kiltera: a simulation language for timed, dynamic structure systems

Ernesto Posse and Hans Vangheluwe
Modelling, Simulation and Design Lab
School of Computer Science
McGill University
Montreal, Quebec, Canada

Abstract

In recent years there has been an increased interest in modelling systems whose structure changes dynamically, for example to study mobility. At the same time, there is a plethora of simulation languages for discrete-event systems that include an explicit notion of time, but few, if any, support explicitly the notion of structural change. In this paper we introduce a process description language which addresses these issues explicitly. We discuss an application to modelling networks of servers.

1. Introduction

Modelling and simulation are concerned with the description of dynamic systems. A system is an aggregate of components. A dynamic system is a system which undergoes changes over time. A dynamic system has structure and behaviour. The structure refers to how a system's components are related to each other and to the whole. The behaviour refers to the actions or events of a system with respect to its environment and to the passage of time.

Discrete-event modelling and simulation [11] focus on the notions of state and event. These assume an underlying notion of time: at any given point in time a system is in some state, and events change the state. These notions are at the core of the description of behaviour of discrete-event systems. When a system has two or more components, the behaviour of one can affect the others. Interaction between components becomes a central aspect of a system's behaviour. Therefore structure and behaviour are closely intertwined.

Most modelling and simulation approaches attempt to make a clear-cut distinction between structure and behaviour. The result are languages and formalisms in which the dynamics refer to changes of state where state is unrelated to structure. Hence only systems with a static stucture can be described in such languages. However, many real world systems are not static in this sense but have a dynamic structure: relations between components change over time. Examples abound in many fields. In telecommunications and computing we have systems such as mobile-phone networks and adaptive computer and network architectures as well as complex software architectures. In biology we can see anything from molecules reacting to full eco-systems as systems undergoing structural change. In the social sciences we can see human organizations as systems with dynamic structure as well.

Linking structure to state, as is done in [2], can give us the power to describe structural changes, but emphasis on explicit descriptions of state often lead to verbose formalisms. Furthermore, the notion of state itself is an abstraction which we use as a means to describe a system's condition at one point in time. But what is important about a state is not the state itself, but whatever relevant information we can observe from it, or what actions or events are possible in that state. Thus, an alternative approach is to focus on the actions, in particular the *interactions* in which the system can engage. If we focus on interaction, component interconnection, the network of communications, becomes the central aspect of structure.

Process description languages such as CSP [4], CCS [7], and LOTOS [5], provide such emphasis on component interaction. Some languages, in particular the so-called π-calculus [8], go further and support structural changes. Another approach is that of the Actor model of Agha and Hewitt ([1], [6], [3]). Nevertheless, to the best of our knowledge, none of these combine the ability to undergo structural changes with an explicit notion of time familiar in the discrete-event setting.

In this paper we introduce a language which we call kiltera that combines the ability to observe the passage of time and describe a system's behaviour in a time-dependent manner, with the ability to describe changes in the network of communications between components. We believe that the modelling and simulation community can benefit from languages that combine these features.

The rest of the paper is organized as follows: section 2 provides a general description of the language's computational model; section 3 introduces the core of the language's syntax; section 4 introduces the language's semantics; section 5 discusses some applications, and finally section 6 provides some concluding remarks.

2. The kiltera computational model

kiltera is a language for describing concurrent, interacting, mobile components which "live" over time.

A kiltera system consists of one or more dynamic components or *processes*. A process is a modular component with a well defined interface consisting of a set of *ports*. The only way to interact with a process is through its ports. Each process is an independent computational unit and proceeds concurrently with all other processes. Processes execute a number of different actions. The most important kind of actions performed by processes are communication actions.

A process is not necessarily a purely sequential computation, as it may be itself composed of parallel subprocesses. In other words, a network of processes is a process, thus enabling modularity. This nesting of subprocesses is central aspect of the structure of a process.

2.1. Interaction

Processes communicate through channels by message-passing. Processes have ports. A channel connects two or more processes through their ports. There are two primitive communication operations: sending a message and receiving a message through some channel. When a process sends or receives a message it does so by specifying the port connected to the relevant channel.

There are two kinds of communication: synchronous and asynchronous.

Synchronous communication means that the sender and the receiver of a message engage in the interaction simultaneously. In other words, execution of a communication action (sending or receiving a message) is a blocking operation: the process executing the action will be blocked until some other process is ready to engage in the interaction. If a process attempts a receiving operation on a synchronous channel, it will be blocked until some other process sends a message through that channel, and dually, if a process attempts a sending operation on a synchronous channel, it will also be blocked until some other process executes a receiving action on that channel.

An alternative way of viewing synchronous communication is in terms of acknowledgement: a send operation waits for acknowledgment from the receiver before completion.

Asynchronous communication means that the sender and receiver do not need to engage in the interaction simultaneously. In other words, only the receiving operation is blocking, while the sender of a message can proceed with execution without waiting for acknowledgement from the receiver.

In the rest of this paper we consider only the synchronous-communication fragment of kiltera.

Channels can connect more than two processes, but communication is by "unicasting" or "two-way" communication rather than "multicasting" or "multi-way" communication. This is, when a process sends a message through a channel connected to two or more receivers, only one of the receivers will get the message and the rest will remain blocked. The selection is non-deterministic: the receivers are considered to be competing for the message. The same is true if there are many senders and one receiver.

2.2. Link mobility

A *process network* is a set of processes connected through channels. A *configuration* is a particular topology of this network. Channels are first-class values and therefore can be communicated between processes. This means that the configuration of a network can change dynamically when processes execute. In particular processes can acquire access to channels and therefore to other processes to which they didn't have access. This is known as *link mobility* and it is the kind of dynamic-structure supported by kiltera.

2.3. Time

kiltera processes execute over time (whether it is logical or physical time.) The execution of a process occurs with respect to a global clock. The time base is the real numbers.

The execution of each action is an *event* which takes place at a particular point in time. Most normal actions do not take (logical) time to complete.

Synchronization actions (blocking actions such as waiting to receive a message,) might take some time: from the point in time when the action is initiated or attempted until the point in time when the synchronization (exchange of information) actually occurs: for instance, if process P_1 attempts to receive a message through some channel a at time t_0 but no input is available on the channel at that time, it will block and wait until some other process sends data. When another process P_2 sends data through a at time $t_1 > t_0$ then synchronization occurs, and the receiving action of P_1 is said to have taken $t_1 - t_0$ time units. The actual synchronization is not considered to take any time itself.

Processes can be made to wait for a given amount of time, or equivalently, a process may schedule events in the future. Processes can also specify *timeouts*: the scheduling

of future events which cancels another process which has not yet finished. Processes can also measure the passage of time and change their behaviour accordingly.

3. Syntax

Table 1 gives a subset of the constructs in kiltera. We call $\mathcal{P}$ the set of all process expressions, $\mathcal{D}$ the set of process definitions, $\mathcal{A}$ the set of actions, $\mathcal{E}$ the set of expressions, and $\mathcal{N}$ the set of all possible names. We use the following convention in the syntax: P, P_i range over $\mathcal{P}$, D,D_i range over $\mathcal{D}$, A over $\mathcal{A}$, E, E_i range over $\mathcal{E}$, a, a_i, x, x_i, t, f, u, u_i range over $\mathcal{N}$, n ranges over the set of (floating-point) numbers, and s ranges over the set of strings. For brevity we use $\{...\}$ to describe nesting and disambiguation, but in the actual implementation, indentation is used instead.

The full language contains more constructs to make it more practical, in particular it contains constructs to describe arrays of processes and channels, useful when describing large systems.

4. Semantics

In this section we give first an informal description of the semantics of the constructs in Table 1. We later provide a formal description.

The process definition process a[u_1,...,u_n]: P defines a class of processes named a, with ports $u_1, ..., u_n$ and body P. A process definition defines a class of processes that can be instantiated. The process definition process a[u_1,...,u_n](x_1,...,x_m): P is the same but declares additional parameters to be passed at the moment of instantiation.

The function definition function f(x_1,...,x_n): E defines a (pure) function.

The nil process is the process that does nothing. In particular, it does not interact with any process. The process done is used to represent successful termination.

Processes of the form $A \rightarrow P$ execute the action A, which may be blocking as described below, and then continue to behave as process P. Processes of the form $A\texttt{at}t \rightarrow P$ behave the same way but bind the variable t to the time elapsed between the start of the action and the time the action finishes. This will be 0 if the action is non-blocking.

A process of the form send E to $u \rightarrow P$ sends the value of the expression E to the port u of the process in which it occurs and then continues as P. This operation is blocking: the sender waits until some receiver is ready to take the message. Since communication is "two-way," only one receiver gets the message even if more than one are connected to the port u.[1]

A process of the form receive x from $u \rightarrow P$ blocks until a message arrives at port u. This message is then bound to the variable x and the process continues as P. The process receive x from u at $t \rightarrow P$ is the same, but the variable t gets bound to the time elapsed from the time the action is attempted until the message is received.

A process of the form let x = E in P behaves like P with all the free occurrences of x replaced by the value of E.

A process of the form if E then P_1 else P_2 behaves as P_1 if E evaluates to $true$ or as P_2 otherwise. A process of the form if E then P behaves as if E then P else done.

A process of the form match E with $E_1 \rightarrow P_1$ | ... | $E_n \rightarrow P_n$ does pattern matching: the value of E is matched against the patterns $E_1, ..., E_n$ in that order. If the pattern E_k succeeds, then process P_k is executed, binding the free variables of E_k to the corresponding values of E.

A process of the form wait $E \rightarrow P$ blocks the process for t time units where t is the value of the expression E. Then the process continues as P.

A process of the form timeout P_1 after $E \rightarrow P_2$ behaves as P_1, but if after t time units, where t is the value of the expression E, P_1 has not finished *and* it has not engaged in any external interaction (i.e. it has not performed any sends or receives,) then P_1 is aborted and the process P_2 starts. If, however, P_1 does finish or engages in external interaction before t time units, then P_2 is discarded.

A process of the form channel x_1,...,x_n in P creates new *local* channels $x_1, ..., x_n$ and executes process P. The scope of the names $x_1, ..., x_n$ is P, hence these channels cannot be directly accessed by another process. This construct provides encapsulation and hiding.

A process of the form par { P_1,...,P_n } executes the subprocesses $P_1, ..., P_n$ in parallel.

A process of the form D_1 ... D_n in P executes the process P in an environment with the process classes defined by $D_1, ..., D_n$. The scope of these definitions is P.

A process of the form a[x_1,...,x_n] creates and runs an instance of the process class a (as defined in the current scope) connecting the channels $x_1, ..., x_n$ to the ports $u_1, ..., u_n$ of the process, and executing the body of the process definition, assuming the process is defined in the current scope as process a[u_1,...,u_n]: P. The process a[x_1,...,x_n](E_1,...,E_n) does the same, but passes as parameters the values of expressions $E_1, ..., E_m$ if the process class declared parameters.

Expressions are either booleans, numbers, strings, tuples, the unit constant (representing a token value), variables or function applications. Channels are considered first-class values and can be sent as messages. This allows the modelling of link mobility as described in section 4.1.

[1] We describe here the synchronous communication semantics, but the implementation also supports asynchronous communication.

$$
\begin{array}{rcl}
D & \rightarrow & \mathsf{process}\ a[u_1,...,u_n] : P \\
 & | & \mathsf{process}\ a[u_1,...,u_n](x_1,...,x_m) : P \\
 & | & \mathsf{function}\ f(x_1,...,x_n) : E
\end{array}
$$

$$
\begin{array}{rcl}
P & \rightarrow & \mathsf{nil} \\
 & | & \mathsf{done} \\
 & | & A \rightarrow P \\
 & | & A\ \mathsf{at}\ t \rightarrow P \\
 & | & \mathsf{let}\ x\ =\ E\ \mathsf{in}\ P \\
 & | & \mathsf{if}\ E\ \mathsf{then}\ P \\
 & | & \mathsf{if}\ E\ \mathsf{then}\ P_1\ \mathsf{then}\ P_2 \\
 & | & \mathsf{match}\ E\ \mathsf{with}\ E_1 \rightarrow P_1 \,|\, ... \,|\, E_n \rightarrow P_n \\
 & | & \mathsf{wait}\ E\ \rightarrow P \\
 & | & \mathsf{timeout}\ P_1\ \mathsf{after}\ E\ \rightarrow P_2 \\
 & | & \mathsf{channel}\ x_1, ..., x_n\ \mathsf{in}\ P' \\
 & | & \mathsf{par}\ \{P_1, ..., P_n\} \\
 & | & D_1\ ...\ D_n\ \mathsf{in}\ P \\
 & | & a[x_1,...,x_n] \\
 & | & a[x_1,...,x_n](E_1, ..., E_m)
\end{array}
$$

$$
\begin{array}{rcl}
A & \rightarrow & \mathsf{send}\ E\ \mathsf{to}\ u \\
 & | & \mathsf{receive}\ x\ \mathsf{from}\ u
\end{array}
$$

$$
\begin{array}{rcl}
E & \rightarrow & \mathsf{unit} \\
 & | & n \\
 & | & x \\
 & | & \mathsf{true} \\
 & | & \mathsf{false} \\
 & | & \text{“}s\text{”} \\
 & | & (E_1,...,E_n) \\
 & | & unop\ E' \\
 & | & E_1\ binop\ E_2 \\
 & | & f(E_1,...,E_n) \\
unop & \rightarrow & - \,|\, \mathsf{not} \\
binop & \rightarrow & + \,|\, - \,|\, * \,|\, / \\
 & | & \mathsf{and} \,|\, \mathsf{or} \,|\, = \\
 & | & < \,|\, > \,|\, <= \,|\, >=
\end{array}
$$

Table 1. kiltera **syntax.**

4.1. Structural changes

As explained in 2.2, the structural changes supported by kiltera are changes to the topology of the network connecting a system's components. This is achieved, as in the π-calculus, by making channels first-class values which can be communicated. A typical example is the following:

```
process A[x,y]: send y to x→done
process B[x]: receive w from x →send 1 to w →done
process C[z]: receive m from z → ...
in
 channel x in
  par {
   B[x],
   channel y in
    par { A[x,y], C[y] } }
```

Here we have three components A, B and C. A and C are linked together through channel y, and B is linked to A through channel x. B is waiting for a message on this channel, and A sends the channel y as a message through x. When B receives this message it sends a message (1) through that channel, so C receives a 1. Hence, initially C could communicate only with A, but after A sent y, it could also communicate with B, and thus, the network's topology changed.

Apart from link mobility, other forms of structural changes are supported: creation of components is achieved with process instantiation, and destruction is achieved either by voluntary termination or by timeout.

4.2. Formal semantics

The operational semantics is given, in the style of Structural Operational Semantics [9], by a timed-labelled transition system defined inductively by inference rules of the form: "RULE: if B_1, B_2, ..., B_n then C". A rule with no premises is an axiom. These rules are presented in Table 2.

We describe the semantics in terms of *configurations*. A configuration $\Delta \vdash P$ consists of a multiset Δ of process definitions, representing the execution environment, and a process P. The semantics are given by a timed-labelled transition system $(\mathcal{C}, \mathcal{L}, \rightarrow, \rightsquigarrow)$ where $\mathcal{C}$ is the set of configurations, $\mathcal{L}$ is the set of possible action labels described below, $\rightarrow \subseteq \mathcal{C} \times \mathcal{L} \times \mathcal{C}$ is the *transition* relation defined in Table 2, and $\rightsquigarrow \subseteq \mathcal{C} \times \mathbb{R}_0^+ \times \mathcal{C}$ is the *evolution* relation defined in Table 2, where $\mathbb{R}_0^+$ denotes the set of positive reals including 0. We write $\Delta \vdash P \xrightarrow{\alpha} \Delta' \vdash P'$ for $(\Delta \vdash P, \alpha, \Delta' \vdash P') \in \rightarrow$, to mean that process P in an environment Δ can become process P' in an environment Δ' by performing action α. Similarly, we write $\Delta \vdash P \overset{d}{\rightsquigarrow} \Delta' \vdash P'$ for $(\Delta \vdash P, d, \Delta' \vdash P') \in \rightsquigarrow$, to mean that process P in an environment Δ evolves into process P' in an environment Δ' after an amount of time d. For brevity of notation we write $P \xrightarrow{\alpha} P'$ to mean that $\Delta \vdash P \xrightarrow{\alpha} \Delta \vdash P'$ for any Δ. Similarly, we write $P \overset{d}{\rightsquigarrow} P'$ to mean that $\Delta \vdash P \overset{d}{\rightsquigarrow} \Delta \vdash P'$ for any Δ.

Intuitively the transition relation is intended to capture the notion of instantaneous change of state while the evolution relation is intended to capture the passage of time.

$$match(p, v, \sigma) \stackrel{def}{=}$$
$$\begin{cases} \sigma & \text{if } p = v \text{ and } v \in \mathcal{K} \text{ or } p \in dom(\sigma) \\ & \quad \text{and } \sigma(p) = v \\ \sigma \cup \{p \mapsto v\} & \text{if } p \in \mathcal{N} \text{ and } p \notin dom(\sigma) \\ \sigma_n & \text{if } p = (p_1, ..., p_n) \text{ and } v = (v_1, ..., v_n), \\ & \quad \text{where } \forall i \in \{1, ..., n\}, \\ & \quad \sigma_i \stackrel{def}{=} match(p_i, v_i, \sigma_{i-1}) \\ & \quad \text{and } \sigma_0 \stackrel{def}{=} \sigma \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 1. Pattern-value match

We denote $\mathcal{V}$ for the set of possible values, which includes the booleans, the real numbers, strings, tuples of values and channels. We call $\mathcal{K} \subseteq \mathcal{V}$ the set of constants including booleans, strings, numbers, and channels (but not tuples.) We assume there is a function $eval : \mathcal{E} \to \mathcal{V}$, that evaluates an expression. We omit the details here due to lack of space.

We take the set of action labels $\mathcal{L}$ to be the set of elements of the form $c!v$, $c?x$, τ or $\surd$. An action $c!v$ represents a message v sent over a channel c, with $c \in \mathcal{N}$ any name and $v \in \mathcal{V}$ any value. An action $c?x$ represents the reception of a message through a channel c, where $x \in \mathcal{N}$. τ is a special action used to denote an unobservable (internal) event and $\surd$ denotes termination. For notational convenience we define a function $act : \mathcal{A} \to \mathcal{L}$ associating syntactic actions with action labels as follows: $act(\mathsf{send}\ E\ \mathsf{to}\ u) \stackrel{def}{=} u!eval(E)$ and $act(\mathsf{receive}\ x\ \mathsf{from}\ u) \stackrel{def}{=} u?x$.

We need the following definitions. A name x is *bound* in P if P is a process definition and x is either a port, a parameter or the name of the process definition, or if P is a process of the form $\mathsf{channel}\ x\ \mathsf{in}\ P'$, $\mathsf{receive}\ x\ \mathsf{from}\ u \to P'$, $A\ \mathsf{at}\ x \to P'$, or $\mathsf{let}\ x = E\ \mathsf{in}\ P'$. A *name substitution* is a function $\sigma : \mathcal{N} \to \mathcal{N}$. We write $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ for the substitution σ where $\sigma(x_1) = y_1$, ..., $\sigma(x_n) = y_n$ and $\sigma(z) = z$ for all $z \notin \{x_1, \ldots, x_n\}$. The notion of substitution is extended to processes in the natural way taking care of avoiding capture. For full details see [10]. We write $P\sigma$ for $\sigma(P)$ denoting the process where all free occurrences of each $x \in dom(\sigma)$ have been substituted by $\sigma(x)$.

In order to define the semantics of pattern-matching we need the following definitions: a *pattern* is an expression that may have variables, a *datum* is an expression with no variables, and *(general) substitution* is a mapping $\sigma : \mathcal{N} \to \mathcal{V}$. We call $\mathcal{S}$ the set of substitutions. We now define a function $match : \mathcal{E} \times \mathcal{V} \times \mathcal{S} \to \mathcal{S}$ as shown in Figure 1.

The core rules defining the transition and evolution relations are given in Table 2.

Some constructs are defined as syntactic sugar: $\mathsf{wait}\ t \to P$ is short for $\mathsf{timeout}\ \mathsf{nil}\ \mathsf{after}\ t \to P$. $\mathsf{if}\ E\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ is short for $\mathsf{match}\ E\ \mathsf{with}\ \mathsf{true} \to P_1 \mid \mathsf{false} \to P_2$ and $\mathsf{if}\ E\ \mathsf{then}\ P$ is short for $\mathsf{match}\ E\ \mathsf{with}\ \mathsf{true} \to P \mid \mathsf{false} \to \mathsf{done}$. $\mathsf{channel}\ x_1,...,x_n\ \mathsf{in}\ P$ is shorthand for $\mathsf{channel}\ x_1\ \mathsf{in}\ \mathsf{channel}\ x_2\ \mathsf{in}\ ...\mathsf{channel}\ x_n\ \mathsf{in}\ P$. We omit the rules for the parametrized forms of process definition and process instantiation due to lack of space, but the corresponding rules are straight-forward.

We impose the following restriction on the definition of our timed-labelled transition system $(\mathcal{C}, \mathcal{L}, \to, \rightsquigarrow)$: for any configuration $\gamma \in \mathcal{C}$ such that there is a configuration $\gamma' \in \mathcal{C}$ where $\gamma \stackrel{\tau}{\to} \gamma'$ then for all $d \in \mathbb{R}^+$ there is no configuration $\gamma'' \in \mathcal{C}$ such that $\gamma \stackrel{d}{\rightsquigarrow} \gamma''$. This restriction requires internal transitions to be urgent, this is, actions are performed as soon as they are enabled.

Given the definitions of transition and evolution we can now define a computation.

For a pair of configurations $\gamma, \gamma' \in \mathcal{C}$ we write

$$\gamma \stackrel{\tilde{\alpha}}{\Longrightarrow} \gamma'$$

where $\tilde{\alpha} \stackrel{def}{=} \alpha_1, ..., \alpha_{n-1}$ is a sequence of actions if there are configurations $\gamma_1, ..., \gamma_n$ such that

$$\gamma = \gamma_1 (\stackrel{\tau}{\to})^* \stackrel{\alpha_1}{\longrightarrow} (\stackrel{\tau}{\to})^* \cdots (\stackrel{\tau}{\to})^* \stackrel{\alpha_{n-1}}{\longrightarrow} (\stackrel{\tau}{\to})^* \gamma_n = \gamma'$$

Note that if $\gamma \stackrel{d}{\rightsquigarrow} \gamma'$ and $\gamma' \stackrel{d'}{\rightsquigarrow} \gamma''$ then $\gamma \stackrel{d+d'}{\rightsquigarrow} \gamma''$. Therefore any computation has the form:

$$\gamma \stackrel{\tilde{\alpha}_1}{\Longrightarrow} \stackrel{d_1}{\rightsquigarrow} \stackrel{\tilde{\alpha}_2}{\Longrightarrow} \stackrel{d_2}{\rightsquigarrow} \cdots \stackrel{\tilde{\alpha}_m}{\Longrightarrow} \stackrel{d_m}{\rightsquigarrow} \gamma'$$

5. Applications

In this section we describe an application of kiltera to the problem of distributing tasks among a group of servers.[2]

For this example it will be useful to have buffers. Figure 2 shows a possible implementation of bounded buffers. Here each cell process can hold a value which is passed to whomever requests it. A bounded buffer is then simply a finite connection of these cells.

We will also use a "generator" process to produce jobs for the servers, as well as a "statistics manager" that will keep track of average waiting times and a "consumer" which will receive the finished jobs. These are depicted in Figure 3.

[2] In this example we use a built-in function supported in the current implementation of kiltera, random, which generates a pseudorandom number between 0 and 1. We also use indentation to denote nesting instead of {...} as described in section 3. Semantically, these examples also rely on some key features, namely synchronous two-way communication (as opposed to asynchronous or multi-way communication,) the ability to create new processes, to send channels in messages, as well as the time-related constructs.

NIL: nil $\overset{\alpha}{\not\longrightarrow}$ ENIL: nil $\overset{d}{\rightsquigarrow}$ nil DONE: done $\xrightarrow{\surd}$ nil ACT1: $A \rightarrow P \xrightarrow{act(A)} P$ ACT2: A at $t \rightarrow P \xrightarrow{act(A)} P\{t \mapsto 0\}$

EACT1: $A \rightarrow P \overset{d}{\rightsquigarrow} A \rightarrow P$ EACT2: A at $t \rightarrow P \overset{d}{\rightsquigarrow} P\{t \mapsto t + d\}$ LET: let $x = E$ in $P \xrightarrow{\tau} P\{x \mapsto eval(E)\}$

MTCH: match E with $E_1 \rightarrow P_1 \mid ... \mid E_n \rightarrow P_n \xrightarrow{\tau} P_i\sigma$ if $\sigma \neq \emptyset$ where $\sigma \overset{def}{=} match(E_i, eval(E), \emptyset)$

TMOUT1: if $P \xrightarrow{\alpha} P'$ and $\alpha \neq \tau$ then timeout P after $E \rightarrow Q \xrightarrow{\alpha} P'$

TMOUT2: if $P \xrightarrow{\surd}$ nil then timeout P after $E \rightarrow Q \xrightarrow{\surd}$ nil

TMOUT3: if $P \xrightarrow{\tau} P'$ then timeout P after $E \rightarrow Q \xrightarrow{\tau}$ timeout P' after $E \rightarrow Q$

TMOUT4: if $eval(E) = 0$ then timeout P after $E \rightarrow Q \xrightarrow{\tau} Q$

ETMOUT: if $P \overset{d'}{\rightsquigarrow} P'$ and $0 < d' \leq eval(E)$ then timeout P after $E \rightarrow Q \overset{d'}{\rightsquigarrow}$ timeout P' after $E - d' \rightarrow Q$

CHN: channel x in $P \xrightarrow{\tau} P\{x \mapsto x'\}$ where x' is a fresh name PAR: if $P \xrightarrow{\alpha} P'$ then par $\{..., P, ...\} \xrightarrow{\alpha}$ par $\{..., P', ...\}$

EPAR: if $\forall i \in \{1, ..., n\}.\ P_i \overset{d}{\rightsquigarrow} P_i'$ then par $\{P_1, ..., P_n\} \overset{d}{\rightsquigarrow}$ par $\{P_1', ..., P_n'\}$

COMM: if $P \xrightarrow{c!v} P'$ and $Q \xrightarrow{c?x} Q'$ then par $\{..., P, ..., Q, ...\} \xrightarrow{\tau}$ par $\{..., P', ..., Q'\{x \mapsto v\}, ...\}$

DEF: $\Delta \vdash D_1...D_n$ in $P \xrightarrow{\tau} \Delta \cup \{D_1, ..., D_n\} \vdash P$

INST: if process $a[x_1, ..., x_n] : P \in \Delta$ and $\Delta \vdash P\{x_1 \mapsto u_1, ..., x_n \mapsto u_n\} \xrightarrow{\alpha} \Delta' \vdash P'$ then $\Delta \vdash a[u_1, ..., u_n] \xrightarrow{\alpha} \Delta' \vdash P'$

Table 2. Operational semantics

```
process Cell[inp, outp]:
  receive x from inp ->
   send x to outp ->
    Cell[inp, outp]

process Buffer[i,o](n):
  if n = 1 then
   Cell[i,o]
  else
    channel h in
     par
      Cell[i,h]
      Buffer[h,o](n-1)
```

Figure 2. Bounded buffers.

```
process Generator[output](g0, g1, s0, s1, c, t):
  let d = (g1-g0)*random()+g0 in
   let s = (s1-s0)*random()+s0 in
    wait d ->
     send ("job", c, t+d, s) to output ->
      Generator[output](g0,g1,s0,s1,c+1,t+d)

process StatsManager[stats](c, sum, avg):
  print ("stats   ", c, fit(sum,6), fit(avg,6)) ->
   receive t from stats ->
    StatsManager[stats](c+1,sum+t,
                        (sum+t)/(c+1))

process Consumer[inp]:
  receive result from inp ->
   Consumer[inp]
```

Figure 3. Generators and statistics.

The generator process has parameters specifying the delay between generated jobs (uniformily distributed over the interval $[g0, g1)$) and the size of the generated jobs (also uniformily distributed over the interval $[s0, s1)$,) the number of jobs so far (c) and the current time (t.) The generated jobs are tagged with an id, the time of creation and their size.

In this example we model a system consisting of a set of *nodes*, each of which has a number of *servers* that will perform tasks assigned to them. Each node receives job requests from the outside and assigns each of them to one of its servers. If all servers are busy, a node asks some other node for help, and if the other node has some idle server, it "moves" it to the requesting node. If job requests continue to arrive, all servers are busy and a neighbouring node cannot provide a spare server, then jobs are queued until some servers becomes free.

In order to distribute tasks and handle other nodes' requests, a node has, in addition to its servers, a buffer for jobs, a job dispatcher and a "move-handler." Figure 4 shows the structure of a node. The specification for nodes is shown in Figure 5. A node has an input port, an output port, a port used to send requests to other nodes for help (ask,) a port where such requests are received (move,) and a port to link with the statistics manager. The parameters are the maximum time the node will wait before asking another node for help (busyt), and the size of the buffer (bsize). Note that all servers share one link (b) with the dispatcher (and the move handler.) The dispatcher uses the ask link to send requests for servers to other nodes when required, and the move-handler takes care of such requests coming from the move link.

Servers are shown in Figure 6. Each server has two ports:

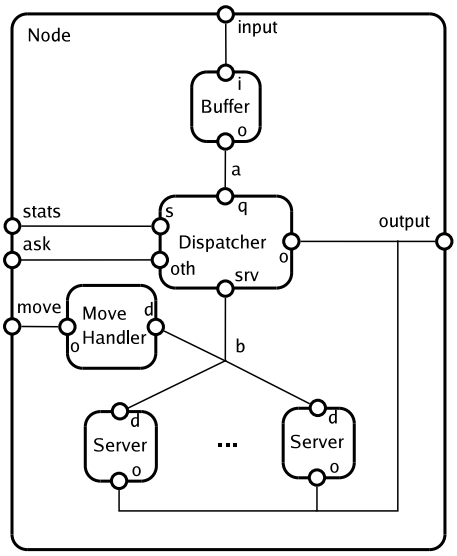

Figure 4. Nodes

```
process Node[input, output, ask, move,
                                stats](busyt, bsize):
  channel a, b in
    par
      Buffer[input, a](bsize)
      Dispatcher[a, output, b, ask, stats](0, busyt)
      MoveHandler[move,b]
      Server[b, output](idlet)
      Server[b, output](idlet)
```

Figure 5. Nodes

one linking it to a dispatcher, and one output port. A server can be in one of two modes: *idle* or *processing*. When idle, a server waits for messages. Messages are either jobs or "move" requests. If a message arrives with a job, it changes to the processing mode. If a move request arrives, it comes with links to the requesting node's dispatcher and output. In this case the server remains idle but becomes connected to the channels received. Since the only way to observe and interact with a process is through its ports, from the point of view of the dispatchers the server behaves as if it moved. In processing mode, the server will remain busy, without accepting any messages for an amount of time associated with the task. When this time is due, the server sends a "done" message to the output port and returns to idle mode.

The specification of dispatchers is shown in Figure 7. Each dispatcher has a port to link to the queue, an output port, a "server link" to connect with all the servers, a port to send requests to other nodes (other,) and one for linking with the statistics manager. Its parameters are the current time (time), and the maximum time before asking another node for help (busyt).

```
process Server[dispatcher, output]:

  process Idle[dispatcher, output]:
    receive message from dispatcher ->
      match message with
        ("job", id, t0, size) ->
          Processor[dispatcher,output](id,size)
      | ("move", new_disp, new_out) ->
          Idle[new_disp, new_out]

  process Processor[dispatcher,output](id,size):
    wait size ->
      send ("done",id) to output ->
        Idle[dispatcher, output]

  in
    Idle[dispatcher, output]
```

Figure 6. Servers

A dispatcher waits for jobs coming from the buffer, and when one arrives the dispatcher attempts to send it through the server link. If one of the servers gets the job, the dispatcher sends a message to the statistics manager and goes back to waiting for more jobs. If after a certain amount of time (busyt) none of the associated servers takes the job, it sends a request to some other node, passing along its server's link and output channel. Once it has been taken, it sends the job to the server link again. At this point several things can happen: 1) the other node has a spare server which got the "move" message with the new links and so receives the job, 2) one of the busy servers in the requesting node becomes idle and accepts the job, or 3) all servers in both nodes are busy, in which case the dispatcher remains blocked (since the send is blocking) and new jobs are buffered. Once the job is taken, it sends a message to the statistics manager and goes back to waiting for more jobs.

Figure 8 shows the "move handler." This is the component in charge of handling requests for free servers from other nodes. It has two ports: one where moving requests are expected from other nodes and one linking with the node's servers (and dispatcher.) When it receives a request, together with the other node's channels, it sends a "move" message through the server channel. If one server accepts the message it goes back to listening for requests. Otherwise it remains blocked until some server accepts the message, and therefore the requesting dispatcher remains blocked as well.

Finally Figure 9 shows a sample network consisting of a generator a statistics manager and two nodes connected.

```
process Dispatcher[queue, out, servers,
                   other, stats](time, busyt):
  receive message from queue at e1 ->
   match message with
    ("job", id, t0, size) ->
     let t1 = time + e1 in
      timeout
       send message to servers at e2 ->
        let t2 = t1 + e2 in
         send t2 - t0 to stats ->
          Dispatcher[queue,out,servers,
                     other, stats](t2, busyt)
      after busyt ->
       send ("req",servers,out) to other ->
        send message to servers at e2 ->
         let t2 = t1 + e2 + busyt in
          send t2 - t0 to stats ->
           Dispatcher[queue,out,servers,
                      other, stats](t2, busyt)
```

Figure 7. Dispatcher

```
process MoveHandler[other,server_link]:
  receive message from other ->
   match message with
    ("req", other_disp, other_out) ->
     send ("move",other_disp,other_out)
                          to server_link ->
      MoveHandler[other,server_link]
```

Figure 8. Move handler

6. Final remarks

We have introduced a process description language with an explicit notion of time and support for dynamic structural changes. While this language shares many characteristics with some existing process description languages such as CSP, CCS and the π-calculus, it differs from them in that, with respect to CCS and CSP, it supports mobility, and with respect to the π-calculus, it supports timed-systems. However the closeness with these languages means that we might be able to take advantage of the theoretical frameworks developed for them.

We have built a thread-based interpreter for this language and we are pursuing several lines of research: we are exploring the possibility of supporting full active-process migration, which may be more adequate in a distributed setting; we are investigating the relationships with other languages and formalisms; and we are working on an event-based simulator supporting both virtual and real-time simulation.

The implementation can be obtained at http://moncs.cs.mcgill.ca/people/eposse/projects/kiltera.

```
process Network[]:
  channel gen, stats, sink, a, b in
   par
    StatsManager[stats](0, 0.0, 0.0)
    Generator[gen](1.0, 2.0, 1.0, 3.0, 0, 0.0)
    Node[gen, sink, a, b, stats](4.0,100)
    Node[gen, sink, b, a, stats](4.0,100)
    Consumer[sink]
```

Figure 9. Node network.

References

[1] Gul A. Agha and Carl Hewitt. Concurrent programming using Actors. In *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[2] F. Barros, M. Mendes, and B. Zeigler. Variable DEVS — variable structure modeling formalism: An adaptive computer architecture application. 1994.

[3] M.-W. Jang et al. An actor based simulation for studying uav coordination. In *Proc. of 15th European Simulation Symposium*, 2003.

[4] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, August 1978.

[5] ISO. LOTOS - language of temporal ordering specification. Technical Report ISO DP 8807, 1987.

[6] M.-W. Jang and G. Agha. Agent framework services to reduce communication overhead in large-scale agent-based simulations. *Simulation Modelling Practice and Theory*, 14(6):679–694, August 2006.

[7] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[8] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and 86, Computer Science Dept., University of Edinburgh, March 1989.

[9] Gordon Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Dept. of Computer Science, Aarhus University, 1981.

[10] Ernesto Posse. kiltera: a language for concurrent, interacting, timed mobile systems. Technical Report SOCS-TR-2006.4, School of Computer Science, McGill University, 2006.

[11] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, second edition, 2000.