Time analysis of actor programs[☆]

Cosimo Laneve[a], Michael Lienhardt[b], Ka I Pun[c,d], Guillermo Román-Díez[e]

[a] *University of Bologna/INRIA, Italy*
[b] *University of Turin, Italy*
[c] *Western Norway University of Applied Sciences, Norway*
[d] *University of Oslo, Norway*
[e] *Universidad Politécnica de Madrid, Spain*

Abstract

This paper proposes a technique for estimating the computational time of programs in an actor model, which is intended to serve as a compiler target of a wide variety of actor-based programming languages. We define a *compositional* translation function returning *cost equations*, which are fed to an automatic off-the-shelf solver for obtaining the time bounds. Our approach is based on a new notion of *synchronization sets*, which captures possible difficult synchronization patterns between actors and helps make the analysis efficient and precise. The approach is proven to correctly over-approximate the worst computational time of an actor model of a concurrent programs. Our technique is complemented by a prototype analyzer that returns upper bound of costs for the actor model.

Keywords: time analysis, behavioural types, resource analysis

1. Introduction

Time computation for programs running on mainstream architectures, for example, multicore, distributed systems or cloud, is intricate and demanding as the execution of a process may be indirectly delayed by other processes running on different machines due to synchronizations. The computational cost of programs is particularly relevant in cloud architectures, where services are bound by so-called *service-level agreements* (SLAs), which regulate the costs in time and assign penalties for their infringements [7]. In particular, the service providers need guarantees that the services meet the SLA, for example in terms of the end-user response time, by deciding on a resource management policy,

[☆]Supported by the Spanish MINECO projects TIN2015-69175-C4-2-R, by the Spanish CM project S2013/ICE-3006, and the *SIRIUS Centre for Scalable Data Access* (`www.sirius-labs.no`).

Email addresses: `cosimo.laneve@unibo.it` (Cosimo Laneve), `michael.lienhardt@di.unito.it` (Michael Lienhardt), `Violet.Ka.I.Pun@hvl.no` (Ka I Pun), `guillermo.roman@upm.es` (Guillermo Román-Díez)

and by determining the appropriate number of virtual machine instances (or containers) and their parameter settings (e.g., their CPU speeds).

In this paper we propose a technique for estimating the computational time of programs in an actor model. This model is intended to serve as a compiler target of a wide variety of actor-based programming languages, such as object-oriented ones, including `Java` and `C#`, which are used in cloud architectures. Our technique aims at (and in fact, has been developed for) helping service providers to select resource management policies in *a correct way*, before actually deploying the service.

Several techniques have been proposed for analyzing the computational cost of sequential programs. See for example [3, 6, 11, 12, 18] and Section 6. Our approach is similar to [16], where a statically typed intermediate language has been defined in order to verify safety properties and certify code optimizations. Different from [16], our language, called `alt`, short for *actor language with time*, is concurrent, and contains an operation defining the number of processing cycles required to be computed, called $wait(n)$ (similar to the `sleep`(n) operation in `Java`). In order to analyze the computational time of `alt`, we define an algorithm that returns a set of cost equations that are adequate for a solver. We demonstrate that the solution of the cost equations over-approximates the computational cost of the `alt` program in input. Given these results, estimating the computational cost of a program in some programming languages amounts to defining a compilation into an `alt` program (and demonstrating its correctness).

The presented work builds upon a previous article by the authors [10], where it captures the actor models of actor-based programming languages in terms of *behavioural types*, and obtains the computational cost of the corresponding program by feeding the cost equations produced by a translation function to a off-the-shelf solver – the `CoFloCo` solver [8]. However, this technique proposed in [10] has a very severe constraint: invocations were admitted only either on the same actor or on newly created ones, i.e., no invocation on parameters. For instance, according to this constraint, an invocation to a method `inner`(y, x), where the first parameter is the actor executing the method, cannot occur in the body of a method `outer`(x, y). The challenge is that, in this case, computing the cost of `outer`(x, y) requires to know whether there is a synchronization between actors x and y. In case there is, one has to consider that `inner`(y, x) might be delayed by other methods running on y, which might be independent from `outer`(x, y).

This paper focuses on overcoming this issue. We first compute *synchronization sets* of actors, which are actors that potentially might interfere with the executions of each other. We then compose the cost of an invocation with the cost of the callee in two ways: (1) it is *added* – corresponding to *sequential compositions* – if the arguments of the invocation and those of the caller are in the same synchronization set; (2) it is the *maximum value* – corresponding to *parallel composition* – otherwise. We then define a new translation function that takes the synchronization sets of an `alt` program into account and returns the corresponding set of cost equations.

The translation of `alt` programs into the solver input code [2, 8] has been

prototyped and can be experimented (see Section 5). This tool, together with the compiler we have defined in [10], allow us to automatically compute the cost of programs in ABS, a prototype language for programming the cloud [14] that is an extension of alt. Experimental results show that our technique is very precise when computing the cost of a number of typical distributed patterns, such as *fork-join* or *map-reduce*.

Paper overview. The alt language is defined in Section 2 and we discuss in Section 3 the issues in estimating the computational cost. Section 4 explains the analysis of computational time by means of a translation function that returns cost equations, and discusses the properties of the translation. In Section 5, we illustrate the conversion of the translation output to the adequate form for an off-the-shelf solver, present our prototype, together with the experimental results of our approach and a comparison with other tools. In Section 6 we discuss the related work and deliver concluding remarks in Section 7.

2. The language alt

In this section, we define the syntax and the semantics of alt and discuss some examples.

Syntax. We use several disjoint sets of *names*: *method names*, ranged over $\mathtt{m}$, $\mathtt{m}'$, $\cdots$; *actor* or *integer names* ranged over x, y, u, $\cdots$; *future names* ranged over f, g, h, $\cdots$. The notation $\overline{x}$ denotes a possibly empty tuple of names. An alt program is a tuple $\left(\mathtt{m}_1(\overline{x}_1) = s_1, \cdots, \mathtt{m}_n(\overline{x}_n) = s_n, s\right)$ namely a sequence of method definitions of the form $\mathtt{m}_i(\overline{x}_i) = s_i$ plus a main statement s. Statements s and expressions e are defined by the following syntax:

$$
\begin{array}{rcl}
s & ::= & \mathtt{0} \mid \nu x; s \mid \nu f\colon \mathtt{m}(\overline{e}); s \mid f^{\checkmark}; s \mid \mathit{wait}(e); s \\
e & ::= & k \mid x \mid e + e \quad (k \text{ are integer constants})
\end{array}
$$

The syntax of alt statements is quite basic: 0 indicates a terminated statement; νx is the creation of a new actor x; $\nu f\colon \mathtt{m}(\overline{e})$ creates a new task of the method m that is associated to the future f. We assume that $\overline{e} = x, \overline{e}'$, namely $\overline{e}$ is a nonempty tuple where the first element is *always* an actor name indicating the callee of the method, that is, $\nu f\colon \mathtt{m}(x, \overline{e}')$ runs the corresponding instance of the body of m on actor x. The term $f^{\checkmark}$ performs the synchronization of the task associated to f. This may cause the (busy) waiting for the termination of the task. The statement $\mathit{wait}(e)$, where e is an integer expression, represents the advance of e time units. This is the only term in our model that consumes time (a.k.a. that has a cost). The term e is an *integer* expression specifying how many processing cycles are needed by the subsequent statement in the code. These expressions are *Presburger arithmetics expressions*, which is a decidable fragment of Peano arithmetics containing only addition. The restriction is necessary because the analysis in Section 4 cannot deal with generic expressions. Remark that actor names are also expressions.

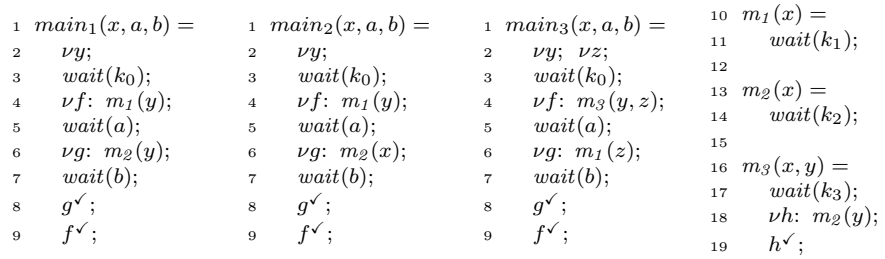


Figure 1: Three *main* methods in `alt`

The method definition $\mathtt{m}(\overline{x}) = s$ binds names $\overline{x}$ in the body s. The sequence $\overline{x}$ is assumed to be $x', \overline{y}, \overline{z}$ where x' is the callee, $\overline{y}$ and $\overline{z}$ are actor names and integer names, respectively. Note that `alt` assumes the presence of a simple type system that verifies the correctness of method applications. Statements $\nu x\mathtt{;}s$ and $\nu f\colon \mathtt{m}(\overline{e})\mathtt{;}s$ link the names x and f in the continuations s. Because of the notion of bound name and the one of free name, we use the standard operations of alpha-conversion and substitution. It is also assumed that method names $\mathtt{m}_1, \cdots, \mathtt{m}_n$ in program declarations are pairwise different.

Two features of `alt` ease our theoretical developments. In particular, in `alt`, actors are *stateless* and methods do not return values. Computing the cost of stateful actor programs requires a leap of the theory developed in this contribution because it is necessary to trace the state of the fields. Similarly, when methods return a value, it is necessary to estimate the value (is it an old actor or a new one? if it is an integer, how large it is?) in order to have sensible cost computations.

Semantics. The semantics of `alt` is a transition system whose states are *configurations* cn that are defined as follows.

$$cn \;::=\; act(x,p,q) \;\mid\; fut(f,val) \;\mid\; invoc(x,f,\mathtt{m},\overline{v}) \;\mid\; cn\ cn$$

$$val \;::=\; \bot \mid \top \qquad\qquad p \;::=\; s\mathtt{;}f \mid f \mid \mathtt{0}$$
$$v \;::=\; x \mid f \mid k \qquad\qquad q \;::=\; \varnothing \mid s\mathtt{;}f \mid q\ q$$

The notation $s\mathtt{;}f$ represents the statement s where the tailing `0` is replaced by the future name f. We use p to range over $s\mathtt{;}f$, f, and `0`. A *configuration* cn is a nonempty set of actors, invocation messages and futures. The associative and commutative union operator on configurations is denoted by whitespace. An actor is written as $act(x,p,q)$, where x is the identity of the actor, p is the *active task*, and q is a pool of either *suspended* or *waiting tasks*. An *invocation message* is denoted as $invoc(x,f,\mathtt{m},\overline{v})$, where x is the callee, f the future to which the call is bound, $\mathtt{m}$ the method name, and $\overline{v}$ the set of parameter values of the call. Configurations also include *futures*, denoted as $fut(f,val)$, where f is the future identity and val indicates whether f has already been computed, written as $\top$, otherwise $\bot$.

$$\frac{cn \to cn'}{cn\ cn'' \to cn'\ cn''}\ (\textsc{Context}) \qquad \frac{z' = \text{fresh}(\,)}{act(x, \nu z\,; p, q) \to act(x, p\{^{z'}/_{z}\}, q)\ \ act(z', \mathsf{0}, \varnothing)}\ (\textsc{New})$$

$$act(x, f^{\checkmark}\,; p, q)\ \ fut(f, \top) \to act(x, p, q)\ \ fut(f, \top)\ (\textsc{Get-True})$$

$$act(x, f^{\checkmark}\,; p, q)\ \ fut(f, \bot) \to act(x, \mathsf{0}, q \cup (f^{\checkmark}\,; p))\ \ fut(f, \bot)\ (\textsc{Get-False})$$

$$\frac{[\![\overline{e}]\!] = \overline{v} \quad f' = \text{fresh}(\,)}{act(x, \nu f\colon \mathtt{m}(z, \overline{e})\,; p, q)\ \to act(x, p\{^{f'}/_{f}\}, q)\ \ invoc(z, f', \mathtt{m}, \overline{v})\ \ fut(f', \bot)}\ (\textsc{Async-Call})$$

$$\frac{s = \text{bind}(x, \mathtt{m}, \overline{v})}{act(x, p, q)\ \ invoc(x, f, \mathtt{m}, \overline{v}) \to act(x, p, q \cup s\,; f)}\ (\textsc{Bind-Fun}) \qquad \frac{[\![e]\!] = 0}{act(x, wait(e)\,; s, q) \to act(x, s, q)}\ (\textsc{Wait-0})$$

$$act(x, \mathsf{0}, q \cup p) \to act(x, p, q)\ (\textsc{Activate}) \qquad act(x, f, q)\ \ fut(f, \bot) \to act(x, \mathsf{0}, q)\ \ fut(f, \top)\ (\textsc{Return}) \qquad \frac{strongstable_t(cn)}{cn \xrightarrow{t} \Phi(cn, t)}\ (\textsc{Tick})$$

Figure 2: Transition rules.

The *transition rules* of `alt` are given in Figure 2. We use an auxiliary function to bind invocations to the corresponding method bodies. Let $\mathtt{m}(x, \overline{z}) = s$ be an `alt` method. Then

$$\text{bind}(y, \mathtt{m}, \overline{v}) = s\{^{y, \overline{v}}/_{x, \overline{z}}\}\ .$$

We also use the function fresh( ) to return either a fresh actor name or a fresh future name. Finally, we use an *evaluation function* $[\![\cdot]\!]$ of expressions e such that $[\![x]\!] = x$, and $[\![e]\!] = v$ whenever e is a constant integer expression and v is its value; $[\![e]\!]$ is undefined otherwise.

The semantics of `alt` is almost standard. See, for example [15], for a similar semantics for a dialect of pi-calculus with time. We discuss the most relevant rules in the following: actor creation, method invocation, method return and the $wait(e)$ operation.

Actor creation is handled by rule New, which extends the configuration with $act(z', \mathsf{0}, \varnothing)$ with z' as a fresh actor identifier. In `alt`, method invocations are asynchronous: rule Async-Call creates a new invocation predicate $invoc(z, f', \mathtt{m}, \overline{v})$ and a new unresolved future predicate $fut(f', \bot)$, which is associated to the invocation with a fresh future identifier f'. The invocation predicate will then be bound to the corresponding callee actor as one of the waiting task (cf. rule Bind-Fun). When a method terminates, rule Return sets the value of the corresponding future f to $\top$. Note that f is added to the end of the method body in rule Bind-Fun. In our model, $wait(e)$ is the unique

operation that consumes time; that is, time does not advance as long as some active task is prefixed by a $wait(e)$ operation. For the trivial case where $e = 0$ (see rule WAIT-0), the operation is simply discarded. On the contrary, when a configuration cn reaches a *stable* state, that is, no other transition is possible apart from those evaluating the $wait(e)$ operations, time advances until an actor with a non-$wait(e)$ operation can proceed. To formalize this semantics, we first introduce the notion of *stability* as follows:

Definition 2.1. Let $t > 0$. A configuration cn is *t-stable*, written $stable_t(cn)$, if every actor in cn matches one of the following forms:

1. $act(x, wait(e)\,;s\,;f', q)$ with $[\![e]\!] \geq t$,
2. $act(x, 0, q)$ and
   i. either $q = \varnothing$,
   ii. or, for every $s \in q$, $s = f^{\checkmark}\,;p$ and $fut(f, \bot) \in cn$.

A configuration cn is *strongly t-stable*, written as $strongstable_t(cn)$, if it is t-stable and there exists an actor $act(x, wait(e)\,;s, q)$ with $[\![e]\!] = t$.

Note that t-stable (and consequently, strongly t-stable) configurations cannot progress anymore because every actor is stuck either on a $wait(\cdot)$-statement or on an unresolved future.

We then define a function to update a configuration cn with respect to a time value t.

$$\Phi(cn, t) = \begin{cases} act(x, wait(k)\,;s, q)\ \Phi(cn', t) & \text{if } cn = act(x, wait(e)\,;s, q)\ cn' \\ & \text{and } k = [\![e]\!] - t \\ act(x, 0, q)\ \Phi(cn', t) & \text{if } cn = act(x, 0, q)\ cn' \\ cn & \text{otherwise.} \end{cases}$$

Together with the rule TICK in Figure 2, we define the *time progress* of a configuration.

The initial configuration of a program with a main body s is

$$act(start, s\,;f_{start}, \varnothing)$$

where $start$ and f_{start} are respectively an actor name and a future identifier for the main body. As usual, $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$ and $\overset{t}{\Rightarrow}$ is $\rightarrow^* \overset{t}{\rightarrow} \rightarrow^*$. A *computation* is $cn \overset{t_1}{\Rightarrow} \cdots \overset{t_n}{\Rightarrow} cn'$, that is, cn' is a configuration reachable from cn with either transitions $\rightarrow$ or $\overset{t}{\Rightarrow}$. When the time labels of transitions are not relevant we also write $cn \Rightarrow^* cn'$.

Definition 2.2. The *execution time of a computation* $cn \overset{t_1}{\Rightarrow} \cdots \overset{t_n}{\Rightarrow} cn'$ is $t_1 + \cdots + t_n$.

The *execution time of a configuration* cn, written $time(cn)$, is the maximum execution time of computations starting at cn. The execution time of an `alt` program is the execution time of its initial configuration.

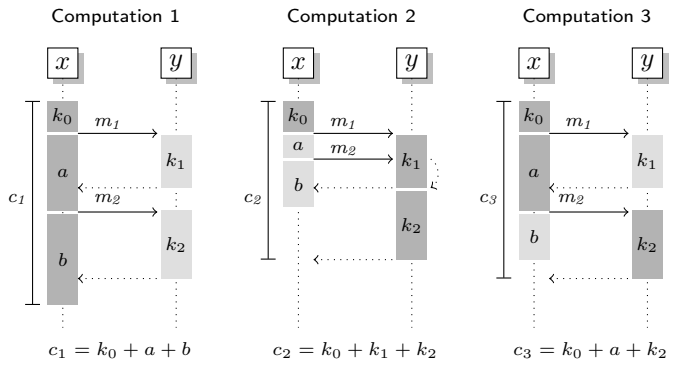
Example 2.1. The method $main_1$ in Figure 1 creates a new actor y at line 2 and spawns two tasks on it at lines 4 and 6, respectively, where the two tasks will execute in parallel with $main_1$. Their terminations are synchronized at lines 8 and 9 by means of $g^{\checkmark}$ and $f^{\checkmark}$. Note that $main_1$ takes its integer arguments as the input parameter of the $wait(\cdot)$-operations at lines 5 and 7, while the one at line 3 uses a constant k_0. Thus, the execution time of $main_1$ depends on the concrete values of a and b. The other $wait(\cdot)$-statements in the example are executed with some constants k_1 and k_2. The other two methods $main_2$ and $main_3$ are interpreted analogously, and we will discuss the executions of these three *main* methods and their corresponding cost in Section 3. ■

Our semantics does not exclude behaviours of methods that perform infinite actions without consuming time (preventing rule TICK to apply), such as $\texttt{foo}(x) = \nu f\colon \texttt{foo}(x); f^{\checkmark}$. This kind of behaviours are well-known in the literature (*cf.* Zeno behaviours, see [15]) and they may be easily excluded from our analysis by constraining recursive invocations to be prefixed by a $wait(e)$-statement, where e is a positive integer.

3. The challenges of cost computation for `alt` programs

Computing the time of `alt` programs is challenging because it is reduced to determining the tasks that *cannot execute in parallel* – therefore their cost must be *added*; and the tasks that *are executed in parallel* – thus, the overall cost is the *maximum value* of the costs. In the following, we illustrate the difficulties by discussing three examples.

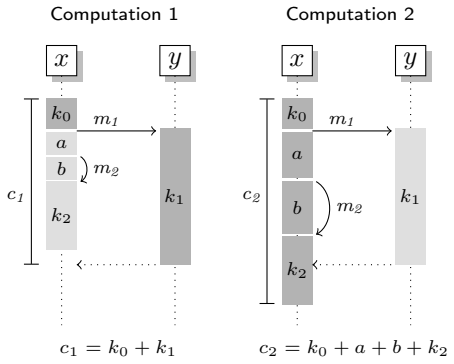
Example 3.1. Consider the method $main_1$ in Figure 1. In this case, methods m_1 and m_2 are invoked on actor y and run in parallel with $main_1$ (which runs on actor x). The graphical representation below illustrates three different computations that are obtained by choosing different values of a and b. Rectangles in dark grey represents the times spent to compute the maximal cost of the computations, which are highlighted by the expressions c_1, c_2 and c_3, respectively.



Computation 1 represents an execution where $a > k_1$, which leads to the execution of m_2 on actor y starts *after* the termination of m_1. In this case, the executions of $wait(b)$ and m_2 run in parallel. Thus, the cost of this execution is $k_0+a+max(b,k_2)$ (the expression c_1 in the figure reflects the case where $b \geq k_2$). Computation 2 describes the execution where $a < k_1$, which leads to delaying m_2 until the execution of m_1 finishes. In this computation, since $a+b < k_1+k_2$, actor x has to wait for the termination of m_2. Finally, Computation 3 illustrates an execution where both actors x and y have to wait either for executing a method or for a method to terminate and return. As our analysis aims at finding a sound approximation of time needed for any execution of the program, the cost of $main_1$ is over-approximated by the expression $k_0 + max(a, k_1) + max(b, k_2)$. Note that the sub-expression $k_0+max(a, k_1)$ determines (an over-approximation of) the starting time of m_2 on actor y. ■

Another challenge to cope with is the case where task is delayed due to an unresolved future (see rule GET-FALSE). In this situation, the carrier actor may start the execution of the pending tasks and may reschedule the initial task after the termination of the other ones.

Example 3.2. Consider $main_2$ in Figure 1 that is almost identical to $main_1$, except that m_2 is invoked on the carrier actor x, on which $main_2$ is also executing. The computations below show two possible executions of $main_2$, depending on the values of a and b.



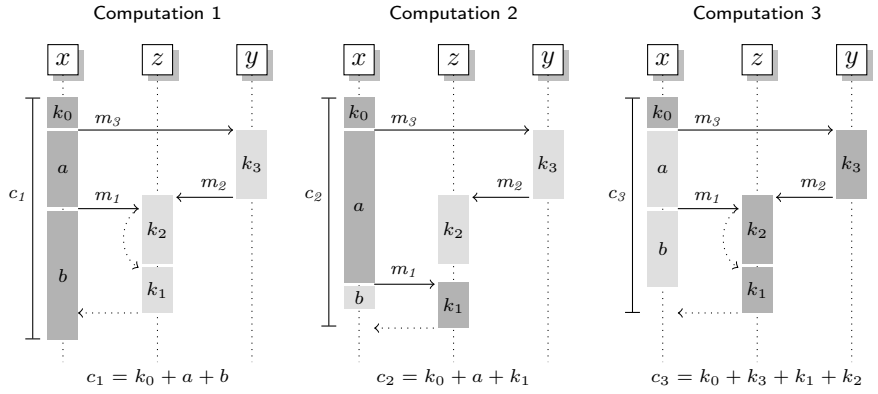
In Computation 1, the synchronization $g^{\checkmark}$ at line 8 happens before m_1 terminates. In this case, the cost of executing m_2 does not contribute to the cost expression c_1. On the contrary, in Computation 2, as the values of a and b are larger, the total time taken for executing $wait(a)$, $wait(b)$ and m_2 (i.e., $wait(k_2)$) on actor x is longer than running m_1 on y. In this case, the cost of executing m_1 does not contribute to c_2. For $main_2$, the expression that over-approximates c_1 and c_2 and defines the worst case execution time is $k_0 + max(a + b + k_2, k_1)$.■

A more complex scenario is that two actors x and y spawn tasks on a third actor z. In this case, the execution of tasks on x and y may delay the starting time of tasks spawned on z. As a consequence, it may lead to postponement at

the synchronization points on x and y. Determining the delays of synchronizations is crucial, as it is discussed in the following example.

Example 3.3. The method $main_3$ in Figure 1 creates two new actors y and z at line 2, then invokes methods m_3 on y at line 4 and m_1 on z at line 6. The actor z is given as a parameter in the invocation of m_3 (line 4) and subsequently also a parameter of m_2 (line 18). The figure below shows three different possible computations of $main_3$.

Computation 1 shows an overall cost computed by summing up the costs of the statements $wait(\cdot)$ in $main_3$. This cost does not include the cost of any task executed on y and z as they are running in parallel with $main_3$ without any delays. Computation 2 combines the costs from actors x and z but not from y. Observe that the time needed to start m_1 is determined by the expression $k_0 + a$ since $a > k_3 + k_2$, where $wait(k_3)$ and $wait(k_2)$ execute sequentially although they run on different actors (y and z). Note that, $wait(b)$ and $wait(k_1)$ are executed in parallel on x and z, and since $k_1 > b$, the maximum cost is determined by the expression $k_0 + a + k_1$.



On the contrary, Computation 3 highlights a possible dependency among tasks executed on y and those on z. The cost expression for this case is $c_3 = k_0 + k_3 + k_2 + k_1$, describing a sum to which all three actors contribute (k_0 on x, k_3 on y and $k_2 + k_1$ on z). The reason is that the statement $wait(k_3)$ on y (line 17) directly affects the starting time of m_2 on z, which in this case delays the execution of m_2 on z as a consequence. Interestingly, one difference between Computation 2 and Computation 3 is determined by the time taken before starting m_1 on actor z. In Computation 1, the cost $k_0 + k_3$ is smaller than $k_0 + a$, causing that m_2 is executed before m_1 on z. However, although in Computation 3 we have that $b > k_1$, b is not considered in the overall cost. When there exists two actors whose tasks could be dependant on each other, their costs must be considered as if the tasks are executed sequentially, that is, their costs are summed up. Thus, the relation between the tasks executed on y and on z must be considered in order to over-approximate all possible executions of $main_3$. Considering all possible values of a and b, the cost expression of this computation is $k_0 + max(a, k_2 + k_3) + max(b, k_1)$. ■

This last example suggests that, when a method invocation has several actors as arguments, the queues of these actors are in relation to each other, as each of them may in principle affect the queues of the others by delaying the execution of the tasks therein. Thus, we say these actors belong to the same *synchronization set*. We over-approximate this situation by considering all the queues of the actors in the same synchronization set as if they were *one unique queue*. It is crucial to note that this synchronization relation is *transitive*. For instance, if x and y belong to the same synchronization set at a program point and if a new synchronization relation between y and z is established by a subsequent invocation, then actors x, y and z belong to the same synchronization set as well.

4. The analysis of `alt` programs

This section defines the translation of an `alt` program into a *cost program* (see below). Given an `alt` program $\mathcal{P}$, the analysis first computes the *synchronization schema* of each method – the (actor) arguments and bound actor names are grouped into disjoint sets according to their potentials to affect the queues of each other (*cf.* Section 4.1) – then translates every method definition $\mathtt{m}(\overline{x}, \overline{n}) = s$ in $\mathcal{P}$ into a cost equation function of the form $m(\overline{n}) = exp$, where m is a (cost) function symbol corresponding to the method name `m` and exp is an expression that may contain (cost) function applications. The translation is detailed in Section 4.2. Note that we do not define the syntax of exp; interested readers may refer to either [2] or [8].

In the following, without loss of generality, we assume that `alt` programs have methods defined by $\mathtt{m}(x, \overline{y}, \overline{n}) = \nu z_1\,;\cdots;\nu z_k\,;s$, and *main* body defined by $\nu w_1\,;\cdots;\nu w_h\,;s'$ such that s and s' do not contain actor creations. We also assume that actor names $x, \overline{y}$ do not clash with names $z_1, \cdots, z_k$.

4.1. Synchronization schemas

A *synchronization set*, ranged over A, B, $\cdots$, is a set of actor names whose tasks are dependant, that is, the task may reciprocally affect the queues of the actors in the same set by means of method invocations and synchronizations. A *synchronization schemas*, ranged over $\mathcal{S}$, $\mathcal{S}'$, $\cdots$, is a collection of pairwise disjoint synchronization sets. Let $\mathcal{S}$ be a synchronization schema, we define two auxiliary functions as follows:

- $\mathcal{S}(x)$ is either A, if $A \in \mathcal{S}$ and $x \in A$, or it is the empty set $\varnothing$;
- $\mathcal{S} \oplus A = \begin{cases} \{A\} & \text{if } \mathcal{S} = \varnothing \\ (\mathcal{S}' \oplus A) \cup \{A'\} & \text{if } \mathcal{S} = \mathcal{S}' \cup \{A'\} \text{ and } A \cap A' = \varnothing \\ \mathcal{S}' \oplus \{A' \cup A\} & \text{if } \mathcal{S} = \mathcal{S}' \cup \{A'\} \text{ and } A \cap A' \neq \varnothing \end{cases}$

The function $\mathcal{S} \oplus A$ merges a schema $\mathcal{S}$ with a synchronization set A. If none of the actors in A belongs to a set in $\mathcal{S}$, the operation reduces to a simple set union.

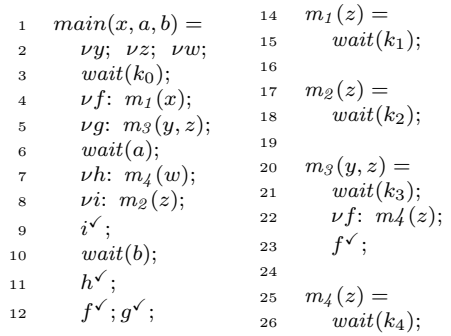


Figure 3: Running example of a `alt` program

Definition 4.1 (Synchronization schema function)**.** Let

$$\texttt{sschem}(\mathcal{S}, s) = \begin{cases} \mathcal{S} \oplus \{y, \overline{z}\} & \text{if } s = \nu f\colon \texttt{m}(y, \overline{z}, \overline{e}) \\ \texttt{sschem}(\texttt{sschem}(\mathcal{S}, s'), s'') & \text{if } s = s'\,;s'' \\ \mathcal{S} & \text{otherwise} \end{cases}$$

Let $\texttt{m}(x, \overline{y}, \overline{e}) = \nu z_1\,; \cdots ;\nu z_k\,; s$ be an `alt` method definition. The *synchronization schema* of $\texttt{m}$, denoted as $\mathcal{S}_\texttt{m}$, is the set $\mathcal{S}_\texttt{m} = \texttt{sschem}(\{\{x, \overline{y}\}\}, s)$.

For instance, consider a synchronization schema $\mathcal{S}$=$\{\{x, y\}, \{u, v\}\}$, then $\texttt{sschem}(\mathcal{S}, \nu f\colon \texttt{m}(y, u)) = \{\{x, y, u, v\}\}$, meaning that all actors belong to the same synchronization set. On the contrary, with the call $\texttt{m}(y, z)$, as y, z do not belong to $\{u, v\}$, we get $\texttt{sschem}(\mathcal{S}, \nu f\colon \texttt{m}(y, z)) = \{\{x, y, z\}, \{u, v\}\}$.

Example 4.1. Let $main(x, a, b) = \nu y\,;\nu z\,;\nu w\,; s_{main}$ represent the main method in Figure 3. The synchronization schema $\mathcal{S}_{main}$ is $\texttt{sschem}(\{\{x\}\}, s_{main}) = \{\{x\}, \{y, z\}, \{w\}\}$; that is, the queues of actors y and z affect each other's because they are in the same synchronization set due to the invocation at line 8, whereas the queues of x and w are independent from the rest. Observe that as the body of m_3 does not contain actor creation statement, its synchronization schema is a singleton set that contains the parameters, e.g., $\mathcal{S}_{m_3} = \{\{y, z\}\}$. ■

4.2. Translation

The translation of an `alt` program into a cost program follows the guideline that actors belonging to the same synchronization set could delay tasks in each other's queue. Since the execution time of a task executing on one of the actors may depend on the scheduling of the tasks on the others, we consider all the tasks in those actors as if they belong to the same actor by putting the tasks in the same synchronization set. Thus, they must all be *sequentialized*. On the contrary, if two tasks belong to actors residing in different synchronization sets, the execution of these two tasks is *parallelized* and we consider the maximal

value of their costs. In order to express these two types of compositions, we use *accumulated costs* ε, which are defined as follows:

$$\varepsilon \ ::= \ e \ \mid \ \varepsilon \cdot \langle m(\overline{n}), e \rangle \ \mid \ \varepsilon \parallel e \ .$$

The term $\varepsilon \cdot \langle m(\overline{n}), e \rangle$ represents the sequentialisation of the two costs, namely, ε and $\langle m(\overline{n}), e \rangle$. In particular, the latter stores the cost of an invocation of m and the *time distance* e_i between the invocation and the subsequent invocation on any actor in the same synchronization set A. The term $\varepsilon \parallel e$ expresses the parallel composition of two costs represented by ε and e. The evaluation of accumulated costs ε is defined as

$$[\![e]\!] = e \qquad [\![\varepsilon \cdot \langle m(\overline{n}), e \rangle]\!] = [\![\varepsilon]\!] + \max(m(\overline{n}), e) \qquad [\![\varepsilon \parallel e]\!] = \max([\![\varepsilon]\!], e)$$

where the cost of sequentialisation is captured by adding the two costs, while parallel composition by selecting the maximum between them. Observe that the cost of the pair $\varepsilon \cdot \langle m(\overline{n}), e \rangle$ is the maximal value between the cost of method m and the time distance e.

The translation of `alt` programs records the accumulated costs of synchronization sets in *translation environments*, ranged over by Ψ, Ψ', $\cdots$, which are maps from (pairwise disjoint) synchronization sets to accumulated costs. The operators $\parallel$ and $+$ on Ψ are described by the following auxiliary definitions:

$$(\Psi \parallel e)(S) = \Psi(S) \parallel e$$

$$(\Psi + e)(S) = \varepsilon \cdot \langle m(\overline{n}), e' + e \rangle \parallel (e_1 + e) \parallel \cdots \parallel (e_m + e) \qquad \text{with } \Psi(S) = \varepsilon \cdot \langle m(\overline{n}), e' \rangle \parallel e_1 \parallel \cdots \parallel e_m$$

Given a synchronization schema $\mathcal{S}$, the translation is defined by a function $\texttt{translate}_{\mathcal{S}}(I, \Psi, x, e', e, s)$ that takes six arguments:

i. I is a map from future names to synchronization sets,

ii. Ψ is a translation environment,

iii. x is the name of the *carrier*, i.e., the actor on which s is executing,

iv. e' is an over-approximated cost of the methods invoked on actors belonging to the same synchronization set where the carrier x resides and not yet synchronized,

v. e is an over-approximation of the current execution cost, which is the computational time accumulated from the beginning of the method execution, and

vi. a statement s, or a prefix μ representing $wait(e)$, $\nu f\colon \mathtt{m}(\overline{e})$, or $f^{\checkmark}$ that must be translated.

The translation function then returns a tuple consisting of four elements:

i. an updated map I',

$$
\texttt{translate}_{\mathcal{S}}(I, \Psi, x, e', e, \mu) \;=\;
\begin{cases}
(1) \quad (I, \Psi + e'', e', e + e'') & \text{when } \mu = \mathit{wait}(e'') \\
(2) \quad (I[f \mapsto \mathcal{S}(x)], \Psi, e' + m(\overline{n}),\ e) & \text{when } \mu = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{n}) \text{ and } y \in \mathcal{S}(x) \\
(3) \quad (I[f \mapsto \mathcal{S}(y)], \Psi[\mathcal{S}(y) \mapsto \varepsilon \cdot \langle m(\overline{n}), 0\rangle], e', e) & \\
\qquad & \text{when } \mu = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{n}) \text{ and } y \notin \mathcal{S}(x) \\
\qquad & \text{where } \mathcal{S}(y) \in \mathrm{dom}(\Psi) \text{ implies } \varepsilon = \Psi(\mathcal{S}(y)) \\
\qquad & \text{and } \mathcal{S}(y) \notin \mathrm{dom}(\Psi) \text{ implies } \varepsilon = e \\
(4) \quad (I \setminus F, \Psi + e', 0, e + e') & \text{when } \mu = f^{\checkmark} \text{ and } x \in I(f) \\
\qquad & \text{where } F = \{f' \mid I(f') = \mathcal{S}(x)\} \\
(5) \quad (I \setminus F, (\Psi \parallel e'') \setminus I(f), 0, e'') & \text{when } \mu = f^{\checkmark} \text{ and } x \notin I(f) \\
\qquad & \text{where } F = \{f' \mid I(f') = \mathcal{S}(x) \text{ or } I(f') = I(f)\} \\
\qquad & \text{and } e'' = \mathit{max}(e + e', [\![\Psi(I(f))]\!]) \\
(6) \quad (I \setminus F, \Psi + e', 0, e + e') & \text{when } \mu = f^{\checkmark} \text{ and } f \notin \mathrm{dom}(I) \\
\qquad & \text{where } F = \{f' \mid I(f') = \mathcal{S}(x)\}
\end{cases}
$$

Figure 4: The translation of `alt` statements

ii. an updated translation environment Ψ',

iii. an update of the cost of methods running on synchronized actors, and

iv. an expression of the updated current execution cost.

The function is defined on statements as follows

$$
\begin{aligned}
\texttt{translate}_{\mathcal{S}}(I, \Psi, x, e', e, \mathsf{0}) \;&=\; (I, \Psi, e', e) \\
\texttt{translate}_{\mathcal{S}}(I, \Psi, x, e', e, \mu\,;s') \;&=\; \texttt{translate}_{\mathcal{S}}(I', \Psi', x, e''', e'', s) \\
&\qquad \text{where } \texttt{translate}_{\mathcal{S}}(I, \Psi, x, e', e, \mu) = (I', \Psi', e''', e'')
\end{aligned}
$$

where $\texttt{translate}_{\mathcal{S}}(I, \Psi, x, e', e, \mu)$ is detailed in Figure 4. Note that, the difficult case of the translation function is when the statement is a task synchronization $f^{\checkmark}$, which is covered by cases (4), (5) and (6). In the following, we discuss the six cases of Figure 4 in depth:

***Case* (1):** When s is a $\mathit{wait}(e'')$ statement, the `translate` function adds the cost e'' to the current cost, which is also reflected on the environment Ψ by updating the duration of the method invocations on top of the sequences in Ψ.

***Cases* (2) *and* (3):** When s is a method invocation on actor y, there are two subcases, depending on whether y is in the same synchronization set of carrier x (*Case* (2)) or not (*Case* (3)). In *Case* (2), the cost of the invocation is added to e' and the updated I binds the corresponding future f to $\mathcal{S}(x)$. In *Case* (3), we add the binding from f to $\mathcal{S}(y)$ to the map I and update the translation environment Ψ by appending the cost of invoking m to the accumulated cost which $\mathcal{S}(y)$ maps to.

***Case* (4):** This is the first sub-case of task synchronisation which captures the situation where the synchronization is performed on a method whose callee belongs to $\mathcal{S}(x)$. Since it is non-deterministic when the task being synchronized will actually be scheduled, we sum the costs of all the tasks running on actors in $\mathcal{S}(x)$ to e (worst case) at this synchronization point. This cost has been stored in e', which is updated to 0 after this task synchronization. At the same time, we remove from I all the corresponding futures since the corresponding costs have been already considered.

***Case* (5):** The second sub-case describes the synchronization that is performed on a task whose callee, say y, does not belong to $\mathcal{S}(x)$. As actors x and y reside in different sets, the invocation on y is executed *in parallel* with the carrier x. Thus, the overall cost is computed as the *maximum* between the total cost of all pending invocations on the actors in $\mathcal{S}(x)$ to which carrier x belongs, captured by e', and the cost of all invocations on actors in $\mathcal{S}(y)$, denoted by $[\![\Psi(\mathcal{S}(y))]\!]$. Since we have considered the worst scheduling, that is, we have already counted the cost of all methods spawned on the actors in $\mathcal{S}(y)$ and $\mathcal{S}(x)$ so far, we remove from the environment all calls to the actors in $\mathcal{S}(y)$ and remove all the futures associated to $\mathcal{S}(y)$ and $\mathcal{S}(x)$ from I.

***Case* (6):** In this case, the future f does not belong to Ψ, which means that the cost of the invocation has been already computed. Nevertheless, it is possible that other methods have been invoked after that computation. Therefore, the actual termination of the invocation corresponding to f may happen *after* the completion of all the invocations. In order to take this into account we need to add the cost of those tasks whose callee belongs to $\mathcal{S}(x)$, which has been accumulated in e', in a similar way as we have done in *Case (4)*.

Observe that the resulting translation environment Ψ_n is always empty because, in our setting, any method invocations in `m` is always synchronized within the method body. For the same reason, e'_n is always equal to 0.

We comment on some relevant aspects of `translate` by discussing an example in detail.

Example 4.2. Figure 5 illustrates the application of `translate` function to *main* in Figure 3. The leftmost column states the line of code i of *main* and we let I_i, R_i, Ψ_i, e'_i, e_i refer to the corresponding argument of `translate` function at line i. The second column contains the output tuple of `translate` function, while the third specifies the calculation of the current cost e_i. The rightmost column indicates the case (c_i) of the translate function that we have applied. For the sake of clarity, as method calls do not contain numeric parameters, we use the method name m for $m(\_)$. Note that `translate` takes the synchronization schema $\mathcal{S}_{main} = \{\{x\}, \{y, z\}, \{w\}\}$ that has been computed in Example 4.1 as input.

At lines 3, 6 and 10, the cost expression e_i increases by summing the previous cost expression and the cost of $wait(\cdot)$-statements at each of these lines (case 1). A local invocation (case 2) to m_1 is added to e'_i at line 4 and its corresponding future variable is added to $I_4 = \{f \mapsto \{x\}\}$. Lines 5, 7 and 8

$3:\ \Big(\varnothing,\ \varnothing,\ 0,\ e_3\Big)$ $[\ e_3 = k_0\ ]$ **(1)**

$4:\ \Big(\{f \mapsto \{x\}\},\ \varnothing,\ m_1, e_4\Big)$ $[\ e_4 = e_3\ ]$ **(2)**

$5:\ \Big(\{f \mapsto \{x\}, g \mapsto \{y,z\}\},\ \{\{y,z\} \mapsto e_4 \cdot \langle m_3, 0\rangle\},\ m_1,\ e_5\Big)$ $[\ e_5 = e_4\ ]$ **(3)**

$6:\ \Big(\{f \mapsto \{x\}, g \mapsto \{y,z\}\},\ \{\{y,z\} \mapsto e_4 \cdot \langle m_3, a\rangle\},\ m_1,\ e_6\Big)$ $[\ e_6 = e_5 + a\ ]$ **(1)**

$7:\ \Big(\{f \mapsto \{x\}, g \mapsto \{y,z\}, h \mapsto \{w\}\},$
$\{\{y,z\} \mapsto e_4 \cdot \langle m_3, a\rangle, \{w\} \mapsto e_6 \cdot \langle m_4, 0\rangle\},\ m_1,\ e_7\Big)$ $[\ e_7 = e_6\ ]$ **(3)**

$8:\ \Big(\{f \mapsto \{x\}, g \mapsto \{y,z\}, h \mapsto \{w\}, i \mapsto \{y,z\}\},$
$\{\{y,z\} \mapsto e_4 \cdot \langle m_3, a\rangle \cdot \langle m_2, 0\rangle,\ \{w\} \mapsto e_6 \cdot \langle m_4, 0\rangle\},\ m_1,\ e_8\Big)$ $[\ e_8 = e_7\ ]$ **(3)**

$9:\ \Big(\{h \mapsto \{w\}\},\ \{\{w\} \mapsto (e_6 \cdot \langle m_4, 0\rangle \parallel e_9)\},\ 0,\ e_9\Big)$
$[\ e_9 = max\big(e_8 + m_1, e_4 + max(m_3, a) + max(m_2, 0)\big)\ ]$ **(5)**

$10:\ \Big(\{h \mapsto \{w\}\},\ \{\{w\} \mapsto (e_6 \cdot \langle m_4, b\rangle \parallel e_9 + b)\},\ 0,\ e_{10}\Big)$ $[\ e_{10} = e_9 + b\ ]$ **(1)**

$11:\ \Big(\varnothing,\ \varnothing,\ 0, e_{11}\Big)$ $[\ e_{11} = max\big(e_{10}, e_6 + max(m_4, b), e_9 + b\big)\ ]$ **(5)**

$12:\ \Big(\varnothing,\ \varnothing,\ 0, e_{12}\Big)$ $[\ e_{12} = e_{11}\ ]$ **(6)**

Figure 5: Application of the `translate` function to *main* of Figure 3. Every line l has pattern $l : (I_i, \Psi_i, e'_i, e_i)\ [e_i]\ (c_i)$, where $[e_i]$ is the calculation of e_i and (c_i) is the translate function case applied.

contain method invocations on actors not belonging to the carrier's synchronization set, which correspond to case 3 of the `translate` function. For instance, at line 5 we have the first method invocation on an actor in the synchronization set $\{y, z\}$. We then add the bindings $g \mapsto \{y, z\}$ to I_5 and $\{y, z\} \mapsto e_4 \cdot \langle m_3, 0\rangle$ to Ψ_5, where e_4 corresponds to the accumulated costs of methods invoked on the carrier's synchronization set so far, and the time distance to the subsequent method invocation on the same synchronization is 0. Observe that it is possible to compute the cost of pending invocations (on the actors of a synchronization set) by combining informations in I and Ψ. (In particular, I returns the synchronization set of a the carrier of a future, Ψ returns the cost of the pending invocations on actors of that set.)

The invocation of m_4 on actor w works analogously, where the associated synchronization set is $\{w\}$. As the call to m_2 at line 8 is invoked on z and $\mathcal{S}(y) = \{y, z\}$, we update Ψ_8 by appending the pair $\langle m_2, 0\rangle$ to $\Psi(\{y, z\})$, obtaining $\Psi_8(\{y, z\}) = e_4 \cdot \langle m_3, a\rangle \cdot \langle m_2, 0\rangle$. This accumulated cost expresses that the actors in $\{y, z\}$ have (i) two pending calls to be synchronized, namely m_3 and m_2; (ii) the time distance between these calls is a; and (iii) the first invocation on this synchronization set $\{y, z\}$ is performed at time e_4. Observe that the time distance a between m_3 and m_2 is set at line 6 by applying case (1) of `translate` function.

There are four synchronizations in this example, namely, $i^{\checkmark}$ at line 9, $h^{\checkmark}$ at line 11, and $f^{\checkmark}$, $g^{\checkmark}$ at line 12. Since $\Psi_8(\{y, z\}) = e_4 \cdot \langle m_3, a\rangle \cdot \langle m_2, 0\rangle$, the

synchronization $i^{\checkmark}$ may have to wait for the termination of the calls to both m_2 and m_3. The function `translate` applies case (5) at line 9, which gives e_9 as the maximum between (i) the cost local to the carrier, that is, $e_8 + m_1$, where m_1 corresponds to a task invoked on the carrier's synchronization set, which is pending to be synchronized and (ii) the expression $e_4 + max(m_3, a) + max(m_2, 0)$, which consists of three parts: (a) e_4 which is the starting time of the first call to an actor in $I(g) = \{y, z\}$, (b) the maximum between the cost of m_2 and the distance a between the invocations of m_3 and m_2, and (c) the time needed to compute m_2. Observe that $\{y, z\}$ is removed from Ψ_9, as the cost of the invocations on these two actors have already been calculated. Similarly, $I(i)$ and $I(g)$ are removed from I_9. Consequently, the synchronizations $f^{\checkmark}$ and $g^{\checkmark}$ at line 12 do not affect the cost expression. Observe also that e_9 is kept *in parallel* with the synchronization set $\{w\}$ in the translation environment, which allow us to calculate the correct costs of synchronizations in subsequent steps. The application of case (1) at line 10 increases the cost local to the carrier to $e_{10} = e_9 + b$, and updates Ψ_{10} by adding b to the time distance in the pair $\langle m_4, b\rangle$ and to the parallel cost e_9. The synchronization $h^{\checkmark}$ at line 11 amounts to the maximal value of three elements: (a) e_{10} which is the local cost before the synchronization; (b) the cost expression $e_6 + max(m_4, b)$, which is the sum of the first invocation on an actor in $\{w\}$ and the cost of method m_4; and (c) the cost $e_9 + b$ which is parallel to the invocation of m_4. Finally, the application of function `translate` at line 12 does not produce any effect. Thus, the cost of the *main* method in Figure 3, e_{main} is the cost expressed by e_{12}, that is:

$$e_{main} = \begin{array}{l} max(k_0 + a + max(b, k_4), \\ \qquad max(k_0 + max(a, k_3 + k_4) + k_2, a + k_0 + k_1) + b) \end{array}$$

■

4.3. Properties of the translation

The correctness of our system relies on the property that transitions may only consume time, i.e., the cost of a configuration never increases during a transition. This means that the cost of the initial configuration is the largest one; therefore, the cost of the program over-approximates the actual cost of every computation.

Cost programs. A *cost program* is a tuple $(m_1(\overline{n}_1) = e_1, \cdots, m_h(\overline{n}_h) = e_h, e)$, where m_i are function names, $\overline{n}_i$ integer formal parameters, and e_i and e Presburger arithmetics expressions. The *solution* of the above program is the evaluation of e.

Definition 4.2 (Cost of $\mathcal{P}$)**.** Let

$$\mathcal{P} = \Big(\mathtt{m}_1(x_1, \overline{y}_1, \overline{n}_1) = \nu\overline{z}_1\,;s_1, \cdots, \mathtt{m}_h(x_h, \overline{x}_h, \overline{n}_h) = \nu\overline{z}_h\,;s_h, \nu\overline{z}\,;s\Big)$$

be an `alt` program. For every $1 \leq i \leq h$, let

1. $\mathcal{S}_{\mathtt{m}_i} = \mathtt{sschem}(\{\{x_i, \overline{y}_i\}\}, s_i)$,

2. $m_i(\overline{n}_i) = e_i$, where $\texttt{translate}_{\mathcal{S}_{\mathtt{m}_i}}(\varnothing, \varnothing, x_i, 0, 0, s_i) = (I_i, \Psi_i, e'_i, e_i)$,

3. $\mathcal{S} = \texttt{sschem}(\{\varnothing\}, s)$ and $\texttt{translate}_{\mathcal{S}}(\varnothing, \varnothing, x, 0, 0, s) = (I, \Psi, e', e)$.

Let also $eq(\mathcal{P})$ be the cost program $(m_1(\overline{n}_1) = e_1, \cdots, m_h(\overline{n}_h) = e_h, e)$. A *cost solution* of $\mathcal{P}$ is a solution of $eq(\mathcal{P})$.

Theorem 4.1 (Correctness)**.** *Let $\mathcal{P}$ be an* `alt` *program and $\mathcal{U}$ be a solution of $eq(\mathcal{P})$. Let also cn be the initial configuration of $\mathcal{P}$. If $cn \Rightarrow^* cn'$ then $time(cn') \leq \mathcal{U}$.*

The proof of this theorem is in the Appendix. It relies on (i) the extension of `translate` function to runtime configurations, on defining the cost of a computation $cn \Rightarrow^* cn'$, noted $time(cn \Rightarrow^* cn')$ to be the sum of the labels of transitions, and on verifying that if $\mathcal{U}$ is a solution of $\texttt{translate}(cn)$ and $cn \Rightarrow^* cn'$ then $\mathcal{U} - time(cn \Rightarrow^* cn')$ is a solution of $\texttt{translate}(cn')$. To this aim, we reason about by induction on the length of $cn \Rightarrow^* cn'$ and we are reduced to two basic cases, according to the type of the last transition in $cn \Rightarrow^* cn'$:

1. either $cn'' \rightarrow cn'$; assuming that $\mathcal{U}$ is a solution of $\texttt{translate}(cn'')$, then it is also a solution of $\texttt{translate}(cn')$, see Lemma Appendix A.16;

2. or $cn'' \stackrel{t}{\rightarrow} cn'$; assuming that $\mathcal{U}$ is a solution of $\texttt{translate}(cn'')$, then $\mathcal{U} - t$ is also a solution of $\texttt{translate}(cn')$, see Lemma Appendix A.15.

The proof is technically complex because we have to deal with a lot of technical details, such as managing the synchronous sets at runtime and extracting the time difference in the cost equations when $cn'' \stackrel{t}{\rightarrow} cn'$ (which is not easy when costs are stored in Ψ).

It is important to observe that Theorem 4.1 is demonstrated using the (theoretical) solution of cost equations in [8]. This allows us to circumvent possible errors in implementations of the theory, such as `CoFloCo` [8] or `PUBS` [2]. As a byproduct of Theorem 4.1, we obtain the correctness of our technique, modulo the correctness of the solver.

5. The prototype

The analysis technique proposed in this paper has been prototyped [1]. In this section, we are going to discuss a number of technical details about converting cost equations in Section 4 to an adequate format for `PUBS` [2], which is an equation solver. We also present a preliminary evaluation of our prototype by discussing a number of examples and compare our analysis with the parallel cost analysis in [4].

5.1. The conversion of cost equations

The `translate` function returns a set of equations that *need to be adapted* before inputting them in `PUBS`. In particular, the equations used by this tool have the form `eq(name,cost,calls,size)`, where `name` is the equation name, `cost` the value produced by this equation, `calls` a list of equations called from `name` and `size` the size relations needed to compute the closed form of the upper bound (see [2] for details). In one equation, `PUBS` accumulates the cost `cost` and `calls`, thus we cannot express $max(m_1, m_2)$ in just one equation, because the cost of m_1 and m_2 will be added together. We deal with expressions of the form $max(m_1, \ldots, m_n)$ by producing as many equations as the number of elements we have in the max-expression. For instance, given an expression $e = max(e_1, \ldots, e_n)$, we produce the following set of equations $e = e_1; e = e_2; \ldots, e = e_{n-1}; e = e_n$. According to the definition of `translate`, we can find max expressions all over the cost expressions. Thus, `PUBS` requires an adaptation of the cost expressions to adequately handle them. Section 4.2 describes the application of `translate` function to all statements of a method. As we have seen in Example 4.2, the cost produced by the application of `translate` to each statement can be expressed in terms of its previous executions (see the right of Figure 5). The following example illustrates the production of the equations according to the application of `translate` function shown in Figure 5.

Example 5.1. Figure 6 outputs the equations corresponding to `translate` function shown in Figure 5. The solver uses $nat(e)$ to refer to $max(e, 0)$ and avoid negative values (see equations `main_6` and `main_10`), and $c(k)$ for constant values (see equations `main_3`, `m1`, `m2`, `m3` and `m4`). As in `translate`function, the cost expressions are composed step by step by invoking the previous cost equation, as shown in equations from `main_2` to `main_12`. For instance, expressions like $e_6 = e_5 + a$ are captured by the equation

```
eq(main_6(A,B),nat(A),[main_5(A,B)],[]).
```

New cost is added when necessary, e.g., `main_3`, which sums the cost of `main_2` and `c(k0)`, or `main_6`, which sums the cost of `main_5` and `nat(A)`. The computation of max-expressions is done by producing two equations. For instance, the cost expression

$$e_9 = max(e_8 + m_1, e_4 + max(m_3, a) + max(m_2, 0))$$

at line 9 in Figure 5 is captured by the two `main_9_max` equations in Figure 6, which respectively includes the cost `main_8` + `m1`, and the cost of `main_4` + `main_9_max1` + `main_9_max2`. Note that, analogously, `main_9_max1` and `main_9_max2` corresponds to the maximums $max(m_3, a)$ and $max(m_2, 0)$, respectively. Similarly, three equations `main_11_max` are needed to compute max-expression of e_{11} at line 11 in Figure 5. By solving the equations shown in Figure 6 we get the expression:

```
eq(e_main(A,B),0,[main_12(A,B)],[]).

eq(main_2(A,B),0,[],[]).
eq(main_3(A,B),c(k0),[main_2(A,B)],[]).
eq(main_4(A,B),0,[main_3(A,B)],[]).
eq(main_5(A,B),0,[main_4(A,B)],[]).
eq(main_6(A,B),nat(A),[main_5(A,B)],[]).
eq(main_7(A,B),0,[main_6(A,B)],[]).
eq(main_8(A,B),0,[main_7(A,B)],[]).

eq(main_9(A,B),0,[main_9_max(A,B)],[]).
eq(main_9_max(A,B),0,[main_8(A,B),m1(A)],[]).
eq(main_9_max(A,B),0,[main_4(A,B), main_9_max1(A,B), main_9_max2(A,B)],[]).

eq(main_9_max1(A,B),0,[m3(A)],[]).
eq(main_9_max1(A,B),nat(A),[],[]).

eq(main_9_max2(A,B),0,[m2(A)],[]).
eq(main_9_max2(A,B),0,[],[]).

eq(main_10(A,B),nat(B),[main_9(A,B)],[]).

eq(main_11(A,B),0,[main_11_max(A,B)],[]).
eq(main_11_max(A,B),0,[main_10(A,B)],[]).
eq(main_11_max(A,B),0,[main_6(A,B),main_11_max1(A,B)],[]).
eq(main_11_max(A,B),nat(B),[main_9(A,B)],[]).

eq(main_11_max1(A,B),0,[m4(A)],[]).
eq(main_11_max1(A,B),nat(B),[],[]).

eq(main_12(A,B),0,[main_11(A,B)],[]).

eq(m1(A),c(k1),[],[]).
eq(m2(A),c(k2),[],[]).
eq(m3(A),c(k3),[m4(A)],[]).
eq(m4(A),c(k4),[],[]).
```

Figure 6: Equations from the `translate` function Figure 5

$$
\begin{array}{l}
max([nat(A) + c(k_0) + max([nat(B), c(k_4)]),\\
\quad nat(B) + max([c(k_0) + max([nat(A), c(k_3) + c(k_4)]) + c(k_2),\\
\qquad\qquad nat(A) + c(k_0) + c(k_1)])])
\end{array}
$$

which reflects the cost expression e_{main} computed in Figure 5 and captures the overall cost of *main* in Figure 3. ■

5.2. Experimental evaluation

The prototype can be experimented online[1]. To deliver preliminary assessments, we have written our programs in a language that is an extension of `alt` – the `ABS` language [14], where only the features presents in `alt` are used). The experiments, whose source codes are available in the tool web site, model a number of typical concurrent and distributed scenarios:

[1] `http://costa.ls.fi.upm.es/timeanalysis`

Program	t_a	t_s	#eq	Upper bound
NWorkers (t1,t2,lt)	1	194	119	`max(t2,lt,t1)`
NWorkersMultiWorks (t1,t2,t3,t4)	5	591	231	`t1+t2+t3+t4`
NWorkersPartSync (t1,t2,lt)	1	73	57	`lt+max(lt+lt,t2,lt+max(t1,lt))`
NWorkersDelayed (t1,t2,t3,t4,lt)	1	237	87	`t1+max(lt,t2)+t3+t4`
ProdCons (pt,ct)	1	132	118	`max(ct+max(pt+pt,ct+pt))`
MapReduce (mt1,mt2,lt1,rt1,rt2,lt2)	2	3665	1330	`lt1+max(mt2,mt1)+lt2+max(rt1,rt2)`
MapRedSubWorkers(mt,rt,sm1,sm2,sr1,sr2)	3	11059	1378	`mt+max(sm1,sm2)+rt+max(sr1,sr2)`

Figure 7: Experimental results (times in ms)

NWorkers(t1,t2,lt) models the standard *fork-join* pattern where multiple workers execute in parallel on different actors parts of a larger task, represented by *wait*(*t1*) and *wait*(*t2*), respectively, and the carrier spends `lt` time units in parallel to the workers.

NWorkersMultiWorks(t1,t2,t3,t4) models a similar scenario but spawning four tasks per worker, with durations t1, t2, t3, t4 and synchronizes all of them at the end of the main method.

NWorkersPartSync(t1,t2,lt) models a *fork-join* pattern with two workers, but the launcher spends lt time units, between the synchronizations;

NWorkersDelayed(t1,t2,t3,t4,lt) also models a *fork-join* pattern with two workers that execute four tasks with durations t1, t2, t3, t4. In this case, the launcher spends lt time units between spawning the second and the third task.

ProdCons(pt,ct) models a *producer-consumer* scenario where the production and the consumption might occur in parallel, taking pt and ct time units for producing and consuming, respectively.

MapReduce(mt1,mt2,lt1,rt1,rt2,lt2) models the *map-reduce* algorithm where the *map* tasks, which take mt1 or mt2, are spawned on multiple workers in parallel. When all *map* tasks are finished, after lt1 time units for modelling the *map* result processing, the *reduce* tasks, which take rt1 or rt2, are spawned on the worker. Then after the synchronizations, the main methods spends lt2 time units to process the results.

MapReduceSubWorkers(mt,rt,sm1,sm2,sr1,sr2) models the *map-reduce* algorithm where the workers share the task, taking times mt for *map* and rt for *reduce*, and launch subtasks with times sm1, sm2 for *map* and sr1, sr2 for *reduce* to other sub-workers.

We have executed the above codes on an Intel Core i7-7500U CPU 2.70GHz with 64Gb of RAM, running Debian 9.3. Table 7 summarizes the upper bounds obtained by means of our prototype. For the sake of clarity, we have simplified the expressions that are redundant, and we have replaced in the ta-

ble `nat(e)` by `e`. For instance, the expression returned for NWorkers, namely `max([nat(t2),nat(lt),nat(t1),nat(lt)])`, is written as `max([t2,lt,t1])`.

The results of Table 7 show that the execution time of our analysis is very short: less than 5 ms for all the programs. Even when the number of equations is high, such as in MapReduce or in MapReduceSubWorkers, the time required to solve them is rather short. Regarding to the precision of our analysis, the results highlight that we do not lose precision in *fork-join* patterns. In fact, in this case, we properly identify those parts that can execute in parallel and use the `max` to capture the parallelism. For instance, in NWorkers and NWorkersMultiWorks, our tool identifies the parts running in parallel and calculates the cost as the maximum cost needed among the workers, independent from the number of workers executing simultaneously (in this programs we have four different workers). Similarly, as highlighted by the programs MapReduce and MapReduceSubWorkers, our tool identifies that *map* and *reduce* tasks might execute in parallel before their synchronization. In this case, their costs are added only once to the total cost inside the two maximum expressions `max(mt2,mt1)` and `max(rt2,rt1)`.

5.3. Tools comparison

In the following, we compare the results obtained by our approach and by the analysis described in [4].

The analysis of [4] uses a language – the `ABS` language [14] – that is actually a superset of `alt`. In particular, in order to solve the syntactic mismatches, we had to modify the $wait(e)$-statement into loops with e iterations so as to model the time progress. Additionally, we ignore the constant times output by [4] as they are not considered by our time analysis.

The table illustrates the results obtained by applying our time analysis and the analysis of [4] to the `alt` programs in Figure 1.

Prog.	**Time Analysis**	**Analysis of [4]**
$main_1$	k_0+max(a,k_1)+max(b,k_2)	max(k_0+a+b,k_0+k_1+k_2+a)
$main_2$	k_0+max(a+b+k_2,k_1)	k_0+max(k_1+k_2,a+b+k_2)
$main_3$	k_0+max(a,k_2+k_3)+max(b,k_1)	k_0+max(k_0+a+b,k_0+k_1+k_2+k_3,k_0+a+k_1+k_2)

The reader may notice that the results obtained by our time analysis are more precise than [4] for the studied programs. To explain this point, we need to recall the technique of [4], which uses *block-level cost-centers* to compute the cost expression of the *sequential parts* of the program. A *distributed flow graph* of the program is then generated to express tasks that might be executed in parallel and their flow relations. The distributed flow graph is used to find all the paths that start from the initial node and end in any possible final node. The cost expressions of every paths is computed accordingly and the overall cost, called the *parallel cost* of the program, is simply the maximum of the costs. The critical point of the technique in [4] is that, in order to take into account every possible scheduler's choice, the distributed flow graph collects edges that connect

the beginning and the end of those methods that might be simultaneously in the queue of pending tasks. These edges introduce cycles in the graph; hence, the collection also includes those paths where parallel methods are sequentialized. This results in a loss of precision. For instance, this precision loss is evident in $main_1$, where the cost of `a` and $\texttt{k}_1$ are not considered to happen in parallel. Similarly, in $main_2$, where the costs $\texttt{k}_1$ and $\texttt{k}_2$ are added; or in $main_3$, where $\texttt{k}_2$ should not be added to the expression $\texttt{k}_0\texttt{+a+}\texttt{k}_1\texttt{+}\texttt{k}_2$.

Regarding the benchmarks of Figure 7, the analysis of [4] gives results similar to our tool, except for NWorkersDelayed. In fact, this program is the only one spends some time between two invocations on the same actor. For this program, the upper bound expression obtained by [4] is `lt+t1+t2+t3+t4`, which sequentialize the cost of all the methods. On the contrary, our tool is able to detect that `lt` and `t2` are costs of two parallel methods. Thus, our analysis does not sequentialized these two costs, but returns their maximal value, which in turn produces the expression `t1+max(lt,t2)+t3+t4`.

6. Related work

Static time analysis techniques for concurrent programs follow two main approaches: those based on type-and-effect systems and those based on abstract interpretation.

Type-and-effect systems [10] (i) collect constraints on type and resource variables and (ii) solve these constraints by means of an off-the-shelf solver [2, 8]. Recent work has applied type-based amortised analysis for deriving upper bounds of parallel first-order functional programs [13]. This work differs from our approach in the concurrency model, as they do not handle references to actors in the programs and there is no distinction between blocking and non-blocking synchronization.

In this paper we do not perform constraint collection on type and resources. This because `alt` is intended to be a behavioural type language; therefore we directly use `alt` to define a (compositional) analysis that returns cost equations. The main difference with respect to the analysis of [10] is that now we are able to compute the cost of functions that contain invocations on arguments, namely on actors that are already alive before the function invocations. An approach similar to our one has been proposed in [16] for verifying safety properties of sequential languages.

Abstract interpretation techniques addressing domains carrying quantitative information, such as resource consumption, have been proposed in the literature – see references in [17]. Consequently, several well-developed automatic solvers for cost analysis already exist. These solvers either use finite domains or use expedients (widening or narrowing functions) to guarantee the termination of the fix-point generation. For this reason, solvers often return inaccurate answers when fed with systems that are finite but not statically bounded. Among the others, [4] defines a technique based on abstract interpretation that targets a language with a similar concurrency model as presented in this paper. We have discussed the technique of [4] in Section 5, where we have also analyzed the

differences with our technique. We recall that our technique returns more precise costs for programs that spawn several invocations without synchronization on the same synchronization set. In particular, [4] manifests a sensible loss of precision when cycles appear in the distributed flow graph, as all nodes in the cycle will be part of the path that leads to the maximum upper bound. We also observe that, since the technique in [4] is not compositional, it does not require any management of synchronization sets, which entangles a lot our technique.

7. Conclusions

This paper presents a technique for computing the time of concurrent programs. We have defined `alt`, an actor calculus that is intended to be a compilation target for concurrent languages featuring actor creation, task invocation and synchronization. In particular, `alt` features an explicit cost annotation that defines the number of machine cycles required before executing the continuation, which abstracts away the actual computation activities of the program. The computational cost is then measured by introducing the notion of (strong) *t-stability* (*cf.* Definition 2.1), which represents the ticking of time and expresses that no control activity is possible up to t time units. In order to relate actors that might potentially delay executions of other actors, we introduce the notion of synchronization sets. Then we define a `translate` function that uses synchronization sets to compute a cost equation function for each method definition. We have also proven that our approach is sound with respect to the actual computational cost (*cf.* Theorem 4.1). The analysis has been prototyped and experimented, which shows that our approach produces accurate over-approximation.

Overall, our technique allows one to estimate the computational time of the source program by computing the cost of the target actor program. The aim is to use our technique in a cloud computing setting because `alt` terms might be considered as abstract descriptions of services and the estimation of the computational time may be used for enforcing the compliance with the service level agreement contract. In this context, the cost expressions in $wait(\cdot)$-statement, might be defined by means of a worst-case execution time analysis [5]

Future lines of work must consider the possibility of obtaining more precise information about *synchronization sets* of actors. In this paper, these sets are computed for method bodies by simply accumulating informations, despite of the fact that two actors, in a stage of the computation may affect each other, while in another stage they are independent. In fact, a more appropriate notion would have been that of *sequence of synchronization sets*. While such improvement will end up in more precise results of the cost analysis, it is not clear how much more complex the theoretical developments will become. Perhaps a manageable extension will be a notion in between the synchronization sets and the sequences of such sets.

In this paper, the cost of a method includes that of the asynchronous invocations in its body because every invocation is synchronized. This constraint does not permit to trigger methods such as drivers, daemons, etc. without waiting for

their termination. In order to take care of costs of unsynchronized methods, one should use *continuations* and compose costs according to the synchronization set they correspond. We will address this extension in a future work.

References

[1] Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G., 2014. SACO: Static Analyzer for Concurrent Objects. In: Proceedings of TACAS 2014. Vol. 8413 of LNCS. Springer, pp. 562–567.

[2] Albert, E., Arenas, P., Genaim, S., Puebla, G., 2011. Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46 (2), 161–203.

[3] Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D., 2012. Cost analysis of object-oriented bytecode programs. Theoretical Computer Science 413 (1), 142–159.

[4] Albert, E., Correas, J., Johnsen, E. B., Román-Díez, G., 2015. Parallel cost analysis of distributed systems. In: Proceedings of SAS 2015. Vol. 9291 of LNCS. Springer, pp. 275–292.

[5] Blazy, S., Maroneze, A., Pichardie, D., 2013. Formal Verification of Loop Bound Estimation for WCET Analysis. In: Proceedings of VSTTE'13. Vol. 8164 of LNCS. Springer, pp. 281–303.

[6] Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J., 2014. Alternating runtime and size complexity analysis of integer programs. In: Proceedings of TACAS 2014. Vol. 8413 of LNCS. Springer, pp. 140–155.

[7] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., Brandic, I., 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Comp. Sys. 25 (6), 599–616.

[8] Flores Montoya, A., Hähnle, R., 2014. Resource analysis of complex programs with cost equations. In: Proceedings of APLAS 2014. Vol. 8858 of LNCS. Springer, pp. 275–295.

[10] Giachino, E., Johnsen, E. B., Laneve, C., I Pun, K., 2016. Time complexity of concurrent programs. In: Proceesdings of FACS 2015. Springer, pp. 199–216.

[11] Gulwani, S., Mehra, K. K., Chilimbi, T., 2009. Speed: precise and efficient static estimation of program computational complexity. In: ACM SIGPLAN Notices. Vol. 44. ACM, pp. 127–139.

[12] Hoffmann, J., Aehlig, K., Hofmann, M., 2012. Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. 34 (3), 14.

[13] Hoffmann, J., Shao, Z., 2015. Automatic static cost analysis for parallel programs. In: Vitek, J. (Ed.), Programming Languages and Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 132–157.

[14] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M., 2011. ABS: A core language for abstract behavioral specification. In: Proceedings of FMCO 2010. Vol. 6957 of LNCS. Springer, pp. 142–164.

[15] Laneve, C., Zavattaro, G., 2005. Foundations of web transactions. In: Proceedings of FOSSACS 2005. Vol. 3441 of LNCS. Springer, pp. 282–298.

[16] Morrisett, G., Walker, D., Crary, K., Glew, N., 1999. From system f to typed assembly language. ACM Trans. Program. Lang. Syst. 21 (3), 527–568.

[17] Trinder, P. W., Cole, M. I., Hammond, K., Loidl, H., Michaelson, G., 2013. Resource analyses for parallel and distributed coordination. Concurrency and Computation: Practice and Experience 25 (3), 309–348.

[18] Wegbreit, B., 1975. Mechanical program analysis. Communications of the ACM 18 (9), 528–539.

Appendix A. Appendix

We introduce this appendix with some helpful notation related to the cost equation of an `alt` program, and with few algebraic notation on graphs and equivalence relations that will be helpful to define synchronization schema for runtime configurations.

Definition Appendix A.1. In this appendix, to simplify future notations, we extend the notion of expression as follows:

$$e \ ::= \quad | \ x \ | \ f \ | \ e + e \ | \ \mathtt{m}(\overline{n}) \ | \ \max(\overline{e})$$

The parameters $\mathcal{S}$, I, Ψ, x, e', e and s of the `translate` function are *consistent* iff: $\mathcal{S}$ is an equivalence relation on a set including all free actor names in s and x; all free future names in s are in $\mathrm{dom}(I)$; all the synchronization sets in $\mathrm{im}(I)$ are in $\mathcal{S}$;[2] and all the synchronization sets in $\mathrm{im}(I)$, minus $\mathcal{S}(x)$, are in $\mathrm{dom}(\Psi)$.

Suppose given consistent parameters $\mathcal{S}$, I, Ψ, x, e', e and s. We write $\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s)$ the cost of the translation of s:

$$\mathtt{translate}_{\mathcal{S}}(I, \Psi, x, e', 0, s) = (I', \Psi', e_1, e_2) \Leftrightarrow \mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s) = e_2$$

Definition Appendix A.2. Suppose given any two expressions e_1 and e_2. We write $e_1 \leq e_2$ iff for all substitution Σ with $fv(e_1) \cup fv(e_2) \subseteq \mathrm{dom}(\Sigma)$ and $\mathrm{im}(\Sigma) \subseteq [0..\infty]$, we have $\Sigma(e_1) \leq \Sigma(e_2)$.

The following definition is a small adaptation of Definition 4.2 where the cost of the main statement of the program, instead of being by itself, is mapped to a name f_{start}. That way, a cost program becomes a simple set of equations, that is easier to manipulate in the proofs (see Lemma Appendix A.14).

Definition Appendix A.3. Suppose given a set of functions $\overline{FD} = \mathtt{m}_1(\overline{x}_1) = s_1, \cdots, \mathtt{m}_n(\overline{x}_n) = s_n$. The *cost program eq* of an `alt` program $P = (\overline{FD}, s_{main})$ is defined as follows:

$$eq(P) \stackrel{def}{=} [f_{start} \mapsto \mathtt{T}_{\mathtt{sschem}(\{\ \{start\}\ \}, s_{main})}(\varnothing, \varnothing, start, 0, 0, s_{main})] \cup \bigcup_i [\mathtt{m}_i(\overline{x}_i) \mapsto \mathtt{translate}(\mathtt{m}_i(\overline{x}_i) = s_i)]$$

A *cost solution* Σ of an `alt` program P is a solution for the equation $eq(P)$.

Definition Appendix A.4. Given a set V and a relation R on V, we write $R^{eq(V)}$ the transitive, reflexive and symmetric closure of R in V. Note that we will write R^{eq} when the set V is clear from the context. Given a set V and an equivalence relation R on V, we write $\tilde{R}$ the partition of V raised by R. Given a

[2] we note $\mathrm{im}(I)$ the image of I, i.e., $\mathrm{im}(I) = \{I(f) \mid f \in \mathrm{dom}(I)\}$

graph $G = (V, E)$ and an equivalence relation R on V, we write $G/\!\!/^{*}R$ the graph $G' = (V', E')$ such that:

$$V' \stackrel{def}{=} \tilde{R} \qquad E' \stackrel{def}{=} \{(A, A') \mid A \neq A' \in R, \exists x \in A, y \in A', (x, y) \in E\}$$

In the rest of the appendix, we write: $parents(G, x)$ for the set of the parent nodes of x in G, i.e., $parents(G, x) \stackrel{def}{=} \{y \mid (y, x) \in E\}$; $paths(G)$ for the set of non-empty paths p in the directed graph G; $start(p)$ for the start node of the path p; $end(p)$ for the end node of the path p; and $nodes(p)$ for the set of nodes traversed by the path p.

Appendix A.1. Runtime Analysis

In this section, we extend the translation function of Fig. 4 to the runtime terms of `alt`. This extension maps all tasks in the runtime to a cost equation encoding an upper bound of its finishing time. In order to do so, we need to define the different parameters of the `translate` function. First of all, let us introduce the necessary notations:

Definition Appendix A.5. Suppose given a runtime configuration cn. We write $x \in cn$ iff there exists p, q and cn' such that $cn = act(x, p, q)\ cn'$. If $cn = act(x, p, q)\ cn'$, we write $cn(x)$ for the configuration $act(x, p, q)$. If $cn = act(x, p, q)\ cn'$, we write $(x : f) \in cn$ iff there exists $s; f \in p\ q$ or if $cn' = invoc(x, f, \mathtt{m}, \overline{v})\ cn''$; we write $f \to f' \in cn$ iff there exists $s; f \in p\ q$ and $f'^{\checkmark}$ is a statement in s; and we write $f \to x' \in cn$ iff there exists $s; f \in p\ q$ such that x' is free in s or if there exists f' suhc that $f \to f' \in cn$ and $(x' : f') \in cn$. Finally, we write $\mathtt{roots}(cn)$ for the set of root futures of cn, i.e., $\mathtt{roots}(cn) = \{f \mid \nexists f',\ f' \to f \in cn\}$; and $A(cn)$ for a set of actor names A, which extracts from cn all actors named in A, with their futures $fut(f, val)$ and invocation messages $invoc(x, f, \mathtt{m}, \overline{v})$.

Synchronization Schemas. The first parameter, the synchronization schema, has a central role in the cost computation, as it captures the actors that may reciprocally affect the queues of each other by means of function invocations and synchronizations. Without such information, the cost computation cannot capture all the tasks that could occur in an actor's process queue, and thus would be erroneous. At static time, such schemas are computed using the parameters of method calls. However, at runtime, this information is missing as running code does not refer back to their originating method and parameters. The way we can recreate information captured by a synchronization schema at runtime is to follow the point-to relation between objects. We recall the point-to relation between object in the following definition:

Definition Appendix A.6. The *point-to* relation of a runtime configuration cn, noted $point\text{-}to(cn)$, is a directed graph (V, E) where $V \stackrel{def}{=} \{x, f \mid (x : f) \in cn\}$; $E \stackrel{def}{=} \{(x, f) \mid (x : f) \in cn\} \cup \{(f, x) \mid f \to x \in cn\}$.

With this, we can construct the synchronization schema of a runtime configuration. Following Definition 4.1, synchronization sets actually capture a notion slightly more subtle than simply affecting each other's queue. Firstly, synchronization sets do not include the obvious parent-son object relation, as this would be a too restrictive approximation that would consider any concurrent computation as put in sequence. Secondly, synchronization sets put in relation not only objects that may influence each other, but also objects that refer to influencing objects: the synchronization set somehow consider the transitive closure of the point-to relation. Finally, we also need to include in the synchronization sets the dependencies that might be created by future method calls. These considerations lead to the following definition of the synchronization sets for runtime configuration.

Definition Appendix A.7. Given a graph $G = (V, E)$, we define the relations H and L on V as follows:

$$x \; H \; y \Leftrightarrow \{y\} \subsetneq \mathit{parents}(G, x)$$
$$x \; L \; y \Leftrightarrow \exists p \in \mathit{paths}(G),\, \mathit{start}(p) = \mathit{end}(p) \wedge \{x, y\} \in \mathit{nodes}(p)$$

The *synchronization graph* of a runtime configuration cn, written G_{cn} is defined as follows:

$$G_{cn} \stackrel{def}{=} (\{\mathrm{flat}(S) \mid S \in V\}, \{(\mathrm{flat}(S), \mathrm{flat}(S')) \mid (S, S') \in E\})$$
$$\text{where} \begin{cases} (V, E) = \mathit{point\text{-}to}(cn) \overset{*}{/} H^{eq} \overset{*}{/} F^{eq} \overset{*}{/} L^{eq} \\ v \; F \; v' \Leftrightarrow x \in v, f \in v', (x : f) \in cn \\ \mathrm{flat}(M) = \bigcup_{N \in M} \bigcup_{O \in N} O \cap X \\ X = \{x \mid x \in cn\} \end{cases}$$

The *synchronization schema* of a runtime configuration cn, written $\mathcal{S}_{cn}$ is the set of vertices of G_{cn}.

Note that, by construction, $\mathcal{S}_{cn}$ is a partition of the object names in cn. Hence, we can use the same notation as in Section 4 by writing $\mathcal{S}_{cn}(x)$ for the set $A \in \mathcal{S}_{cn}$ containing x.

Future Localization. The parameter I is the simplest to define, now that we have a precise definition of synchronization schema. Indeed, this parameter simply maps the futures living in the configuration to its actor's synchronization set. Given a runtime configuration cn, we define:

$$I_{cn} \stackrel{def}{=} [f \mapsto \mathcal{S}_{cn}(x) \mid x \in cn, f \in cn(x)]$$

Translation Environments. The `translate` function has a last important parameter: the environment Ψ giving the accumulated cost of every running synchronization set. At runtime, waiting statements are not only bound to numerical expression or method calls, but partially to executed processes identified by futures. This motivates the expression extension we gave in the introduction of the appendix. During the static time analysis, and also in the runtime analysis,

Ψ contains an entry for all synchronization sets that have a non-empty process queue. Hence, this parameter is relevant only when translating a process, and collects data about the futures and synchronization sets related to the analyzed process.

The translation function. We present in Figure A.8 the translation function of runtime configurations into mappings from future names to cost equations. Note that when translating a specific process, we need to use a local future localizations I, which takes into account objects that are not created yet, and to create the corresponding translation environment Ψ. This is done with the following function:

$$[\mathcal{S}, I]^x_s \stackrel{def}{=} (\mathcal{S}', I', \Psi, e) \quad \text{with} \begin{cases} \mathcal{S}' = \texttt{sschem}(\mathcal{S}, s) \\ I' = [f \mapsto A \mid f \in \mathrm{dom}(I),\, I(f) \subset A,\, A \in \mathcal{S}'] \\ \Psi = [A \mapsto \langle \sum_{I(f)=A,\, f \in fv(s)} f, 0\rangle \mid A \in I'(fv(s)) \wedge x \notin A] \\ e = \sum_{\substack{f \in I^{-1}(\mathcal{S}(x)) \\ f \text{ free in } s}} f \end{cases}$$

$$\texttt{translate}_{\mathcal{S}}(I, cn) = \begin{cases} (1) & \varnothing & \text{when } cn = \varepsilon \\ (2.1) & \varnothing & \text{when } cn = \mathit{fut}(f, \bot) \\ (2.2) & \{f \mapsto 0\} & \text{when } cn = \mathit{fut}(f, \top) \\ (3) & \bigcup_{p' \in p, q} \texttt{translate}_{\mathcal{S}}(I, x, p') & \text{when } cn = \mathit{act}(x, p, q) \\ (4) & \{f \mapsto \mathtt{m}(x, \overline{v})\} & \text{when } cn = \mathit{invoc}(x, f, \mathtt{m}, \overline{v}) \\ (5) & \texttt{translate}_{\mathcal{S}}(I, cn_1) \cup \texttt{translate}_{\mathcal{S}}(I, cn_2) & \text{when } cn = cn_1 \ cn_2 \end{cases}$$

$$\texttt{translate}_{\mathcal{S}}(I, x, p) = \begin{cases} (6) & \varnothing & \text{when } p = 0 \\ (7) & \{f \mapsto \mathtt{T}_{\mathcal{S}'}(I', \Psi, x, e, 0, s)\} & \text{when } p = s\,;f \text{ and with } [\mathcal{S}, I]^x_s = (\mathcal{S}', I', \Psi, e) \\ (8) & \{f \mapsto 0\} & \text{when } p = f \end{cases}$$

Figure A.8: The translation of `alt` runtime terms

Definition Appendix A.8. Suppose given a set of functions $\overline{FD} = \mathtt{m}_1(\overline{x}_1) = s_1, \cdots, \mathtt{m}_n(\overline{x}_n) = s_n$. The *cost configuration eq* of a runtime configuration cn

based on the functions $\overline{FD}$, written $eq(cn, \overline{FD})$, is defined as follows:

$$eq(cn, \overline{FD}) \stackrel{def}{=} \texttt{translate}_{\mathcal{S}_{cn}}(I_{cn}, \Psi_{cn}, cn) \cup \bigcup_i [\mathtt{m}_i(\overline{x}_i) \mapsto \texttt{translate}(\mathtt{m}_i(\overline{x}_i) = s_i)]$$

A *cost solution* Σ of a runtime configuration cn based on the functions $\overline{FD}$, is a solution for the cost configuration $eq(cn, \overline{FD})$.

The following two lemmas give some properties about the image of a cost equation, so we will be able to manipulate them in the rest of the proofs. The first lemma states that a cost equation maps functions and futures to expressions e_x.

Lemma Appendix A.1. *Consider a consistent set of parameters $\mathcal{S}$, I, Ψ, x, e', e and s. Then, $\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s)$ is an expression. Consider additionally a runtime configuration cn and a set of functions $\overline{FD}$. Then, the image of $eq(cn, \overline{FD})$ is a set of expressions e''.*

Proof. Inspecting the rules in Figure 4, it is clear that e_r in $(I, \Psi, e'', e_r) = \texttt{translate}_{\mathcal{S}}(I, \Psi, x, e', e, s)$ is an expression. Consequently, by Definition Appendix A.1, $\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s)$ is an expression as well. Additionally, it is easy, looking at the rules in Figure A.8, and by Definition Appendix A.8, to see that the image of $eq(cn, \overline{FD})$ are all constructed from $\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', 0, s)$ where e' is a sum over futures. Hence, the image of $eq(cn, \overline{FD})$ is a set of expressions e'' (a sum of future is an expression). ■

Lemma Appendix A.2. *An expression e is always equal to some $\max(e_1, \ldots, e_n)$ where none of the e_i contain a* max *operator.*

Proof. This property is a consequence of the following equations that can move all the max functions in e to the top level:

$$\max(e_1, \ldots, e_n) + e = \max(e_1 + e, \ldots, e_n + e)$$
$$\max(\max(\overline{e}), \overline{e}') = \max(\overline{e}, \overline{e}')$$

■

In the rest of this appendix, we will always consider expressions e to be equivalent modulo the previous equalities that allow to move the max function around.

Appendix A.2. Upper Bound Correction

In this section, we prove that the cost we computed is a correct upper bound for the execution time of the program.

Appendix A.2.1. Properties of the Runtime Configuration's Graph Structure

Lemma Appendix A.3. *The synchronization graph of a runtime configuration cn is a forest.*

Proof. This is a direct consequence of merging in the same equivalence classes the nodes that break acyclicity in the point-to graph. ■

Lemma Appendix A.4. *Suppose given a runtime configuration cn,* $(x : f) \in cn$, $(x' : f') \in cn$ *and* $y \in cn$ *such that* $f \to y \in cn$, $f' \to y \in cn$ *and* $\mathcal{S}_{cn}(x) \neq \mathcal{S}_{cn}(y)$. *Then,* $x = x'$ *and* $f = f'$.

Proof. This is a direct consequence of the construction of the synchronization graph, where we quotient the point-to graph of *cn* by H, putting every actors accessible from different processes in the same equivalence class. ■

Lemma Appendix A.5. *Given a runtime configuration cn and a process* $p = s; f$ *such that* $cn = act(x, p', q)\ cn'$ *with* $p \in p'\ q$. *Then, for all* $A_1 \neq A_2 \in \mathcal{S}_{cn}$, *there exists* $A'_1 \neq A'_2 \in \texttt{sschem}(\mathcal{S}_{cn}, s)$ *with* $A_1 \subset A'_1$ *and* $A_2 \subset A'_2$.

Proof. This is directly proven by induction on s, and by the fact that $\mathcal{S}_{cn}$ is constructed by taking into account the dependency between actors in p. ■

Definition Appendix A.9. The *process graph* of a runtime configuration *cn* is a graph (V, E) such that $V = \{f \mid f \in cn(x), x \in cn\}$ and $E = \{(f, f') \mid f, f' \in V \wedge f \to f' \in cn\}$

Lemma Appendix A.6. *The process graph G of a runtime configuration cn is a tree.*

Proof. By induction on the reduction steps to reach *cn* from an initial configuration. Note that only the (Async-Call) reduction rule creates a link between futures; moreover, it creates this link to a fresh future name, so it is not possible to create loop. Hence, the process graph is a forest. Finally, because all futures must be synchronized, it is not possible to break a link towards a running process. Therefore, all nodes in the process graph are reachable. Thus, the process graph is a tree. ■

Appendix A.2.2. Properties of Runtime Evolution

Lemma Appendix A.7. *Suppose given* $\mathcal{S}$, I_1, I_2, Ψ, x, e, e', s *such that: the sets of parameters* $\mathcal{S}$, I_i, Ψ, x, e, e' *and s are consistent, for* $1 \leq i \leq 2$; *and for all f free in s, we have* $f \in \mathrm{dom}(I_1) \cap \mathrm{dom}(I_2)$ *and* $I_1(f) = I_2(f)$. *Then, we have*

$$\mathtt{T}_{\mathcal{S}}(I_1, \Psi, x, e, e', s) = \mathtt{T}_{\mathcal{S}}(I_2, \Psi, x, e_x, e'_x, s)$$

Proof. This is directly proven by induction on s, by remarking that all futures that are accessed in I_i (with $1 \leq i \leq 2$) are the ones free in s (rules (4) and (5) of Figure 4). ■

Lemma Appendix A.8. *Suppose given $\mathcal{S}_1$, $\mathcal{S}_2$, I_1, I_2, Ψ_1, Ψ_2, x, e_1, e_2, e'_1, e'_2 and s such that: the parameters $\mathcal{S}_i$, I_i, Ψ_i, x, e_i, e'_i, and s are consistent (for $1 \leq i \leq 2$). Moreover, suppose that: for all $A_1 \in \mathcal{S}_1$, there exists $A_2 \in \mathcal{S}_2$ such that $A_1 \subseteq A_2$; $\mathrm{dom}(I_1) \supseteq \mathrm{dom}(I_2)$, for all $f \in \mathrm{dom}(I_2)$ such that $I_1(f) \subseteq I_2(f)$; for all A_1 and A_2 such that $I_1^{-1}(A_1) \cap I_2^{-1}(A_2) \neq \varnothing$, we have $A_1 \subset A_2$ and $I_1^{-1}(A_1) \subseteq I_2^{-1}(A_2)$; $e_1 \leq e_2$ and $e'_1 \leq e_2$; and for all $A_1 \in \mathrm{dom}(\Psi_1)$, either:*

- *there exists $A_2 \in \mathrm{dom}(\Psi_2)$ such that $A_1 \subseteq A_2$ and $\Psi_1(A_1) \leq \Psi_2(A_2)$; or*
- *$A_1 \cap \mathcal{S}_2(x) = \varnothing$ and $[\![\Psi_1(A_1)]\!] \leq e_2$; or*
- *$A_1 \subseteq \mathcal{S}_2(x)$ and $[\![\Psi_1(A_1)]\!] \leq e_2 + e'_2$;*

Then, we have

$$\mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e'_1, e_1, s) \leq \mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e'_2, e_2, s)$$

Proof. We prove this result by induction on s:

(i) Case $s = \mathsf{0}$. By construction, we have the result:

$$\mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e'_1, e_1, s) = e_1 \leq e_2 = \mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e'_2, e_2, s)$$

(ii) Case $s = \nu x; s'$. By construction, we have that

$$\mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e'_1, e_1, s) = \mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e'_1, e_1, s')$$
$$\mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e'_2, e_2, s) = \mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e'_2, e_2, s')$$

Hence, with the induction hypothesis, we conclude the case.

(iii) Case $s = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e}); s'$ with $y \in \mathcal{S}_1(x)$ (by hypothesis, this implies that $y \in \mathcal{S}_2(x)$). By rule (2) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e'_1, e_1, s) = \mathtt{T}_{\mathcal{S}_1}(I_1[f \mapsto \mathcal{S}_1(x)], \Psi_1, x, e'_1 + \mathtt{m}(\overline{e}), e_1, s')$$
$$\mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e'_2, e_2, s) = \mathtt{T}_{\mathcal{S}_2}(I_2[f \mapsto \mathcal{S}_2(x)], \Psi_2, x, e'_2 + \mathtt{m}(\overline{e}), e_2, s')$$

Hence, with the induction hypothesis, we conclude the case.

(iv) Case $s = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e}); s'$ with $y \in \mathcal{S}_2(x) \setminus \mathcal{S}_1(x)$. Let us write $S_1 = \mathcal{S}_1(y)$. By rules (2) and (3) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e'_1, e_1, s) = \mathtt{T}_{\mathcal{S}_1}(I_1[f \mapsto S_1], \Psi'_1, x, e'_1, e_1, s')$$
$$\mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e'_2, e_2, s) = \mathtt{T}_{\mathcal{S}_2}(I_2[f \mapsto \mathcal{S}_2(x)], \Psi_2, x, e'_2 + \mathtt{m}(\overline{e}), e_2, s')$$
$$\text{with } \Psi'_1 = \Psi_1[S_1 \mapsto \varepsilon_1 \cdot \langle \mathtt{m}(\overline{e}), 0 \rangle]$$
$$\text{and} \quad \varepsilon_1 = \Psi_1(S_1) \text{ if } S_1 \in \mathrm{dom}(\Psi_1) \quad \varepsilon_1 = e_1 \text{ if } S_1 \notin \mathrm{dom}(\Psi_1)$$

It is straightforward to see that $\Psi'_1(S_1) \leq e_2 + e'_2 + \mathtt{m}(\overline{e})$ in all cases ($S_1 \in \mathrm{dom}(\Psi_1)$ or $S_1 \notin \mathrm{dom}(\Psi_1)$). Hence, with the induction hypothesis, we conclude the case.

(v) Case $s = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e})\,;s'$ with $y \notin \mathcal{S}_1(x) \cup \mathcal{S}_2(x)$. Let us write $S_1 = \mathcal{S}_1(y)$ and $S_2 = \mathcal{S}_2(y)$. By rule (3) of Figure 4, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e_1', e_1, s) = \mathtt{T}_{\mathcal{S}_1}(I_1[f \mapsto S_1], \Psi_1', x, e_1', e_1, s') \\
\mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e_2', e_2, s) = \mathtt{T}_{\mathcal{S}_2}(I_2[f \mapsto S_2], \Psi_2', x, e_2', e_2, s') \\
\quad \text{with } \Psi_1' = \Psi_1[S_1 \mapsto \varepsilon_1 \cdot \langle \mathtt{m}(\overline{e}), 0\rangle],\ \Psi_2' = \Psi_2[S_2 \mapsto \varepsilon \cdot \langle \mathtt{m}(\overline{e}), 0\rangle] \\
\quad \text{and } \left\{ \begin{array}{ll} \varepsilon_1 = \Psi_1(S_1) \text{ if } S_1 \in \mathrm{dom}(\Psi_1) & \varepsilon_1 = e_1 \text{ if } S_1 \notin \mathrm{dom}(\Psi_1) \\ \varepsilon_2 = \Psi_2(S_2) \text{ if } S_2 \in \mathrm{dom}(\Psi_2) & \varepsilon_2 = e_2 \text{ if } S_2 \notin \mathrm{dom}(\Psi_2) \end{array} \right.
\end{array}$$

By hypothesis, we have three cases:

(a) Case $S_1 \notin \mathrm{dom}(\Psi_1)$ and $S_2 \notin \mathrm{dom}(\Psi_2)$. We thus have that $\varepsilon_1 = e_1 \leq e_2 = \varepsilon_2$.

(b) Case $S_1 \in \mathrm{dom}(\Psi_1)$ and $S_2 \notin \mathrm{dom}(\Psi_2)$. We thus have that $\varepsilon_1 = \Psi_1(S_1) \leq e_2 = \varepsilon_2$.

(c) Case $S_1 \in \mathrm{dom}(\Psi_1)$ and $S_2 \in \mathrm{dom}(\Psi_2)$. We thus have that $\varepsilon_1 = \Psi_1(S_1) \leq \Psi_2(S_2) = \varepsilon_2$.

In all cases, we have that $\Psi_1'(S_1) \leq \Psi_2'(S_2)$. Hence, with the induction hypothesis, we conclude the case.

(vi) Case $s = f^{\checkmark}\,;s'$ with $x \in I_1(f)$ and $f \in \mathrm{dom}(I_2)$ (by hypothesis, this implies that $x \in I_2(f)$). By rule (4) of Figure 4, we have that

$$\begin{array}{rl}
\mathtt{T}_{\mathcal{S}_i}(I_i, \Psi_i, x, e_i', e_i, s) = \mathtt{T}_{\mathcal{S}_i}(I_i \backslash F_i, (\Psi_i + e_i'), x, 0, e_i + e_i', s') & (1 \leq i \leq 2) \\
\qquad \text{with } F_i = \{f' \mid I_i(f') = \mathcal{S}_i(x)\} & (1 \leq i \leq 2)
\end{array}$$

It is straightforward to see that $(\Psi_1 + e_1')$ holds the induction hypothesis with respect to $(\Psi_2 + e_2')$ and $e_2 + e_2'$. Hence, with the induction hypothesis, we conclude the case.

(vii) Case $s = f^{\checkmark}\,;s'$ with $x \in I_2(f) \setminus I_1(f)$. Let $S_1 = I_1(f)$. By rules (4) and (5) of Figure 4, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e_1', e_1, s) = \mathtt{T}_{\mathcal{S}_1}(I_1 \setminus F_1, \Psi_1', x, 0, e_1'', s') \\
\mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e_2', e_2, s) = \mathtt{T}_{\mathcal{S}_2}(I_2, (\Psi_2 + e_2'), x, 0, e_2 + e_2', s') \\
\qquad \text{with } F_1 = \{f' \in \mathrm{dom}(I_1) \mid I(f') = \mathcal{S}(x) \text{ or } I(f') = S_1\} \\
\qquad \text{and } e_1'' = \max(e_1 + e_1', [\![\Psi_1(S_1)]\!]) \quad \Psi_1' = (\Psi_1 \parallel e_1'') \setminus S_1
\end{array}$$

By construction, we have $I_1(f') \subseteq \mathcal{S}_2(x)$ for all $f' \in F_1$. Therefore, we have $\mathrm{dom}(I_1 \setminus F_1) \supseteq \mathrm{dom}(I_2)$. By hypothesis, we have that $[\![\Psi_1(S_1)]\!] \leq e_2 + e_2'$, hence we have $e_1'' \leq e_2 + e_2'$. Consequently, we have that Ψ_1' holds the induction hypothesis with respect to $(\Psi_2 + e_2')$ and $e_2 + e_2'$. Hence, with the induction hypothesis, we conclude the case.

(viii) Case $s = f^{\checkmark}; s'$ with $x \notin I_2(f)$ (by hypothesis, this implies that $x \notin I_1(f)$). Let $S_1 = I_1(f)$ and $S_2 = I_2(f)$. By rule (5) of Figure 4, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}_i}(I_i, \Psi_i, x, e'_i, e_i, s) = \mathtt{T}_{\mathcal{S}_i}(I_i \setminus F_i, \Psi'_i, x, 0, e''_i, s') \qquad (1 \leq i \leq 2) \\
\text{with } F_i = \{f' \in \mathrm{dom}(I_i) \mid I_i(f') = \mathcal{S}_i(x) \text{ or } I_i(f') = S_i\} \ (1 \leq i \leq 2) \\
\text{and} \left\{ \begin{array}{l} e''_i = \max(e'_i + e_i, [\![\Psi_i(S_i)]\!]) \ (1 \leq i \leq 2) \\ \Psi'_i = (\Psi_i \parallel e''_i) \setminus S_i \ (1 \leq i \leq 2) \end{array} \right.
\end{array}$$

It is straighforward to see that $\mathrm{dom}(I_1 \setminus F_1) \supseteq \mathrm{dom}(I_2 \setminus F_2)$, and that Ψ'_1 holds the induction hypothesis with respect to (Ψ'_2) and $e_2 + e'_2$. Hence, with the induction hypothesis, we conclude the case.

(ix) Case $s = f^{\checkmark}; s'$ with $x \in I_1(f)$ and $f \notin \mathrm{dom}(I_2)$. By rules (4) and (6) of Figure 4, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}_i}(I_i, \Psi_i, x, e'_i, e_i, s) = \mathtt{T}_{\mathcal{S}_i}(I_i \setminus F_i, (\Psi_i + e'_i), x, 0, e_i + e'_i, s') \qquad (1 \leq i \leq 2) \\
\qquad\qquad \text{with } F_i = \{f' \mid I_i(f') = \mathcal{S}_i(x)\} \qquad (1 \leq i \leq 2)
\end{array}$$

This case is thus identical to case (vi), and thus we conclude the case.

(x) Case $s = f^{\checkmark}; s'$ with $x \notin I_1(f)$ and $f \notin \mathrm{dom}(I_2)$. Let $S_1 = I_1(f)$. By rules (5) and (6) of Figure 4, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}_1}(I_1, \Psi_1, x, e'_1, e_1, s) = \mathtt{T}_{\mathcal{S}_1}(I_1 \setminus F_1, \Psi'_1, x, 0, e''_1, s') \\
\mathtt{T}_{\mathcal{S}_2}(I_2, \Psi_2, x, e'_2, e_2, s) = \mathtt{T}_{\mathcal{S}_2}(I_2 \setminus F_2, (\Psi_2 + e'_2), x, 0, e_2 + e'_2, s') \\
\text{with } F_1 = \{f' \in \mathrm{dom}(I_1) \mid I_1(f') = \mathcal{S}_1(x) \text{ or } I_1(f') = S_1\}, \ F_2 = \{f' \mid I_2(f') = \mathcal{S}_2(x)\} \\
\text{and} \left\{ \begin{array}{l} e''_1 = \max(e'_1 + e_1, [\![\Psi_1(S_1)]\!]) \\ \Psi'_1 = (\Psi_1 \parallel e''_1) \setminus S_1 \end{array} \right.
\end{array}$$

By construction, we have $I_1(f') \notin \mathrm{dom}(I_2)$ for all f' such that $I_1(f') = S_1$. Hence, we have $\mathrm{dom}(I_1 \setminus F_1) \supseteq \mathrm{dom}(I_2 \setminus F_2)$. By hypothesis, we have that $[\![\Psi_1(S_1)]\!] \leq e_2 + e'_2$, hence we have $e''_1 \leq e_2 + e'_2$. Consequently, we have that Ψ'_1 holds the induction hypothesis with respect to $(\Psi_2 + e'_2)$ and $e_2 + e'_2$. Hence, with the induction hypothesis, we conclude the case.

(xi) Case $s = f^{\checkmark}; s'$ with $f \notin \mathrm{dom}(I_1) \cup \mathrm{dom}(I_2)$. By rule (6) of Figure 4, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}_i}(I_i, \Psi_i, x, e'_i, e_i, s) = \mathtt{T}_{\mathcal{S}_i}(I_i \setminus F_i, (\Psi_i + e'_i), x, 0, e_i + e'_i, s') \qquad (1 \leq i \leq 2) \\
\qquad\qquad \text{with } F_i = \{f' \mid I_i(f') = \mathcal{S}_i(x)\} \qquad (1 \leq i \leq 2)
\end{array}$$

This case is thus identical to case (vi), thus, we conclude the case.

(xii) Case $s = \mathit{wait}(e); s'$. By rule (1) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}_i}(I_i, \Psi_i, x, e'_i, e_i, s) = \mathtt{T}_{\mathcal{S}_i}(I_i, (\Psi_i + e), x, e'_i, e_i + e, s') \qquad (1 \leq i \leq 2)$$

Hence, with the induction hypothesis, we conclude the case.

■

Lemma Appendix A.9. *Suppose given $\mathcal{S}$, I, Ψ, Ψ', x, e_1, e_2, e', s and t with: $\mathcal{S}$, I, Ψ, x, e_1 and e' are coherent parameters;* $\mathrm{dom}(\Psi) = \mathrm{dom}(\Psi')$*; and for all* $A \in \mathrm{dom}(\Psi)$*, there exists* e^{A_1} *and* e^{A_2} *such that* $[\![\Psi'(A)]\!] = \max(e^{A_1} + t, e^{A_2})$ *where* $[\![\Psi(A)]\!] = \max(e^{A_1}, e^{A_2})$*. We then have*

$$\mathtt{T}_{\mathcal{S}}(I, \Psi', x, e', \max(e_1 + t, e_2), s) = \max(t + e^{r_1}, e^{r_2})$$

$$\textit{with} \begin{cases} \mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', \max(e_1, e_2), s) = \max(e^{r_1}, e^{r_2}) \\ e^{r_1} = \max(e_1 + {e_1}', \max_{A \in \mathrm{dom}(\Psi)}(e^{A_1} + {e^{A_1}}')) \textit{ for some } {e_1}' \textit{ and } {e^{A_1}}' \\ e^{r_2} = \max(e_2 + {e_2}', \max_{A \in \mathrm{dom}(\Psi)}(e^{A_2} + {e^{A_2}}')) \textit{ for some } {e_2}' \textit{ and } {e^{A_2}}' \end{cases}$$

Proof. We write $e = \max(e_1, e_2$ and $e^t = \max(e_1 + t, e_2)$. We prove this result by induction on s:

(i) Case $s = \mathsf{0}$. By construction, the lemma holds: I and Ψ are empty in this case (all the futures must be synchronized at the end of a method), $e' = 0$ (for the same reason), and

$$\mathtt{T}_{\mathcal{S}}(\varnothing, \varnothing, x, 0, e^t, \mathsf{0}) = e^t$$
$$\mathtt{T}_{\mathcal{S}}(\varnothing, \varnothing, x, 0, e, \mathsf{0}) = e$$

(ii) Case $s = \nu x; s'$. By construction, we have that

$$\mathtt{T}_{\mathcal{S}}(I, \Psi', x, e'_x, e^t, s) = \mathtt{T}_{\mathcal{S}}(I, \Psi', x, e', e^t, s')$$
$$\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e'_x, e, s) = \mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s')$$

Hence, with the induction hypothesis, we conclude the case.

(iii) Case $s = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e}); s'$ with $y \in \mathcal{S}(x)$. By rule (2) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}}(I, \Psi', x, e', e^t, s) = \mathtt{T}_{\mathcal{S}}(I[f \mapsto \mathcal{S}(x)], \Psi', x, e' + \mathtt{m}(\overline{e}), e^t, s')$$
$$\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s) = \mathtt{T}_{\mathcal{S}}(I[f \mapsto \mathcal{S}(x)], \Psi, x, e' + \mathtt{m}(\overline{e}), e, s')$$

Hence, with the induction hypothesis, we conclude the case.

(iv) Case $s = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e}); s'$ with $y \notin \mathcal{S}(x)$. Let $S = \mathcal{S}(y)$. By rule (3) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}}(I, \Psi', x, e', e^t, s) = \mathtt{T}_{\mathcal{S}}(I[f \mapsto S], \Psi'_f, x, e', e^t, s')$$
$$\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s) = \mathtt{T}_{\mathcal{S}}(I[f \mapsto S], \Psi_f, x, e', e, s')$$
$$\text{with } \Psi'_f = \Psi'[S \mapsto \varepsilon' \cdot \langle \mathtt{m}(\overline{e}), 0 \rangle],\ \Psi_f = \Psi[S \mapsto \varepsilon \cdot \langle \mathtt{m}(\overline{e}), 0 \rangle]$$
$$\text{and } \begin{cases} \varepsilon' = \Psi'(S),\ \varepsilon = \Psi(S) & \text{if } S \in \mathrm{dom}(\Psi) \\ \varepsilon' = e^t,\ \varepsilon = e & \text{else} \end{cases}$$

It is easy to see that $\mathrm{dom}(\Psi'_f) = \mathrm{dom}(\Psi_f)$ and that

$$\begin{cases} [\![\Psi'_f(S)]\!] = \max(e^{S_3} + t, e^{S_4}), [\![\Psi_f(S)]\!] = \max(e^{S_3}, e^{S_4}) & \text{if } S \in \mathrm{dom}(\Psi) \\ [\![\Psi'_f(S)]\!] = \max(e^3 + t, e^4), [\![\Psi_f(S)]\!] = \max(e^3, e^4) & \text{else} \end{cases}$$
$$\text{with } e^{S_3} = e^{S_1} + \mathtt{m}(\overline{e}),\ e^{S_4} = e^{S_2} + \mathtt{m}(\overline{e}),\ e^3 = e_1 + \mathtt{m}(\overline{e}),\ e^4 = e_2 + \mathtt{m}(\overline{e})$$

Hence, with the induction hypothesis, we conclude the case.

(v) Case $s = f^{\checkmark}; s'$ with $x \in I(f)$. By rule (4) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}}(I, \Psi', x, e', e^t, s) = \mathtt{T}_{\mathcal{S}}(I, (\Psi' + e'), x, 0, e^t + e', s')$$
$$\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s) = \mathtt{T}_{\mathcal{S}}(I, (\Psi + e'), x, 0, e + e', s')$$

By construction, it is straightforward to see that $\mathrm{dom}(\Psi' + e') = \mathrm{dom}(\Psi + e')$ and that for all $A \in \mathrm{dom}(\Psi + e')$, we have

$$\begin{cases} [\![(\Psi + e)(A)]\!] = \max(e^{A_1} + e', \max(e^{A^1_2} + \mathtt{m}(\overline{e}), e^{A^1_2} + e^{A^2_2} + e')) \\ [\![(\Psi' + e)(A)]\!] = \max(e^{A_1} + e' + t, \max(e^{A^1_2} + \mathtt{m}(\overline{e}), e^{A^1_2} + e^{A^2_2} + e')) \end{cases}$$

for some $e^{A^1_2}$ and $e^{A^2_2}$. Hence, with the induction hypothesis and the following equalities

$$e^t + e' = \max(e_1 + e' + t, e_2 + e') \qquad e + e' = \max(e_1 + e', e_2 + e')\ ,$$

we conclude the case.

(vi) Case $s = f^{\checkmark}; s'$ with $x \notin I(f)$. Let $S = I(f)$. By rule (5) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}}(I, \Psi', x, e', e^t, s) = \mathtt{T}_{\mathcal{S}}(I \setminus F, \Psi'_f, x, 0, e^3, s')$$
$$\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e', e, s) = \mathtt{T}_{\mathcal{S}}(I \setminus F, \Psi_f, x, 0, e^4, s')$$
$$\text{with } F = \{f' \in \mathrm{dom}(I) \mid I(f') = \mathcal{S}(x) \text{ or } I(f') = S\}$$
$$\text{and} \begin{cases} e^3 = \max(e^t + e', [\![\Psi'(S)]\!]),\ e^4 = \max(e + e', [\![\Psi(S)]\!]) \\ \Psi'_f = (\Psi' \parallel e^3) \setminus S,\ \Psi_f = (\Psi \parallel e^4) \setminus S \end{cases}$$

It is straightforward to see that the following equalities hold:

$$\begin{cases} e^3 = \max(e_1 + e' + t, \max(e_2 + e', [\![\Psi'(S)]\!])) \\ e^4 = \max(e + 1 + e', \max(e_2 + e', [\![\Psi'(S)]\!])) \end{cases}$$

Additionally, we have that $\mathrm{dom}(\Psi'_f) = \mathrm{dom}(\Psi_f)$, and let us consider $A \in \mathrm{dom}(\Psi_f)$, we then have

$$\begin{cases} [\![\Psi'_f(A)]\!] = \max(\max(e^{A_1} + e_1 + e') + t, \max(e_2 + e', [\![\Psi'(S)]\!])) \\ [\![\Psi_f(A)]\!] = \max(\max(e^{A_1} + e_1 + e'), \max(e_2 + e', [\![\Psi'(S)]\!])) \end{cases}$$

Hence, with the induction hypothesis, we conclude the case.

(vii) Case $s = \mathit{wait}(e'')\,;s'$. By rule (1) of Figure 4, we have that

$$\begin{aligned}\mathtt{T}_{\mathcal{S}}(I,\Psi',x,e',e^t,s) &= \mathtt{T}_{\mathcal{S}}(I,\Psi'+e'',x,e',e^t+e'',s')\\ \mathtt{T}_{\mathcal{S}}(I,\Psi,x,e',e,s) &= \mathtt{T}_{\mathcal{S}}(I,\Psi+e'',x,e',e+e'',s')\end{aligned}$$

By construction, it is straightforward to see that $\mathrm{dom}(\Psi'+e'') = \mathrm{dom}(\Psi+e'')$ and that for all $A \in \mathrm{dom}(\Psi+e'')$. We then have

$$\begin{cases}[\![(\Psi+e'')(A)]\!] = \max(e^{A_1}+e'', \max(e^{A_2^1}, e^{A_2^1}+e^{A_2^2}+e''))\\ [\![(\Psi'+e'')(A)]\!] = \max(e^{A_1}+e''+t, \max(e^{A_2^1}, e^{A_2^1}+e^{A_2^2}+e''))\end{cases}$$

for some $e^{A_2^1}$ and $e^{A_2^2}$. Hence, with the induction hypothesis, we conclude the case.

■

Lemma Appendix A.10. *Suppose given $\mathcal{S}$, I, Ψ, x, e, s, t and t' with: $\mathcal{S}$, I, Ψ, x, e, 0 and s are coherent parameters; for all $A \in \mathrm{dom}(\Psi)$, we have $\Psi(A) = 0 \cdot \langle e_x^A, 0\rangle$; and $0 < t' \leq t$. We then have*

$$\begin{aligned}&\mathtt{T}_{\mathcal{S}}(I,\Psi,x,e_x,0,\mathit{wait}(t)\,;s) = \max(t'+e_x^1, \max_{A\in\mathrm{dom}(\Psi)}(\Psi(A)+e_x^A))\\ &\quad \textit{with } \max(e_x^1, \max_{A\in\mathrm{dom}(\Psi)}(\Psi(A)+e_x^A)) = \mathtt{T}_{\mathcal{S}}(I,\Psi,x,e_x,0,\mathit{wait}(t-t')\,;s)\end{aligned}$$

Proof. By rule (7) of Figure 4, we have that

$$\begin{aligned}\mathtt{T}_{\mathcal{S}}(I,\Psi,x,e_x,0,\mathit{wait}(t)\,;s) &= \mathtt{T}_{\mathcal{S}}(I,\Psi+t,x,e'_x,t,s)\\ \mathtt{T}_{\mathcal{S}}(I,\Psi,x,e_x,0,\mathit{wait}(t-t')\,;s) &= \mathtt{T}_{\mathcal{S}}(I,\Psi+(t-t'),x,e'_x,(t-t'),s)\end{aligned}$$

Hence, this lemma is a direct consequence of Lemma Appendix A.9. ■

Lemma Appendix A.11. *Suppose given $\mathcal{S}$, I, Ψ, x, e, f and s with: $\mathcal{S}$, I, Ψ, x, 0 e, and s are coherent parameters; for all $A \in \mathrm{dom}(\Psi)$, we have $\Psi(A) = 0 \cdot \langle e^A, 0\rangle$; and $I(f) = \mathcal{S}(x)$. We then have*

$$\begin{aligned}&\mathtt{T}_{\mathcal{S}}(I,\Psi,x,e,0,f^{\checkmark}\,;s) = \max(e+e', \max_{A\in\mathrm{dom}(\Psi)}(\Psi(A)+e^A))\\ &\qquad\qquad \textit{with } \max(e', \max_{A\in\mathrm{dom}(\Psi)}(\Psi(A)+e^A)) = \mathtt{T}_{\mathcal{S}}(I,\Psi,x,e,0,s)\end{aligned}$$

Proof. By rules (4) and (7) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}}(I,\Psi,x,e,0,f^{\checkmark}\,;s) = \mathtt{T}_{\mathcal{S}}(I,\Psi+e,x,0,e,s)$$

This result is thus a direct consequence of Lemma Appendix A.9. ■

Lemma Appendix A.12. *Suppose given $\mathcal{S}$, I, Ψ, x, e, f and s with: $\mathcal{S}$, I, Ψ, x, e, 0 and s are coherent parameters; for all $A \in \mathrm{dom}(\Psi)$, we have $\Psi(A) = 0 \cdot \langle e^A, 0 \rangle$; and $I(f) \neq \mathcal{S}(x)$. We then have*

$$\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e, 0, f^{\checkmark}\,;s) = \max(\max(e, [\![\Psi(I(f))]\!]) + e', \max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A))$$

$$\textit{with} \begin{cases} \max(e', \max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A)) = \mathtt{T}_{\mathcal{S}}(I \setminus F, \Psi \setminus I(f), x, 0, 0, s) \\ F = \{f' \in \mathrm{dom}(I) \mid I(f') = \mathcal{S}(x) \textit{ or } I(f') = I(f)\} \end{cases}$$

Proof. Let $S = I(f)$. By rule (5) of Figure 4, we have that

$$\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e, 0, f^{\checkmark}\,;s) = \mathtt{T}_{\mathcal{S}}(I \setminus F, \Psi', x, 0, e', s)$$
$$\text{with } F = \{f' \in \mathrm{dom}(I) \mid I(f') = \mathcal{S}(x) \text{ or } I(f') = S\}$$
$$\text{and} e' = \max(e, [\![\Psi(S)]\!]),\ \Psi' = (\Psi \parallel e') \setminus S$$

The result can thus be proven in a similar way to Lemma Appendix A.9. ■

Lemma Appendix A.13. *Suppose given $\mathcal{S}$, I, Ψ, x, e_x, f and s with: $\mathcal{S}$, I, Ψ, x, e, 0 and s are coherent parameters; for all $A \in \mathrm{dom}(\Psi)$, we have $\Psi(A) = 0 \cdot \langle e^A, 0 \rangle$; and $f \in \mathrm{dom}(I)$. We then have*

$$\mathtt{T}_{\mathcal{S}}(I, \Psi, x, e, 0, f^{\checkmark}\,;s) \geq \mathtt{T}_{\mathcal{S}}(I \setminus \{f\}, \Psi, x, e, 0, f^{\checkmark}\,;s)$$

Proof. This is directly proven by induction on s, by remarking that $\Psi(I(f))$ is a simple expression. ■

Appendix A.2.3. Main Results

Lemma Appendix A.14 (Upper Bound Stability 1/3)**.** *Suppose given a cost solution Σ of the cost program generated from an* `alt` *program $P = \left(\mathtt{m}_1(\overline{x}_1) = s_1, \cdots, \mathtt{m}_n(\overline{x}_n) = s_n, s_{main}\right)$. Then Σ is also a cost solution of the runtime configuration $cn \stackrel{def}{=} act(start, s_{main}\,; f_{start}, \varnothing)$.*

Proof. Let us consider the cost program eq_1 of the program P, and the cost configuration eq_2 of the runtime configuration cn. By construction, eq_1 and eq_2 are identical, except maybe for the entry f_{start}: we indeed have:

$$eq_1(f_{start}) = \mathtt{translate}_{\mathtt{sschem}(\{\ \{start\}\ \}, s_{main})}(\varnothing, \varnothing, start, 0, 0, s_{main})$$
$$eq_2(f_{start}) = \mathtt{translate}_{\mathtt{sschem}(\mathcal{S}_{cn}, s_{main})}(I_{cn}, \Psi_{cn} \setminus \mathcal{S}_{cn}(start), start, 0, 0, s_{main})$$
$$\text{with} \quad \begin{cases} \mathcal{S}_{cn} = \{\ \{start\}\ \} \\ I_{cn} = [f_{start} \mapsto \{start\}] \\ \Psi_{cn} = [\{start\} \mapsto f_{start}] \end{cases}$$

Remark that $\Psi_{cn} \setminus \mathcal{S}_{cn}(start) = \varnothing$, by Lemma Appendix A.7, we thus have that

$$\mathtt{translate}_{\mathtt{sschem}(\mathcal{S}_{cn}, s_{main})}(I_{cn}, \Psi_{cn} \setminus \mathcal{S}_{cn}(start), start, 0, 0, s_{main})$$
$$= \mathtt{translate}_{\mathtt{sschem}(\{\ \{start\}\ \}, s_{main})}(\varnothing, \varnothing, start, 0, 0, s_{main})$$

which proves the result. ■

Lemma Appendix A.15 (Upper Bound Stability 2/3)**.** *Suppose given a run-time configuration cn such that there exists cn′ with $cn \xrightarrow{t} cn'$. Suppose moreover given a solution Σ of the cost configuration of cn. Then, there exists a solution Σ' of the cost configuration of cn′ such that for all $f \in \mathrm{dom}(\Sigma)$, we have $\Sigma'(f) + t \leq \Sigma(f)$.*

Proof. Let eq_1 be the cost configuration of cn and eq_2 the cost configuration of cn': we remark that $\mathrm{dom}(eq_1) = \mathrm{dom}(eq_2)$, and we first prove by induction on cn that for all $f \in \mathrm{dom}(eq_1)$, we either have that:

$$eq_1(f) = \max(t' + e^1, \max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A)) \quad \text{with } eq_2(f) = \max(e^1, \max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A))$$

or $eq_1(f) = eq_2(f) = \sum_{f \in F} f + e$ for some e and with $F = \{f' \mid f \to f' \in cn\} \neq \varnothing$. If the configuration is empty, then its cost configuration is empty and the result trivially holds. Now let us consider that cn is not empty. By the definition of $cn \xrightarrow{t} cn$, triggered by the rule (Tick), cn is strongly *t-stable* (see Definition 2.1). Let us consider $f \in \mathrm{dom}(\Sigma)$, we have three cases:

- Case $cn = act(x, wait(e); s; f, q)\ cn''$. By Definition 2.1, we have $cn' = act(x, wait(k); s; f, q)\ \Phi(cn'', t)$ with $k = [\![e]\!] - t$. By Lemma Appendix A.10, we have that $eq_1(f) = \max(t' + e^1, \max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A))$ with $eq_2(f) = \max(e^1, \max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A))$.

- Case $cn = act(x, f'^{\checkmark}; s; f, q)\ cn'$ with $I_{cn}(f) = I_{cn}(f')$. By Definition 2.1, we have $cn' = act(x, f'^{\checkmark}; s; f, q)\ \Phi(cn'', t)$, which implies that $eq_1(f) = eq_2(f)$. Using Lemma Appendix A.9, it is straightfoward to see that $eq_2(f) = \sum_{f \in F} f + e$ for some e, with $F = \{f'' \mid I_{cn}(f) = I_{cn}(f''),\ f \to f'' \in cn\}$ and $f' \in F$.

- Case $cn = act(x, f'^{\checkmark}; s; f, q)\ cn'$ with $I_{cn}(f) \neq I_{cn}(f')$. By Definition 2.1, we have $cn' = act(x, f'^{\checkmark}; s; f, q)\ \Phi(cn'', t)$, which implies that $eq_1(f) = eq_2(f)$. Using Lemma Appendix A.9, it is straightfoward to see that $eq_2(f) = \sum_{f \in F} f + e$ for some e, with $F = \{f'' \mid I_{cn}(f') = I_{cn}(f''),\ f \to f'' \in cn\}$ and $f' \in F$.

Using Lemma Appendix A.6, we have that the process graph G of cn is a tree, and so we can construct the solution Σ' inductively, starting from the leaves of G. ■

Lemma Appendix A.16 (Upper Bound Stability 3/3)**.** *Suppose given a configuration cn such that there exists cn′ with $cn \to cn'$. Suppose moreover given a solution Σ of the cost configuration of cn. Then, there exists a solution Σ' of the cost configuration of cn′ such that for all $f \in \mathrm{dom}(\Sigma)$, we have $\Sigma'(f) \leq \Sigma(f)$.*

Proof. Let us consider the cost configuration eq_1 of cn and eq_2 of cn'. We prove that for all $f \in \mathrm{dom}(eq_1)$, we have $eq_2(f) \leq eq_1(f)$, which thus proves the result. We construct our demonstration by case distinction on the reduction rule, modulo the rule (Context).

- Case (New). It is straightforward to see that, with the definition of $\mathcal{S}_{cn}$, that $eq_1 = eq_2$. Hence, the result holds for the case.
- Case (Get-True). We have that $cn = act(x, {f_1}^{\checkmark}; p, q)\ cn'$, with $p = s; f$. We can first remark that for all $(x : f') \in cn$ such that $f' \neq f$, we have that $eq_1(f) = eq_2(f)$. Moreover, from the construction of the future localization, we have that $f_1 \notin \mathrm{dom}(I_{cn})$. Hence, using the rule (6) of Figure 4, we have that $eq_1(f) = eq_2(f)$.
- Case (Get-False). It is easy in this case to see that $eq_1 = eq_2$, which gives us the result.
- Case (Async-Call). We have that $cn = act(x, \nu f_1\colon \mathtt{m}(z, \overline{e}); p, q)\ cn'$, with $p = s; f$. Let f_2 be the fresh future name created in this rule. We now have two sub-cases.

  Let first consider that $\mathcal{S}_{cn}(z) = \mathcal{S}_{cn}(x)$. By construction, we have $\Psi_{cn'}(\mathcal{S}_{cn'}(z)) = \Psi_{cn}(\mathcal{S}_{cn}(z))$, as f_2 is free in p. Moreover, considering the rule (7) of Figure A.8, for all $f' \in \mathrm{dom}(eq_1) \setminus \{f_2\}$, we have $eq_1(f') = eq_2(f')$. Now, by rule (4) of Figure A.8, we have that $eq_2(f_2) = \mathtt{m}(z, \overline{e})$, and $eq_1(f) = eq_2(f)\{{}^{\mathtt{m}(z,\overline{e})}/_{f_2}\}$. By stating that $\Sigma' = \Sigma[f_2 \mapsto \Sigma(\mathtt{m}(z, \overline{e}))]$, we have that $\Sigma(eq_1(f)) = \Sigma'(eq_2(f))$, which concludes this sub-case.

  Second, consider that $\mathcal{S}_{cn}(z) \neq \mathcal{S}_{cn}(x)$. By Lemma Appendix A.4, we have that f is the only future name such that $f \to z \in cn$. Hence, by construction, we have that for all $f' \in (\mathrm{dom}(eq_1) \setminus \{f\}$, we have $eq_1(f') = eq_2(f')$. Now, by rule (4) of Figure A.8, we have that $eq_2(f_2) = \mathtt{m}(z, \overline{e})$, and $eq_1(f) = eq_2(f)\{{}^{\mathtt{m}(z,\overline{e})}/_{f_2}\}$. By stating that $\Sigma' = \Sigma[f_2 \mapsto \Sigma(\mathtt{m}(z, \overline{e}))]$, we have that $\Sigma(eq_1(f)) = \Sigma'(eq_2(f))$, which concludes this sub-case.
- Case (Bind-Fun). Here, we have that $eq_1(f) = \mathtt{m}(x, \overline{v})$, and $eq_2(f) = e$ where $eq_1(\mathtt{m}(x, \overline{v})) = e$. Thus, we conclude the case.
- Case (Wait-0). It is easy in this case to see that $eq_1 = eq_2$, which gives us the result.
- Case (Activate). It is easy in this case to see that $eq_1 = eq_2$, which gives us the result.
- Case (Return). We have that $cn = act(x, f, q)\ cn'$. By construction, we have that $eq_1(f) = 0 = eq_2(f)$ (by rule (2.2) of Fig. A.8). By Lemma Appendix A.6, either: i) $f = f_{start}$, and so we have the result, as $eq_2(f) = 0$ for all $f \in \mathrm{dom}(eq_2)$; or ii) there exists exactly one f' such that $f' \to f \in cn$. By construction, for all $f'' \in \mathrm{dom}(I_{cn}) \setminus f'$, we have $eq_1(f'') = eq_2(f'')$. Finally, by Lemma Appendix A.13, we have that $eq_1(f') \geq eq_2(f')$, which concludes the case.

■

Theorem Appendix A.17 (Correction)**.** *Given an* `alt` *program P, and a solution Σ of the cost equation of P, we have that $\Sigma(f_{start})$ is an upper bound of the execution time of P.*

Proof. By induction on the reduction time of P, using the three previous lemma.
■