add(x,a) < a < Add(y,z)
(Sum3(4,5,f1) < f1 < Sum3(1,2,3)) > x > print(x)
```

This changes pruning from the previous example to sequential to make sure that scope sharing and value passing is done correctly through pruning as well as sequential expressions.

Example B.3.9. Nested Definition and Call
Tests:

- Nested expression definition and call.
- Calling inside a nested pruning expression.
- Scope of variables by calling with identical variable names.

Command	Expected output
`krun`	Prints `15` and publishes `signal`.

```
Sum3(x,y,z) := Add(x,a) < a < Add(y,z)
(Sum3(4,5,x) < x < Sum3(1,2,3)) > f2 > print(f2)
```

This has a small variable name change over the previous example to test if the name would be confused with the one in the definition.

Example B.3.10. Factorial
Tests recursion through the factorial function.

```
// Define factorial
Factorial(x) := (((if(r) < r < Equals(x,0)) >> 1) | ((if(r
    ) < r < Gr(x,0)) >> (Mul(a,x) < a < (Factorial(b) < b <
     Sub(x,1)))))

// Call factorial(5)
Factorial(5) > f > print(f)
```

B.4 Time

Example B.4.1. Tests time through the sites *Rtimer* and *Clock*.

Command	Expected output
`krun`	Publishes two `signal`'s and a `1`. Clock reaches 3.

```
Rtimer(3) | Rtimer(2) | Add(0,1)
```

Example B.4.2. Tests time through the sites *Rtimer* and *Clock*.

Command	Expected output
`krun`	Prints `3` and publishes `signal`.

```
Rtimer(3) >> (print(x) < x < clock)
```

Example B.4.3. Tests time through the sites *Atimer*, *Rtimer* and *Clock*.

Command	Expected output
`krun`	Prints `6` and publishes `signal`.

```
Atimer(3) >> Atimer(4) >> Atimer(5) >> Rtimer(1) >> (print
    (x) < x < clock)
```

Example B.4.4. Tests time. In particular, it test a subtle point in the semantics that publishing has higher precedence than passing time.

Command	Expected output
`krun`	Publishes `3`.
`krun --ltlmc "<>Ltl isGPublished(3)"`	`true`

```
(1|1|1) >> count.inc() >> zero() | Rtimer(1) >> count.read
    ()
```

Before one time unit passes, all ready publishes should be done. That will increment `count` to `3`, and then after one time unit passes, `count.read()` is called and it publishes `3`. If it published less than that, then it means that time has passed before publishing.

B.5 Robot Environment

Apart from the examples shown in Chapter 5, the following examples were tested on the robot module. Some of these examples define and use the following expressions (which I put here instead of repeating inside every example code):

```
ChangeLane(b) := ((bot.rotateRight(b) >> bot.stepFwd(b) >>
     bot.rotateLeft(b)) | (bot.rotateLeft(b) >> bot.stepFwd
    (b) >> bot.rotateRight(b)))
```

```
SmartStep(b) :=  bot.scan(b) > isBlocked > (ifNot(
    isBlocked) >> bot.stepFwd(b) | if(isBlocked) >>
    ChangeLane(b))
SmartStep4Ever(b) :=  SmartStep(b) >> SmartStep4Ever(b)
```

Example B.5.1.

```
bot.init() >> bot.stepFwd() >> bot.stepFwd() >> bot.
    rotateLeft() >> bot.stepFwd()
```

Example B.5.2.

```
// test with search: --search --pattern "<gVars>... \"bot.
    myCow.direction\" |-> Dir </gVars>". should give three
    solutions. one where the robot rotates right, one
    rotates left, one moves forward.
bot.mapInit() >> bot.init("myCow") >> (bot.rotateRight("
    myCow") | bot.rotateLeft("myCow") | bot.stepFwd("myCow
    "))
```

Example B.5.3.

```
// Use krun without search. This will initialize two
    robots and move each one step forward.
bot.mapInit() >> (bot.init("myCow") | bot.init("yourCow"))
     > x > (bot.stepFwd(x))
```

Example B.5.4.

```
// Both bots will move one step forward. Next, put blocks
    and see if 'if' works.
bot.mapInit() >> (bot.init("myCow") | bot.init("yourCow"))
     > x > SmartStep(x)
```

Example B.5.5.

```
// this will initialize two bots but they both will be in
    the same position, the default position. They should
    both collide with the obstacle
bot.mapInit() >> bot.setObstacles(<1,5>) >> (bot.init("
    myCow") | bot.init("yourCow")) > x > bot.stepFwd(x)
```

Example B.5.6.

```
// this will move both bots. yourCow will hit but mine won
    't.
bot.mapInit() >> bot.setObstacles(<5,1>,<7,2>) >> (bot.
    init("myCow") | bot.init("yourCow",<5,0>,<0,1>)) > x >
    bot.stepFwd(x)
```

B.5.1 Testing the State Search

Example B.5.7.

```
// yourCow will change lane. test with search to see it
    end up in two places, changing lane once to the left
    and once to the right. Through the next few examples,
    we build up from a less complex version to a more
    complex one, debugging 'search' because it is taking
    long. krun is working fine though.
bot.mapInit() >> bot.setObstacles(<5,1>) >> (bot.init("
    myCow") | bot.init("yourCow",<5,0>,<0,1>)) > x >
    SmartStep(x)
```

Example B.5.8.

```
// search working as expected with sequential delta.
bot.mapInit(<6,6>) >> bot.setObstacles(<2,1>,<4,2>) >> (
    bot.init("myCow",<0,3>,<1,0>) | bot.init("yourCow
    ",<2,0>,<0,1>)) > x > (bot.rotateRight(x) | bot.
    rotateLeft(x))
```

Example B.5.9.

```
// search working. Next, add one more smart step
bot.mapInit(<6,6>) >> bot.setObstacles(<2,1>,<4,2>) >> bot
    .init("yourCow",<2,0>,<0,1>) > x > SmartStep(x)
```

Example B.5.10.

```
// search works but takes around 10 minutes
bot.mapInit(<6,6>) >> bot.setObstacles(<2,1>,<4,2>) >> bot
    .init("yourCow",<2,0>,<0,1>) > x > (SmartStep(x) >>
    SmartStep(x))
```

Example B.5.11.

```
// In this one, search took exactly 9 minutes
bot.mapInit(<6,6>) >> bot.setObstacles(<2,1>,<4,2>) >> bot.init("yourCow",<2,0>,<0,1>) >> SmartStep("yourCow") >> SmartStep("yourCow")
```

Example B.5.12.

```
// search gives an error after 22 minutes. probably out of memory because of the maude backend.
bot.mapInit(<6,6>) >> bot.setObstacles(<2,1>,<4,2>) >> bot.init("yourCow",<2,0>,<0,1>) >> SmartStep("yourCow") >> SmartStep("yourCow") >> SmartStep("yourCow")
```

Example B.5.13.

```
// simplified the previous example but still search is running forever.
bot.mapInit(<6,6>) >> bot.setObstacles(<2,1>,<4,2>) >> (bot.init("myCow",<0,3>,<1,0>) | bot.init("yourCow",<2,0>,<0,1>)) > x > SmartStep(x)
```

Example B.5.14.

```
// Using the recursive expressions that goes forever causes search to take forever as well
bot.mapInit() >> (bot.init("myCow") | bot.init("yourCow")) > x > SmartStep4Ever(x)
```

B.5.2 Testing LTL Model Checking

The following examples helped fix problems with time and synchronization. LTL model checking used to run for upto 20 minutes in some of these examples, and sometimes even go out of memory. Now it takes a matter of seconds.

Example B.5.15.

```
// test with: --ltlmc "<>Ltl gVarEqTo(\"bot.yourCow.is_bumper_hit\",false)"
// ltlmc successful in a matter of seconds, returns true.
bot.mapInit(<6,6>) >> bot.setObstacles(<2,1>,<4,2>) >> bot.init("yourCow",<2,0>,<0,1>) >> SmartStep("yourCow") >> SmartStep("yourCow") >> SmartStep("yourCow")
```

Example B.5.16.

```
// test with: --ltlmc "<>Ltl gVarEqTo(\"bot.yourCow.
    is_bumper_hit\",false)"
// ltlmc successful in a matter of seconds, returns true.
bot.mapInit() >> bot.setObstacles(<5,1>) >> (bot.init("
    myCow") | bot.init("yourCow",<5,0>,<0,1>)) > x >
    SmartStep(x) // ltlmc taking forever
```

Example B.5.17.

```
// test with: --ltlmc "[]Ltl <>Ltl gVarEqTo(\"bot.yourCow.
    is_bumper_hit\",false)"
// ltlmc also takes a matter of seconds, return true.
bot.mapInit(<6,6>) >> bot.setObstacles(<2,1>,<4,2>) >> bot
    .init("yourCow",<2,0>,<0,1>) > x > (SmartStep(x) >>
    SmartStep(x))
```

Example B.5.18.

```
// this example runs correctly, but search will run
    forever.
Bot_FwdForever() := bot.stepFwd() >> Bot_FwdForever()
bot.mapInit() >> bot.init() >> Bot_FwdForever()
```

Example B.5.19.

```
// tests nondeterminism in robot movement
Bot_2Fwd() := Bot_MoveFwd() >> Bot_MoveFwd()
Bot_FwdForever() := Bot_MoveFwd() >> Bot_FwdForever()
Bot_ChangeCourse() := (Bot_TurnLeft() | Bot_TurnRight())
    >> Bot_MoveFwd() >> (Bot_TurnLeft() | Bot_TurnRight())
Bot_ManeuverAround() := Bot_TurnLeft() >> Bot_MoveFwd() >>
     Bot_TurnRight() | Bot_TurnRight() >> Bot_MoveFwd() >>
    Bot_TurnLeft()
Bot_Protocol() := Bot_Scan() >> (if(ObstacleAhead()) >>
    Bot_ManeuverAround() | if(Not(ObstacleAhead()))) >>
    Bot_MoveFwd() >> Bot_Protocol()
//Bot_Setup() := Bot_Init() >> Bot_SetObstacles
    ([2,5],[1,4],[1,6])

Bot_Init() >> Bot_Protocol()
```

Example B.5.20.

```
// test using:
//krun --search --pattern "<gVars> M:Map </gVars>"
DummyExp(a,b) := Add(a,b)
bot.init() >> (bot.turnRight() | bot.turnLeft() | bot.
    moveFwd()) // WORKING! test with search. should give
    three solutions. one where the robot turns right, one
    turns left, one moves forward.
```

Example B.5.21.

```
RandomMove() := Bot_MoveFwd() | Bot_TurnLeft() >>
    Bot_MoveFwd() | Bot_TurnRight() >> Bot_MoveFwd()
Bot_Init() >> RandomMove() >> RandomMove()
```

Example B.5.22.

```
// this tests bot initialization and setting the
    environment
bot.init() >> bot.setObstacles(<1,5>) >> bot.moveFwd()
```

Example B.5.23.

```
FwdForever(thisBot) := bot.moveFwd(thisBot) >> FwdForever(
    thisBot)
bot.mapInit(<10,10>) >> bot.setObstacles(<5,2>) >> (bot.
    init(<0,5>) | bot.init(<5,0>)) > myCow > FwdForever(
    myCow)
```

B.6 Publishing and Number of Solutions

This set of examples were made with the goal of understanding the behavior of the `--search` command with regard to the publishing rule, which resembles the only observable transition in these examples. After all the testing and debugging, simplifying the example to pin-point any inconsistencies seen in its behavior, the conclusion drawn was that the order at which the transition rule applies for different threads is what makes a more-than-expected number of solutions.

All the examples in this subsection are tested using the command:
`krun test.orc --search --pattern "<gPublish> L:List </gPublish>"`.

Example B.6.1.

```
// imitates recursive calls. tests propagation of variable
    mappings.
signal >> if(false) | signal >> if(false) | Add(b,1) < b < 1
```

Example B.6.2.

```
//Publishes 2 and 3
if(false) | (( Add(b,1) | ( Add(b,1) < b < Add(b,1) )) < b <
     let(1) )
```

Example B.6.3.

```
//Publishes 2, 3, 4. Gives 6 solutions.
if(false) | (( Add(b,1) | ( ( Add(b,1) | ( Add(b,1) < b <
    Add(b,1) )) < b < Add(b,1) )) < b < let(1) )
```

Example B.6.4. This outputs 2, 3, 4. It's two transitions more than the previous example because of the two sequential operations. This gives 56 solutions. This means that the change from Example B.6.5 to Example B.6.6 increased the solutions and `tid` numbers. Let's minimize to pin-point the problem.

```
signal >> if(false) | signal >> (( Add(b,1) | ( ( Add(b,1) |
     ( Add(b,1) < b < Add(b,1) )) < b < Add(b,1) )) < b < Let
    (1) )
```

Example B.6.5. Minimizing from Example B.6.4. This gave 6 solutions.

```
if(false) | signal >> (( Add(b,1) | ( ( Add(b,1) | ( Add(b
    ,1) < b < Add(b,1) )) < b < Add(b,1) )) < b < 1)
```

Example B.6.6. Example 3.3 minimizing from Example B.6.4. this gave 50 solutions.

```
signal >> if(false) | (( Add(b,1) | ( ( Add(b,1) | ( Add(b
    ,1) < b < Add(b,1) )) < b < Add(b,1) )) < b < 1)
```

Example B.6.7. Minimizing from Example B.6.6 by removing top pruning. This gave 1 solution.

```
signal >> if(false) | ( Add(b,1) | ( ( Add(b,1) | ( Add(b,1)
     < b < Add(b,1) )) < b < Add(b,1) ))
```

Example B.6.8. Minimizing from Example B.6.6 by removing mid pruning. This gave 27 solutions.

```
signal >> if(false) | ( Add(b,1) | ( ( Add(b,1) | ( Add(b,1)
     < b < Add(b,1) )) ) < b < 1)
```

Example B.6.9. Minimizing from Example B.6.8 by removing bottom pruning. This gave 9 solutions.

```
signal >> if(false) | ( Add(b,1) | Add(b,1) | Add(b,1) < b <
     1)
```

Example B.6.10. Minimizing from Example B.6.9. This gives 1 solution.

```
Add(b,1) | Add(b,1) | Add(b,1) < b < 1
```

Example B.6.11. Minimizing from Example B.6.9. This gives 5 solutions.

```
signal >> if(false) | Add(b,1) | Add(b,1) < b < 1
```

Example B.6.12. Minimizing from Example B.6.11. This gives one solution. This helped in the debugging since it used to give 3 solutions. This is probably as minimal as it could get.

```
signal >> if(false) | Add(b,1) < b < 1
```

Example B.6.13. This gives two solutions.

```
signal >> if(false) | 1
```

Example B.6.14. This gives two solutions.

```
signal >> if(false) | 1
```

Example B.6.15. This gives two solutions.

```
signal >> if(false) | signal >> if(false)
```

Example B.6.16. This one is derived from Example B.6.12.

```
0 >> 3 | b < b < 1
```

B.7 Stressing State Search

Because publishing is defined as an observable transition, running `search` on two publishes in parallel will yield two solutions because of the permutations of the order of publishing. This means that the number of solutions will increase exponentially with respect to the number of publishes in parallel. The following set of examples was designed to find the limit of `search`.

Example B.7.1. This outputs 2, 3 and 4. Here we added more sequential operations. It has 12 concurrent publish transitions.

```
signal >> if(false) | signal >> (( signal >> Add(b,1) |
    signal >> ( ( signal >> Add(b,1) | signal >> ( Add(b,1) <
     b < Add(b,1) )) < b < Add(b,1) )) < b < Let(1) )
```

Example B.7.2. This has one less publish. It has 11 concurrent publishes. Here, `--search-final` gave 1295 solutions while `--search --depth 5` gave 149 solutions.

```
signal >> if(false) | signal >> (( Add(b,1) | signal >> ( (
    signal >> Add(b,1) | signal >> ( Add(b,1) < b < Add(b,1)
    )) < b < Add(b,1) )) < b < let(1) )
```

B.8 Variable Lookup

Having designed our own mechanism for variable lookup and scope sharing, this specific part needed rigorous testing and debugging. The following series of examples was made to trace and debug problems with that module, to refine the theory behind it and to optimize its mechanism. Ultimately, these examples were leading to the recursive factorial program.

B.8.1 Phase 1

We have a subset of three examples that have a pruning operation at different levels.

Example B.8.1. This outputs `10` and the topmost pruning prunes everything else. To see the value of the pruning that is below it, we change something as seen in the next example. After that, we allow the pruning below that in Example B.8.3 and see the result of the bottom-most pruning.

```
signal >> if(false) | signal >> ( Mul(a,5) < a < (( signal
    >> Add(b,1) | signal >> ( Mul(a,5) < a < ( ( signal >>
    Add(b,1) | signal >> ( Mul(a,5) < a < ( Add(b,1) < b <
    Add(b,1) ))) < b < Add(b,1) ))) < b < let(1) ) )
```

Example B.8.2. Outputs two solutions: 500 and 75

```
signal >> if(false) | signal >> ( Mul(a,5) < a < (( signal
    >> freeze | signal >> ( Mul(a,5) < a < ( ( signal >> Add(
    b,1) | signal >> ( Mul(a,5) < a < ( Add(b,1) < b < Add(b
    ,1) ))) < b < Add(b,1) ))) < b < let(1) ) )
```

Example B.8.3. Outputs 500.

```
signal >> if(false) | signal >> ( Mul(a,5) < a < (( signal
    >> freeze | signal >> ( Mul(a,5) < a < ( ( signal >>
    freeze | signal >> ( Mul(a,5) < a < ( Add(b,1) < b < Add(
    b,1) ))) < b < Add(b,1) ))) < b < let(1) ) )
```

B.8.2 Phase 2

Now we remove `freeze`.

Example B.8.4. Outputs three solutions: 10, 75, 500.

```
signal >> if(false) | ( Mul(a,5) < a < (( signal >> Add(b,1)
     | ( Mul(a,5) < a < ( ( signal >> Add(b,1) | ( Mul(a,5) <
     a < ( Add(b,1) < b < Add(b,1) ))) < b < Add(b,1) ))) < b
     < let(1) ) )
```

Example B.8.5. Outputs two solutions: 75 and 500.

```
signal >> if(false) | ( Mul(a,5) < a < (( signal >> if(false
    ) | ( Mul(a,5) < a < ( ( signal >> Add(b,1) | ( Mul(a,5)
    < a < ( Add(b,1) < b < Add(b,1) ))) < b < Add(b,1) ))) <
    b < let(1) ) )
```

Example B.8.6. Outputs one solution: 500.

```
signal >> if(false) | ( Mul(a,5) < a < (( signal >> if(false
    ) | ( Mul(a,5) < a < ( ( signal >> if(false) | ( Mul(a,5)
     < a < ( Add(b,1) < b < Add(b,1) ))) < b < Add(b,1) ))) <
     b < let(1) ) )
```

B.8.3 Phase 3

Now we remove `signal`'s.

Example B.8.7. Outputs three solutions: 10, 75 and 500.

```
if(false) | ( Mul(a,5) < a < (( Add(b,1) | ( Mul(a,5) < a <
    ( ( Add(b,1) | ( Mul(a,5) < a < ( Add(b,1) < b < Add(b,1)
     ))) < b < Add(b,1) ))) < b < 1 ) )
```

Example B.8.8. Outputs two solutions: 75 and 500.

```
if(false) | ( Mul(a,5) < a < (( if(false) | ( Mul(a,5) < a <
     ( ( Add(b,1) | ( Mul(a,5) < a < ( Add(b,1) < b < Add(b
    ,1) ))) < b < Add(b,1) ))) < b < 1 ) )
```

Example B.8.9. Outputs one solution: 500.

```
if(false) | ( Mul(a,5) < a < (( if(false) | ( Mul(a,5) < a <
    ( ( if(false) | ( Mul(a,5) < a < ( Add(b,1) < b < Add(b
   ,1) ))) < b < Add(b,1) ))) < b < 1 ) )
```

B.8.4 Phase 4

These also have the same pattern. Let's simplify more.

Example B.8.10. Outputs three solutions: 10, 15 and 20.

```
if(false) | ( Mul(a,5) < a < (( Add(b,1) | ( ( Add(b,1) | (
   Add(b,1) < b < Add(b,1) )) < b < Add(b,1) )) < b < 1 ) )
```

Example B.8.11. Outputs two solutions: 15 and 20.

```
if(false) | ( Mul(a,5) < a < (( if(false) | ( ( Add(b,1) | (
    Add(b,1) < b < Add(b,1) )) < b < Add(b,1) )) < b < 1 ) )
```

Example B.8.12. Outputs one solution: 20.

```
if(false) | ( Mul(a,5) < a < (( if(false) | ( ( if(false) |
   ( Add(b,1) < b < Add(b,1) )) < b < Add(b,1) )) < b < 1 )
   )
```

B.8.5 Final Simplification

Example B.8.13. Outputs two solutions: 15 and 20.

```
if(false) | (( if(false) | ( ( Add(b,1) | Add(b,1) ) < b <
   Add(b,1) )) < b < 1 )
```