Journal of Universal Computer Science, vol. 11, no. 6 (2005), 850-873
submitted: 16/2/04, accepted: 16/2/05, appeared: 28/6/05 © J.UCS

Time Costs in Actor Computations

M. Di Santo, F. Frattolillo
(Research Centre on Software Technology, Dept. of Engineering,
University of Sannio, Italy
disanto,frattolillo@unisannio.it)

Abstract: Actor programs give rise to computation structures that evolve dynamically and unpredictably both in shape and size. Therefore, their execution times cannot be statically determined. This paper describes an approach to the problem of estimating time costs of actor programs. The approach takes into account the constraints imposed both by the semantics and implementation of the model. In particular, implementation constraints can be captured and exploited to drastically reduce the number of computations generable by the program, thus simplifying the execution time evaluation. Moreover, execution times are expressed in a parametric form by using a variant of the *LogP* model able to synthetically characterize the target hardware/software platform.
Key Words: actor computations, execution time, parallel computations, implementation constraints
Category: C.4, D.2.8

1 Introduction and motivations

Developing parallel and distributed applications is a difficult task that can be greatly simplified by using high-level, object-oriented languages [Agha 90]. A prototypical model of such languages is the well known *Actor* programming model [Agha 86], based on autonomous agents, called actors, provided with internal state and communicating solely via asynchronous and reliable message passing. The model supports the development of highly dynamic applications and its specific characteristics make it suitable to be implemented on parallel and distributed computing systems [Agha et al. 97, Varela, Agha 98].

The adoption of high-level, object-oriented programming models, however, usually makes it difficult to statically evaluate the execution time (in the following also called "time cost") of a program on a given hardware/software computing platform, because of the abstraction gap existing between the programming mechanisms provided by the models and the ones implemented at the hardware level. In fact, the different implementations allowed by the semantics of a high-level, object-oriented programming model on a given computing platform and the vagaries of timings caused by concurrent executions on multiple processors often make the static performance analysis of programs a burdensome activity.

The above considerations particularly apply to the Actor model, as pointed out in [Skillicorn, Talia 96]. In fact, due to the nature of the communication delivery system characterizing the most common implementations on parallel computing platforms, messages asynchronously exchanged between two actors may be received out of their sending sequence. Consequently, different executions of an actor program on the same input data may give rise to different computations and even to different results. However, if the model is implemented according to specific constraints, the number of the generable computations can be drastically reduced and, consequently, time cost evaluation simplified. In particular, this is true when model implementations force computations to be “fifo” or “causally ordered” [Charron-Bost et al. 96]. To this end, it is worth noting that there exist communication protocols which can be adopted to manage communications among actors at low level and which are able to ensure the fifo or causally ordered property to computations [Mattern, Fünfrocken 95].

Since the time costs of an actor program directly depend on the runtime behavior of the whole actor system implemented on a given computing platform, the problem of estimating such costs can be actually tackled only if actor systems as well as the computations they generate can be described and then formally specified. To this end, it is worth noting that, over the last decade, there has been a growing interest in the software performance analysis, since performance problems may be so severe that they can require changes in the software architecture of programs as well as specific choices of the target computing platforms. The most recent research in this field has focused on the approaches which propose performance models to characterize the quantitative behavior of software systems [Balsamo et al. 04]. The main goal has been to integrate performance analysis in the software development process, in order to correctly drive the early phases of the software life cycle. However, although several of such approaches have been successfully applied, the proposed solutions do not usually take into account the actual implementations of software systems, thus ignoring their runtime behaviors. On the contrary, this has to be considered a crucial factor when quantitative evaluations are required. On the other hand, most of the recent work related to the Actor model only focuses on presenting and discussing rigorous semantics for actors in the general context of modern distributed open systems [Agha et al. 97, Talcott 98, Smith, Talcott 99, Smith, Talcott 02, Thati et al. 04], rather than on the problem of evaluating time costs in order to meet nonfunctional requirements.

Therefore, at the state of the art, the research results achieved in the field of the performance analysis applicable to programs based on high-level, object-oriented programming models cannot be directly exploited to estimate the time costs of actor programs, and motivate the work presented in this paper, which just describes a specific methodology for estimating such costs. The main goals of the proposed methodology are [Blelloch 96, Blumofe et al. 95, Skillicorn 98]:

1. to allow the a priori evaluation of different algorithmic solutions of a given problem;

2. to enable the performance prediction of parallel algorithms by using a few

parameters characterizing the target hardware/software platform;

3. to follow an algorithmic approach, rather than an algebraic or semantic one, to evaluate time costs, so as to better constitute a useful basis on which to implement a software tool able to estimate such costs.

The proposed methodology makes it possible to describe the computations generable during the execution of a program, on the basis of the constraints imposed by both the semantics and the implementation of the programming model. In particular, two sequential phases are distinguishable. In the former, by using a set-based approach, all the computations generable by a program execution are derived and synthesized in a graphical description called C-TREE. As reported above, the realization of the former phase can be greatly simplified if the model is implemented according to specific constraints that reduce the number of the different possible computations. The latter phase focuses on the computations synthesized in the C-TREE, in order to estimate their time costs. To this end, each computation is graphically organized into a TTD (Time Dependencies Diagram), where actors are represented by means of their *lifelines* [Agha 86]. The nodes on the lifelines represent the processing of messages and are labeled with "weights" characterizing their execution times. The weights are calculated by using a variant of the *LogP* model [Culler et al. 96] able to capture the execution characteristics of a parallel abstract machine supporting actor programming, without having to specify unnecessary details about the underlying hardware/software platform.

After briefly describing the Actor programming model which we refer to, the methodology for evaluating time costs, particularly in fifo and causally ordered actor computations, is presented. Finally, a simple example of use of the methodology is shown.

2 The Actor programming model

Actors are objects which manifest a pure reactive nature and interact with other actors solely via message passing. They unify both data and code in local states, called *behaviors*, and are dynamically created and referred through system-wide identifiers, called *mail addresses*.

The communication mechanism is point-to-point, asynchronous and one-directional. Because mail addresses may be transmitted via messages, the actor-net, which shows the potential flow of information, may dynamically change. Messages are guaranteed to be delivered to their destinations with "finite delay", that is, delay is allowed to be arbitrary, but not infinite [Clinger 81]. Incoming messages are buffered into *mail queues* associated to receiving actors, before being serially dequeued and processed.

The processing of a message triggers the execution of the actor *script*, the code in the behavior of the receiver. A script consists of *methods* which specify the actions to be performed when messages are processed. During processing, new actors can be created, messages asynchronously sent and the current behavior substituted by a new one (*replacement behavior*). In practice, replacements implement local state changes, which can span from simple updates in the values of state variables to radical changes of the behavior.

The processing of a message gives rise to a sequential activity associated to the message and the current actor behavior. Therefore, an actor cannot process any other message until the replacement behavior is specified and the running method terminated.

The hypothesis of finite delay in delivering messages to their destinations may cause transmission order between two communicating actors not to be preserved at delivery. Therefore, the insertion order of messages into mail queues and the behaviors used to process them may vary from execution to execution, for the same application and input. As a consequence, the repeated execution of an application on a given input may generate different computations.

More precisely, the following definitions are assumed [Clinger 81, Emrath et al. 88]. An actor application is "determinate" if it always leads to the same result on a given input, "non-determinate" otherwise. In particular, a determinate application is "internally determinate" if it always generates the same computation, "externally determinate" otherwise. Furthermore, a non-determinate application exhibits a "bounded non-determinism" if, on a given input, it always returns a result and the set of possible results is finite. Parallel branch-and-bound programs are classic examples of externally determinate applications, while an application affected by a bounded non-determinism often implements a parallel search for a proof or a counterexample in artificial intelligence domains [Foster 95, Grama, Kumar 99, Kanal, Kumar 88].

3 The C-TREE

In this section we describe what is the C-TREE and how it can be constructed, both in the general case and when the implementation constrains generable computations to be "fifo" or "causally ordered", thus significantly reducing their number.

3.1 Describing actor computations

Let us consider an execution of the actor application $\mathcal{AP}$ on the given input $\mathcal{I}$. We take the view that:

- the significant events in the execution are the processing of messages;

- the computation developed by each actor is the temporal ordered sequence (lifeline) of the events occured to the actor;
- the whole computation is the set of the computations developed by all the actors taking part in the execution.

Moreover, we use the expression *pending event* in order to denote a message sent but not yet processed [Clinger 81].

We also use the following notations:

- Ψ denotes the whole computation;
- if a is an actor, χ_a denotes its computation;
- the pair $[t(b).m, s^j]^i$ denotes the event in which: m is the message, s the sender actor, t the target actor, b the behavior used by t to process m, i the ordinal number specifying the position of the event in the target lifeline, and j the ordinal number specifying the position of the event in the sender lifeline which generates m;
- t^i shortly denotes the event $[t(b).m, s^j]^i$;
- the pair $[t.m, s^j]$ denotes the pending event corresponding to the event $[t(b).m, s^j]^i$, from which it differs for the lack of the behavior and of the position in the target lifeline.

At any given point of the execution, we characterize the computational state by means of the following three sets, which globally contain all the information needed to determine the ultimate courses of the computation:

- Ex, the set of all the (already happened) events;
- Px, the set of all the pending events;
- Bx, the set including, for each existing actor a, the term $a(b)^i$, in which b is the current behavior of a, and the ordinal number i specifies that a has already processed i messages.

We assume that, at the start of the execution, the system creates the *root* actor r, with initial behavior b, and the pending event $[r.\perp, -]$, where $\perp$ is a default initial message [Clinger 81]; the message $\perp$ is provided to r by the system and so it has not a sender. Thus, the initial computational state is $C_{(0)} \equiv < Ex_{(0)}, Bx_{(0)}, Px_{(0)} >$, where $Ex_{(0)} \equiv \emptyset$, $Bx_{(0)} \equiv \{r(b)^0\}$ and $Px_{(0)} \equiv \{[r.\perp, -]\}$.

Starting by this initial state, the computation unfolds by repeatedly applying the following randomized transition rule: a pending event is extracted from Px and *realized*. In detail, if the extracted pending event is $[t.m, s^j]$ and the pertinent term in Bx is $t(b)^i$, then the following actions are performed:

- the event $[t(b).m, s^j]^{i+1}$ is formed and added to Ex;
- the message m is processed by the corresponding method in b.

Processing the message m means determining which programming primitives are invoked during the execution of the method that processes the message. This information can be derived from the *reaction rules* [Talcott 98] that, in the current behavior b of the actor t, determine the response to the message.

During the processing of m:

- new actors are created and the corresponding terms are added to Bx, one for each created actor; if a is a new actor and b is the initial behavior, the new term has the form $a(b)^0$;
- new pending events are added to Px, one for each new message sent by t; in particular, if the message c is sent to the actor a, the new pending event is $[a.c, t^{i+1}]$;
- the new behavior nb of t is evaluated and the term $t(nb)^{i+1}$ substitutes $t(b)^i$ in Bx.

The final state, if it exists, is obtained when the set Px becomes empty. In this case, the finite sequence of all the computational states defines the computation.

3.2 Generating the C-TREE

The algorithm described in [Section 3.1] is a randomized one, in that, at each step, one of the pending events is arbitrarily chosen and realized. Obviously, if we pursue all the possible alternative choices, we are able to generate all the computations which the execution of $\mathcal{AP}$ on the input $\mathcal{I}$ can give rise to.

Initially, since $Px_{(0)}$ has cardinality equal to 1, only one pending event can be realized, thus giving rise to the next computational state, denoted by $C_{(0,1)} \equiv <Ex_{(0,1)}, Bx_{(0,1)}, Px_{(0,1)}>$.

In particular:

- $Ex_{(0,1)}$ includes the first realized event $[r(b).\perp, -]^1$;
- $Bx_{(0,1)}$ includes the term $r(nb)^1$, where nb is the replacement behavior of r, and, for each new actor a with initial behavior b, a term $a(b)^0$;
- $Px_{(0,1)}$ includes, for each message c sent to the target actor t, a pending event $[t.c, r^1]$.

Now, let us suppose that $Px_{(0,1)}$ has cardinality equal to n. By applying a step further our algorithm, the computational state $C_{(0,1)}$ gives rise to n different evolutions, $C_{(0,1,1)}, C_{(0,1,2)}, \ldots, C_{(0,1,n)}$, where each $C_{(0,1,i)} \equiv< Ex_{(0,1,i)}, Bx_{(0,1,i)}, Px_{(0,1,i)} >$ is obtainable by realizing one of the n pending events in $Px_{(0,1)}$.

In general, let:

$$C_{(0,1,g2,g3,\ldots,gn)} \equiv < Ex_{(0,1,g2,g3,\ldots,gn)}, Bx_{(0,1,g2,g3,\ldots,gn)}, Px_{(0,1,g2,g3,\ldots,gn)} >$$

denote one of the computational states generated after n unfolding steps, in which $g2, g3, \ldots, gn$ are natural values. If $Px_{(0,1,g2,g3,\ldots,gn)}$ contains h pending events, h different evolutions are allowed, denoted by:

$$C_{(0,1,g2,g3,\ldots,gn,i)} \equiv< Ex_{(0,1,g2,g3,\ldots,gn,i)}, Bx_{(0,1,g2,g3,\ldots,gn,i)}, Px_{(0,1,g2,g3,\ldots,gn,i)} >, \\ \forall i = 1 \ldots h$$

In particular, $Ex_{(0,1,g2,g3,\ldots,gn,i)}$ is built by realizing the *i-th* pending event in $Px_{(0,1,g2,g3,\ldots,gn)}$. It is worth noting that, since the creation of an actor always precedes the sending of a message to it, at each step, the current set Bx always includes the current behavior of the actor that must realize the selected pending event [Talcott 98].

This procedure makes it possible to derive all the computations generable by running an actor application that, on the given input, is either determinate or affected by a bounded non-determinism. These computations are generated according to only the constraints imposed by the semantics of the Actor model, without any concern for the implementation and the target hardware/software architecture.

The result of the procedure may be graphically synthesized in the C-TREE, a tree whose nodes and edges represent respectively the computational states and state transitions generated by the algorithm. The root of the C-TREE represents the initial computational state $C_{(0)}$, while the leafs represent all the possible final states. Thus, each path of the C-TREE corresponds to a different computation.

Let $C_{(0,1,g2,g3,\ldots,gn)}$ be a leaf of the C-TREE. The sets $Ex_{(0,1,g2,g3,\ldots,gn)}$ and $Bx_{(0,1,g2,g3,\ldots,gn)}$ can be used to express the corresponding computation in the form $\Psi \equiv (\chi_{a1}, \chi_{a2}, \ldots, \chi_{am})$:

- Ψ includes as many subsets χ as the cardinality of $Bx_{(0,1,g2,g3,\ldots,gn)}$, since there exists a set χ_a in Ψ for each term $a(b)^i$ in $Bx_{(0,1,g2,g3,\ldots,gn)}$;
- each set χ_a can be built by including in it all the terms contained in $Ex_{(0,1,g2,g3,\ldots,gn)}$ representing the events realized by a, that is, the events in the form $[a(b).m, as^j]^i$, whatever b, m, as, i, j may be.

3.3 *Equivalent* states

The building of the C-TREE can be simplified by identifying pairs of nodes representing equivalent computational states. For instance, let $C_{(0,1,g2,g3,\ldots,gn)}$ and $C_{(0,1,h2,h3,\ldots,hm)}$ be two nodes of a C-TREE. They are considered *equivalent* if both the conditions hold:

– the sets $Bx_{(0,1,g2,g3,\ldots,gn)}$ and $Bx_{(0,1,h2,h3,\ldots,hm)}$ contain the same actors, each one provided with the same behavior in both the sets;

– the sets $Px_{(0,1,g2,g3,\ldots,gn)}$ and $Px_{(0,1,h2,h3,\ldots,hm)}$ contain the same pending events, compared without considering sending actors.

Two equivalent states unfold identical computations. Therefore, the C-TREE can be further developed by unfolding only one of the two equivalent states, without loss of information.

In order to identify the equivalent states while the C-TREE is generated, a method that exploits the *Cyclic Redundancy Checks* (CRCs) [Press et al. 92] can be employed. Such a method is based upon maintaining all the sets Bx and Px associated to the nodes of a C-TREE as ordered data structures. This way, each set, regarded as an ordered aggregate of data (whether numbers, records, lines or whole files), can be tagged with a short, constant-length, statistically unique "key": its CRC. As a consequence, the sets Bx or Px[1] associated to the nodes of a C-TREE can be compared for identity by comparing only their short CRC keys: differing keys imply nonidentical sets, while identical keys imply, to high statistical certainty, identical sets. Therefore, any two nodes of the C-TREE can be considered equivalent to high statistical certainty if their pairs of CRCs, each calculated respectively for the sets Bx and Px, result to be equal. However, if the very small probability of being wrong is not allowed to be tolerated, a full comparison of the sets only when their keys are identical has to be made. It is worth noting that such a method can be used only if the ordered structure of the sets Bx and Px is preserved whenever they are updated while the C-TREE is generated.

Under these assumptions, whenever a new node is added to a C-TREE, the pair containing the CRCs of Bx and Px can be computed. Such a pair gives synthetic information about computations that can unfold from the new node of the C-TREE. Therefore, the new pair should be compared to all the other pairs associated to the already generated nodes of the C-TREE, in order to detect the computational states which the new state results equivalent to.

By using CRCs to identify equivalent computational states, it is possible to avoid burdensome comparisons among complex data structures. Moreover, it

[1] Fot this set, the CRC is calculated taking into account the pending events without considering sending actors.

is also possible to drastically reduce the number of comparisons to be made by employing a data structure in which all the nodes of a C-TREE characterized by the same pair of CRCs can be grouped and directly referred to. To this end, it is possible to use a matrix[2] whose generic element (i,j) refers to all the nodes in the C-TREE whose sets Bx and Px are identified by CRCs just equal respectively to the values i and j. This way, any newly generated node characterized by the CRC values a and b has to be compared only to the nodes included in the list referred by the entry (a,b) of the matrix.

3.4 The *causality* relation

Let $C_{(0,1,g2,g3,\ldots,gn)}$ be a leaf of the C-TREE and $\Psi \equiv (\chi_{a1}, \chi_{a2}, \ldots, \chi_{am})$ the associated computation. We know that, for each actor a, all the events belonging to χ_a in Ψ are totally ordered by their occurrence numbers. This order, denoted by $\prec_a$, implies a *causal* ordering on the events of χ_a, in the sense that an earlier event may affect a later event in the a computation [Charron-Bost et al. 96, Lamport 78]. $\prec_a$ is an irreflexive total order in χ_a and defines the so called *arrival* order at the actor a [Baker, Hewitt 77, Clinger 81]:

$$a^n \prec_a a^m \iff n < m, \forall a$$

Communications exchanged among actors are the sole interaction form allowed by the programming model. They relate the events in which messages are sent with the events in which the messages are processed. These relations define the *activation* order [Baker, Hewitt 77, Clinger 81]. Therefore, the causal order relation can be extended to the events in Ψ according to the following definition: the *causality* relation $\prec$ in Ψ is the smallest relation that satisfies the following three properties:

1. if $a^n \prec_a a^m$, then $a^n \prec a^m$;
2. if $[a(b).m, as^j]^i \in \chi_a \subseteq \Psi$, then $as^j \prec a^i$;
3. if $a1^i \prec a2^j$ and $a2^j \prec a3^k$, then $a1^i \prec a3^k$.

This relation defines a partial order on the events of the computation Ψ and is based on the consideration that message sending always precedes message processing. In particular, the relation $\prec$ represents the *combined* order in Ψ [Baker, Hewitt 77, Clinger 81].

The events in Ψ may happen according to any order compatible with the one defined by the relation $\prec$. This means that an event may happen only if all the

[2] The matrix can be implemented as a linked data structure or in any other, more effective technique if the "generator polynomial" is of degree 16 or higher and there may exist limits on run-time memory usage.

events that precede it according to the relation $\prec$ have already happened. If two events are not ordered by the relation $\prec$, they are said "concurrent", since one event may happen independently of the other, that is, neither can causally affect the other [Lamport 78]. It is worth noting that the axiom of "strong realizability" assures that there exists a one-one mapping from the set of events in Ψ to the set of natural numbers. This mapping gives a total order on Ψ that is compatible with the *combined* order, i.e. with the relation $\prec$ [Clinger 81].

3.5 $Fifo$ computations

In a fifo computation any two messages exchanged between the same two actors are received preserving the sending order. To formalize this assumption, a new notation is introduced: in order to differentiate among pending events generated during the execution of the same method, the notation $[t.m, s^{i,j}]$ is used, in which j indicates that m is the *j-th* message sent during the *i-th* event realized by the actor s. Consequently, since method execution is a sequential activity, the dispatch of the *n-th* message precedes the dispatch of the *m-th* message if and only if $n < m$. This way, the sequential order existing among messages sent during the execution of a method can be evidenced.

Let us consider two pending events $[t.m1, s^{i,n}]$ and $[t.m2, s^{j,m}]$ tied to messages sent by the same actor s to the same target t. In a fifo computation, if $i<j$ or if $i=j$ and $n<m$ then $[t.m1, s^{i,n}] \prec [t.m2, s^{j,m}]$, whatever t, s, i, j, n, m may be.

In the following, we show how to exploit the condition reported above in order to only generate fifo computations, so as to simplify the construction of the C-TREE.

Let us consider the set $Px_{(0,1,g2,\ldots,gn)}$ and all its partitions $px(t,s)$, each one including all the pending events $[t.m, s^{i,n}]$ tied to a given pair of target and sending actors t and s. If only fifo computations have to be generated, whatever two events $[t.m1, s^{i,n}]$ and $[t.m2, s^{j,m}]$ may be in $px(t,s)$, then $[t.m1, s^{i,n}]$ must be realized before $[t.m2, s^{j,m}]$ if $i < j$ or if $i = j$ and $n < m$. Consequently, it is always possible to state the precedence between any two events $[t.m1, s^{i,n}]$ and $[t.m2, s^{j,m}]$ in $px(t,s)$ by examining the pairs (i,n) and (j,m). This induces, according to the relation $\prec$, a total order in $px(t,s)$ and makes it possible to identify in each partition the *minimal* event $min^{fifo}_{px(t,s)}$, defined as follows:

$$[t.m, s^{i,n}] \in px(t,s) \;\text{ is }\; min^{fifo}_{px(t,s)} \Longleftrightarrow (i<j)\,\text{or}\,(i=j\,\text{and}\,n<m),\\ \forall\,[t.c, s^{j,m}] \in px(t,s)$$

The minimal events are the unique events in $Px_{(0,1,g2,\ldots,gn)}$ that are allowed to be realized, if only fifo computations have to be generated. Therefore, the procedure used to generate the C-TREE must be modified as follows. Let

$C_{(0,1,g2,\ldots,gn)}$ be a computation and n_p the cardinality of $Px_{(0,1,g2,\ldots,gn)}$. At each step, the randomized selection of the pending event to be realized has to be restricted to the only minimal events in $Px_{(0,1,g2,\ldots,gn)}$. This means that if their number is n_{fifo}, only n_{fifo} possible evolutions of the computation are allowed, with $n_{fifo} \leq n_p$.

3.6 *Causally ordered* computations

The fifo condition can be strengthened by requiring that all the messages sent to the same actor are received in an order consistent with the causal order of the corresponding send events. This property qualifies the causally ordered computations and can be modelled by the following condition. Let us consider two pending events $[t.m1, s1^i]$ and $[t.m2, s2^j]$ tied to messages received by the same target actor t; in a causally ordered computation, if $s1^i \prec s2^j$ then $[t.m1, s1^i] \prec [t.m2, s2^j]$, whatever t, $s1$, $s2$, i, j may be.

Let us consider the set $Px_{(0,1,g2,\ldots,gn)}$ and all its partitions $px(t)$, each one including all the pending events $[t.m, s^i]$ tied to the messages received by a same target actor t. If only causally ordered computations have to be generated, whatever two events $[t.m1, s1^i]$ and $[t.m2, s2^j]$ may be in $px(t)$, then $[t.m1, s1^i]$ must be realized before $[t.m2, s2^j]$, if $s1^i \prec s2^j$. On the contrary, if $s1^i$ and $s2^j$ are concurrent events, then $[t.m1, s1^i]$ and $[t.m2, s2^j]$ are said "independent" and can be realized in any order. This induces, according to the relation $\prec$, a partial order in $px(t)$ and makes it possible to identify in each partition the *minimal* events, each one denoted by $min^{causal}_{px(t)}$ and defined as follows:

$$[t.m, s^i] \in px(t) \text{ is } min^{causal}_{px(t)} \Longleftrightarrow \nexists\, [t.c, a^j] \in px(t) : a^j \prec s^i, \forall a, i, j$$

Obviously, in each partition there may be more than one minimal event. The minimal events are the unique events in $Px_{(0,1,g2,\ldots,gn)}$ that are allowed to be realized, if only causally ordered computations have to be generated. Therefore, the procedure used to generate the C-TREE must be modified as follows. Let $C_{(0,1,g2,\ldots,gn)}$ be a computation and n_p the cardinality of $Px_{(0,1,g2,\ldots,gn)}$. At each step, the randomized selection of the pending event to be realized has to be restricted to the only minimal events in $Px_{(0,1,g2,\ldots,gn)}$. This means that if their number is n_{causal}, only n_{causal} possible evolutions of the computation are allowed, with $n_{causal} \leq n_p$.

4 The evaluation of time costs

Let us consider an actor computation in the form $\Psi \equiv (\chi_{a1}, \chi_{a2}, \ldots, \chi_{am})$. It can be graphically visualised by using a *time dependencies diagram* (TDD). Each actor is depicted by means of its lifeline, a directed line on which the events are located as nodes. On a lifeline, an event a^i precedes the event a^j, according to the line direction, if $i < j$.

Actor creations are depicted as arrows connecting the events in which the creation primitives are invoked with the origins of the created actor lifelines. Messages are depicted as arrows connecting sending events with the events processing them. Each event may have several outgoing arrows, but only one incoming arrow.

Causal dependencies among events are expressed by the *paths* existing among them. In particular, paths follow arrow directions and run on actor lifelines according to their directions. Events can be realized in any order provided that all the causal dependencies have been respected. Therefore, an event is allowed to be realized only if it is "ready", that is, all the events that precede it according to the relation $\prec$ have been already realised, or, in other words, if all the causal dependencies affecting it have been satisfied.

Each node on a TDD must be labelled with a *weight* that specifies the execution time of the method associated to the event depicted by the node. Such a time can be estimated as the sum of two main contributions: the *kernel time* and the *local time*. The former is the sum of the execution times of the Actor programming primitives invoked during the method execution. The latter is the execution time of all the other instructions in the method. The local time depends on the hardware characteristics of the target processor and can be estimated by means of direct measures on the application code. The kernel time also depends on the characteristics of the underlying communication system. Therefore, to make cost evaluation independent of low level details of the target hardware/software platform, a variant of the $LogP$ model is used to capture the behavior of an actor abstract parallel machine and to estimate the time costs of the methods executed during a computation. The variant is based on the following parameters, among which the first two have to be specifically evaluated for each programming primitive requiring a communication, such as the *send* and *new* primitives:

- o_s: the *sending overhead*, defined as the length of time that a processor is engaged in sending a message ($o_{s(send)}$) or requesting an actor creation ($o_{s(new)}$); during this time, the processor cannot perform any other operation.
- o_r: the *receiving overhead*, defined as the length of time that a processor is engaged in receiving a message ($o_{r(send)}$) or a creation request ($o_{r(new)}$) and in performing the corresponding actions, such as starting up a method execution or creating an actor.
- L: an upper bound on the *latency*, or delay, incurred in communicating a message or a creation request.

– g: the *gap*, defined as the minimum time interval between consecutive invocations of send or new primitives.

– P: the number of processor/memory modules.

It is assumed a unit time for a local operation. This unit is called *cycle* and the parameters g, L, $o_{s(...)}$ and $o_{r(...)}$ are all measured as multiples of the cycle. Moreover, in evaluating the cost of a method, it has to be considered that: (1) $o_{s(...)}$ is to be attributed to the method invoking the primitive; (2) $o_{r(...)}$ is to be attributed according to the following considerations: $o_{r(send)}$ is to be attributed to the method that processes the message, while $o_{r(new)}$ is to be considered a start-up cost to be associated to the created actor. Thus, a node representing the start-up cost must be depicted at the beginning of each actor lifeline. Finally, it is worth noting that the replacement behavior primitive does not involve any communication. As a consequence, it is characterised by an overhead $o_{(beh)}$ to be attributed to the method invoking the primitive.

After having assigned weights to all the nodes in a TDD, it is possible to calculate the *Work* and *Depth* [Blelloch 96] associated to the computation represented by the TDD. In particular, *Work* is defined as the total time cost of a computation, and *Depth* is defined as the the total time cost associated to the longest chain of sequential dependencies in a computation. Consequently, *Work* can be calculated by summing the weights of all the methods in a TDD, and *Depth* can be determined by identifying the maximum value among the sums of the weights associated to the different paths in a TDD. *Work* and *Depth* can be considered as the running times of an application at two limits: when the application is executed on a one-processor machine (*Work*) or on an ideal machine with an unlimited number of processors (*Depth*). In fact, *Work* and *Depth* are often referred to as T_1 and T_∞ [Blumofe et al. 95].

Let us consider the execution of an application on a given input. All the generable computations are described by the leaves of the C-TREE. For each leaf, the associated computation can be described in the form Ψ and then graphically visualised by a TDD. *Work* and *Depth* can be calculated for each TDD by applying the variant of *LogP*. Then, the computations can be compared on the basis of these two measures, parametric with respect to the hardware/software characteristics of the target computing system.

5 An example

This section presents an example that shows how the methodology described can be used to calculate the time cost of a simple Actor program, whose pseudo code is shown in [Figure 1]. The program exploits a "divide-and-conquer" strategy to apply in parallel the non-commutative *module* operator to all the integers in

```
behaviour root ( )
  method start accepts start ( )
    var from, to: int
    read from, to
    send new node(), range(self, from, to)
    become joinCont(self, 0, true, true)
  end start
end root

behaviour node ( )
  method range accepts range ( cont: maddr; from, to: int )
    var diff, mid: int
    diff := to−from
    if diff = 0 then
      send cont, join(from)
      dispose self
    else if diff = 1 then
      send cont, join(to+from)
      dispose self
    else
      mid := (from+to)/2
      send new node(), range(self, from, mid)
      send new node(), range(self, mid+1, to)
      become joinCont(cont, 0, false, false)
    end if
  end range
end node

behaviour joinCont ( cont: maddr; result: int; isRoot, isValue: boolean )
  method join accepts join ( value: int )
    if isValue then
      if isRoot then
        write result
      else
        send cont, join(value mod result)
      end if
      dispose self
    else
      result := value
      isValue := true
      become theBehaviour
    end if
  end join
end joinCont
```

Figure 1: A simple "divide-and-conquer" Actor program.

the range from `from` to `to`. The computation is characterized by a binary tree structure, where: (1) each leaf actor adds the numbers in the received range and passes on the sum to its parent; (2) each internal actor splits the received range into two, passes on them to two new created actors, receives back the two sums, combines them by applying the *module* operator, and passes on the result to its parent.

Since the applied operator is non-commutative, the simple program exhibits a bounded non-determinism. As a consequence, different computations and outputs can be produced. In particular, if the program is executed on the range from `1` to `4`, the methodology (see [Figure 2]) identifies two possible computations: the former produces the output `3`, while the latter produces the output `1`.

Legend
behaviours: rt (root), nd (node), jc (joinCont)
messages: st (start), rg (range), jn (join)
constants: t (true), f (false)

$Ex_{(0)} \equiv \emptyset,\ Bx_{(0)} \equiv \{r(rt())^0\},\ Px_{(0)} \equiv \{[r.st(), -]\}$

$Ex_{(0,1)} \equiv \{[r(rt()).st(), -]^1\},\ Bx_{(0,1)} \equiv \{r(jc(r,0,t,t))^1, a1(nd())^0\},\ Px_{(0,1)} \equiv \{[a1.rg(r,1,4), r^1]\}$

$Ex_{(0,1,1)} \equiv Ex_{(0,1)} \cup \{[a1(nd()).rg(r,1,4), r^1]^1\},$
$Bx_{(0,1,1)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,0,f,f))^1, a11(nd())^0, a12(nd())^0\},$
$Px_{(0,1,1)} \equiv \{[a11.rg(a1,1,2), a1^1], [a12.rg(a1,3,4), a1^1]\}$

$Ex_{(0,1,1,1)} \equiv Ex_{(0,1,1)} \cup \{[a11(nd()).rg(a1,1,2), a1^1]^1\},$
$Bx_{(0,1,1,1)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,0,f,f))^1, a12(nd())^0\},$
$Px_{(0,1,1,1)} \equiv \{[a1.jn(3), a11^1], [a12.rg(a1,3,4), a1^1]\}$

$Ex_{(0,1,1,2)} \equiv Ex_{(0,1,1)} \cup \{[a12(nd()).rg(a1,3,4), a1^1]^1\},$
$Bx_{(0,1,1,2)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,0,f,f))^1, a11(nd())^0\},$
$Px_{(0,1,1,2)} \equiv \{[a1.jn(7), a12^1], [a11.rg(a1,1,2), a1^1]\}$

$Ex_{(0,1,1,1,1)} \equiv Ex_{(0,1,1,1)} \cup \{[a12(nd()).rg(a1,3,4), a1^1]^1\},$
$Bx_{(0,1,1,1,1)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,0,f,f))^1\},$
$Px_{(0,1,1,1,1)} \equiv \{[a1.jn(3), a11^1], [a1.jn(7), a12^1]\}$

$Ex_{(0,1,1,1,2)} \equiv Ex_{(0,1,1,1)} \cup \{[a1(jc(r,0,f,f)).jn(3), a11^1]^2\},$
$Bx_{(0,1,1,1,2)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,3,f,t))^2, a12(nd())^0\},$
$Px_{(0,1,1,1,2)} \equiv \{[a12.rg(a1,3,4), a1^1]\}$

$Ex_{(0,1,1,2,1)} \equiv Ex_{(0,1,1,2)} \cup \{[a11(nd()).rg(a1,1,2), a1^1]^1\},$
$Bx_{(0,1,1,2,1)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,0,f,f))^1\},$
$Px_{(0,1,1,2,1)} \equiv \{[a1.jn(3), a11^1], [a1.jn(7), a12^1]\}$

$Ex_{(0,1,1,2,2)} \equiv Ex_{(0,1,1,2)} \cup \{[a1(jc(r,0,f,f)).jn(7), a12^1]^2\},$
$Bx_{(0,1,1,2,2)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,7,f,t))^2, a11(nd())^0\},$
$Px_{(0,1,1,2,2)} \equiv \{[a11.rg(a1,1,2), a1^1]\}$

$Ex_{(0,1,1,1,1,1)} \equiv Ex_{(0,1,1,1,1)} \cup \{[a1(jc(r,0,f,f)).jn(3), a11^1]^2\},$
$Bx_{(0,1,1,1,1,1)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,3,f,t))^2\}, \quad Px_{(0,1,1,1,1,1)} \equiv \{[a1.jn(7), a12^1]\}$

$Ex_{(0,1,1,1,1,2)} \equiv Ex_{(0,1,1,1,1)} \cup \{[a1(jc(r,0,f,f)).jn(7), a12^1]^2\},$
$Bx_{(0,1,1,1,1,2)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,7,f,t))^2\}, \quad Px_{(0,1,1,1,1,2)} \equiv \{[a1.jn(3), a11^1]\}$

$Ex_{(0,1,1,1,2,1)} \equiv Ex_{(0,1,1,1,2)} \cup \{[a12(nd()).rg(a1,3,4), a1^1]^1\},$
$Bx_{(0,1,1,1,2,1)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,3,f,t))^2\}, \quad Px_{(0,1,1,1,2,1)} \equiv \{[a1.jn(7), a12^1]\}$

$Ex_{(0,1,1,2,2,1)} \equiv Ex_{(0,1,1,2,2)} \cup \{[a11(nd()).rg(a1,1,2), a1^1]^1\},$
$Bx_{(0,1,1,2,2,1)} \equiv \{r(jc(r,0,t,t))^1, a1(jc(r,7,f,t))^2\}, \quad Px_{(0,1,1,2,2,1)} \equiv \{[a1.jn(3), a11^1]\}$

$Ex_{(0,1,1,1,1,1,1)} \equiv Ex_{(0,1,1,1,1,1)} \cup \{[a1(jc(r,3,f,t)).jn(7), a12^1]^3\},$
$Bx_{(0,1,1,1,1,1,1)} \equiv \{r(jc(r,0,t,t))^1\}, \quad Px_{(0,1,1,1,1,1,1)} \equiv \{[r.jn(1), a1^3]\}$

$Ex_{(0,1,1,1,1,2,1)} \equiv Ex_{(0,1,1,1,1,2)} \cup \{[a1(jc(r,7,f,t)).jn(3), a11^1]^3\},$
$Bx_{(0,1,1,1,1,2,1)} \equiv \{r(jc(r,0,t,t))^1\}, \quad Px_{(0,1,1,1,1,2,1)} \equiv \{[r.jn(3), a1^3]\}$

$Ex_{(0,1,1,1,1,1,1,1)} \equiv Ex_{(0,1,1,1,1,1,1)} \cup \{[r(jc(r,0,t,t)).jn(1), a1^3]^2\},$
$Bx_{(0,1,1,1,1,1,1,1)} \equiv \emptyset, \quad Px_{(0,1,1,1,1,1,1,1)} \equiv \emptyset,$ Write 1

$Ex_{(0,1,1,1,1,2,1,1)} \equiv Ex_{(0,1,1,1,1,2,1)} \cup \{[r(jc(r,0,t,t)).jn(3), a1^3]^2\},$
$Bx_{(0,1,1,1,1,2,1,1)} \equiv \emptyset, \quad Px_{(0,1,1,1,1,2,1,1)} \equiv \emptyset,$ Write 3

Figure 2: The computational states generated by the program in [Figure 1].

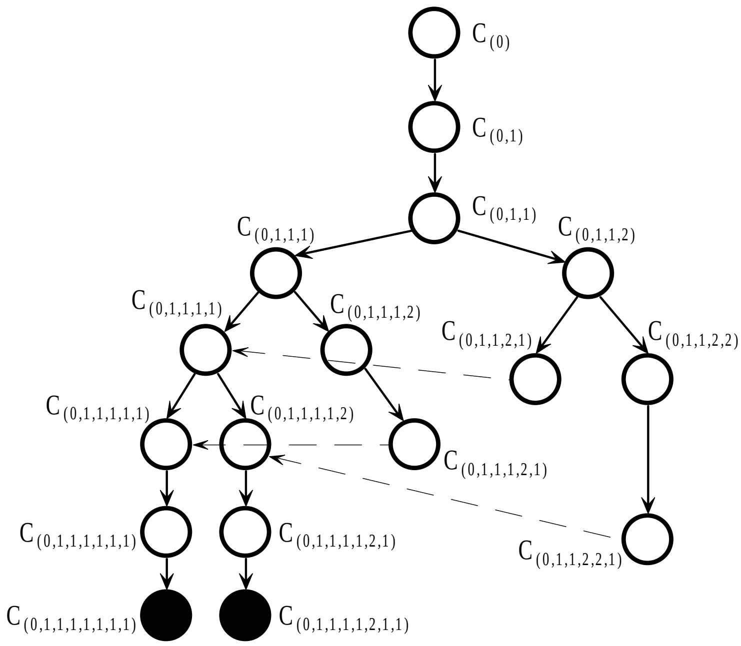

Figure 3: C-TREE representing the computational states shown in [Figure 2].

Consequently, the corresponding C-TREE shown in [Figure 3] has two leaves: each one is represented by a filled circle and is associated to a different TDD.

For the sake of brevity, each possible intermediate state of the computation in [Figure 2] is indicated only by the sets Ex, Bx and Px. Moreover, some intermediate states result to be *equivalent* according to the definition given in [Section 3.2]. Therefore, they give rise to identical unfolded computations. This is represented in [Figure 3] by dotted lines connecting the following equivalent states: $C_{(0,1,1,1,1)}$ and $C_{(0,1,1,2,1)}$, $C_{(0,1,1,1,1,1)}$ and $C_{(0,1,1,1,2,1)}$, $C_{(0,1,1,1,1,2)}$ and $C_{(0,1,1,2,2,1)}$. Therefore, only the computations originating from $C_{(0,1,1,1,1)}$, $C_{(0,1,1,1,1,1)}$ and $C_{(0,1,1,1,1,2)}$ are explored.

Finally, in the code of [Figure 1] the primitive *dispose* is used to deallocate actors. Consequently, when this primitive is invoked, the deallocated actor is removed from the corresponding set Bx. This means that, to identify all the actors that take part in the computation described by a leaf of the C-TREE, the evolution of the set Bx along the path associated to the leaf has to be considered.

Focusing only on the computation $C_{(0,1,1,1,1,1,1,1)}$, the associated TDD is shown in [Figure 4]. In particular, each vertical line is a lifeline representing an actor. The filled circles on these lines represent actor creation overheads, while the remaining circles represent events, i.e. method executions. The arrows show

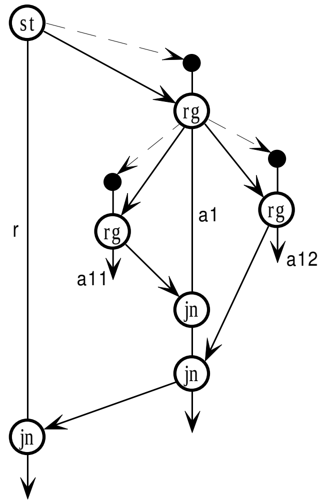

Figure 4: The TDD associated to the computation $C_{(0,1,1,1,1,1,1,1)}$.

the *send* primitives, while the dotted arrows show the *new* primitives.

In [Figure 5] the weighted version of the TDD shown in [Figure 4] is presented in order to calculate the *Work* and *Depth*.

As for the *Work*, it results:

$$Work = \sum_{i=1}^{10} W_i$$

in which W_i is the weight, i.e. the execution time, associated to the *i-th* circle. In particular, the weights associated to filled circles are always equal to the receiving overheads of actors creation, while the weights relative to the other circles are to be expressly calculated on the basis of what referred to in [Section 4]. In [Figure 6] the expressions for the terms W_i are reported. The terms with label *local* represent the *local time* contributions to node weights, while the sums of all remaining contributions represent the *kernel time* tied to actor primitives execution. Among these contributions there is also the overhead $o_{(dispose)}$ associated to the *dispose* primitive.

To calculate the *Depth*, a weight has to be associated to each edge of the TDD in [Figure 5], according to the following considerations. For an edge belonging to a lifeline, two conditions are possible: (1) if the edge originates from a node representing an actor creation overhead, its weight is the sum of the weight of the node originating the edge and the weight of the incoming edge of the node; (2) if the edge originates from a node representing a method execution, its weight

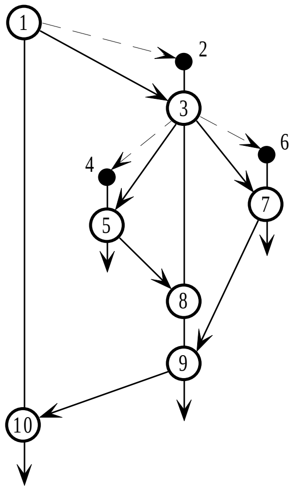

Figure 5: The weighted version of the TDD shown in [Figure 4].

$W_1 = o_{r(send)} + o_{s(new)} + o_{s(send)} + o_{(beh)} + t_{1(local)}$
$W_2 \equiv W_4 \equiv W_6 = o_{r(new)}$
$W_3 = o_{r(send)} + 2 \cdot (o_{s(new)} + o_{s(send)}) + o_{(beh)} + t_{3(local)}$
$W_5 \equiv W_7 = o_{r(send)} + o_{s(send)} + o_{(dispose)} + t_{5(local)}$
$W_8 = o_{r(send)} + o_{(beh)} + t_{8(local)}$
$W_9 = o_{r(send)} + o_{s(send)} + o_{(dispose)} + t_{9(local)}$
$W_{10} = o_{r(send)} + o_{(dispose)} + t_{10(local)}$

Figure 6: The weights assigned to the nodes of the TDD shown in [Figure 5].

is the sum of the weight of the node originating the edge and the maximum between the weights of the incoming edges of the node. If the edge represents a *send* primitive or a communication tied to the execution of a *new* primitive, its weight is the sum of three contributions: the weight of the node issuing the communication, the maximum between the weights of the two incoming edges of the node, and the latency L.

In [Figure 7] all the weights associated to the edges of the TDD shown in [Figure 5] are reported. They are calculated on the basis of the relations reported in [Figure 6]. In particular, the *Depth* is associated to the path identified by the nodes `1`, `2`, `3`, `4`, `5`, `8`, `9`, `10`.

6 Related work

Most of the work developed in the context of the performance analysis of object-oriented programs implemented on given computing platforms is based on the

$$
\begin{aligned}
d_{1,2} &\equiv d_{1,3} = W_1 + L; \quad d_{1,10} = W_1 \\
d_{2,3} &= d_{1,2} + W_2 = W_1 + W_2 + L \\
d_{3,4} &= W_3 + \max\{d_{1,3}, d_{2,3}\} + L = W_3 + d_{2,3} + L = W_1 + W_2 + W_3 + 2L \\
d_{3,5} &\equiv d_{3,7} \equiv d_{3,6} \equiv d_{3,4} \\
d_{3,8} &= W_3 + \max\{d_{1,3}, d_{2,3}\} = W_3 + d_{2,3} = W_1 + W_2 + W_3 + L \\
d_{4,5} &\equiv d_{6,7} = d_{3,4} + W_4 = W_1 + W_2 + W_3 + W_4 + 2L \\
d_{5,8} &= W_5 + \max\{d_{4,5}, d_{3,5}\} + L = W_5 + d_{4,5} + L = W_1 + W_2 + W_3 + W_4 + W_5 + 3L \\
d_{7,9} &= W_7 + \max\{d_{3,7}, d_{6,7}\} + L = W_7 + d_{6,7} + L = W_1 + W_2 + W_3 + W_4 + W_7 + 3L \\
d_{8,9} &= W_8 + \max\{d_{5,8}, d_{3,8}\} = W_8 + d_{5,8} = W_1 + W_2 + W_3 + W_4 + W_5 + W_8 + 3L \\
d_{9,10} &= W_9 + \max\{d_{8,9}, d_{7,9}\} + L = W_9 + d_{8,9} + L = \\
&= W_1 + W_2 + W_3 + W_4 + W_5 + W_8 + W_9 + 4L \\
Depth &= W_{10} + \max\{d_{1,10}, d_{9,10}\} = W_{10} + d_{9,10} = \\
&= W_1 + W_2 + W_3 + W_4 + W_5 + W_8 + W_9 + W_{10} + 4L
\end{aligned}
$$

Figure 7: The weights assigned to the edges of the TDD shown in [Figure 5].

use of a wide variety of notations for the specification of concurrent/distributed systems. In fact, such notations usually try to capture and abstractly describe the basic aspects concerning both the software execution model and the system execution model. The former derives from the semantics of the adopted programming model, while the latter takes into account the implementation of the software execution model and the main characteristics of the hardware/software computing platform on which the software runs. However, even though these notations and their numerous variants have achieved widespread usage as specification tools for concurrent/distributed systems, such as actor systems, they have been rarely exploited to estimate time costs of actor programs, since they have some weaknesses with respect to this specific goal.

Petri nets [Peterson 81] are used to mathematically represent discrete distributed systems. Because of their ability to express concurrent events, they generalize automata theory. Consequently, Petri nets, and their successor Statecharts [Harel 87], can essentially describe actor systems as finite automata, that is, as systems which have a strong state-based behavior. However, the primary weakness of finite automata is that an actor system may not have a meaningful global state, while the properties concerning time costs of actor programs are often more naturally expressed in terms of events and relations on events.

The π-calculus [Honda, Tokoro 91, Milner et al. 92] is a form of process algebra able to model systems of autonomous agents which interact with each other in ways that they are free to select spontaneously. Although many approaches based on π-calculus have been proposed in order to describe and analyze both functional and performance properties of programs, this form of process algebra does not generally address the issue of the so-called “fairness”, which, on the contrary, appears to be very important in actor systems in that it determines much of the complexity in estimating time costs.

In [Talcott 98], two semantic models are studied: open event diagrams (OEDs) and interaction diagrams (IDs). OEDs generalize the event diagrams formalized in [Clinger 81] to the open system settings, by making the interactions with the execution environment explicit. Event diagrams express the view that what is important about actor computations are the messages sent, the actors created in response to a message delivery, and the "precedes" ordering among these events. IDs are the result of hiding the internal events of complete event diagrams. What remains are the arrivals of messages from the environment (input events), the deliveries of messages to the environment (output events), and the ordering relation on input and output events obtained by restricting the precedes relation. OEDs can model stages of an actor system activity and complete computations in which all the messages have been delivered, while IDs are abstractions of complete computations providing a way of forgetting non-relevant ordering information. However, the two works [Agha et al. 97, Talcott 98], while giving a mathematical and semantic characterization of OEDs and IDs, need to postulate the existence of both arrival and input/output orderings satisfying the specific constraints imposed by applications. On the contrary, this existence has not to be postulated when the main goal is to analyze the execution behavior of an actor application in order to estimate its time cost. In fact, to this end, the "combined orders" generable by repeatedly running an application on a given input are what has to be determined. They cannot be assumed as given, in that it is only after the determination of all the possible combined orders that the time cost of an application can be estimated.

Specification Diagrams (SDs) [Smith, Talcott 99, Smith, Talcott 02, Thati et al. 04] are mainly targeted to actor systems. They combine a great expressivity with formal underpinnings, and are characterized by two forms of notations: the former is graphical and highly intuitive, and is intended for use in practice; the latter is textual and is perhaps easier to manipulate for mathematical study. Therefore, the latter can be used for formal reasoning about actor systems, while the former is more suitable to describe the dynamics of actor programs. However, the graphical form of SDs expresses abstract causal ordering on computation events in terms of "causal threads", which are entities that exist only at semantic level. Consequently, there is no necessary connection between threads and actors, since a single thread of causality may involve multiple actors, and a single actor may appear to have multiple threads of causality. This ends up making the use the SDs very difficult when quantitative evaluations concerning actor systems on given hardware/software computing platforms are required.

The methodology proposed in this paper makes use of diagrams, the TDDs, that can be related to other forms of message passing diagrams, such as the UML Sequence Diagrams [Rumbaugh et al. 98], in which vertical lines represent processes/threads, while horizontal lines represent exchanged messages. To this end, it is worth noting that the UML formalism is widely used to describe behavioral characteristics of programs. In particular, in [Cortellessa et al. 01] the UML diagrams of a program can be enriched with performance annotations concerning software architecture, workload distribution, parameters of hardware devices, and the mapping of the software modules to the target computing platform, in order to produce a global precedence graph which identifies the execution flow and the interconnections among the system software components. This graph is then used to generate a Layered Queueing Network (LQN) [Trivedi 01] based model that represents the whole system platform, including hardware and software components, and that can be exploited to get performance indices, such as the program response time. Furthermore, in [Petriu, Wang 99] the UML framework is used to describe programs whose structure can be expressed by means of architectural patterns, such as pipe and filters, master/slave, broker, client/server, etc. The UML descriptions are exploited to build LQN based models of the programs by applying graph transformation techniques, automatically performed by a general-purpose graph rewriting tool. The LQN based models are then used to obtain performance indices of the programs. However, although the most recent versions of the UML framework have been largely extended, they do not still allow asynchronous "fair" messaging to be expressed. Therefore, differently from the proposed methodology, they cannot easily capture and describe actor behaviors, and remain primarily designed to show possible scenarios of execution, and not to give all the possible scenarios.

In [Andolfi et al. 00] the dynamic behavior of programs is described by Queueing Network [Kant 92] based models automatically generated from software architecture specifications obtained by means of Message Sequence Charts (MSCs). In fact, the approach analyzes the program dynamic specification in terms of the execution traces (the sequences of events exchanged between components) it defines. The analysis generates performance indices characterizing the programs. However, MSCs are affected by the same drawbacks of the UML approaches. Thus, differently from the proposed methodology, they are able to describe only some possible scenarios of execution.

Different approaches to the study of time costs are followed in [Blelloch 96] and [Blumofe et al. 95, Frigo et al. 98]. In particular, in [Blelloch 96] NESL is presented, a parallel programming language that provides a language-based performance model. The model gives a formal way to calculate the Work and Depth of a program. In particular, work and depth costs are assigned to each primitive instruction of the language and rules are specified for combining both parallel and sequential expressions. This way, it is possible to determine how the running time grows as a function of the input size. However, the developed analysis cannot be exploited to estimate the running time of an application on actual hardware/software platforms, because the costs assigned to primitives are evaluated without properly taking into account communication latency and throughput. In fact, the performance model associated to NESL is not able to significantly capture the behavior of hardware/software platforms used to run programs. Moreover, NESL is not an object-based language and only supports data-parallelism.

Cilk [Blumofe et al. 95, Frigo et al. 98] is a language for general-purpose, multithreaded parallel programming based on ANSI C. It is especially effective for exploiting dynamic, highly asynchronous parallelism, difficult to write in data-parallel or message-passing style. Unlike many other parallel programming systems, Cilk is *algorithmic* [Blumofe et al. 95], in that its runtime system employs a scheduler that allows the time costs of programs to be estimated accurately on the basis of abstract measures of complexity. In particular, the scheduling model based on *work stealing* has led to the development of a performance model that predicts the efficiency of a Cilk program by using two simple parameters: work and critical-path length. As a consequence, time costs of Cilk applications can be evaluated only under specific implementation hypotheses and this strongly limits the validity of the performance model.

7 Conclusions

This paper describes a methodology for estimating the time costs of Actor applications. The methodology takes properly into account the constraints imposed by both the semantics of the programming model and its implementation on a given hardware/software platform. Our approach is particularized to the implementations of the Actor model generating only fifo or causally ordered computations, but can be further specialized to capture even more specific implementation constraints, thus dramatically reducing the number of computations actually generated and making the study of time costs practicable.

The methodology exploits a set-based approach to generate the C-TREE, in which all the computations satisfying the specific constraints imposed by the implementation are synthesized, and a variant of the $LogP$ model, in which few parameters are able to capture the execution behavior of a parallel abstract machine supporting Actor programming. So, it is possible to express the execution times of methods and calculate the *Work* and *Depth* of computations. This enables to study time costs of Actor applications without having to consider unnecessary and low-level details about the computing system used.

The methodology can be also exploited to specialize the implementation of the Actor model on a given hardware/software platform, in order to force a specific computation unfolding. In fact, as shown for fifo and causally ordered computations, by applying the methodology, it is possible to find the constraints able to determine the required execution behavior. These constraints can then be used to devise the scheduling algorithms or low-level communication protocols to be adopted in order to obtain an implementation of the Actor model able to guarantee the expected execution behavior.

Finally, the methodology can be considered a basis to implement a software tool able to automate the estimate of Actor applications.

References

[Agha 86] Agha, G.: "Actors: A Model of Concurrent Computation in Distributed Systems"; The MIT Press (1986).

[Agha 90] Agha, G.: "Concurrent object-oriented programming"; Communications of the ACM, 33, 9 (1990), 125–141.

[Agha et al. 97] Agha, G., Mason, I. A., et al.: "A foundation for actor computation"; Journal of Functional Programming, 7 (1997), 1–72.

[Andolfi et al. 00] Andolfi, F., Aquilani, F., Balsamo, S., Inverardi, P.: "Deriving Performance Models of Software Architectures from Message Sequence Charts"; 2^{nd} ACM Intl Workshop on Software and Performance, Ottawa, Canada (2000), 47–57.

[Baker, Hewitt 77] Baker, H. G., Hewitt, C.: "Laws for communicating parallel processes"; IFIP Congress (1977), 987–992.

[Balsamo et al. 04] Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: "Model-Based Performance Prediction in Software Development: A Survey"; IEEE Transactions on Software Engineering, 30, 5 (2004), 295–310.

[Blelloch 96] Blelloch, G. E.: "Programming Parallel Algorithms"; Communications of the ACM, 39, 3 (1996), 85–97.

[Blumofe et al. 95] Blumofe, R. D., Joerg, C. F., at al.: "Cilk: An Efficient Multithreaded Runtime System"; *5th* ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Santa Barbara, California, USA (1995), 207–216.

[Charron-Bost et al. 96] Charron-Bost, B., Mattern, F., Tel, G.: "Synchronous, asynchronous, and causally ordered communication"; Distributed Computing, 9, 4 (1996), 173–191.

[Clinger 81] Clinger, W.: "Foundation of Actor Semantics"; AI-TR-633, MIT Artificial Intelligence Laboratory (1981).

[Cortellessa et al. 01] Cortellessa, V., D'Ambrogio, A., Iazeolla, G.: "Automatic Derivation of Software Performance Models from CASE Documents"; Performance Evaluation, 45 (2001), 81–105.

[Culler et al. 96] Culler, D., Karp, R., Patterson, D., et al.: "LogP: A Practical Model of Parallel Computation"; Communications of the ACM, 39, 11 (1996), 78–85.

[Emrath et al. 88] Emrath, P. A., Ghosh, S., Padua, D. A.: "Automatic detection of nondeterminacy in parallel programs"; Workshop on Parallel and Distributed Debugging, Madison, Wisconsin, USA (1988), 89–99.

[Foster 95] Foster, I.: "Designing and Building Parallel Programs"; Addison Wesley, Boston (1995).

[Frigo et al. 98] Frigo, M., Leiserson, C. E., Randall, K. H.: "The Implementation of the Cilk-5 Multithreaded Language"; ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada (1998), 212–223.

[Grama, Kumar 99] Grama, A. Y., Kumar, V.: "State of the Art in Parallel Search Techniques for Discrete Optimization Problems"; IEEE Transactions on Knowledge and Data Engineering, 11, 1 (1999), 28–35.

[Harel 87] Harel, D.: "Statecharts: A visual formalism for complex systems"; Science of Computer Programming, 8, 3 (1987), 231–274.

[Honda, Tokoro 91] Honda, K., Tokoro, M.: "An object calculus for asynchronous communication"; European Conference on Object-Oriented Programming, LNCS 512, Springer-Verlag (1991), 133–147.

[Kanal, Kumar 88] Kanal, L. N., Kumar, V.: "Search in Artificial Intelligence"; Springer-Verlag, Berlin (1988).

[Kant 92] Kant, K.: "Introduction to Computer System Performance Evaluation"; McGraw-Hill (1992).

[Lamport 78] Lamport, L.: "Time, Clocks, and the Ordering of Events in a Distributed System"; Communications of the ACM, 21, 7 (1978), 558–565.

[Mattern, Fünfrocken 95] Mattern, F., Fünfrocken, F.: "A Non-Blocking Lightweight Implementation of Causal Order Message Delivery"; Theory and Practice in Distributed Systems, LNCS 938, Springer-Verlag (1995), 197–213.

[Milner et al. 92] Milner, R., Parrow, J., Walker, D.: "A calculus of mobile processes" (Parts I and II); Information and Computation, 100 (1992), 1–77.

[Peterson 81] Peterson, J. L.: "Petri Net. Theory and the Modeling of Systems"; Prentice Hall (1981).

[Petriu, Wang 99] Petriu, D. C., Wang, X.: "From UML Descriptions of High-Level Software Architectures to LQN Performance Models"; Intl Workshop on Applications of Graph Transformations with Industrial Relevance, LNCS 1779, Springer-Verlag (1999), 47–62.

[Press et al. 92] Press, W. H., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T: "Numerical Recipes in C: The Art of Scientific Computing"; Cambridge University Press (1992), 896–901.

[Rumbaugh et al. 98] Rumbaugh, J., Jacobson, I., Booch, G.: "Unified Modeling Language Reference Manual"; Addison-Wesley (1998).

[Skillicorn, Talia 96] Skillicorn, D. B., Talia, D.: "Models and Languages for Parallel Computation"; ACM Computing Surveys, 30, 2 (1996), 123–169.

[Skillicorn 98] Skillicorn, D. B.: "Architectures, Costs, and Transformations"; Invited talk, Workshop on Constructive Methods for Parallel Programming, Marstrand, Sweden (1998).

[Smith, Talcott 99] Smith, S. F., Talcott, C. A.: "Modular Reasoning for Actor Specification Diagrams"; Formal Methods in Object-Oriented Distributed Systems, Kluwer Academic Publishers (1999).

[Smith, Talcott 02] Smith, S. F., Talcott, C. A.: Specification Diagrams for Actor Systems"; Higher-Order and Symbolic Computation, 15, 4 (2002), 301–348.

[Talcott 98] Talcott, C. A.: "Composable Semantic Models for Actor Theories"; Higher-Order and Symbolic Computation, 11, 3 (1998), 281–343.

[Thati et al. 04] Thati, P., Talcott, C. A., Agha, G.: "Techniques for Executing and Reasoning About Specification Diagrams"; 10^{th} Intl Conference on Algebraic Methodology and Software Technology, LNCS 3116, Springer-Verlag (2004), 521–536.

[Trivedi 01] Trivedi, K. S.: "Probability and Statistics with Reliability, Queuing, and Computer Science Applications"; John Wiley and Sons (2001).

[Varela, Agha 98] Varela, C. A., Agha, G.: "What after Java? From objects to actors"; 7^{th} International Conference on World Wide Web 7, Brisbane, Australia (1998).