Resource Consumption of Concurrent Objects over Time

Antonio Flores-Montoya and Reiner Hähnle

TU Darmstadt, Dept. of Computer Science
aeflores|haehnle@cs.tu-darmstadt.de

Abstract. We present the first static resource analysis of timed concurrent object programs. Instead of measuring the total resource consumption over a complete execution, we measure the resource consumption at different moments in time, that is, the *resource consumption over time*. To obtain such a measure we perform a program transformation that generates a program without time whose resource consumption corresponds to the resource consumption of the timed program during time t. The transformed program can then be analyzed with a combination of existing tools. These provide upper bounds that safely approximate the resource consumption of all possible behaviors of the program at all possible times. We implemented a prototype of the approach and evaluated it on a complex program to demonstrate its feasibility.

1 Introduction

The use of static analysis to infer upper bounds on the resource consumption of software systems is a very active area of research. Many tools have been recently developed [7, 8, 10, 11, 15]. However, most effort has been focused on the analysis of sequential programs. There is some initial research that tries to apply resource analysis techniques to distributed and concurrent systems [1, 2, 12]. In contrast to sequential systems, in a concurrent system it is not only important to estimate the amount of consumed resources but also *when* they are consumed. If we have a set of tasks to be executed, a system will behave differently, depending on whether the tasks are scheduled simultaneously, sequentially, etc. Some aspects of this richer concept of resource consumption are already explored in [2, 12].

We propose a novel approach for measuring concurrent resource consumption based on a discrete time model. We present a static analysis that infers upper bounds of the *resource consumption over time*. These upper bounds are expressed as a function of the entry parameters of the program and the elapsed time and allow to explore how resource consumption varies in a given program as time advances. In our program the explicit aspects of time are captured in two primitives, **await duration**(t) and **until**(t'), that suspend a task for a certain period of time t or until the absolute time has reached t'. The main advantage of this approach is that, by simply placing these time primitives at different locations inside a program, we can obtain different kinds of behavior and analyze their resource consumption.

The target of our analysis is a language based on concurrent objects. In this language, each object owns a processor and executes in parallel with others. Each object contains a set of tasks and only one task per object can be executed at any given time, while the scheduling of tasks is non-deterministic and non-preemptive. Communication and synchronization among objects is based on asynchronous message passing and future variables. The language is inspired by ABS [13] and the time primitives are (slightly simplified) taken from [6, 14]. However, the presented approach could be easily adopted to other concurrency models based on creating and joining tasks.

To the best of our knowledge, our paper presents the first static resource analysis that analytically derives sound symbolic upper bounds for timed concurrent programs. The analysis generates a set of upper bounds that summarize analytically the resource consumption of a program over time. The main contributions of this work are:

- We define the novel concept of *resource consumption over time* (Sec. 3).
- We develop a sound program transformation that generates an untimed program from a timed program (Sec. 4).
- We show how we can analyze the resulting untimed program with a combination of existing tools and how the results can be interpreted (Sec. 5).
- We illustrate the flexibility of the timed approach by inferring the *peak cost* of an example borrowed from [2] (Sec. 6).

2 Timed Concurrent Programs

We adopt a lightweight object-oriented language. A program $\mathcal{P}$ consists of a set of classes, each of them defines a set of fields and a set of methods. The set of types includes the class names, the integer (Int) primitive type, the set of *future* variable types $fut(T)$ and $Unit$. The latter is the default return type of methods (like $Void$ in C). The notation $\overline{T}$ is used as a shorthand for $T_1, ...T_n$, and similarly for other names. Pure expressions (p) have no side effects. A pure expression is *local* (naming convention p_l) if it does not depend on any field. The abstract syntax of class declarations CL, method declarations M, types T, variables v (x for local variables), and statements s is:

$$
\begin{array}{l}
CL ::= \textsf{class}\ C\ \{\overline{T}\ \overline{f}; \overline{M}\} \quad M ::= T\ m(\overline{T}\ \overline{x})\{s; \textsf{return}\ p\} \quad v ::= x \mid \textsf{this}.f \\
s ::= s; s \mid v = e \mid \textsf{if}\,(p)\ s\ \textsf{else}\ s \mid \textsf{while}(p)\ s \\
\qquad \textsf{await}\ x? \mid \textsf{await duration}(p_l) \mid \textsf{until}\,(p_l) \mid \textsf{release} \mid \textsf{cost}\ p \\
e ::= \textsf{new}\ C(\overline{p}) \mid x!m(\overline{p}) \mid x.\textsf{get} \mid p \quad T ::= C \mid Int \mid fut(T) \mid Unit
\end{array}
$$

We assume that all methods have a single return instruction at the end of the method. If the method returns $Unit$, the return instruction is empty *return*. Each object encapsulates a *local heap* which is not accessible from outside this object, i.e., fields are always accessed using the **this** object, and any other object can only access such fields through method calls. We assume that each program includes a method called $\textsf{main}(\overline{z})$ with a set of initial parameters $\overline{z}$, from which

```
class Hndset(ts,smss){
Unit normalBehavior(mx,dur,tc){
  if (now()>tc && now()<tc+20)
    midnightWindow(mx,dur,tc);
  else{
    ts!call(dur);
    await duration(1);
    normalBehavior(mx,dur,tc);
 }}
Unit midnightWindow(mx,dur,tc){
   if (now() >= tc+20) {
    normalBehavior(mx,dur,tc);
   } else {
     Int i = 0;
     while (i < mx) {
       smss!sendSMS();
       i = i + 1;
     }
    await duration(1);
    midnightWindow(mx,dur,tc);
 }}
}
```

```
class PhoneSvr{
  Unit call(dur){
    while (dur>0) {
     cost 1;
     dur = dur-1;
     await duration(1);
  }}
}

class SMSSvr{
  Unit sendSMS() {
    cost 1;
  }
}

Unit main(mx,dur,tc){
 SMSSvr sms=new SMSSvr();
 PhoneSvr ts=new PhoneSvr();
 Hndset hs=new Hndset(ts,sms);
 hs!normalBehavior(mx,dur,tc);
}
```

Fig. 1. A timed program

execution will start. The main method does not belong to any class and has no fields. The concurrency model is as follows: each object has a lock that is shared by all the tasks that belong to the object. Data synchronization is by means of future variables (denoted y): an **await** y? instruction is used to synchronize with the result of executing a task $y = x!m(\overline{p})$, where **await** y? is suspended until the result assigned to the future variable y is available (i.e., the task is finished). During suspension the object's lock is released so that another pending task on that object can take it. In contrast to await, the expression y.**get** blocks its object (no other task of the same object can run) until y is available. Finally, the instruction **release** releases the object's lock unconditionally.

Time is a discrete magnitude. A timed program has a global clock common to all concurrent components. The current time can be accessed through the pure expression $now()$ (which simply reads the value of the clock). Time is advanced through the primitive **await duration**(p_l) which releases the object's lock until time has advanced p_l units (relative time progress) or **until**(p_l) which releases the object's lock until time is at least p_l (absolute time progress). Time only advances when no task can progress any further.

2.1 Explicit Cost Model

We adopt the standard approach to obtain a parametric cost model. We introduce a new statement **cost** p where p is a pure expression of integer type. When

a statement **cost** p is executed, the result of evaluating p determines the amount of resources consumed. The choice of the cost model (where the cost statements are introduced) determines the resources that we want to observe. For example, if we introduce a statement **cost** 1 after every instruction, we will infer an upper bound on the number of executed instructions. Or we could add a cost instruction to each **new** C instruction to measure the number of objects created. Both time and cost annotations can be introduced automatically according to the underlying architecture, network topology, or any other given criterion.

Example 1. Fig. 1 illustrates an example of a timed program with cost annotations. The return instructions have been omitted given that all the methods return $Unit$. The parameter types have also been omitted (they are all integers). The program contains 3 classes: A phone server PhoneSvr that might process calls of different duration. Each call lasts calltime time units and consumes one resource per time unit; An SMS server SMSSvr that can process SMSs. Each SMS is processed instantly consuming a resource unit. And the class Hndset that models a possible behavior of the clients. In particular, it simulates a scenario where the servers are receiving calls. The duration of calls is given by the parameter dur. At time tc, the behavior changes and we enter the midnight window where we receive mx SMS per time unit. This exceptional behavior lasts until time is $tc+20$ when it changes back to normal. This behavior is modelled by two mutually recursive methods normalBehavior and midnightWindow. Finally, the main method creates an SMS server sms, a phone server ts and calls normalBehavior.

2.2 Operational Semantics

A *program state* S is a set $S = \mathtt{Ob} \cup \mathtt{T} \cup \{clk(t)\}$ where Ob is the set of all created objects, T is the set of tasks (including finished tasks) and $clk(t)$ is the global clock with the current time t. The associative and commutative union operator on states is denoted by white-space. An *object* is a term $ob(o, a, lk)$ where o is the object identifier, a is a mapping from the object fields to their values, and lk the identifier of the *active task* that holds the object's lock or $\bot$ if the object's lock is free. Only one task can be *active* (running) in each object and hold its *lock*. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. A *task* is a term $tsk(tk, o, l, s)$ where tk is a unique task identifier, o identifies the object to which the task belongs, l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(val)$ if the task has terminated and the return value val is available. Created objects and tasks never disappear from the state in the semantics.

Given a program $\mathcal{P}$ with a main method $\mathsf{main}(\overline{z})$ and a set of initial values $\overline{val}$, the execution of a program starts from the initial state $S_0(\overline{val}) = \{obj(0, f, \bot), tsk(0,\ 0, buildLoc(\overline{val}, main), body(\mathsf{main})), clk(1)\}$ where we have an initial object with identifier 0 with a free lock $\bot$. f is an empty mapping (since main has no fields). The local state is generated by $buildLoc(\overline{val}, \mathsf{main})$ that maps the main method parameters $\overline{z}$ to the given entry values $\overline{val}$ and the rest

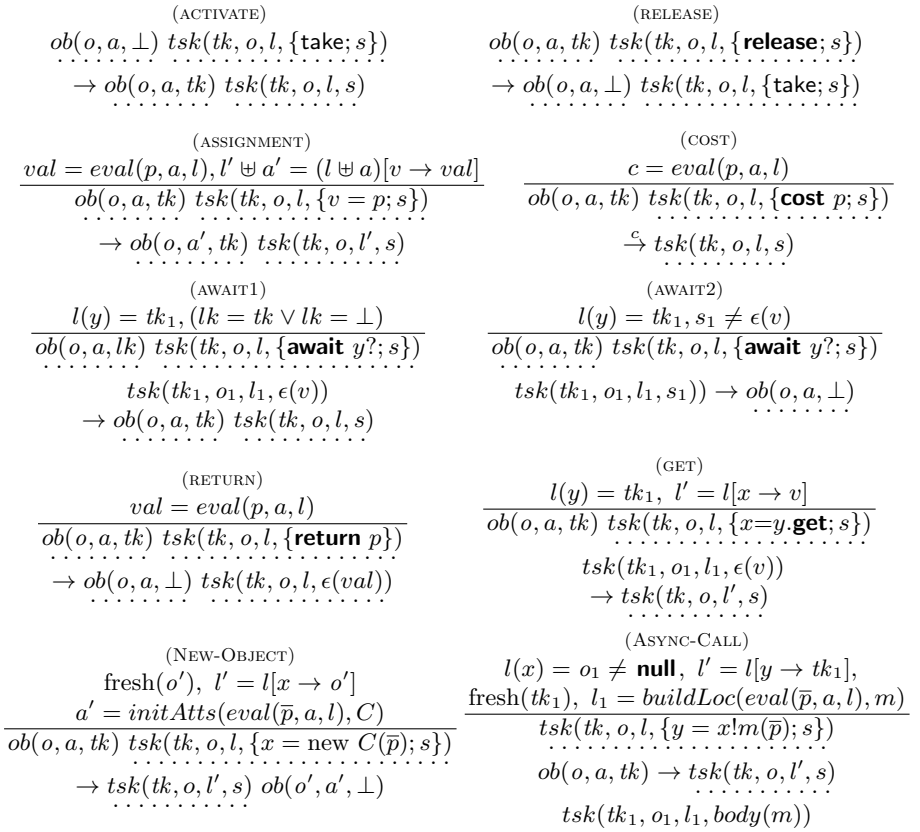

Fig. 2. Semantics

of local variables to default values. The initial time is one. Given a method $M ::=T\ m(\overline{T}\ \overline{x})\{s; \mathbf{return}\ p\}$, the function $body(M)$ returns $\{\mathsf{take}; s; \mathbf{return}\ p\}$ i.e. the sequence of statements of the method preceded by an auxiliary instruction take that fetches the lock.

Execution proceeds from S_0 by applying *non-deterministically* the semantic rules depicted in Figs. 2 and 3. We omit the rules for **if** and **while** as they are standard, they can be found in the App. B. The operational semantics is given in a rewriting style where a step is a transition of the form $a\underline{b} \xrightarrow{c} \underline{b'}\ n$, where the dotted underlining indicates that term b is rewritten into b′; we look up the term a but do not modify it and hence it is not included in the subsequent state; term n is newly added to the state; and c is the cost of the transition according to the cost model. We omit c if it is zero. For simplicity, we assume that only local variables are assigned in the rules New-Object,Async-Call and Get. Fields can still be modified with the rule Assignment. Transitions are applied according to the rules as follows.

$$\frac{t' = t}{tsk(tk, o, l, \{\textbf{await duration}(p_l); s\}) \; ob(o, a, tk) \; clk(t) \rightarrow tsk(tk, o, l, \{\textbf{until}\,(p_l + t'); s\})} \; (\textsc{await duration})$$

$$\frac{t' = eval(p_l, l), \; t' > t}{tsk(tk, o, l, \{\textbf{until}\,(p_l); s\}) \; ob(o, a, tk) \; clk(t) \rightarrow ob(o, a, \bot) \; tsk(tk, o, l, \{\,\mathsf{untilp}\,(p_l); s\})} \; (\textsc{until2})$$

$$\frac{t' = eval(p_l, l), \; t' \leq t}{ob(o, a, tk) \; tsk(tk, o, l, \{\textbf{until}\,(p_l); s\}) \; clk(t) \rightarrow tsk(tk, o, l, s)} \; (\textsc{until1})$$

$$\frac{t' = eval(p_l, l), \; t' \leq t}{ob(o, a, \bot) \; tsk(tk, o, l, \{\,\mathsf{untilp}\,(p_l); s\}) \; clk(t) \rightarrow ob(o, a, tk) \; tsk(tk, o, l, s)} \; (\textsc{until3})$$

$$\frac{canAdv, t' = t + 1}{clk(t) \rightarrow clk(t')} \; (\textsc{tick})$$

Fig. 3. Time semantics

Activate: A task that has a take statement obtains its object's lock if it is free.
Release: **release** unconditionally yields the object's lock so any other task of the same object can take it. The instruction take is added to the pending instructions.
Assignment: The variable v gets assigned to the result of evaluating the pure expression p given the current state a and l (denoted $eval(p, a, l)$). The notation $l' = l[v \rightarrow val]$ denotes that the mapping l' is equal to l for all values except v where $l'(v) = val$. We assume local variables and fields are always different. If v is a local variable, l is updated $l' = l[v \rightarrow val]$. If v is a field, a is updated $a' = a[v \rightarrow val]$. We express that as $l' \uplus a' = (l \uplus a)[v \rightarrow val]$.
Cost: c resource units are consumed where c is the result of evaluating the expression p in the current state. As the object o is not modified, it is not included in the resulting state.
Await1: The (local) future variable we are waiting for points to a finished task and the await is completed. The finished task t_1 is only looked up but it does not disappear from the state as its return value may be needed later on. To complete the await, the object's lock must be free or in the task tk. As a result of applying the rule, the task tk obtains (or keeps) the object's lock.
Await2: If the task we are awaiting for is not finished, We release the lock.
Return: When **return** is executed, the return value val is stored (by adding the instruction $\epsilon(val)$) so that it can be obtained by the future variables that reference the task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is finished.
Get: An $x = y.\textbf{get}$ instruction waits for the (local) future variable but without yielding the lock. It stores the value associated with the future variable y in x.
New-Object: An active task tk in object o creates an object o' of type C, its fields are initialized with default values and the given parameters $eval(\overline{p}, a, l)$ ($initAtts$) and o' is introduced to the state with a free lock.

ASYNC_CALL: A method call creates a new task (the initial state is created by *buildLoc* using the calling values $eval(\overline{p}, a, l)$) with a fresh task identifier t_1 which is associated to the corresponding future variable y in l'.
AWAIT DURATION: It generates an $\textbf{until}(p_l + t')$ instruction that substitutes the original $\textbf{await duration}(p_l)$ where t' is a constant with the current time of the clock.
UNTIL1: If the current time is greater or equal than the time we are waiting for and the task has the object's lock, we complete the $\textbf{until}(p_l)$ instruction. The time we are waiting for is the result of evaluating the pure local expression p_l. We write $eval(p_l, l)$ to emphasize that p_l does not depend on the object's state.
UNTIL2: If we have not reached the time we are waiting for, we release the lock. we substitute $\mathsf{untilp}(p_l)$ for $\textbf{until}(p_l)$. $\mathsf{untilp}(p_l)$ will be able to take back the lock. Its behavior is analogous to the auxiliary instruction take with respect to **release**.
UNTIL3: If the current time is greater or equal than the time we are waiting for and the lock is free, we complete the instruction $\mathsf{untilp}(p_l)$ and obtain the lock.
TICK: The rule tick makes the time advance one unit. It is only applicable if no task can progress. This behavior reflects the run-to-completion policy and is enforced by the function *canAdv*. The latter is true only when *no other semantic rule can be applied* to any task of the current state.

3 Cost over Time

The traditional definition of cost used in static resource analysis is concerned with the total amount of resources consumed during the complete execution of a program. Given program $\mathcal{P}$ and input values $\overline{val}$, we can define the cost of a trace as follows:

Definition 1 (Cost of a trace). *Let* $tr(\overline{val}) = S_0(\overline{val}) \xrightarrow{c_0} S_1 \xrightarrow{c_1} \cdots$ *be a (possibly infinite) execution trace where* $\overline{val}$ *are the input values of the main method and* c_i *the cost of the semantic transition leading to state* S_{i+1}*. The cost of executing* $tr(\overline{val})$ *is* $Cost_{tr(\overline{val})} = \sum_i c_i$ *i.e. the sum of the costs of the all transitions in the trace* tr*.*

Let $\mathcal{TR}_\mathcal{P}$ be the set of all possible traces of a program for all possible input values, we define an upper bound of such a program:

Definition 2 (Upper bound). *Let* $\overline{T}$ *be the types of the input parameters of a program* $\mathcal{P}$*, an upper bound function is defined as* $ub : \overline{T} \to \mathbb{R}$*.* $ub(\overline{x})$ *is a valid upper bound of* $\mathcal{P}$ *iff for all input values* $\overline{val}$ *and traces* $tr(\overline{val}) \in \mathcal{TR}_\mathcal{P}$*:* $Cost_{tr(\overline{val})} \leq ub(\overline{val})$*.*

A *conditional upper bound* is an upper bound ub with a precondition such that the upper bound is only valid when the input parameters satisfy the precondition.

The main problem with this upper bound definition is that it does not give any information on how and *when* the resources are consumed. In the example from Fig. 1 the total resource consumption is not bounded since it consists of an

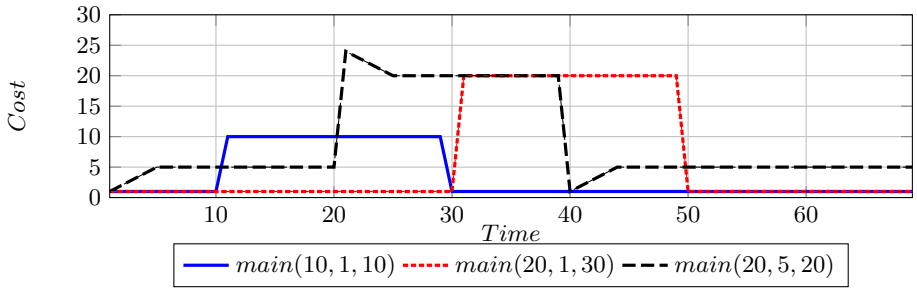

Fig. 4. Resource consumption profiles of the model from Fig. 1 which coincide with their inferred upper bounds

infinite execution that consumes a non-zero amount of resources at each iteration. For programs with timed behavior, we are interested in their consumption over time, that is, the maximum possible amount of resources consumed at a moment in time t.

Example 2. In Fig. 4, we show the resource consumption over time of our program from Fig. 1 given different possible input parameters. We generated values by executing the program with specific input values, and derived concrete values from our *analytically obtained* upper bounds. In all three cases, the upper bounds are precise. The first parameter of main determines the height of the midnight window, the second the duration of the calls, and the third when the midnight window takes place. Note that when a call is started, its cost is distributed in time. In the third profile, we can see how at the start of the midnight window there are still some calls in progress which cause an extra peak in resource consumption. On the other hand, when the midnight window ends, it takes some time until the cost goes back to 5 as calls are started one at a time.

Definition 3 (Cost of a trace in time). *Let* $tr(\overline{val}) = S_0(\overline{val}) \xrightarrow{c_0} S_1 \xrightarrow{c_1} \cdots$ *be an execution trace as above. The cost of executing such a trace at time* tg *is the sum of the cost of all transitions where the time is* tg*.* $Cost_{tr(\overline{val})}(tg) = \sum_{i|clk(tg)\in S_{i+1}} c_i$.

The goal of our analysis is to approximate (safely) the behavior of systems over time by obtaining a set of (conditional) upper bounds that are parametrized by the time we want to observe.

Definition 4 (Time-parametrized upper bound). *Let* $\overline{T}$ *be the types of the input parameters of a program* $\mathcal{P}$*. A time-parametrized upper bound function has type* $ub : (\overline{T}, \mathbb{N}) \to \mathbb{R}$*. Then* $ub(\overline{x}, tg)$ *is a valid time-parametrized upper bound of* $\mathcal{P}$ *iff for all input values* $\overline{val}$*,* $tg \in \mathbb{N}$ *and* $tr(\overline{val}) \in \mathcal{TR}_{\mathcal{P}}$*:* $Cost_{tr(\overline{val})}(tg) \leq ub(\overline{val}, tg)$.

$$
\begin{array}{lll}
1 : \alpha(\textbf{cost}\ c) & \Rightarrow \textbf{if}\,(\texttt{tg==}\textbf{now}())\ \textbf{cost}\ \texttt{c}; & \\
2 : \alpha(y = x!m(\overline{v})) & \Rightarrow y = x!m(\overline{v}, tg) & \\
3 : \alpha(s_1; s_2) & \Rightarrow \alpha(s_1); \alpha(s_2) & \\
4 : \alpha(\textbf{if}\,(p)\ s_1\ \textbf{else}\ s_2) & \Rightarrow \textbf{if}\,(p)\ \alpha(s_1)\ \textbf{else}\ \alpha(s_2) & \\
5 : \alpha(\textbf{while}(p)\ s) & \Rightarrow \textbf{while}(p)\ \alpha(s) & \\
6 : \alpha(s) & \Rightarrow s & \text{otherwise}
\end{array}
$$

Fig. 5. Transformation over the statements

4 Program Transformation

To analyze timed programs we use a sound program transformation that generates untimed programs which can be analyzed by standard tools. In a first step (Sec. 4.1) we transform a given program $\mathcal{P}$ into a program $\mathcal{P}_{tg}$ such that the resource consumption of $\mathcal{P}$ at time tg is the same as the total resource consumption of $\mathcal{P}_{tg}$. Then we perform a second transformation (Sec. 4.2) that safely models the advance of time.

4.1 Cost Model Specialization

To achieve the desired effect, every method (including the main method) is equipped with an extra parameter tg that holds the time we want to observe. We assume tg is disjoint with the existing variable names. The statements of each method are transformed using the function α defined in Fig. 5. A method declaration $T\ m(\overline{T}\ \overline{x})\{s; \textbf{return}\ p;\}$ becomes $T\ m(\overline{T}\ \overline{x}, Int\ tg)\{\alpha(s); \textbf{return}\ p;\}$.

The function α defined in Fig. 5 works as follows: (1) we substitute every cost statement by a conditional statement that only consumes resources if the current time is tg; (2) tg is added to the method calls as a parameter; (3),(4) and (5) the tranformation of non-atomic statements is the transformation of the statements that compose them; and (6) the remaining statements are not modified.

To prove soundness of the transformation, we define the concept of extended local state and α-equivalent state.

Definition 5 (Extended local state). *Let l be a local state of a task, then $l_{tg} = l + [tg \to y]$ is a mapping where for every $x \in Dom(l)$, $l(x) = l_{tg}(x)$ and $l_{tg}(tg) = y$. We call l_{tg} an extended local state of l with tg.*

Definition 6 (α-equivalent state). *Define an extended execution state S^α:*
$S^\alpha = \{e \mid e \in S \wedge e \neq tsk(tk, o, l, s)\} \cup$
$\{tsk(tk, o, l + [tg \to tg_o], \alpha(s)) \mid tsk(tk, o, l, s) \in S\}$
We apply the transformation α to all the pending statements of all the tasks in S and take their extended local state. The objects and the clock remain unchanged; tg_0 denotes the unique initial value of the variable tg throughout the execution.

We use the function α for two purposes: the function defines the transformation over the original program and, at the same time, we use it to establish a relation between execution states of the original and transformed program.

$$
\begin{array}{lll}
1: \tau(y = x!m(\overline{p})) & \Rightarrow y = x!m(\overline{p}, time) & \\
2: \tau(\textsf{until}\,(p_l)) & \Rightarrow release; time = max(p_l, time); & \\
3: \tau(\textsf{await duration}(p_l)) & \Rightarrow release; time = max(time + p_l, time); & \\
4: \tau(\textsf{await}\ y?) & \Rightarrow \textsf{await}\ y?; time = max(time, y.time); & \\
5: \tau(\textsf{untilp}\,(t)) & \Rightarrow take; time = max(t, time); & \\
6: \tau(s_1; s_2) & \Rightarrow \tau(s_1); \tau(s_2) & \\
7: \tau(\textsf{if}\,(p)\ s_1\ \textsf{else}\ s_2) & \Rightarrow \textsf{if}\,(p)\ \tau(s_1)\ \textsf{else}\ \tau(s_2) & \\
8: \tau(\textsf{while}(p)\ s) & \Rightarrow \textsf{while}(p)\ \tau(s) & \\
9: \tau(s) & \Rightarrow s & \text{otherwise}
\end{array}
$$

Fig. 6. Second transformation over the statements

Theorem 1 (Soundness). *Given a program $\mathcal{P}$ and its transformed program $\mathcal{P}_{tg}$, for every trace $tr(\overline{val}) \in \mathcal{TR}_{\mathcal{P}}$ whose final state is S, there is a trace $tr'(\overline{val}, tg_0) \in \mathcal{TR}_{\mathcal{P}_{tg}}$ whose final state is S^α and $Cost_{tr(\overline{val})}(tg_0) = Cost_{tr'(\overline{val}, tg_0)}$.*

This states that each behavior of the original program can be simulated by the transformed program. In addition, the transformed program $\mathcal{P}_{tg}$ captures the amount of the total cost consumed at time tg_0 (the input parameter).

Proof idea By induction on the length of a trace. We show that for each step $S \xrightarrow{c} S_{new}$ in $\mathcal{P}$, there is a step or a sequence of steps in the transformed program between α-equivalent states $S^\alpha \xrightarrow{c_1} S_1 \xrightarrow{c_2} \cdots \xrightarrow{c_n} S^\alpha_{new}$. Moreover, the cost $\sum_{i=1}^{m} c_i$ is c if $clk(tg_0) \in S$ and 0 if $clk(t') \in S$ for $t' \neq tg_0$. We can apply the same semantic rules in the original and the transformed program for all cases except for rule (Cost) and reach an α-equivalent state (the addition of the variable tg does not affect the behavior). For rule (Cost) we show that the cost is consumed only if the time is equal to tg (thanks to the conditional statement) and the resulting state is an α-equivalent state. A detailed proof is in App. B.1. □

Corollary 1. *An upper bound of $\mathcal{P}_{tg}$ is a time-parametrized upper bound of $\mathcal{P}$.*

4.2 Rendering Time Explicit

Now we eliminate timing behavior from our programs. That transformation takes a program $\mathcal{P}_{tg}$ and generates $et\mathcal{P}_{tg}$ (explicit time program). The program $et\mathcal{P}_{tg}$ does not contain any timing constructs. The transformation adds a new parameter *time* to each method of the program (disjoint from all existing variable names). We transform the program in such a way that the local variable *time* of every task coincides with the global clock whenever the task is executing.

After the execution of each method the value of variable *time* is returned together with its original return value. For the technical realization we assume that future variables now store a pair of values. The original value is accessed with y.**get** and the time value with $y.time$. A method declaration

Transformed call

```
1 call(dur,tg,time){
2  while (dur>0) {
3    if(tg==time)
4        cost 1;
5    dur= dur-1;
6    release;time=max(
     time+1,time);
7   }
8  return time }
```

Transformed normalBehavior

```
9 normalBehavior(mx,dur,tc,tg,time){
10  if (time>tc && time<tc+20)
11   midnightWindow(mx,dur,tc,tg,time);
12  else{
13   ts!call(dur,tg,time);
14   release; time=max(time+1,time);
15   normalBehavior(mx,dur,tc,tg,time);
16  }
17  return time }
```

Fig. 7. Some methods of the transformed program

$T\ m(\overline{T}\ \overline{x})\{s;\textbf{return}\ p\}$ becomes $(T, Int)\ m(\overline{T}\ \overline{x}, Int\ time)\{\tau(s);\textbf{return}\ (p, time)\}$. Methods that did not return a value now return *time*.

Method statements are transformed using the function τ defined in Fig. 6. (1) Whenever a method is called, the current time is passed as a parameter. Hence, new tasks have a local time that starts when the task is created. For the **until**(p_l) (2) and **await duration**(p_l) (3) statements the lock is released and time is updated. (4) When we execute an **await** y? we compare the current local time and the time returned by the future and keep the maximum. (5) untilp(t) is an auxiliary statement used when the lock of the task has been released. We substitute it by the auxiliary instruction take generated by **release** and then update the time value. This way the transformations of **until**(p_l) and untilp(t) are coherent. Note that untilp(t) cannot appear in a program because it is an auxiliary instruction. However, by defining its transformation, we can use τ to define τ-equivalent execution states and prove the soundness of the transformation (see Def. 8 and Thm. 4.2). (6) (7) and (8) the tranformation of non-atomic statements is the transformation of the statements that compose them. (9) The remaining statements are not modified. Finally, every reference to *now*() is substituted by a reference to the new local variable *time*.

Example 3. In Fig. 7 we show some of the transformed methods. Every method has two additional parameters *tg* and *time*; the references to *now*() have been substituted by *time* (lines 3 and 10); The **await duration**(1) instructions have been replaced by a **release** followed by an update of the variable *time* (lines 6 and 14); the methods return the time (lines 8 and 17) and the cost statement in method call is wrapped into a conditional statement (line 3).

This transformation is valid for non-blocking programs. Intuitively, a program is blocking if we can have a situation where a task is waiting for the completion of another without releasing its object's lock (only possible with y.**get** instructions). In that case, other tasks in the same object could be delayed in time although they are ready to execute because they cannot access the lock. A sufficient condition to guarantee that a program is non-blocking is that every y.**get** instruction is preceded by an **await** y?. Given that condition, the task related to

y is guaranteed to be finished when $y.\textbf{get}$ is reached and the $y.\textbf{get}$ instruction will be completed immediately without blocking.

Definition 7 (Non-blocking program). *A program $\mathcal{P}$ is non-blocking iff for every reachable state S, if $tsk(tk, o, l, y.\textbf{get}; s) \in S$ such that $l(y) = tk_1$, then $tsk(tk_1, o_1, l_1, \epsilon(v)) \in S$.*

Similar to the previous transformation we can define a τ-equivalent execution state of S by applying the transformation to all the pending statements of all the tasks in S, obtaining their extended local state and removing the clock. The objects remain unchanged. A necessary condition for a state to be τ-equivalent is that the values of the local variables *time* in all the tasks that hold the object's lock correspond to the global clock. The tasks without the lock may have an outdated time value (smaller or equal than the global clock).

Definition 8 (τ-equivalent state). *Let S be a state of the original program with $clk(t) \in S$. Then a τ-equivalent state S^τ to S is defined as:*
$S^\tau = \{ob(o, a, lk) \mid ob(o, a, lk) \in S\} \cup$
$\quad \{tsk(tk, o, l + [time \to T_{tk}], \{\tau(s); \textbf{return } (p, time)\})$
$\quad\quad \mid tsk(tk, o, l, \{s; \textbf{return } p\}) \in S\}$
where T_{tk} represents the current time of each task. We require that $T_{tk} \leq t$ for all tasks and $T_{tk} = t$ for the tasks that have the lock.

Theorem 2 ($et\mathcal{P}_{tg}$ Simulates $\mathcal{P}_{tg}$). *Given a non-blocking program $\mathcal{P}_{tg}$ and its transformed program $et\mathcal{P}_{tg}$, then for every trace $tr(\overline{val}) \in \mathcal{TR}_{\mathcal{P}_{tg}}$ with states $S_{1..n}$ there is a trace $tr'(\overline{val}, 1) \in \mathcal{TR}_{et\mathcal{P}_{tg}}$ that contains the τ-equivalent states $S^\tau_{1..n}$, in the same order, S^τ_n being the final state of tr' and $Cost_{tr(\overline{val})}(tg) = Cost_{tr'(\overline{val},1)}$. Note that tr' can have intermediate states that do not have an equivalent in tr. Additionally, consecutive states in tr might have a single τ-equivalent state in tr'.*

Proof idea By induction on the length of a trace. The base case is trivial. In the inductive step, we assume that we have a trace $tr \in \mathcal{TR}_{\mathcal{P}_{tg}}$ of length n and a trace $tr' \in \mathcal{TR}_{et\mathcal{P}_{tg}}$ such that tr' contains n τ-equivalent states to the ones appearing in tr in the same order (tr' can have additional intermediate states). The cost of tr and tr' is equal. We prove that for any step from the final state of tr $S \xrightarrow{c} S_{new}$, there is a step or sequence of steps in the transformed program that reaches a τ-equivalent state with the same resource consumption.

For the rules that are applied to a task with a lock and that are not affected by τ we apply the same rule to the transformed program and obtain the desired state. The fact that the program is non-blocking implies that when we apply (Tick), all the locks must be free. If we apply (Tick) in $\mathcal{P}_{tg}$, all the local variables *time* become outdated but as the tasks do not have the lock, the state in tr' is still τ-equivalent. For the rules (Await1), (Until1), (Until2), and (Until3) we have to prove that the assigned time in tr' corresponds to the clock in tr. The idea is that since the rules are applicable (the awaited task is finished or the awaited time is reached), the clock cannot advance before they are applied (this is only

valid for non-blocking programs). Therefore, $max(time, v.time)$, $max(p_l, time)$ and $max(t, time)$ yield the correct value of the clock and the reached state in tr' is τ-equivalent. Similarly, since the rule (ACTIVATE) is applicable (a **release** or task creation), the clock cannot have advanced and the variable $time$ is not outdated. The statement **await duration**(d) can be reduced to **until**$(t + d)$. A detailed proof can be found in App. B.2. □

Corollary 2. *An upper bound of $et\mathcal{P}_{tg}$ is an upper bound of $\mathcal{P}_{tg}$.*

Note that theorem states the soundness of the transformation but not its completeness. The transformation is in fact not complete. That is, the transformed program can have additional behaviors that are not present in the original program. In particular, if we have an **until**(p_l) statement, it is possible that when executed $eval(p_l, l) \leq t$ and the semantic rule UNTIL1 is applied. This rule does not release the lock. However, the corresponding transformed program will always release the lock $\tau($**until**$(p_l)) = release; time = max(p_l, time);$. The transformed program could then schedule another pending task whereas the original program cannot do the same. However, we do not expect any additional loss of precision for this reason because the later analyses already conservatively approximate a possible release of the lock in any instruction that contains a conditional release (such as **await** y?).

5 Analyzing Untimed Programs

We are able to apply existing tools for resource analysis on transformed programs to obtain an upper bound or a set of upper bounds, but it is crucial to ensure high precision of the analysis (cf. Fig. 4). To achieve this we combine the frontend of the resource analyzer COSTABS [1] with the more recent solver CoFloCo[1] [10]. The input of COSTABS is an untimed ABS program (i.e., a superset of the language in Sect. 2) and generates a set of *cost equations* that characterize the cost of the program and can be solved by CoFloCo. Cost equations are non-deterministic recurrence equations annotated with constraints.

Example 4. COSTABS+CoFloCo generates 15 conditional upper bounds for the main method of our example. We show some of them, the complete data from the example can be found in App. A.

#	Upper Bound	Precondition
6	tg	$(mx \geq 1 \wedge tg \geq 2 \wedge dur \geq tg + 1 \wedge tc \geq tg + 1)$
8	$mx + tc$	$(mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc+2 \wedge dur \geq tg+1 \wedge tc+18 \geq tg) \vee (dur = tg \wedge mx \geq 1 \wedge tc \geq 2 \wedge tc + 18 \geq dur \wedge dur \geq tc + 2) \vee (tg = tc + 19 \wedge mx \geq 1 \wedge tg \geq 20 \wedge dur \geq tg + 1) \vee (tg = tc + 1 \wedge mx \geq 1 \wedge tg \geq 2 \wedge dur \geq tg + 1)$
12	$tg - tc - 19$	$(mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc+21 \wedge tg \geq dur+tc \wedge dur+tc+18 \geq tg)$
15	$mx + dur + tc - tg$	$(mx \geq 1 \wedge tg \geq dur+1 \wedge tg \geq tc+2 \wedge tc+18 \geq tg \wedge dur+tc \geq tg+2)$

[1] Other resource analysis tools could be used as long as they admit a parametric cost model, i.e., it is must be possible to establish the cost at each program point.

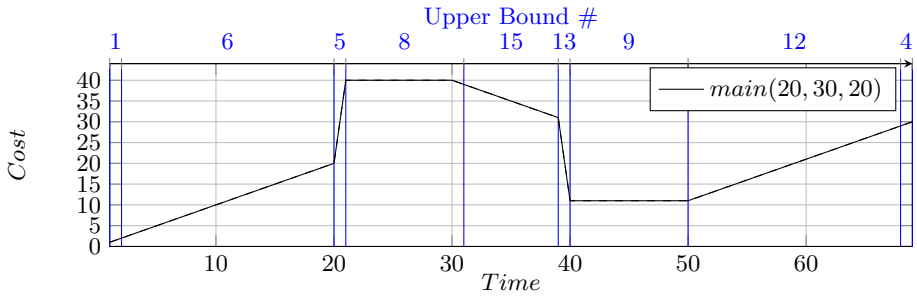

Fig. 8. A resource consumption profile generated to observe other upper bounds

```
1Unit m(p,q,r,s,n){
2 await duration(nondet);
3 this!r(r);
4 while(n>0){
5  this!p(p);
6  y=this!q(q);
7  await y?;
8  await duration(nondet);
9  await duration(1);
10  n=n-1;
11 } this!s(s);
12}
```

```
13Unit r/p/q/s(c){
14 await duration(nondet);
15 cost c;
16}
```

#	Upper bound	Precondition
1	$max(s,q)+r+n*p$	$n \geq 3 \wedge tg \geq n$
2	$q+r+(tg+1)*p$	$tg \geq 2 \wedge n \geq tg+2$

Fig. 9. A timed model annotated to obtain peak cost

In Fig. 4, we could observe the precision of the upper bounds that correspond to the resource consumption in simulations for concrete inputs. By examining the upper bounds, we can get a good approximation of the different behaviors of the program. Preconditions can be useful to see under which conditions a program behaves differently. In particular, we can generate test data from preconditions that generate a certain behavior. We used this idea to create execution patterns of our example that were not observed in the initial simulations, for example upper bound #15 in Fig. 8. The upper bounds here are also tight and the intervals where each upper bound is valid are annotated with its number.

6 Inferring Peak Cost

In Fig. 9, we introduce a new example to illustrate how explicit time primitives can be used for the purpose of modelling peak cost. The example is an adaptation of the running example from [2]. In the original example, the methods r,p,q and s are left unspecified. We assume they have all the same implementation and consume the amount c of resources which is given by the input parameter. With peak cost we mean the following: at any time tg there is a set of pending tasks that could be executed simultaneously. Let $C(tg)$ be the sum of the cost of these

tasks at tg. Then the peak cost is the maximum value that $C(tg)$ attains. In [2] the notion of simultaneity is defined based on synchronization points **await** y? (line 7). In order to emulate the behavior that corresponds to peak cost analysis, we simply make time advance one unit right after each synchronization point (line 9 in gray). In addition, with this concept of simultaneity, tasks do not need to start executing immediately but can be delayed an indefinite amount of time. We can model that by inserting a non-deterministic **await duration**($nondet$) whenever a method starts or restarts to execute (lines 2, 8 and 14).

After adding these annotations we apply our analysis and obtain upper bounds of the peak cost. On the right side of Fig. 9 we display the two main conditional upper bounds (we leave out some border cases) computed by our analysis. The first one corresponds to the result obtained by [2] and captures that there can be n pending instances of task p to be executed simultaneously, but only one instance of q as the program waits for each instance to finish before calling the next one. This is the peak cost for executing the complete program as reflected in the precondition $tg \geq n$. The second upper bound gives more fine-grained information on how the tasks might accumulate in the loop as time tg passes. This kind of analysis cannot be obtained with the method of [2], because it involves analyzing timed behavior.

7 Related Work, Conclusions and Future Work

To the best of our knowledge, we present the first resource analysis for timed concurrent object-oriented programs. The analysis is based on an inexpensive program transformation into untimed programs. The untimed programs can be analyzed with existing tools for resource analysis [7, 8, 10, 11, 15] so our analysis directly benefits from any improvement of these techniques. Because the timing behavior is parametric, our analysis opens multiple possibilities for reasoning about concurrent programs. For instance, we could generate timing annotations according to a network model and observe how processing power is consumed.

The most closely related work is about resource analysis of concurrent objects [1, 2]. The main focus of [1] is on how to deal with shared memory in concurrent applications and they propose the notion of *cost centers* as a way of computing the cost of different components (objects) separately. In contrast, our approach analyzes cost over time—it can be seen as a layer on top of their analysis. As mentioned in Sec. 6, paper [2] does not measure total cost but *peak cost*, i.e., the maximum cost a component can consume at any given time. The crucial difference is that in our analysis the cost is determined by the explicit time behavior, whereas [2] abstracts away from timed behavior and uses the maximal degree of parallelism obtained from a *May-Happen-in-Parallel* analysis [3]. Because we have explicit time primitives, we can infer upper bounds that depend on time. Additionally, we can infer peak cost parametrized with time (see Sec. 6). The authors of paper [12] define a type system to infer upper bounds on two cost models (*work* and *depth*) for functional programs. *work* is the total cost of the

program and *depth* is similar to the maximum time that can be reached in any part of the program if we make time advance each time resources are consumed.

Timed automata [4] have been widely used in the analysis of timed systems. In particular, they have been applied to concurrent object-oriented programs [9], however, with a different focus: to transform such programs into timed automata to check their schedulability. *Priced timed automata* [5] model systems with both time and resources, but they differ from our setting in crucial ways, having continuous time and no input parameters. They also focus on different properties, such as *minimum-cost reachability*.

Our current analysis does not support blocking programs. When we have a blocking program, there can be extra delays in tasks after instructions that release an object's lock (**await** $y?$, **release**, and **await duration**(t)). These extra delays depend on the blocking instructions that might interleave with such tasks and they must be taken into account to update the *time* variable correctly. How to approximate blocking programs safely and precisely is left to future work.

References

1. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *P*roc. of APLAS'11, volume 7078 of *L*NCS, pages 238–254. Springer, Dec. 2011.
2. E. Albert, J. Correas, and G. Román-Díez. Peak Cost Analysis of Distributed Systems. In *2*1st International Static Analysis Symposium (SAS'14), volume 8723 of *L*NCS, pages 18–33. Springer-Verlag, 2014.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of may-happen-in-parallel in concurrent objects. In H. Giese and G. Rosu, editors, *F*ORTE, volume 7273 of *L*NCS, pages 35–51. Springer-Verlag, 2012.
4. R. Alur and D. L. Dill. A theory of timed automata. *T*heoretical Computer Science, 126:183–235, 1994.
5. G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced time automata. In M. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *H*SCC, volume 2034 of *L*NCS, pages 147–161. Springer, 2001.
6. J. Bjørk, F. Boer, E. Johnsen, R. Schlatte, and S. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *I*nnov. Syst. Softw. Eng., 9(1):29–43, Mar. 2013.
7. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *T*ACAS, 2014.
8. Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *P*LDI, 2015. To Appear.
9. F. de Boer, T. Chothia, and M. Jaghoori. Modular schedulability analysis of concurrent objects in creol. In F. Arbab and M. Sirjani, editors, *F*SEN, volume 5961 of *L*NCS, pages 212–227. Springer, 2010.
10. A. Flores Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In J. Garrigue, editor, *A*PLAS, volume 8858 of *L*NCS, pages 275–295. Springer, Nov. 2014.
11. S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *P*OPL, pages 127–139, New York, NY, USA, 2009. ACM.

12. J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs. In *E*SOP, 2015. To Appear.
13. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *P*roc. of FMCO 2010, volume 6957 of *L*NCS, pages 142–164. Springer-Verlag, 2011.
14. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *J*ournal of Logical and Algebraic Methods in Programming, 2014. DOI: 10.1016/j.jlamp.2014.07.001.
15. M. Sinn, F. Zuleger, and H. Veith. A simple and scalable approach to bound analysis and amortized complexity analysis. In *C*AV, volume 8559 of *L*NCS, pages 743–759. Springer, 2014.

The appendix is provided for the reviewers' convenience, it is not part of the paper

A Complete Solution of the Example

We analyzed the example program with the following precondition: $dur \geq 1 \wedge mx \geq 1 \wedge tc \geq 1 \wedge tg \geq time = 1$. It determines that dur mx and tc are positive, the initial time is 1 and that the time we want to observe (tg) is positive. We added the precondition to avoid analyzing uninteresting cases (such as when $tg < time$ at the beginning of the program). The complete results of the analysis are in Fig. 10.

#	Upper Bound	Precondition
1	1	$(dur = 1 \wedge tc = tg \wedge mx \geq 1 \wedge tc \geq 2) \vee (dur = 1 \wedge mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc + 21) \vee (dur = 1 \wedge mx \geq 1 \wedge tg \geq 2 \wedge tc \geq tg + 1) \vee (tc + 20 = tg \wedge 20 \geq dur \wedge mx \geq 1 \wedge dur \geq 1 \wedge tc \geq 1) \vee (tg = 1 \wedge mx \geq 1 \wedge dur \geq 1 \wedge tc \geq 1)$
2	2	$(dur = 2 \wedge mx \geq 1 \wedge tg \geq 3 \wedge tc \geq tg + 1) \vee (dur = 2 \wedge tc = tg \wedge mx \geq 1 \wedge tc \geq 3) \vee (dur = 21 \wedge tc + 20 = tg \wedge mx \geq 1 \wedge tc \geq 2) \vee (dur = 2 \wedge mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc + 21) \vee (dur = 21 \wedge tc = 1 \wedge tg = 21 \wedge mx \geq 1) \vee (dur = 2 \wedge tg = 2 \wedge mx \geq 1 \wedge tc \geq 2)$
3	mx	$(mx \geq 1 \wedge dur \geq 1 \wedge tc \geq 1 \wedge tc + 18 \geq tg \wedge tg \geq dur + tc) \vee (tc + 19 = tg \wedge 19 \geq dur \wedge mx \geq 1 \wedge dur \geq 1 \wedge tc \geq 1)$
4	dur	$(tc = tg \wedge mx \geq 1 \wedge dur \geq 3 \wedge tc \geq dur + 1) \vee (mx \geq 1 \wedge dur \geq 3 \wedge tc \geq 1 \wedge tg \geq dur + tc + 19) \vee (mx \geq 1 \wedge dur \geq 3 \wedge tg \geq dur + 1 \wedge tc \geq tg + 1) \vee (dur = tg \wedge mx \geq 1 \wedge dur \geq 3 \wedge tc \geq dur)$
5	tc	$(tg = tc \wedge mx \geq 1 \wedge tg \geq 2 \wedge dur \geq tg + 1)$
6	tg	$(mx \geq 1 \wedge tg \geq 2 \wedge dur \geq tg + 1 \wedge tc \geq tg + 1)$
7	$mx + 1$	$(dur + tc = tg + 1 \wedge mx \geq 1 \wedge tc \geq 2 \wedge tg \geq tc + 2 \wedge tc + 18 \geq tg) \vee (dur = 2 \wedge tc + 1 = tg \wedge mx \geq 1 \wedge tc \geq 2) \vee (tc = 1 \wedge tg = dur \wedge 20 \geq tg \wedge mx \geq 1 \wedge tg \geq 3) \vee (dur = 20 \wedge tc + 19 = tg \wedge mx \geq 1 \wedge tc \geq 2) \vee (dur = 2 \wedge tc = 1 \wedge tg = 2 \wedge mx \geq 1)$
8	$mx + tc$	$(mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc + 2 \wedge dur \geq tg + 1 \wedge tc + 18 \geq tg) \vee (dur = tg \wedge mx \geq 1 \wedge tc \geq 2 \wedge tc + 18 \geq dur \wedge dur \geq tc + 2) \vee (tg = tc + 19 \wedge mx \geq 1 \wedge tg \geq 20 \wedge dur \geq tg + 1) \vee (tg = tc + 1 \wedge mx \geq 1 \wedge tg \geq 2 \wedge dur \geq tg + 1)$
9	$dur - 19$	$(tc + 20 = tg \wedge mx \geq 1 \wedge dur \geq 22 \wedge tc + 19 \geq dur) \vee (dur = tg \wedge mx \geq 1 \wedge tc \geq 2 \wedge dur \geq tc + 21) \vee (dur + tc = tg + 1 \wedge mx \geq 1 \wedge dur \geq 22 \wedge tc \geq 2) \vee (dur = tc + 20 \wedge dur = tg \wedge mx \geq 1 \wedge dur \geq 22) \vee (mx \geq 1 \wedge tg \geq dur + 1 \wedge tg \geq tc + 21 \wedge dur + tc \geq tg + 2) \vee (tc = 1 \wedge dur = tg \wedge mx \geq 1 \wedge dur \geq 22)$
10	$tc + 1$	$(tg = tc + 20 \wedge mx \geq 1 \wedge tg \geq 21 \wedge dur \geq tg + 1)$
11	$tg + -19$	$(mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc + 21 \wedge dur \geq tg + 1)$
12	$tg - tc - 19$	$(mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc + 21 \wedge tg \geq dur + tc \wedge dur + tc + 18 \geq tg)$
13	$mx + dur - 19$	$(tc + 19 = tg \wedge mx \geq 1 \wedge dur \geq 21 \wedge tc + 18 \geq dur) \vee (dur = tc + 19 \wedge dur = tg \wedge mx \geq 1 \wedge dur \geq 21)$
14	$mx + dur - 1$	$(tc + 1 = tg \wedge mx \geq 1 \wedge dur \geq 3 \wedge tc \geq dur) \vee (dur = tc + 1 \wedge dur = tg \wedge mx \geq 1 \wedge dur \geq 3)$
15	$mx + dur + tc - tg$	$(mx \geq 1 \wedge tg \geq dur + 1 \wedge tg \geq tc + 2 \wedge tc + 18 \geq tg \wedge dur + tc \geq tg + 2)$

Fig. 10. Complete results

$$\frac{eval(p,a,l) = true}{tsk(tk,o,l,\{\,\textsf{if}\,(p)\ s_1\ \textsf{else}\ s_2;s\}) \cdots\cdots ob(o,a,tk) \to tsk(tk,o,l,s_1;s)} \text{(IF-TRUE)}$$

$$\frac{eval(p,a,l) = false}{tsk(tk,o,l,\{\,\textsf{if}\,(p)\ s_1\ \textsf{else}\ s_2;s\}) \cdots\cdots ob(o,a,tk) \to tsk(tk,o,l,s_2;s)} \text{(IF-FALSE)}$$

$$\frac{eval(p,a,l) = true}{ob(o,a,tk)\ tsk(tk,o,l,\{\textsf{while}(p)\ s_1;s\}) \cdots\cdots \to tsk(tk,o,l,\{s_1;\textsf{while}(p)\ s_1;s)} \text{(WHILE-TRUE)}$$

$$\frac{eval(p,a,l) = false}{ob(o,a,tk)\ tsk(tk,o,l,\{\textsf{while}(p)\ s_1;s\}) \cdots\cdots \to tsk(tk,o,l,s)} \text{(WHILE-FALSE)}$$

Fig. 11. Semantic rules for **if** and **while**

B Proofs

In order to perform the proofs according to the semantics, we include the semantic rules for **if** and **while** in Fig. 11.

B.1 Proof of Theorem 1

Proof. We perform the proof using induction on the length of the trace:

Base case The cost in a trace of length 0 in both $\mathcal{P}$ and $\mathcal{P}_{tg}$ is zero. Let $S_0(\overline{val})$ be the initial state of $\mathcal{P}$, $S_0^\alpha(\overline{val})$ corresponds to the initial state of $\mathcal{P}_{tg}$ by definition.

Inductive case The induction hypothesis is: For every trace tr of length n, there is a trace tr' such that $Cost_{tr(\overline{val})}(tg_0) = Cost_{tr'(\overline{val},tg_0)}$ and the final states of tr and tr' are S and S^α.

Let us have a trace $tr_1 = tr \xrightarrow{c} S_{new}$ of length $n+1$. By induction hypothesis, we know the last state of tr is S and there is a trace tr' such that its last state is S^α and $Cost_{tr(\overline{val})}(tg_0) = Cost_{tr'(\overline{val},tg_0)}$. We have to prove that for any step $S \xrightarrow{c} S_{new}$ there is a step or sequence of steps that reaches the transformed state S_{new}^α: $S^\alpha \xrightarrow{c_1} S_1 \xrightarrow{c_2} \cdots \xrightarrow{c_m} S_{new}^\alpha$ and the cost $\sum_{i=1}^{m} c_i$ is c if $clk(tg_0) \in S$ or 0 if $clk(t') \in S\ t' \neq tg_0$.

If we perform any step that does not consume resources from S to S_{new}, we can perform the same step in S^α and obtain S_{new}^α (See Figs.2 and 3). None of the original instructions refers or modifies tg so the extended states do not have any effect in the executions of the original instructions. The pending instructions, locks, tasks and objects coincide. From these, the only transformed statement is the method call. The addition of tg as a parameter guarantees that the local state of the new method in S_{new}^α is the extended local state of the new method in S_{new}.

If we perform a step (Cost) in S and obtain S_{new}, we distinguish two cases: If $clk(t) \in S\ t = tg_0$, then we can apply (If-True) followed by (Cost) to S^α and obtain S_{new}^α. the cost of the first step is 0 and the cost of the second step is $eval(c, l + [tg \to tg_0], a)$. The cost in the original program is $eval(c, l, a)$. Because

c cannot contain references to tg, $eval(c, l, a) = eval(c, l + [tg \rightarrow tg_0], a)$. If otherwise ($clk(t) \in S$ such that $t \neq tg_0$), we can apply (If-False) and obtain S^{α}_{new}. The cost is 0 as expected.

B.2 Proof of Theorem 4.2

The theorem 4.2 is valid for *well formed non-blocking* (See def.7) programs. A program is *well formed* as long as it cannot have runtime failures such as calling a method on an object reference that points to **null**. In order to prove theorem 4.2, we require the following auxiliary result:

Lemma 1. *Given a well formed non-blocking program, if canAdv is true in a state S, all the object's locks are free: $\forall\ ob(o, a, lk) \in S,\ lk = \bot$.*

Proof (Lemma 1). If *canAdv* is true in S, no other rule but (Tick) can be applied. Suppose there is a task $tsk(tk, m, o, l, s) \in S$ that has the object's lock. Considering the rules applicable to each statement of a task with the object's lock: (Await1) and (Await2), (Until1) and (Until2), (If-True) and (If-False) and (While-True) and (While-False) have complementary conditions (at least one of them has to be applicable). From the rest of the rules, only (Get), and (Async-Call) have conditions that need to be satisfied.

- If $s = \{y = x!m(\overline{p}; s')\}$ and (Async-Call) cannot be applied, $l(x) =$ **null** and the program is *not well formed*.
- If $s = \{x = y.\textbf{get}; s')\}$ and (Get) cannot be applied, the program is *blocking*

□

Proof (Theorem 4.2). We prove the theorem using induction on the length of the trace in the original program: For a trace of length 0, the initial states of $\mathcal{P}_{tg}$ and $et\mathcal{P}_{tg}$ are defined as S and S^{τ} respectively. $clk(1) \in S$ and the initial and only task in $et\mathcal{P}_{tg}$ is $tsk(tk, o, l + [time \rightarrow 1], \tau(body(main)))$ so the initial state of $et\mathcal{P}_{tg}$ is τ-equivalent to S. The cost of both traces is 0.

The induction hypothesis is: For every trace tr of $\mathcal{P}_{tg}$ of length n with states $S_{1..n}$, there is a trace tr' that contains the τ-equivalent states $S^{\tau}_{1..n}$, S^{τ}_n is the final state and $Cost_{tr(\overline{val})} = Cost_{tr'(\overline{val},1)}$.

Let us have a trace $tr_1 = S_1 \rightarrow \cdots \rightarrow S_n \xrightarrow{c} S_{new}$ of length $n + 1$. The sub-trace $tr = S_0 \rightarrow \cdots \rightarrow S_n$ has length n so we know there is a trace that contains $S^{\tau}_{1..n}$ such that its last state is S^{τ}_n, and $Cost_{tr(\overline{val})} = Cost_{tr'(\overline{val},1)}$. We have to prove that for any step $S_n \xrightarrow{c} S_{new}$ there is a step or sequence of steps that reaches a τ-equivalent state S^{τ}_{new}: $S^{\tau}_n \xrightarrow{c_1} S'_1 \xrightarrow{c_2} S'_2 \xrightarrow{c_3} \cdots \xrightarrow{c_m} S^{\tau}_{new}$, and the cost $\sum_{i=1}^{m} c_i$ is c.

We consider each of the semantic rules that can be applied to S_n:

- For rules (New-Object), (Release), (Async-Call), (Await2), (Return), (Get), (Return), (Assignment), (Cost), (If-True), (If-False), (While-True) and (While-False), applied to a task $tsk(tk, o, l, s)$. We can apply the same rule to the τ-equivalent task $tsk(tk, o, l + [time \rightarrow t], \tau(s)) \in S^{\tau}_n$ (where

$clk(t) \in S_n$) and obtain S'. *eval* will yield the same result in both S_n and S_n^τ and neither the clock $clk(t) \in S_n$ nor the values of *time* in the tasks of S_n^τ change, therefore $S' = S_{new}^\tau$.
If we apply (Cost), the cost is the same in both programs (*eval* yields the same result). Otherwise the cost in 0.
For the rule (Async-Call), we create a new task $tsk(tk_1, o_1, l_1, body(m_1)) \in S_{new}$ and a corresponding task $tsk(tk_1, o_1, l_1 + [time \to t], \tau(body(m_1))) \in S'$ that contains a local variable *time*. The value of *time* is received form the calling task and is equal to the clock $l_1[time \to t](time) = l[time \to t](time) = t$.

– (Activate) applied to a task $tsk(tk, o, l, \{\mathsf{take}; s\}) \in S_n$ with $ob(o, a, \bot) \in S_n$ and with a clock $clk(t) \in S_n$. The corresponding task in S_n^τ is

$$tsk(tk, o, l + [time \to t'], \{\mathsf{take}; \tau(s)\})$$

We apply (Activate) and we have to prove that $t = t'$ so the new state will have the task

$$tsk(tk, o, l + [time \to t], \tau(s))$$

and will be τ-equivalent S_{new}^τ. The take statement can be only generated by a previous application of (Release) or (Async-Call) (Remember that $body(m)$ adds a take before the statements of m).

- If (Release) was applied, Let $S_{release}$ be the state from where (Release) was applied and $tr_{frag} = S_{release} \to \cdots \to S_n$ the sub-trace of tr from that state until S_n. *canAdv* cannot be true in tr_{frag}. If *canAdv* was true, by the lemma 1 we would conclude that all the locks are free. However, if the lock of o is free, (Activate) would be applicable which contradict the conditions of *canAdv*. Consequently, no (Tick) step can be part of tr_{frag} and $clk(t) \in S_{release}$.
  We have an τ-equivalent state $S_{release}^\tau$. In $S_{release}^\tau$ tk has the lock and therefore its local state has $l(time) = t$. Because the value of $l(time)$ in a task cannot decrease, we know that $t' \geq t$. By induction hypothesis $t' \geq t$. Therefore, $t' = t$ and the resulting state is τ-equivalent.
- If (Async-Call) was applied, Let $S_{afterCall}$ be the state after (Async-Call) was applied and $tr_{frag} = S_{afterCall} \to \cdots \to S_n$ the sub-trace of tr from that state until S_n. We can reason as before that no (Tick) step can be part of tr_{frag} and $clk(t) \in S_{afterCall}$.
  In the τ-equivalent state $S_{afterCall}^\tau$ the task tk has just been created from another task tk' that has the lock. The value of $l(time)$ in tk is received from the value in tk' and therefore it must be t. With that and the induction hypothesis, we can conclude that $t' = t$ as before.

– (Await1) applied to a task $tsk(tk, o, l, \{\mathbf{await}\ y?; s\}) \in S_n$ with $ob(o, a, lk) \in S_n$ and with a clock $clk(t) \in S_n$. The corresponding task in S_n^τ is

$$tsk(tk, o, l + [time \to t'], \{\mathbf{await}\ y?; time = max(time, y.time); \tau(s)\})$$

We apply (Await1) followed by (Assignment). If we prove that the maximum value $max(time, y.time)$ is t, then, the new state will have the task

$$tsk(tk, o, l + [time \to t], \tau(s))$$

and will be τ-equivalent S^{τ}_{new}.
We distinguish two cases whether the task has the lock $lk = tk$ or not $lk = \bot$.

- If the task has the lock, the local time t' is equal to the global time $t = t'$ (by induction hypothesis). The value $y.time$ was evaluated with the local time of the finished task in a previous step so $y.time \leq t$ ($y.time$ cannot be modified). Therefore, $max(time, y.time)$ will yield t.
- If the task does not have the lock, there has to be an step (Await2) in the trace tr where the task lost the lock. Let S_{lost} be the state from where (Await2) was executed and $tr_{frag} = S_{lost} \to \cdots \to S_n$ the sub-trace of tr from that state until S_n. In S_{lost}, the waited task $l(y)$ cannot be finished (since (Await2) is applied), so there has to be an intermediate step in tr_{frag} where such task is finished (the (Return) rule is applied). Let S_{return} be the state from where (Return) was executed and $tr_{frag2} = S_{return} \to \cdots \to S_n$ the sub-trace of tr_{frag} from that state until S_n. $canAdv$ cannot be true in tr_{frag2}. If $canAdv$ was true, by the lemma 1 we would conclude that all the locks are free. However, if the lock of o is free, (Await1) would be applicable which contradict the conditions of $canAdv$. Consequently, no (Tick) step can be part of tr_{frag2}.
  We have an τ-equivalent state to S_{return} where (Return) is applied and $y.time$ receives the value t ((Return) is applied to a task that has the lock). Therefore, $max(time, y.time)$ will yield t.

– (Until 1) applied to a task $tsk(tk, o, l, \{\textsf{until}\,(p_l); s\}) \in S_n$ with $ob(o, a, tk) \in S_n$ and with a clock $clk(t) \in S_n$. The corresponding task in S^{τ}_n is

$$tsk(tk, o, l + [time \to t], \{\textsf{release}; time = max(time, p_l); \tau(s)\})$$

The result of p_l is $t' = eval(p_l, l)$. If we can apply (Until 1), then $t' \leq t$ so $max(time, t')$ will yield t (the task has the lock so the variable $time$ has the right time value t). We can apply (Release), (Activate) and (Assignment). The evaluation of p_l in the step (Assignment) yields the same result t' (which is $\leq t$) as $l + [time \to t]$ has not been modified. We obtain

$$tsk(tk, o, l + [time \to t], \tau(s)) \in S'$$

Therefore, the new state is τ-equivalent $S' = S^{\tau}_{new}$.

– (Until 2) applied to a task $tsk(tk, o, l, \{\textsf{until}\,(p_l); s\}) \in S_n$ with $ob(o, a, tk) \in S_n$ and with a clock $clk(t) \in S_n$ yields $tsk(tk, o, l, \{\textsf{untilp}\,(p_l); s\}), ob(o, a, \bot) \in S_{new}$. The corresponding task in S^{τ}_n is

$$tsk(tk, o, l + [time \to t], \{\textsf{release}; time = max(time, p_l); \tau(s)\})$$

We can perform (Release) and obtain a state S'

$$tsk(tk, o, l + [time \to t], \{take; time = max(time, p_l); \tau(s)\}), ob(o, a, \bot) \in S'$$

We have that $\tau(\,\mathsf{untilp}\,(p_l); s) = take; time = max(time, p_l); \tau(s)$ so S' is τ-equivalent to S_{new} ($S' = S^{\tau}_{new}$).

– (Until 3) applied to a task $tsk(tk, m, o, l, \{\,\mathsf{untilp}\,(p_l); s\}) \in S_n$ in an object $ob(o, a, \bot) \in S_n$ where $eval(p_l.l) = t'$. This configuration can only be obtained if there is a previous step (Until 2) for the same task in the trace tr. Let S_{lost} be the state from where (Until 2) was executed and $tr_{frag} = S_{lost} \to \cdots \to S_n$ the sub-trace of tr from that state until S_n. In S_{lost}, $eval(p_l, l) = t' > t$ where $clk(t) \in S_{lost}$ so there has to be an intermediate step in tr_{frag} where $t \geq t'$ becomes true. This can only happen through a (Tick) step (the evaluation of p_l cannot change since the task tk does not have the lock).
Let S_{After_Tick} be the state immediately after applying (Tick) that reaches the awaited time $t = t'$. Let $tr_{frag2} = S_{After\_Tick} \cdots \to S_n$ be the sub-trace of tr_{frag} from S_{After_Tick} until S_n. $canAdv$ cannot be true in tr_{frag2}. If it was true, by corollary 1 all the locks should be free and (Until 1) would be applicable to $tsk(tk, o, l, \{\,\mathsf{untilp}\,(p_l); s\})$ which contradicts the definition of $canAdv$. Consequently, $t' = t$ for $clk(t) \in S_{After\_Tick..n}$.
The corresponding task in S^{τ}_n is

$$tsk(tk, o, l + [time \to t''], \{take; time = max(time, p_l); \tau(s)\})$$

where $t'' \leq t$. We can apply (Activate) and (Assignment) and obtain S' with

$$tsk(tk, o, l + [time \to t'], \tau(s)) \in S'$$

The result of $max(time, p_l)$ is t' which coincides with t. Therefore, $S' = S^{\tau}_{new}$.

– (Await Duration) applied to a task $tsk(tk, o, l, \{\mathsf{await\ duration}(p_l); s\}) \in S_n$ with $clk(t) \in S_n$ produces $tsk(tk, o, l, \{\,\mathsf{until}\,(p_l + t); s\}) \in S_{new}$. The corresponding task in S^{τ}_n is

$$tsk(tk, o, l + [time \to t], \{\mathsf{release}; max(time + p_l, time); s\})$$

which corresponds (because $time$ will evaluate to t) to

$$tsk(tk, o, l + [time \to t], \{\mathsf{release}; max(t + p_l, time); s\})$$

which is the τ-equivalent version of $tsk(tk, o, l, \{\,\mathsf{until}\,(p_l + t); s\}) \in S_{new}$. Therefore, S^{τ}_n is τ-equivalent to S_{new}.

– When (Tick) is applied to S_n with $clk(t)$, we obtain S_{new} with $clk(t + 1)$. Given the lemma 1, no task has the lock. By induction hypothesis, all tasks in S^{τ}_n have $tsk(tk, o, l + [time \to t'], s)$ such that $t' \leq t$ which implies $t' \leq t + 1$. Therefore, S^{τ}_n is τ-equivalent to S_{new}.