Int J Software Informatics, Volume 9, Issue 2 (2015), pp. 205–231
International Journal of Software and Informatics, ISSN 1673-7288
©2015 by ISCAS. All rights reserved.

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Knorc Calculus and Its Formal Semantics

— To Honor my Friend Prof. Krieg-Brueckner's 66th Birthday

Ruqian Lu

(Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China)

(Key Lab of Management, Decision and Information Systems, Chinese Academy of Sciences, Beijing 100190, China)

(Sino-Australian Joint Lab of Quantum Computing and Quantum Information Processing, Beijing, China)

Abstract This paper introduces the orchestration calculus Knorc, which is a conservative extension of the Orc calculus designed by J. Misra et. al. Orc is a simple but powerful calculus for wide area computing, whose simplicity makes it a solid kernel for orchestration programming. But on the other hand Orc leaves everything else to the programmer, which often makes the programming task complicated. The design idea of Knorc was to provide Orc with a delicately selected set of facilities to greatly increase the expressive power of the calculus and at the same time keep the calculus concise. The distinguished features of Knorc include, but not limited to: combination of process algebra and logic programming, site considered as remote Boolean procedure, Horn-like logic programming and inference, diversity of different parallelism mechanisms, network of abstract knowledge sources, open world assumption as opposed to closed world assumption where OWA means existing sites need not be known to the programmer, symmetric process-to-process communication, batch processing facilities of knowledge and data, as well as broad band message transmission.

Besides introducing the general structures of the language Knorc, we present also a formal structural operational semantics. This is one of the major foci of this paper.

Key words: orchestration; process algebra; logic programming; formal semantics; Orc language; Knorc language

Lu RQ. Knorc calculus and its formal semantics – to honor my friend Prof. Krieg-Brueckner's 66th birthday. *Int J Software Informatics*, **Vol.9, No.2 (2015): 205–231. http://www.ijsi.org/1673-7288/9/i215.htm**

1 Background and A Quick Overview of Orc

There were two major driving forces which made the orchestration technique popular, namely business information processing and Web service engineering. Various concepts of orchestration under different names have been proposed, such as computation orchestration, service orchestration, business orchestration, cloud orchestration, etc. Accordingly, different programming languages have been

This work is sponsored by CNSF key project 61232015.
Corresponding author: Ruqian Lu, Email: rqlu@math.ac.cn
Received 2015-01-06; Revised 2015-04-01; Accepted 2015-04-30.

designed for programming orchestration, which are in particular suitable for programming concurrent and distributed applications. While most of the early orchestration languages are based on XML representation[25], it was only after entering the 21st century that people have started to design orchestration languages in higher level form. The Orc calculus designed by Misra et al. is one of the most successful languages in this area[16], which is now well-known for its simple but powerful character. By disregarding technical details of programming and focusing on Web resource orchestration, Orc attracts interest of many researchers.

The development of Orc has experienced three major stages. That Orc first published in 2002 by Young-ri Choi et al[5] was basically a programming model for task orchestration, where the basic program units are tasks. But many fundamental principles like wide area computation, remote task call, restricted communication, tree concurrency have been decided in that version. It was J. Misra who has led Orc into its second stage: Orc as a programming calculus (2004) taking a more elegant form of process algebra[15]. In this version the tasks were renamed as sites to emphasize their original motivation as abstraction of Web services. The calculus is simple but still very powerful with a group of salient features regarding real time asynchronous concurrency making it different from traditional process algebras such as CSP, CCS, ACS and π calculus. However, Orc as a calculus lacks conventional computational features, even elementary actions such as addition and multiplication should be done by site calls. The third stage (2006) was opened by D. Kitchen et al, who have developed Orc as a functional language and made it to a real programming language[12].

As pointed out by its authors, the power of Orc is in its computation model as a calculus for wide area computing or wide area orchestration. An Orc program is conceived as one of orchestrating Web services where each Web service is abstracted as a site. To call a Web service is considered as calling the service of a site, written as $M(\bar{p})$, where M is the site name and $\bar{p}$ is a list of parameters. A site call is effective only when all its parameters are instantiated. Otherwise the call will be pending. Site call is the fundamental program unit of Orc. Expressions are site calls connected by combinators. To make the language as simple as possible, Orc has only three combinators: parallel composition $f \mid g$; sequenced computation $f >x> g$, where x is published by f and sent to initiate g; backwards computation $f <x< g \mid h$, also written as f **where** $x :\in g \mid h$, where f, g and h are called in parallel, x is published by either g or h and sent to instantiate pending parameters of f, which terminates $g \mid h$ and prunes it away from the whole expression. In addition, recursive definitions are allowed to increase the programming power. It is natural that in Orc the communication is thought of as asynchronous message passing.

The basic grammar of Orc is simple and concise as shown in table 1[16].

Table 1 Basic grammar of orc

$f, g, h \in$ *Expression*	$::=$	$M(\bar{p}) \parallel E(\bar{p}) \parallel f >x> g \parallel f \mid g \parallel f$ **where** $x :\in g$
$p \in$ *Actual*	$::=$	$x \parallel v$
Definition	$::=$	$E(\bar{x}) \triangleq f$

So the entities of an Orc program are at three levels: the data level: variables, values, parameters; the site call level and the expression level (sites structured with combinators), where the site level is the most fundamental one. There are no arithmetic neither Boolean expressions. As for semantics, the most well established formal semantics of Orc is its asynchronous semantics which is operational and structured. In the following we list some most interesting transition rules and explain them shortly[11]

1.1 Semantics for site call

$$\frac{k \text{ fresh}}{M(v) \xrightarrow{M_k(v)} ?k} \qquad ?k \xrightarrow{k?v} let(v) \qquad let(v) \xrightarrow{!v} 0 \tag{1}$$

Each site call $M(v)$ sets up a fresh handle k, a specific call instance $M_k(v)$ and transits to $?k$. Once the response arrives, it receives the value v with the event $k?v$ and transits to $let(v)$. The later v publishes the value v with event $!v$ and transits to the zero site 0 (unable to do anything).

1.2 Semantics for forwards value transmission

$$\frac{f \xrightarrow{!v} f'}{f > x > g \xrightarrow{\tau} f' > x > g \mid g\,[v/x]} \tag{2}$$

Expression f publishes a value v to become f' and sends it via channel $>x>$ to instantiate expression g's variable x. The result is a parallel composition of instantiated g with $f' >x> g$.

1.3 Semantics for backwards value transmission

$$\frac{g \xrightarrow{!v} g'}{f \textbf{ where } x :\in g \xrightarrow{\tau} f\,[v/x]} \tag{3}$$

Expression g publishes value v and sends it back via the **where** channel to instantiate expression f's variable x. As a consequence, expression g together with the **where** structure disappear.

2 Works Related to Orc

The first echo to the publication of the Orc language might be the wave of studying its semantics, which started almost immediately. The operational semantics mentioned in the above section was provided by David Kitchen et al. and is in fact a trace semantics for Orc. As usual, they extended the Orc execution traces to include failures, refusals, ready states etc. In addition, the traces include also communication and substitution events. It was proved that this Orc's trace semantics is equivalent to its observational semantics. Furthermore, they showed that the equality relation of trace sets is actually a congruence on programs. A more general congruence is also given which is more delicate than strong bisimulation.

Another operational semantics proposed for Orc is Tony Hoare's dynamic labelled tree semantics[9]. The name tree semantics comes from the fact that the execution traces of an Orc program are sets of threads dynamically created and pruned along a labelled tree. The equivalence of two Orc programs is defined by the equality of their corresponding labelled trees. In his paper, Tony Hoare proved a series of algebraic laws of this semantics which can be used to design Orc compilers and to analyze Orc programs' properties as well as proving their correctness.

Rewriting logic semantics was studied by Marti-Oriet, Jose Meseguer and Rosu since the nineties of the last century as an executable rewriting logic theory[14], which is a wide spectrum approach that can be used to describe formal semantics of a system covering its design, implementation and analysis phases. M. AlTurki and J. Meseguer applied Maude, which is a system specification language implementing rewriting logic semantics, to define a rewrite logic semantics for Orc[1] resulting in both a structured operational semantics and a reduction semantics. The authors called this technique DIST-ORC[3].

This paper is perhaps not the right place to list all approaches proposed to study the formal semantics of Orc. Rather, we will use a few lines to mention the further developments of Orc in its generalization or specialization.

While bearing its key concern in distributed orchestration of Web services, the original Orc does not provide much facility on data models and data processing. Kristi Morton, B.A. proposed an XML based data model for Orc in his master of art thesis and developed an extension of Orc called Orc-X which combines Orchestration with XQuery[17]. The latter is a query language for XML documents. Orc-X has been applied to exercise distributed resource management by enriching the data management facilities of Orc with XQuery. On the other hand, transaction processing often raises complicated problems. There are different approaches for treating this issue. Coons proposed a multi-strategy approach for Orc which can apply different processing alternatives to meet different situations[7]. Kitchen studied atomicity and coatomicity in his PhD thesis[10]. Both concepts are based on causality (a partial order) and used for defining sound transactional properties. The former says that the events in an atom are causally either all or none before an event outside the atom and reversely, the events outside an atom are causally either all or none before an event inside the atom. The resulting language is called Ora (Orc with Atomicity).

For programming service oriented architecture, there are two basic paradigms: the orchestration approach and the coordination approach. While the former orchestrates distributed applications based on a central control, the latter coordinates a set of distributed agents with appropriate protocols and is called choreography. It was De Nicola et. al. who have developed a language Korc[19] to combine the functions of Orc and Klaim[18]. The latter is a tuple based coordination language which extends the single parameter language Orc to a polyadic one. (At this place it may be adequate to give few words about the name use. In the early stage of Knorc development, we were not aware of Klaim and its further development Korc. So we named our calculus as Korc. It was only later that we knew the Klaim-Korc-story and renamed our Korc to Knorc. It may be meaningful to point out that their 'K' means Klaim, while that of ours means Knowledge).

Besides programming paradigm, the programming style of Orc also affects its application. Nicolas, C. et al. proposed to combine multi-tier programming facilities with mash-up functions with Orc. Web programming is generally supported by a server programming language, a graphical user interface and a client oriented language. Hop (a Lisp dialect) is a multi-tier language unifying all these three layers to a single tool[20]. Mash-ups are essentially data oriented activities in form of Web services constructed using various kinds of Web sources. Orc is good in Mash-ups programming. Nicolas, C. et al. introduced Orc functions in HOP to unite the two.

As a language for wide area computing, program security is one of the major concerns for implementation. Thywissen, Quark and Yew studied the concept of secure information flow and security type checking[21,24,26]. An information flow is said to be secure if it does not leak information to unauthorized third parties[21]. They introduced a characterization of secure information flow in Orc and built the Orc compiler stOrc[24] extended with security type checks to assure information flow security (don't care the in-site security). Regarding the static and/or dynamic type checking, they focused on the static integrity of Orc programs and built an extension to Orc called cOrcS[26], meaning: continuation of Orc security.

3 Motivation and New Ingredients of Knorc

Usually, every principle of programming has its positive but also negative aspects. This is also true for Orc. While concentrating at wide area service orchestrating, it gives up many technical constructs. Programmers have to take all these in charge. On the one hand this strategy often increases the burden of writing applications, while on the other hand it provides less instruction for implementation and leaves big space free for it. This was one of the major concerns on which our motivation of designing Knorc was based. Partially due to our interest in process algebra and its application, our approach followed the line of Orc as a calculus. The following example 1 is to illustrate why we are interested in extending the calculus Orc to Knorc, in particular to illustrate the usefulness of introducing logic programming facilities in orchestration languages. In particular, we are interested in combining process algebra with logic programming. We believe that this will increase the power of Orc-like languages.

To explain the application background of Orc as a wide area orchestration language, Misra used an example of office work for inviting a speaker[15]. Usually, the secretary in charge of arranging this visit has to do lots of things, including sending invitation letter, negotiating on the date of visit, subscribing to hotel room and booking flight tickets, registering for a lecture room, etc. To complete all these, a program in Orc for orchestrating relevant function modules on the internet is helpful. However, many practical decisions have to be made and many potential conflicts have to be resolved, such as:

- Date of visit: should fit both guest's and host's time schedule;
- Flight ticket reservation: different airlines, departure and arrival times, number of stops, etc.;
- Hotel reservation: number of stars, price class, distance to the host university;

Facing so many details, it is not reasonable to write all of them in one Orc program at least due to two major reasons. From the software engineering aspect, the stepwise refinement of program in a top down way is appreciated. Programmers are encouraged to first write a concise but rough program and then try to refine it. From the software reuse aspect, it is appropriate to separate the knowledge content from the coding part of the program. Once the program is written, it is then easy to modify it for arranging the visit of next speakers. Based on this consideration, we think that logic programming facilities are good candidate for representing the knowledge part.

In Knorc, the knowledge part is programed with Horn-like logic rules. The wanted result is obtained with resolution-like proof procedures. Each time a new task (plan of orchestration) is to be programmed, the amount of necessary revision of the reused program can be kept at the minimal level. Let us consider an example written in Knorc, where *Invitation* is a site deciding on the visiting date, airline number and accommodating hotel:

Example 1. Knorc program for arranging a visit.
Knorc expression:

$$\begin{aligned} Invitation(arrival, flight, hotel) > & [arrival, flight, hotel] \\ & > Letter(arrival, flight, hotel) \end{aligned} \tag{4}$$

Knorc Knowledge base:

$$\begin{aligned}
Invitation(x,y,z) &:- Visit\text{-}date(x), Flight(x,f), Hotel(x,h) \\
Visit\text{-}date(t) &:- Fit(t,host), Fit(t,visitor) \\
Flight(t,f) &:- Fquery(t,ftable), In(f,F), Eq(f\text{-}time(f), \\
&\qquad \min(f\text{-}time(ftable))) \\
Hotel(t,h) &:- Hquery(t,htable), In(h,htable), Eq(distance(h), \\
&\qquad \min(distance(htable))) \\
Fit(t,host) &:- In(t,[21,22,23]) \\
Fit(t,visitor) &:- In(t,[19,20,21])
\end{aligned}$$

In this program, *Invitation* is a site for making arrangement while *Letter* is another site for sending invitation letters. In the sense of Orc the expression (4) cannot be called because the parameters are variables. However they can be instantiated by reasoning with rules in the knowledge base where t is the date of visit to be fixed, *fit* is a predicate for checking the appropriateness of this date regarding both the *host* and the *visitor*, *fquery* is a site for acquiring all information *ftable* on flights available on that day, *f-time*is the time of flying needed, *hquery* is a site for acquiring all information *htable* on hotels available on that day, *distance* is the distance from hotel to university. This example shows that once the Knorc program is written, it can be used for any visitor at any time, except the last two data (the *Fit* rules) which have to be modified to meet each concrete visit.

We see that in the rule bodies there are also site calls such as *f-query* and *h-query* besides predicates *Visit-date*, *Flight* and *Hotel* etc. In fact we even allow rule bodies to contain expressions. Furthermore sites such as *Invitation* appear also as rule heads and play the role of predicates. These are generalizations of Horn logic and explain why we call them Horn-like.

Other new facilities Knorc introduces to Orc include the following:

3.1 Site as Boolean procedure

Knorc considers each site as a Boolean procedure like those in high level programming languages. Beyond the possibility of publishing a value, each site call in Knorc, if performed, always returns a Boolean value. It will be reserved, used or discarded depending on different contexts.

3.2 Site instantiation with logic rules

A site call can only be performed if all its parameters are instantiated. In Knorc it is possible to instantiate the variable parameters via rule inference. The idea is to prove the site with Horn-like logic rules. The site call can then be performed if the proved copy is a fully instantiated one.

3.3 Logic programming with predicates and site calls

In any Knorc's sequential rule, the body components (called terms) can be either predicates or sites, even expressions. The rule head is proved if all terms of its body are proved. A predicate (as a term) can only be proved by rule inference. An expression (as a term) is proved if its call terminates and returns the Boolean True. Instead of Prolog-like backtracking, a Knorc rule searches a solution in a fully non-deterministic way.

3.4 Multiple forms of parallelism

Beyond sequential rules, Knorc also allows parallel rules. The components of parallel rules are expressions but not predicates. There are four kinds of parallelism in Knorc: the or-parallelism (among components of an or-parallel rule) as well as the and-parallelism (among components of an and-parallel rule), where the number of components is known and not too large; the or-set parallelism (or-parallelism looped for a set of parameters) and the and-set parallelism (and-parallelism looped for a set of parameters) where the number of terms is not known or too large such that it would be too cumbersome to list them.

3.5 Chained network of abstract knowledge sources

While Orc does not care any particular data types Knorc assumes the existence of an abstract form of data, the network of abstract knowledge sources (AKS or simply KS). Each KS mimics a Web site, or a Web page, or an information piece on the Web, etc. Each KS consists of its content and a set of links pointing to other KSs. Search is the keyword for the KS network. While Orc assumes a closed world in the way that each site it calls either exists or not, Knorc assumes an open world by allowing the system searching a site of which the existence is unknown.

3.6 Symmetric and asymmetric communication

Orc only allows tree-like asymmetric communication between parent and son processes. Knorc reserves this tree-like communication and introduces symmetric process-to-process communication in addition, which is necessary for cooperating parallel searches. Knorc's communication primitives include both peer-to-peer communication and broadcasting. Each time when communicating the sender sets up a fresh mailbox with keys (passwords) and message to be exchanged. Each receiver owning a key may pick up the message.

3.7 Value tuples and broadband communication

Knorc extends the control parallelism (parallel rules) to data parallelism. Similar as the authors of Korc did[19], we introduce tuple values in Knorc, not only as data representations of parameters, but also as value groups exchanged through channels and mailboxes.

In next sections we will present more details on the design of Knorc.

4 Syntax and Informal Semantics of Knorc

The syntax of Knorc is a conservative extension of Orc's syntax. In the syntax of Orc people classify the computational units in three categories: site calls, combinatory expressions and (usually recursively) defined expressions. The syntax of Knorc consists of two parts: the basic syntax and the rule syntax (including syntax for parallelism and communication).

Table 2 Basic syntax

$$
\begin{aligned}
KP \in KnorcProgram ::= \quad & \bar{f} \in \textit{expression list};\\
& rbe \in \textit{sequential rulebase or empty};\\
& prbe \in \textit{parallel rulebase or empty};\\
& \overline{de} \in \textit{definition list or empty}
\end{aligned}
$$

$$
\begin{aligned}
f, g, h \in expression ::= \ & M(\bar{p})\\
& \| \ E(\bar{p})\\
& \| \ [M(\bar{p})]\\
& \| \ \mathbf{0}\\
& \| \ f >(\bar{x}; b)> g\\
& \| \ f \mid g\\
& \| \ f \ \textbf{where} \ (\bar{x}; b) :\in g\\
& \| \ f \ \textbf{while} \ (\bar{x}; b) :\in g\\
& \| \ f \ \textbf{where} \ y = s \ \textbf{while} \ (\bar{x}; b) :\in g\\
& \| \ f \ \textbf{while} \ y = s \ \textbf{while} \ (\bar{x}; b) :\in g\\
\bar{f} \in \textit{expression-list} ::= \ & f \ \| \ f, \bar{f}\\
\bar{p} \in \textit{parameter-list} ::= \ & p \ \| \ p, \bar{p}\\
d \in definition ::= \ & D(\bar{x}) \underline{\Delta} f\\
\bar{de} \in \textit{definition-list} ::= \ & d \ \| \ d, \bar{de} \ \| \ empty\\
x \in \ & \textit{variable name}\\
v \in \ & \textit{value name}\\
b \in \ & \textit{Boolean variable name}
\end{aligned}
$$

$$
\begin{aligned}
s &\in \textit{knowledge source name}\\
M &\in \textit{site name}\\
E &\in \textit{expression name}\\
p &\in \textit{parameter name} \bigcup \textit{data}
\end{aligned}
$$

As a matter of fact, most of the Knorc's basic syntax rules are similar with the corresponding Orc syntax rules, except that instead of limiting on a single parameter x we have lists of parameters $\bar{x}$. Besides, the basic syntax of Knorc makes a little but important extension to Orc's syntax. In particular, the restricted site call $[M(\bar{p})]$ means it is prevented from using rule inference to instantiate its parameters. In other words, restricted site calls have only the same power as if they were Orc's site calls. Besides, the last three expression formats need a few explanations. The **while** loop keeps repeating calling $f(\bar{x})$ unless the site g cannot generate enough $\bar{x}$ values. The **where-while** and **while-while** loops are used to do ergodic searches on the knowledge source network. The first structure follows one link each time and searches only along a single line. The second structure does search in a breath first way and follows all links having the same start point. Its search traces form a tree. For details see the formal semantics of the knowledge source part.

Table 3 (Sequential) rule syntax

$$
\begin{aligned}
rbe \in \textit{sequential rulebase or empty} &::= rb \parallel \textit{empty}\\
rb = \bar{r} \in \textit{rulebase} &::= r \parallel r, \bar{r}\\
r \in \textit{rule} &::= \textit{head}(r) :- \textit{body}(r)\\
\textit{head}(r) \in \textit{rule head} &::= rn(\overline{pa})\\
\textit{body}(r) \in \textit{rule body} &::= \bar{t}\\
\bar{t} \in \textit{term sequence} &::= t \parallel t, \bar{t} \parallel \textit{empty}\\
t \in \textit{term} &::= f \parallel pr\\
pr \in \textit{predicate} &::= pn(\overline{pa})\\
\overline{pa} \in \textit{argument-list} &::= pa \parallel pa, \overline{pa}\\
pa \in \textit{argument} &::= v \parallel x \parallel fn(\overline{pa})\\
& pn \in \textit{predicate name}\\
& fn \in \textit{function name}\\
& rn \in \textit{rule name}
\end{aligned}
$$

The second part of syntax, the rule syntax, is completely new. It introduces two kinds of rules: the sequential ones and the parallel rules.

A sequential rule consists of a rule head and a rule body. Each rule body contains a finite sequence of expressions and/or predicates which are all called terms. A rule with an empty body is called a datum. In Knorc programming the rules are used to instantiate site calls and expressions in situations when they are not fully instantiated and remain pending. Rules are also useful in finding the best fit instantiations for site calls and expressions (when there are several possibilities) in different circumstances. In Knorc's terminology, instantiating a site call using rules is called a proof. To prove a site call $M(\bar{x})$, the proof procedure consists of the following steps:

1. Find a rule r, such that $M(\bar{x})$ has a mgu (most general unifier) with the rule head which should have the form $M(\bar{x})$. Assume the mgu is φ;
2. If r's body is empty, then $M(\bar{x})\varphi$ is proved if it does not contain variables, which is the wanted instantiation;
3. Otherwise $M(\bar{x})\psi \circ \varphi$ is proved if it does not contain variables and all terms in the body of $r\psi \circ \varphi$ are proved with returned Boolean value True, where ψ is an appropriate substitution;
4. If the term is a predicate in the above process, use this procedure recursively to prove it;
5. If the term is a site call then call this site if it is instantiated, otherwise call this procedure recursively to instantiate it. This site call is proved if it finally gets fully instantiated and the call successfully terminated with returned Boolean value True;
6. If the term is an expression then try to prove all its site calls accordingly with returned Boolean value True.

Note that the form of a sequential rule is quite similar to that of the logic programming language Prolog. However, our proof procedure is not necessary the same as in Prolog which is based on depth first backwards reasoning according to the classical resolution principle. In Knorc the inference procedure of rule reasoning is strongly non-deterministic. It is well known that a Prolog program does not guarantee to provide a solution even if a solution exists. However in the same circumstance Knorc provides a solution anyway and in principle no solution will be excluded. We will see the non-deterministic solution finding mechanism of Knorc rules in the latter section on formal semantics.

Let's compare sequential composition in Orc, sequential composition in Knorc and sequential rule with Table 4:

Table 4 Comparison of sequential program structure of Knorc with Orc

	sequential composition in Orc	sequential composition in Knorc	sequential rule in Knorc
Form	$f_1 >x_1> f_2$	$f_1 >\bar{x}_1> f_2$	$H(\bar{p}) :- f_1(\bar{p}_1), f_2(\bar{p}_2)$
Value passing $f_1 \to f_2$	yes	yes	no
f_1 initially fully instantiated	yes	not necessary	not necessary
f_i instantiation propagates to f_j	no	no	yes
f_i instantiation propagates to H	no	no	yes
Result deterministic	yes	not always	not always

We omit the comparison of Knorc sequential rule with Orc's reverse sequential composition $f_1 <x_1< f_2$ which is simple. Now we consider the following

Example 2.

$$expression: \; N(x,y,z) >(w;b)> M(w)$$

$$\begin{aligned} \mathit{rule}: &\ N(u,v,w) :- Q(u,v), R(v,w), W(w) \\ \mathit{data}: &\ Q(1,2), Q(3,4), R(2,8), R(4,6), W(6), W(8) \end{aligned}$$

We see that although the parameters of $N(x,y,z)$ are variables they can still be instantiated using the rule and data by inference to produce $N(3,4,6)$ and $N(1,2,8)$, which will be called successfully and possibly publish values.

Now the problem arises: what will happen if the site can be instantiated by the expression itself:

$$\mathit{expression} : N(x,y,z) >(w;b)> M(w) \textbf{ where } x :\in h \textbf{ where } y :\in k \textbf{ where } z :\in e \tag{5}$$

In this case the result can be non-deterministic. Namely $N(x,y,z)$ can be fully instantiated and publish values by using logical reasoning as said above, but can also be instantiated to other instances of $N(x,y,z)$ by the three **where** operations and publish other values for w. Some implementations may chose the strategy of letting the instantiation be depending on what happens first: instantiation by rule reasoning or by h,k,e value publication. But any particular strategy does not belong to the formal semantics and is left to the decision of implementation. This is a new form of non-determinism introduced in addition to the existing ones of Orc. Furthermore, this is also a new form of concurrency, since the rule inference and the generation of x,y,z values by the three **where** branches are running in parallel. We adopt the strategy of winner-takes-all such that the variable parameters are either instantiated by rule inference or by the expression value generators (in above example h,k,e) themselves. Knorc refuses a mixed instantiation.

Note that the call to $N(x,y,z)$ will remain pending if rule reasoning does not produce a full instantiated copy of $N(x,y,z)$.

Example 3. Mary won't be notified if the (Boolean) parameter 'paid' is not set to 'yes'.

$$\begin{aligned} \mathit{expression}: &\ \mathit{Registration}(\mathit{name}, \mathit{fee}, \mathit{paid}) >(\mathit{name}; b)> \mathit{Notice}(\mathit{name}) \\ \mathit{data}: &\ \mathit{Registration}(\mathit{John}, 3000, \mathit{paid}); \mathit{Registration}(\mathit{Mary}, 3000, \mathit{paid}), \ldots \end{aligned}$$

On the other hand, a fully instantiated site like that in $N(2,3,4) >w> M(w)$ will always be called immediately without rule inference, even if there are such rules and even the call $N(2,3,4)$ would succeed.

Table 5 (Parallel) rule syntax

$\mathit{prbe} \in$ *parallel rulebase or empty*	$::=$	$\mathit{prb} \parallel \mathit{empty}$
$\mathit{prb} \in$ *prule base*	$::=$	$\mathit{par} \parallel \mathit{par}, \overline{\mathit{par}}$
$\mathit{par} \in$ *prule*	$::=$	*or-rule* $\parallel$ *and-rule* $\parallel$ *orset-rule* $\parallel$ *andset-rule*
$\mathit{oru} \in$ *or-rule*	$::=$	$\mathit{head}(\mathit{par}) :- \mathit{or}(\overline{[M] : f})$
$\mathit{aru} \in$ *and-rule*	$::=$	$\mathit{head}(\mathit{par}) :- \mathit{and}(\overline{[M] : f})$
$\mathit{ors} \in$ *orset-rule*	$::=$	$\mathit{head}(\mathit{par}) :-$ *or-set*$(\overline{x_i \in g_i})(\overline{[M] : f})$
$\mathit{ans} \in$ *andset-rule*	$::=$	$\mathit{head}(\mathit{par}) :-$ *and-set*$(\overline{x_i \in g_i})(\overline{[M] : f})$
$\mathit{head}(\mathit{par}) \in$ *prule head*	$::=$	$\mathit{prn}(\overline{\mathit{pa}})$
$\mathit{prn} \in$ *parallel rule name*	$::=$	*characterstring*

In this syntax *par* is a parallel rule and *prb* is a parallel rule base. There are four types of rule bodies, implementing four types of parallelism: the or-parallelism, the and-parallelism, the or-set-parallelism and the and-set-parallelism. Different from sequential rules, a parallel rule body consists only of a group $\overline{[M] : f}$ of guarded expressions with $[M]$ as guards (which are limited site calls) but no predicates. Orset-rules and andset-rules have a set indication $\overline{x_i \in g_i}$ between the rule head and rule body which includes a group of components (i.e. expressions) who are allowed to run in parallel according to the following laws:

- The calling to a parallel rule consists of four steps: the caller (a predicate or a site call to be proved) tries to match the rule head (find a most general unifier with it, mgu for short); if succeeded, propagate the mgu (substitution) to the whole rule body; invoke a parallel call of all components of the rule body; for each component, test the guard first, if succeeded, then try to prove the expression itself behind the guard;
- Checking the guard will not invoke any extra rule inference since it is a limited site call. No matter the proof of the guard is successful or not, it does not leave any side effects. Any potential effects will be removed after the proof procedure is terminated;
- The call to an or-rule is completed if any one of the components' calls is completed and returns the Boolean value True;
- The call to an and-rule is completed if all components' calls in it are completed and return the Boolean value True;
- The call to an orset-rule is completed if any one of the components' calls is completed with respect to all elements of the set indication and returns the Boolean value True;
- The call to an andset-rule is completed if all components' calls are completed with respect to all elements of the set indication and return the Boolean value True;
- Note that in all cases above, the Boolean value returned by the rule calling depends on the rule head itself. Thus, a successful rule call may also return the Boolean value false;
- Roughly speaking, the call to an orset-rule or andset-rule can be considered as a call to a matrix of components. Since calls to the same site with different parameter instantiations can be run in parallel, this is matrix parallelism instead of just linear parallelism.

Example 4. A class of students will fail to pass the competition if any of them fails to pass the exam for any of the subject. They will win the competition if all students of the class get the score 'very good' in all subjects. Class A is the set of all students. All guards are omitted.

$Lose(A)\ :-\ or\text{-}set(x \in classA)(Fail(math, x), Fail(physics, x), Fail(chemistry, x));$

$$\begin{aligned} Win(A)\ &:-\ \textit{and-set}(x \in classA)(Super(math, x), Super(physics, x),\\ &\quad Super(chemistry, x)); \end{aligned}$$

Example 5. Search for all Web sites relating to some key topic, *key* for short. Assume that to each link p is assigned a meaning presenting a hint on the topic x of the target Web site (pointed to by p). Link p will be selected for transition only if p's meaning is close enough to *key*. The following program looks for all Web sites whose topic is close to key.

$$\begin{aligned} Search(con, s, key)\ &:-\ \textit{and-set}(p \in link(s))([Close(p, key)] : Collect(s, p, key));\\ Collect(s, p, key)\ &:-\ Include(con, linked(s, p)), Search(con, linked(s, p), key); \end{aligned}$$

This is a nested cross-calling of sequential and parallel rules.

Example 6. Sometimes it is meaningful to apply the set-parallel structure even to the cases where the rule body has only one component. The Goldbach conjecture says that any even integer larger than 2 can be represented as the sum of two primes. A prime is a natural number which is remainderless divisible only by 1 and itself. The first rule below checks the Goldbach conjecture for the first one million even integers. The second one investigates the first one million odd integers to find all primes in this region.

$$\begin{aligned} Goldbach\ &:-\ \textit{or-set}(x \in \left[1, 2 \times 10^6\right])([Even(x)] : \textit{Check-conjecture}(x));\\ Euler\ &:-\ \textit{and-set}(x \in \left[1, 2 \times 10^6\right])([Odd(x)] : \textit{Check-prime}(x));\\ \textit{Check-conjecture}(z)\ &:-\ \textit{and-set}(x \in [1 : \left\lfloor \frac{z}{2} \right\rfloor])(\textit{Check-prime}(x), \textit{Check-prime}(z - x)) \end{aligned}$$

This program allows 2×10^6 computers to run in parallel to complete the computation at once. This shows that our set-parallelism device is able to program massive parallelism in an elegant way. The set involved in the above example is additive: each element can be produced with simple mathematical operation iteratively. For more sophisticated sets the advantage of set parallelism would be even more remarkable.

4.1 Fundamental sites

Knorc retains all Orc's fundamental sites and introduces two new ones for linked search and two for process-to-process communication in addition:

Table 6 Fundamental functions and sites

Fundamental Functions:	
$link(s)$:	generates the set of all links in s, where s is a KS;
$linked(s, p)$:	generate the KS pointed to by link p of KS s;
Fundamental Sites:	
$send(v, \bar{k})$:	establish a fresh mailbox $m\#(v, \bar{k})$ with a value v and a set of keys $\bar{k}$;
$receive(x, k)$:	find a mailbox $m\#(v, \bar{k})$ whose key set $\bar{k}$ contains the key k, picks the value v contained in m and use v to instantiate receiver's variable x to become $recieve(v, k)$. At the same time the key k is removed from the mailbox.

Parallelism, non-determinism and communication are all important and interesting issues. Since the major new feature of Knorc is the introduction of logic programming, it would be interesting to compare Knorc with some well-known parallel logic programming languages in these aspects. This is shown in Table 7. Let's explain some terminology. By "in rule or parallelism" and "in rule and parallelism" we mean the or-parallelism and and-parallelism among the components of the same rule body of Knorc. By "inter rule or parallelism" and "inter rule and parallelism" we mean those usually called or-parallelism and and-parallelism in the literature[23]. "Don't care non-determinism" means that whenever a heuristically chosen parallel component fails to be proved no new choice will be made. "Don't know non-determinism" means "backtrack whenever possible" after every failed choice. We propose the concept "don't miss non-determinism" by which we mean trying every possibility to find a solution whenever one exists.

Table 7 A comparison of rule parallelism, non-determinacy and communication

	Prolog	PARLOG	Concurrent Prolog	Knorc
In rule and-parallelism	No	Yes	Yes	Yes + Set
In rule or-parallelism	No	No	No	Yes + Set
Inter-rule and-parallelism	No	No	No	Yes + Set
Inter-rule or-parallelism	No	Committed choice	Candidate clause	Yes + Set
Non-determinism	don't know non-determinism	don't care non-determinism	don't care non-determinism	don't miss non-determinism
Guard	No	Yes	Yes	Yes
Communication	No	(shared memory) Stream Communication	Message passing (three primitives)	asymmetric + symmetric communication

More precisely, Knorc provides tree-like asymmetric communication and symmetric process to process refreshable mailbox communication. Note that Table 7 illustrates only rule parallelism which is a specific property of Knorc. This explains why we did not include Orc or other Orc-like languages in Table 7 because rule inference is not a part of Orc programming style. As for parallelism within expressions there is no explicit difference between Orc and Knorc except that Knorc does have an implicit concurrency which does not exist in Orc. Namely before the firing of any expression which is not fully instantiated like (R 1), the exploratory instantiation of its uninstantiated parameters goes in two ways concurrently: by rule inference and by the **where** structure. This parallelism affects the result of the program, but is not to see explicitly.

5 Formal Semantics of Knorc

5.1 Basic site call semantics

In the first part of Knorc's operational semantics we list only those 'normal' transition rules without reference to the rule part. We basically adopt the notation and style of Orc's semantics except that some extensions are made to cover the enriched syntax. The extensions of basic syntax are along three lines. First we introduce 'broadband communication' with the data list ymbol $\bar{x}$ in most of the syntax rules, which is in accord with the 'tuple based data' of Korc[19]. Second we include conditional site call $if(M(\bar{v}))$ to enrich the programming style and third we extend the **where** structure to **while** structure to easy loop programming. Furthermore since we allow normal site calls to make rule inference to instantiate their variable parameters (shown below later) we should have a way to limit this function in case of need. We use bracketed site calls to express the limitation. Last but not least we include the Boolean value returned by site calls explicitly in the transition rule representations. For the sake of clarity we include the basic site calls such as if and let in the semantics description.

Table 8 Basic transition rules of knorc

$$\frac{k \text{ fresh}}{M(\bar{v}) \xrightarrow{(M_k(\bar{v});b)} ?k} \quad [M(x,\bar{a})] \xrightarrow{\perp} c \in \{M_k(\bar{v}), k?\bar{v}, !\bar{v}, \tau\} \tag{6}$$

$$?k \xrightarrow{(k?\bar{v};b)} let(\bar{v}) \frac{f \xrightarrow{(c;b)} f'}{f \textbf{ where } (\bar{x};b) :\in g \xrightarrow{(c;b)} f' \textbf{ where } (\bar{x};b) :\in g} \tag{7}$$

$$let(\bar{v}) \xrightarrow{(!\bar{v};b)} \mathbf{0} \frac{g \xrightarrow{(c;b)} g' \;\; c \neq !v}{f \textbf{ where } (\bar{x};b) :\in g \xrightarrow{(c;b)} f \textbf{ where } (\bar{x};b) :\in g'} \tag{8}$$

$$\frac{f \xrightarrow{(c;b)} f'}{f \mid g \xrightarrow{(c;b)} f' \mid g} \frac{g \xrightarrow{(!v;b)} g'}{f \textbf{ where } (\bar{x};b) :\in g \xrightarrow{(\tau;b)} f\,[\bar{v}/\bar{x}]} \tag{9}$$

$$\frac{g \xrightarrow{(c;b)} g'}{f \mid g \xrightarrow{(c;b)} f \mid g'} \frac{f \xrightarrow{(c;b)} f', f \neq if, c \neq !\bar{v}}{f >(\bar{x};b)> g \xrightarrow{(c;b)} f' >(\bar{x};b)> g} \tag{10}$$

$$\frac{f \xrightarrow{(!\bar{v};b)} f', f \neq if}{f >(\bar{x};b)> g \xrightarrow{(\tau;b)} f' >(\bar{x};b)> g \mid g\,[\bar{v}/\bar{x}]} \frac{M(\bar{v}) \xrightarrow{(c;\text{False})} N(\bar{u})}{if(M(\bar{v})) >(\bar{x};b)> g \xrightarrow{(\tau;\text{False})} \mathbf{0}} \tag{11}$$

$$\frac{M(\bar{v}) \xrightarrow{(c;\text{True})} N(\bar{u}), c \neq !u}{if(M(\bar{v})) >(\bar{x};b)> g \xrightarrow{(\tau;\text{True})} if(N(\bar{u})) >(\bar{x};b)> g} \tag{12}$$

$$\frac{M(\bar{v}) \xrightarrow{(!\bar{u};\mathrm{True})} N(\bar{w})}{if(M(\bar{v})) >(\bar{x};b)> g \xrightarrow{(\tau;\mathrm{True})} if(N(\bar{w})) >(\bar{x};b)> g \mid g\,[\bar{u}/\bar{x}]} \tag{13}$$

$$\frac{f \xrightarrow{(c;b)} f'}{f \textbf{ while } (\bar{x};b) :\in g \xrightarrow{(c;b)} f' \textbf{ while } (\bar{x};b) :\in g} \tag{14}$$

$$\frac{g \xrightarrow{(c;b)} g', c \neq !v}{f \textbf{ while } (\bar{x};b) :\in g \xrightarrow{(c;b)} f \textbf{ while } (\bar{x};b) :\in g'} \tag{15}$$

$$\frac{g \xrightarrow{(!\bar{v},\mathrm{True})} g'}{f \textbf{ while } (\bar{x};b) :\in g \xrightarrow{\tau} f\,[\bar{v}/\bar{x}] \mid f \textbf{ while } (\bar{x};b) :\in g'} \quad \frac{(E(\bar{x})\underline{\Delta} f) \in D}{E(\bar{p}) \xrightarrow{\tau} f\,[\bar{p}/\bar{x}]} \tag{16}$$

The transition rules (6)–(11) are basically (above mentioned) modified versions of similar rules of Orc. (6 left) means $M(\bar{v})$ setting up a new call with a fresh handle $?k$; (7 left) means the fresh call receiving a result $\bar{v}$; (8 left) means the result published and the process terminated. (6 middle) shows the pending bracketed site call $[M(x,\bar{a})]$ with variable parameter x to mean that it is not allowed to use rule inference to instantiate its parameters as the unbracketed one $M(x,\bar{a})$ may do. (6 right) defines c for all following rules. Rules (7 right)–(10 right) and (9 left)–(10 left) are similar with Orc. Rules (11)–(13) introduce conditional site calls where the Boolean value returned by $M(\bar{v})$ determines the value of the condition. Rules (14)–(16 left) introduce the **while** structure in addition to the **where** structure.

It may be adequate to say a few words about the Boolean value b produced by a site call. That in Knorc each site call returns a Boolean Value (besides published values) is a necessity of the fact that the site calls play also the role of predicates or Boolean procedures in a logic rule where each term must have a Boolean value True or False.

On the other hand, since each site call returns a Boolean value this value can be used in the conditional control of Knorc programs. As a matter of fact Misra himself[16] has already come to the idea of using Boolean value in program control in some specific program contexts. What we have done was just to generalize this idea to a universal principle of site call mechanism. Note that this Boolean value can only be used to support conditional control either explicitly serving as a parameter in if-site call (12-8) or implicitly serving to decide the success or failure of a rule call.

The following part of semantics is specifically defined for Knorc's rule part and is therefore completely new. We invite readers' attention to the following two points. The first one is that an and-parallel rule finishes only when all components finish which is a sharp difference to Orc's parallel composition of sites where the components act freely and independently from each other. Our second idea is to adopt the mechanism of continuation for and-parallel rules. Note that this continuation produces no real effect if all components of the rule are predicates. This means it does not change the semantics of Prolog.

5.2 Horn logic rule part semantics

The key new ingredient of Knorc vs. Orc is the inclusion of logic programming facilities used to instantiate the variable parameters of site calls in case they are pending. The logic used is like Horn logic?-a subset of predicate logic. A Horn clause is a disjunctive clause with at most one positive atom. The Horn clauses and their widely used proof procedure–the resolution procedure (proposed by A. Robinson) are the semantic basis of the logic programming language Prolog. The rule form of a Horn clause is written as:

$$P \ :- \ P_1, \ldots, P_n \ or \ P \tag{17}$$

where P is the rule head and constitute the rule body. P is also called a datum when the body part is empty. In order to increase efficiency Prolog implementations are usually based on a recursive, backwards and depth first proof procedure with backtracking.

In Knorc semantics of logic programming the rules are not strict in Horn form since some literals may be site calls or even expression calls. We call it Horn-like logic and rely upon the same syntactic form like Prolog mentioned above with some modifications. We adopt its backwards recursive semantics but without backtracking which is replaced with non-determinism. The latter is semantically much simpler and can obtain all solutions, while the backtracking approach may lose some solutions because the set of solutions it obtains depends on the ordering of rules. Another advantage of relying upon non-determinism is to provide us with chances of benefitting parallelism.

Remember in Orc calculus there are four base events: the publication event $!v$, the internal event τ, the site call event $M_k(v)$ and the response event $k?v$. Misra used the unified symbol c to represent them. Knorc inherits these notations from Orc with the enrichment of a Boolean value $b \in \{\text{True}, \text{Flase}\}$ which is an independent information accompanying these events. As a matter of fact we have noticed that Misra has been already aware of the usefulness of introducing a Boolean value as part of the result of calling a site[15]. Now these four events are represented as $(!v; b)$, $(\tau; b)$, $(M_k(v); b)$ and $(k?v; b)$. In case the returned Boolean value b does not affect the result, the symbol b can be omitted, or even the simple representations $!v$, τ, $M_k(v)$ and $k?v$ can be used instead. Moreover, the value transmission notations $>(x; b)>$ and **where** $(x; b) :\in g$ can be simply written as $>x>$ and **where** $x :\in g$. In order to include Horn logic proof procedure in our semantics, we only need to add one more base event—the substitution event $@\varphi$, where φ is a substitution $[\bar{v}/\bar{x}]$. We will see below that this base event is very essential in describing our Horn logic semantics. It is also powerful enough such that we don't need to add any other base events for characterizing rule inference. We use symbol $c \in BE = \{!v, \tau, M_k(v), k?v\}$ and $any \in AN = BE \bigcup \{@\varphi\}$ to denote the base events. Besides, we introduce composite events $c(\varphi)$, $any(\varphi)$, $c^*(\varphi)$, $any^*(\varphi)$, $c^+(\varphi)$, $any^+(\varphi)$ to denote first performing substitution φ and then the event(s) c, any, c^*, any^*, c^+, and any^+.

Example 7. $add(x, 1, y) \xrightarrow{@[1/x]} add(1, 1, y)$
The transition sequence

$$add(x, 1, y) >y> let(y) \xrightarrow{@[1/x]} add(1, 1, y) >y> let(y) \xrightarrow{!2} let(2)$$

may be rewritten as

$$add(x,1,y) >y> let(y) \xrightarrow{!2([1/x])} let(2)$$

Table 9 shows the transition rules for Knorc's rule part reasoning:

Table 9 Transition rules of Knorcs Horn logic rule part

$$\frac{verify\text{-}site(M) \xrightarrow{any^*(\varphi)} M\varphi \xrightarrow{c'} M', M \text{ not in } [M]}{M \xrightarrow{c' \circ any^*(\varphi)} M'} \tag{18}$$

$$\frac{mgu(A, head(r)) \xrightarrow{@\varphi} A\varphi, prove\text{-}seq(body(r)\varphi) \xrightarrow{any^*(\psi)} let(body(r)\psi \circ \varphi)}{verify\text{-}atom(A) \xrightarrow{any^*(\psi \circ \varphi)} A\psi \circ \varphi} \tag{19}$$

$$\frac{mgu(A, head(r)) \xrightarrow{@\varphi} A\varphi, body(r) = empty}{verify\text{-}atom(A) \xrightarrow{@\varphi} A\varphi} \tag{20}$$

$$\frac{verify\text{-}atom(M) \xrightarrow{(any^*(\psi), True)} M\psi, Fv(M\psi) = \varnothing}{verify\text{-}site(M) \xrightarrow{any^*(\varphi)} M\varphi} \tag{21}$$

$$\frac{verify\text{-}atom(P) \xrightarrow{(any^*(\varphi); True)} let(P\varphi)}{prove\text{-}pred(P) \xrightarrow{any^*(\varphi)} let(P\varphi)} \tag{22}$$

$$\frac{verify\text{-}site(M) \xrightarrow{any^*(\varphi)} M\varphi, Fv(M\psi) = \varnothing}{verify\text{-}exp(E\{M\}) \xrightarrow{any^*(\varphi)} E(\{M\})} \tag{23}$$

$$\frac{verify\text{-}exp(E) \xrightarrow{any^*(\varphi)} E\varphi \xrightarrow{(any'^+, True)} E' \xrightarrow{\perp}}{prove\text{-}exp(E) \xrightarrow{any'^+ \circ any^*(\varphi)} E'} \tag{24}$$

$$\frac{prove\text{-}pred(hd(x)) \xrightarrow{any^*(\varphi_0)} let(hd(x)\varphi_0), |x|>1, prove\text{-}seq(tl(x)) \xrightarrow{any'^*(\varphi_1)} let(tl(x)\varphi_1)}{prove\text{-}seq(x) \xrightarrow{any'^*(\varphi_1) \bigcup any^*(\varphi_0)} let(x\varphi_1 \circ \varphi_0)} \tag{25}$$

$$\frac{prove\text{-}exp(hd(x)) \xrightarrow{any^+(\varphi_0)} hd(x)', |x| > 1, prove\text{-}seq(tl(x)) \xrightarrow{any'^*(\varphi_1)} let(tl(x)\varphi_1)}{prove\text{-}seq(x) \xrightarrow{any'^*(\varphi_1) \bigcup any^+(\varphi_0)} let(x\varphi_1 \circ \varphi_0)} \tag{26}$$

$$\frac{prove\text{-}pred(hd(x)) \xrightarrow{any^*(\varphi)} let(hd(x)\varphi), hd(x) = x}{prove\text{-}seq(x) \xrightarrow{any^*(\varphi)} let(x\varphi)} \tag{27}$$

$$\frac{prove\text{-}exp(hd(x)) \xrightarrow{any^+(\varphi_0)} hd(x)', hd(x) = x}{prove\text{-}seq(x) \xrightarrow{any^+(\varphi_0)} hd(x)'} \tag{28}$$

Different from Table 8 the rules in Table 9 make use of a group of specific site calls which are not part of Knorc syntax. They can be considered as system reserved internal functions which are not available to the programmer and are only for describing semantics.

The transition rule (18) is the bridge between the traditional site call-oriented semantics of Orc and the new added semantics of logical rule inference in Knorc. It says that a site M may be called even if it is not fully instantiated. The way to do that is to try to use rule inference to find a legal instantiation. We call this process 'verify'. More concretely the call to (the not fully instantiated) M may be performed if it is verified (by instantiating the parameters with substitution φ and side effects any^* generated by rule inference with procedure 'verify-site'). In other words, the site M may perform the (composite) event $c' \circ any^*(\varphi)$ to become M'. There is however a restriction. A site call in form of $[M]$ is not allowed to make use of rule inference to get instantiation, since there are situations where rule inference is not applicable to site calls even if the latter is pending.

To make the presentation simple we use the unified notation 'atom' to represent rule head, predicate and site call. Transition rule (19) says that if atom A matches the head of some Horn rule r with most general unifier φ and $body(r)\varphi$ can be proved by performing the composite event $any^*(\psi)$ then the substituted atom $A\psi \circ \varphi$ is verified. Transition rule (20) says that if the rule body is empty (rule r degenerates to data) then matching the head of r is enough for atom A to be verified. These two rules together show how an atom can be verified. Transition rule (21) says that site M is verified if it is verified as an atom and the result contains no free variables. Thus (18)–(21) show roughly how an uninstantiated site can be instantiated and called using rule inference.

From (19) we see that each rule inference consists of two steps: first matching the rule head with a most general unifier and then proving the rule body with further substitutions and possibly some side effects. A rule body consists of an ordered set of terms each of which is a predicate or an expression. (22) says that a predicate (as a body term) is proved if it is verified (under some substitution φ) with Horn logic inference. (23) shows that the verify-exp procedure verifies an expression only one site (in the expression) each time where the notation $E\{M\}$ means expression E contains (among others) a site call M. (24) shows that an expression (as a body term) is proved if it performs a finite but non empty set of events (including those generated by rule inference) and finally returns the Boolean value True. Note the difference between *hd* and *head* that $hd(s)$ means the first item of a sequence s, while $head(r)$ means the head part of a Horn logic rule r. The transition symbol $E \xrightarrow{\perp}$ means E cannot perform an event anymore.

Given that all terms (predicates and expressions) of a rule body are proved, transition rules (25)–(28) show how the body of a Horn rule is proved. They are designed as a recursive procedure by separating the Horn rule body x in its head part $hd(x)$ and tail part $tl(x)$. The transition rules (25) and (26) apply to the cases where $hd(x)$ is a predicate or an expression, respectively. The transition rules (27) and (28) apply when $tl(x)$ is empty. Note that in (25) and (26) the composite events generated by $prove\text{-}seq(x)$ are the unions of the events generated by the *hd*-part and the *tl*-part respectively because the hd-part and tl-part run concurrently. The events $any'^{*}(\varphi_1)\bigcup any^{+}(\varphi_0)$ produced by these two parts are interleaved in any possible order, except that substitutions φ_0 and φ_1 must happen before any^+ and any'^* respectively. This is different from (24) where the event combinations are sequential ones. All events in any^* should happen before those in any'^+.

5.3 Parallel rule part semantics

Unlike the (sequential) logic rules discussed above the parallel rules in Knorc do not contain any predicates. The rule body of each parallel rule consists of a group of component expressions running in parallel where each component is preceded by a guard. That means each component is guarded where each guard is a restricted site call because we don't want the guard test invokes extra rule inference. Testing a guard is running this site call where no matter the test succeeds or fails, all side effects produced during the test will be cancelled after testing.

Table 10 Transition rules of knorc's parallel rule part

$$\frac{mgu(A, head(r)) \xrightarrow{@\varphi} A\varphi, or(body(r)\varphi) \xrightarrow{any^+} f' \xrightarrow{\perp}}{A \xrightarrow{any^+(\varphi)} A(subs(any^+(\varphi)))} \tag{29}$$

$$\frac{[M] \xrightarrow{(c;\text{True})} [N]\,, f \xrightarrow{(any^+, True)} f' \xrightarrow{\perp}}{or([M]:f, \overline{[M_i]:f_i}) \xrightarrow{any^+} f' \xrightarrow{\perp}} \tag{30}$$

$$\frac{mgu(A, head(r)) \xrightarrow{@\varphi} A\varphi, and(body(r)\varphi) \xrightarrow{any^+} and(\overline{f'}) \xrightarrow{\perp}}{A \xrightarrow{any^+(\varphi)} A(subs(any^+(\varphi)))} \tag{31}$$

$$\frac{\forall i, [M_i] \xrightarrow{(c_i;\text{True})} [N_i]\,, \prod_{i\in\{1,2,\ldots,|body(r)|\}} f_i \xrightarrow{(any_i^+, True)} f_i' \xrightarrow{\perp}}{and(\overline{[M_i]:f_i}) \xrightarrow{\bigcup_i any_i^+} and(\overline{f_i'}) \xrightarrow{\perp}} \tag{32}$$

$$\frac{mgu(A, head(r)) \xrightarrow{@\varphi} A\varphi, or\text{-}set(\overline{x_i \in g_i})(body(r)\varphi) \xrightarrow{any^+} f' \xrightarrow{\perp}}{A \xrightarrow{any^+(\varphi)} A(subs(any^+(\varphi)))} \tag{33}$$

$$\frac{\forall \overline{u_i \in g_i}, [M(u_i)] \xrightarrow{(c_i;\text{True})} [N_i]\,, \forall \overline{u_i \in g_i}, f(u_i) \xrightarrow{(any_i^+, True)} f_i' \xrightarrow{\perp}}{or\text{-}set(\overline{x_i \in g_i})([M]:f, \overline{[M_i]:g_i}) \xrightarrow{\bigcup_i any_i^+} f' \xrightarrow{\perp}} \tag{34}$$

$$\frac{\begin{array}{c} mgu(A, head(r)) \xrightarrow{@\varphi} A\varphi, \\ and\text{-}set(\overline{x_i \in g_i})(body(r)\varphi) \xrightarrow{any^+} and(\overline{f'_{i,j}}) \xrightarrow{\perp} \end{array}}{A \xrightarrow{any^+\varphi} A(subs(any^+(\varphi)))} \tag{35}$$

$$\frac{\forall i, \forall v_j \in g_j, [M_i(v_j)] \xrightarrow{(c_{i,j};\text{True})} [N_{i,j}]\,, \forall f_i \in \overline{f_i}, f_i(v_j) \xrightarrow{(any_{i,j}^+, True)} f'_{i,j} \xrightarrow{\perp}}{and\text{-}set(\overline{x_j \in g_j})(\overline{[M_i]:f_i}) \xrightarrow{\bigcup_{i,j} any_{i,j}^+} and(\overline{f'_{i,j}}) \xrightarrow{\perp}} \tag{36}$$

In the context of parallel rule transitions, we abuse the notations 'or' and 'and' to think them as if they were site names such that $or(\ldots)$ as well as $and(\ldots)$ were site calls. $Body(r)$ denotes their rule body and is a sequence of expression calls. The transition rule (29) illustrates the proof procedure of atom A (a site call M or a predicate P) using or-parallel rule r. If A matches the rule head of r the result is an mgu (most general unifier) φ. A is then proved in form of $A(subs(any^+(\varphi)))$ where any^+ is the set of events happening during the proof procedure of $body(r)\varphi$ (rule body substituted with φ) where $subs(any^+(\varphi))$ is the composition of all substitutions contained in $any^+(\varphi)$. That is to say: though the proof procedure of A may raise a set of side effects in any^+ only the substitutions $subs(any^+(\varphi))$ appear in the proved form of A. All other events included in any^+ (are produced in the inference process as side effects and) will not affect the result. Note also that the guard before each component is 'just' a site (no rule inference is permitted). Any effect produced by the guard will be discarded.

Rule (30) says that if the guard $[M]$ of any component f of the *or*-body is successfully checked and f itself may perform a composite event any^+ to become f' and terminates then the *or*-body as a whole successfully terminates. Note that although we write f as the first component in (30) the generality is not violated because each component of the *or*-body can be repositioned to the first place without affecting the semantics of or-parallel rules. Note also an important assumption: since we allow all components 'race' in parallel only the events generated by the winner (first terminating component) are effective. All other events possibly generated by other components will be cancelled and have no influence on the result.

Transition rules (31) and (32) show a similar procedure as PR (1–2) with regard to and-parallel rules. (31) is almost the same as (29). (32) requests all components f_i of the *and*-body produce events any_i^+ to become f_i' and terminate but also allows these f_i be able to run in any interleaved order (the $\prod$ operator) to assure the successful termination of the whole rule body.In this process the events of all individual components are put together in form of $\bigcup_i any_i^+$ in any interleaved order to form the synthesized event set produced by the rule body.

Transition rules (33)–(36) repeat the same procedure for set-oriented parallelism instead of simple or/and parallelisms where the only difference is that in this case we have two indices: one index i for the expressions and another index j for set elements. Thus the expressions to be evaluated form a matrix. It means roughly: for each tuple of $(x_1, \ldots, x_n) \in g_1 \times \ldots \times g_n$, run the expression list $\overline{f_j(x_1, \ldots, x_n)}$ in or-parallelism or and-parallelism. Also the event sets generated form a matrix. There aren't too many extra complicacies.

Regarding the implementation aspect, Knorc assumes that all branches of the *or*-body and *and*-body are running in parallel. Since some of them may modify the environment by performing site calls and therefore produce side effects affecting the behavior of other branches the results become somewhat non-deterministic. However, considering that the implementation environment often pose limitations to the scale of parallelism Knorc leaves the freedom to the implementation by allowing a variety of different degrees of parallelism from total sequentiality to total parallelity depending on real system conditions. Note that unwanted results may appear such as 'undefined' results shown by the following:

Example 8.

$$Q \ :- \ \mathit{and}\ ([\text{True}] : \mathit{Minus\text{-}one}\ (\mathit{account}), [\text{True}] : \mathit{Reciprocal\text{-}account}\ (\mathit{account}))$$

A call to this rule will raise exception if account's initial value is 1 and the *Minus-one* expression will be performed first. To calculate the reciprocal value of zero is meaningless. But in this case the premise of (32) is not satisfied therefore the whole conditional transition rule is not violated.

5.4 Abstract knowledge source part semantics

We introduce the concept of (abstract) knowledge source (KS) which is a structured value consisting of a value in the sense of Orc and a set of links pointing to other KSs. A link is also called a pointer if we want to emphasize its directedness. In this way the set of all knowledge sources form a network. Examples of such networks are traffic networks, bio-informatics networks, Web site networks, etc. We introduce two basic functions: $\mathit{link}(s)$ denotes all links starting from the knowledge source s, $\mathit{linking}(s,p)$ denotes that knowledge source pointed to by the link p from knowledge source s.

Table 11 Search rules on knowledge source network

$$\frac{p \in \mathit{link}(s)}{M(s) \xrightarrow{+p} M(\mathit{linked}(s,p))} \tag{37}$$

$$\frac{p \in \mathit{link}(s), g \xrightarrow{(!s;\text{True})} g'}{f \ \textbf{where}\ x{=}s \ \textbf{while}\ \mathit{linked}(x,p){:}{\in} g \xrightarrow{\tau} f[s/x] | f \ \textbf{where}\ x{=}\mathit{linked}(s,p) \ \textbf{while}\ \mathit{linked}(x,p){:}{\in} g'} \tag{38}$$

$$\frac{p \in \mathit{link}(s), g \xrightarrow{(!s;\text{True})} g'}{f \ \textbf{while}\ x{=}s \ \textbf{while}\ \mathit{linked}(x,p){:}{\in} g \xrightarrow{\tau} f[s/x] \mid f \ \textbf{while}\ x{=}\mathit{linked}(s,p) \ \textbf{while}\ \mathit{linked}(x,p){:}{\in} g'} \tag{39}$$

To be more precise, sometimes we add the condition '$s \in S$' to the premise to mean that s should be a knowledge source (KS). But this is not necessary because only KS may have links. (37) introduces a new base event $+p$ to mean that performing $+p$ makes $M(s)$ transformed to $M(\mathit{linked}(s,p))$. This function is particularly useful for chained search. In the last section we have presented axioms for proving terms in a rule. With the new base event $+p$ we are now ready to extend these axioms for the cases of linked search by redefining the general base event *any* as: $\mathit{any} \in AN = BE \bigcup \{@\varphi\} \bigcup \{+p\}$. Rule (38) introduces a new structure for chained search, or more precisely, for KS processing based on chained search. It benefits both keywords **where** and **while**. The former specifies the current KS being processed, while the latter specifies the continuing search on the KS network. More literally, it first performs the expression $f(s)$ where s is the starting KS. Provided that s has a link p pointing to some other KS the next call to f is $f(\mathit{linked}(s,p))$ which then leads to further search. However, this is a single line of search where each time only one of the next KS will be fetched. Rule (39) extends the **where–while** structure to **while–while** structure where the search trace is a tree. Each time when starting from a particular KS all links from this KS to other KS will be searched.

5.5 Communication part semantics

As was pointed out by the authors of Orc the communication events in Orc are limited to message exchange between parent–son processes. Sometimes this is not powerful enough. Take the Web search as an example. Whenever we send out some network spiders for searching and collecting useful information from Web sites their traces represent different processes. To increase the efficiency of search, real time communication between the network spiders is desired. But the spider processes are sibling processes rather than parent–son processes. In order to solve this problem we define in Knorc two communication primitives: the sender and the receiver. A communication event transmits information from sender to receiver(s). Each time when a sender wants to send information it establishes a fresh mailbox with a finite set of keys as a wrapper for the information. Only processes having corresponding keys are endowed with right to open the mailbox and access the information. Mailbox with a particular key $\tilde{k}$ contains broadcast information which can be accessed by everyone. Communication events appear only in parallel rules.

Table 12 Transition rules of knorc's communication part

$$\frac{m\ fresh\ mailbox}{send(v,\bar{k}) \xrightarrow{out_m(v,\bar{k})} send(x,\bar{k}) \mid m\#(v,\bar{k})} \tag{40}$$

$$\frac{k_0 \in \bar{k}}{receive(x,k_0) \mid m\#(v,\bar{k}) \xrightarrow{in_m(v,k_0)} receive(v,k_0) \mid m\#(v,\bar{k}-\{k_0\})} \tag{41}$$

$$receive(x,k_0) \mid m\#(v,\tilde{k}) \xrightarrow{in_m(v,\tilde{k})} receive(v,k_0) \mid m\#(v,\tilde{k}) \tag{42}$$

Table 12 lists the transition rules of Knorc's communication part. Transition rule (40) specifies the mechanism of information sending where the sender $send(v,\bar{k})$ establishes a fresh mailbox $m\#(v,\bar{k})$ containing a value v with the keys $\bar{k}$. In this way it generates an output event out_m. In this process sender's information v is consumed (becoming a variable x without value). Transition rules (41) and (42) specify the mechanism of information receiving where the parallel composition of mailbox and receiver illustrates the typical scene of a distributed system which makes it possible to transmit a value from mailbox to receiver. Each time the information is accessed by receive (v,k_0) the corresponding key k_0 in the mailbox is consumed. The mailbox becomes garbage after all keys in it are consumed except the broadcast key $\tilde{k}$ which remains kept in the mailbox. It remains there to enable any further accesses.

Comparing these communication primitives with those proposed by Orc demonstrates another novelty of Knorc's communication facilities. In Knorc there are no constant communication channels. Rather Knorc proposes to establish a fresh mailbox for each instance of information sending. With a fresh mailbox it is easy to specify the receivers and keys exactly as a finite group or as the whole public (broadcast). It supports dynamic change of mailboxes. Also the secrete keys of communication are dynamic such that one can have different keys for different instances of communication. Remember the difference of communication channels between π calculus and CCS, where the CCS channels are fixed while the π calculus channels are dynamically changeable.

With these two new primitives for communication we extend the set of base events once again to: $any \in AN = BE \bigcup \{@\varphi\} \bigcup \{+p\} \bigcup \{in_m(v,k), out_m(v,\bar{k})\}$

Example 9. Two parallel processes $found(s, key)$ and $found(t, key)$ are initiated to find the knowledge source (KS) containing 'key' in a collaborative way where s and t are two different starting points. Each of the two processes is represented as an or-parallelism consisting of a 'find' component for finding the wanted KS and (if found) sending a signal to the other process, and a 'receive' component for receiving signal from the other process. There are two rules for proving the 'find' component, one for sending the signal and terminates the process when the KS with 'key' is found, and one goes over to the next connected KS and continues the search.

$$\begin{aligned} Found(s, key) &\mid Found(t, key); \\ Found(x, key) &:- \ or(Find(x, key), receive(x, k)); \\ Find(x, key) &:- \ Contain(x, key), send(signal, k); \\ Find(x, key) &:- \ \textit{Is-a}(p, link(x)), Find(linked(x, p), key). \end{aligned}$$

By using our set-parallelism, the second 'find' rule can be modified to a more ambitious one:

$$Find(x, key) \ :- \ \textit{and-set}(p \in link(x))(Find(linked(x, p), key))$$

6 Conclusion

As we have illustrated above Knorc is a language or calculus combining orchestration computation with logic programming. It combines also shared memory concurrency and message passing concurrency. In this final section we will summarize shortly what we have done and why we did it in this way.

First we want to mention the Orc calculus once again which stimulated our idea of designing Knorc. We were in particular interested in Orc's properties of site calls as abstract form of Web service requests; its asymmetric sequential program composition, value passing and partial concurrent execution; its tree form multi-thread computation and asymmetric tree-like parent-son process communication; its synchronizing and terminating facilities, and other interesting programming devices. We appreciate its simple and concise form and its process algebra paradigm.

In the Knorc calculus we introduced following facilities to enhance the representation power of Orc:

– All advanced properties of the Orc calculus;

– Combining process algebra with logic programming;

– Site call as remote Boolean procedure call;

– Horn-like logic rule inference and problem solving;

– Site call instantiation by Horn rule inference;

– Four types of parallel rules with concurrency, synchrony and non-determinacy;

– Network of abstract knowledge sources and search mechanism;

– Tupled values and broadband value transmission;

– Symmetric process-to-process communication;

– Peer to peer communication and broadcasting.

Among all these new facilities that of logic programming is in the central place. There are at least four advantages that we get when introducing logic programming in Knorc:

– Advantage 1: Enhance the structuredness of programming. In the early stage of computer programming code and data were mixed together in the same program which often made the program error prone and difficult to check its correctness. It was only in the late sixty and early seventy of last century that structural programming was proposed. The separation of data from code was one of its principles. Logic programming is the means by which Knorc could separate knowledge (many details) from expressions. That is to separate Knorc's declarative part from its imperative part. A straightforward consequence is that an otherwise complicated program could become more concise and readable.

– Advantage 2: Enhance the knowledge and program reuse. Separating knowledge from expressions has the additional benefit that many parts of a routine Knorc program could be easily reused as shown by example 1.

– Advantage 3: Logic programming (based on Herbrand Semantics and resolution principles) and process algebra (based on universal algebra semantics) are two very different programming paradigms. To combine them in one calculus is a serious challenge to language designers, which stimulated researches of various interests. In particular it was not easy to define its formal semantics.

– Advantage 4: Orc has a strict law that only fully instantiated expressions could be fired. This is not very comfortable for programmers in some circumstances. It is for instance the case if the instantiation has to be calculated with a complicated procedure. Another situation is when the instantiation is non-deterministic and has multiple possibilities.

It may be interesting to make a (somewhat superficial) analytics on the various Orc related works introduced in section 2 and their relations to Knorc. Though they seem to be completely orthogonal to each other in the sense that no extension of Orc functionally covers everything of another extension, all these extensions can be aligned in a hierarchy of levels. At the lowest is the infrastructure level which provides fundamental support to Orc to have a reliable implementation. Examples are Dist-Orc (using socket semantics to refine communication semantics) and cOrcS (data and program integrity as security). Next is the platform level which provides extra functionalities and utilities for supporting Orc programming in some specific aspects. Examples are Ora (transactional Orc, guaranteeing full-fledged implementation of transactional activities) and Orc-X (introducing XML data structure with query function). Also the knowledge source data structure of our Knorc belongs to this category. Higher than the platform level is the programming methodology level which provides new programming structures of orchestration. The introduction of logic programming in Knorc should be a typical example in this direction which brings a new programming style of orchestration. The last and highest level in this hierarchy is the system level which provides the possibility of combining Orc with some non-orchestration systems. An example of that is the Korc language mentioned earlier in this paper[19] which combines orchestration and choreography. Note that the division of this hierarchy is not absolute. The knowledge source data structure in Knorc is more abstract vs. XML in Orc-X which is concrete and can be considered as one of the implementation forms of the former. On the other hand we are also looking forward to a more perfect integration of orchestration with logic programming such that Knorc will become a system level extension of Orc.

In next future we will refine further the Knorc calculus in particular its operational semantics. We will also study other forms of Knorc's formal semantics. Last but not least we are starting to implement Knorc on the basis of Orc.

Acknowledgement

I thank Yu Huang for reading through my manuscript, finding notational inconsistency problems and helping correcting typing errors and transforming it to LaTeX format. I thank anonymous reviewers for pointing out various structural and representational problems so that I could make essential improvement of this paper.

References

[1] AlTurki M, Meseguer J. Real-time rewriting semantics of Orc. In: Leuschel M, Podelski A. eds. Proc. of 9th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. 2007. 131–142.

[2] AlTurki M, Meseguer J. Reduction semantics and formal analysis of Orc programs. Electronic Notes in Theoretical Computer Science, 2008, 200(3): 25–41.

[3] AlTurki M, Meseguer J. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In: Olvecaky PC, ed. First International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS'10). 2010. EPTCS 36. 26–45.

[4] Campos DM, Barbosa LS. Implementation of an orchestration language as a Haskell domain specific language. Electronic Notes in Theoretical Computer Science 255. 2009. 245–64.

[5] Choi Y, Garg A, Rai S, Misra J, Vin H. Orchestrating computations on the world-wide Web. In: Monien RFB, ed. Parallel Processing: 8th International Euro-Par Conference, Vol 1, LNCS 2400. Springer. 2002. 1–20.

[6] Cook W, Misra J. A structured orchestration language. http://www.cs.utexas.edu/users/wcook/Projects/orc, 2009.

[7] Coons KE. Transactional Orc. https://orc.csres.utexas.edu/papers/coonske_sp08.pdf[Student Report], University of Texas at Austin, 2008.

[8] Gregory S. Parallel programming in PARLOG. Addison-Wesley, 1987.

[9] Hoare T, Menzel G, Misra J. A tree semantics of an orchestration language. In: Broy M, ed. Proc. of NATO ASI series. 2004. 331–350.

[10] Kitchen DW. Orchestration and atomicity[PhD. Thesis], University of Texas at Austin, 2013.

[11] Kichen DW, Cook W, Misra J. A language for task orchestration and its semantic properties. CONCUR, LNCS 4137. 2006. 477–491.

[12] Kitchen DW, Quark K, Cook W, Misra J. The Orc programming language. LNCS 5522. 2009. 1–25.

[13] Lu R. Korchestration and the Korc calculus. 7th International conference on KSEM, LNAI 8793, Keynote Abstract, 2014.

[14] Marti-Oriet N, Meseger J. Rewriting logic, roadmap and bibliography. Theoretical Computer Science, 2002, 285(2): 121–154.

[15] Misra J. Computation orchestration: a basis for wide-area computing, In: Broy M, ed. Proc. of NATO ASI series. 2004. 285–330.

[16] Misra J, Cook WR. Computation orchestration: a basis for wide-area computing. JSSM, 2006, 6(1): 83–110.

[17] Morton K. Orc-X: Combining Orchestrations and XQuery[Thesis], University of Texas at Austin, 2008.

[18] De Nicola R, Ferrari G, Pugliese R. KLAIM: A kernel language for agents interaction and mobility. TSE, 1998, 24(5): 315–330.

[19] De Nicola R, Margheri A, Tiezzi F. Orchestrating tuple-based languages. LNCS 7173. 2012. 160–178.

[20] Nicolas C, Serrano, Loquori L. Hop and Orc: Blending orchestration and multi-tier programming languages. TR. INRIA. 2010. http://www-sop.inria.fr/members/Cyprien.Nicolas/uns/m2/Nicolas-Internship-report.pdf

[21] Quark A. Secure information flow in Orc (draft)[Student Report], University of Texas at Austin, 2009. https://orc.csres.utexas.edu/papers/quark_sif_09_draft.pdf.

[22] Shapiro E, ed. Concurrent Prolog, MIT Press, 1986.

[23] Talia D. Survey and Comparison of PARLO and Concurrent Prolog, SIGPLAN Notice, 1990, 25(1): 33–42.

[24] Thywissen JA. Secure information flow in the Orc concurrent programming language. https://orc.csres.utexas.edu/papers/Secure-Information-Flow-Orc.pdf [Student Report], University of Texas at Austin, 2009.

[25] Wu Y. An introduction to BPEL standard and its extensions. http://www.soberit.hut.fi/T-86/T-86.5161/2007/BPEL_final_report_yanbowu.pdf

[26] Yew L, Young WD, Cook WR. cOrcS: Continuation of Orc Security with static integrity checking. https://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2113.pdf, 2012