# Live heap space bounds for real-time systems

Martin Kero, Paweł Pietrzak, and Johan Nordlander

Department of Computer Science and Electrical Engineering
Luleå University of Technology
{martin.kero, pawel.pietrzak, johan.nordlander}@ltu.se

**Abstract.** Live heap space analyses have so far been concerned with the standard sequential programming model. However, that model is not very well suited for embedded real-time systems, where fragments of code execute concurrently and in orders determined by periodic and sporadic events. Schedulability analysis has shown that the programming model of real-time systems is not fundamentally in conflict with static predictability, but in contrast to accumulative properties like time, live heap space usage exhibits a very state-dependent behavior that renders direct application of schedulability analysis techniques unsuitable.
In this paper we propose an analysis of live heap space upper bounds for real-time systems based on an accurate prediction of task execution orders. The key component of our analysis is the construction of a non-deterministic finite state machine capturing all task executions that are legal under given timing assumptions. By adding heap usage information inferred for each sequential task, our analysis finds an upper bound on the inter-task heap demands as the solution to an integer linear programming problem. Values so obtained are suitable inputs to other analyses depending on the size of a system's persistent state, such as running time prediction for a concurrent tracing garbage collector.

## 1 Introduction

Recent years have seen a respectable development in techniques for analysis of live heap space usage of programs [3, 14, 9, 18]. The common goal of this line of research is to obtain an a priori upper bound on the size of heap memory reachable from various points in a program, expressed as a function of its input data. To this end, a standard sequential programming model has been assumed, where a program reads all its input initially, computes internally without further interaction, and eventually terminates with a deterministic result.

Unfortunately, very few embedded real-time systems – for which static predictability and failure-free operation are of particular concern – fit such a traditional programming model. Instead of terminating with a result, an embedded system typically maintains an ongoing interaction with its environment, executing fragments of code at predefined intervals or in response to sporadic events. Moreover, code fragments are often allowed to execute in parallel or under arbitrary interleaving, which introduces another source of non-determinism in such

systems. It is clear that both these deviations from purely sequential execution adds significant complexity to the problem of predicting heap space usage.

The substantial body of results in *real-time scheduling theory* has however demonstrated that sporadic events and concurrent execution are not fundamentally at odds with static predictability. What is required is just some carefully chosen restrictions on how tasks (i.e., code fragments) may interact with each other, and what time-patterns external events may exhibit [16]. In this paper we present a technique for lifting live heap space analysis to a real-time programming model of a similar vein, sufficiently restricted to make static analysis feasible, but still expressive enough to fit a large class of real-world systems.

However, unlike schedulability analysis – which is only concerned with the number of CPU cycles a task needs to be allocated before its deadline expires – a prediction of heap space usage cannot ignore the order in which deadline-avoiding tasks actually execute at run-time. For an example, consider a task $A$ that allocates heap memory and a task $B$ that frees up any previous allocations. To the combined heap demand of these tasks, it makes a fundamental difference whether an $A$ is always followed by a $B$ or if two $A$ can sometimes occur in a row, even if this distinction might be irrelevant for the purpose of meeting deadlines. For the same reason, heap space analysis cannot ignore the actual interleaving of tasks that are allowed to run concurrently, unless the effect each task has on live memory can be considered atomic.

The main contribution of this paper is a technique for calculating upper bounds on live heap memory of real-time systems, that is safe even in the presence of state- and order-dependent tasks driven by external sporadic events. Our strategy for doing so consists of the following key ideas:

1. We impose a modest restriction on the tasks we consider: every root of live memory must be protected by some locking mechanism, and all the locks a task requires must be held throughout its whole execution (Section 2). This is arguably a stronger restriction than necessary to guarantee atomicity, but it is appealingly simple and "obviously" correct for our purpose. We further elaborate on the realism of our task model at the end of Section 2.
2. We assume a uniform event model where each task is characterized by a minimum and (possibly infinite) maximum distance in time between the events that may trigger it. This allows us to employ techniques from *timed automata* [5] to construct a non-deterministic finite state machine (FSM) for every given task set, which adequately models all possible task execution orderings that are possible according to the given timing assumptions (Section 3).
3. We apply a standard variant of abstract interpretation to each task for inferring *size relations* [11], which capture how each individual task affects an abstract notion of size for every persistent state variable (Section 5). The input to this step is a variant of the rule-based representation (RBR) introduced in [2] for describing sequential imperative code that may involve iterative or recursive computations (Section 4).

4. We combine the results from the FSM construction and the size relation analysis in order to obtain an *integer linear programming problem*, whose solution includes a provably safe upper bound on the total live heap size observable between all possible task executions (Section 6). A set of examples illustrating how the implemented analysis algorithm behaves in practice are also given in Section 7.

Our interest in this paper is to bound the size of the heap-allocated state a system needs to preserve between its event-triggered activations, to serve as input to other analyses that crucially depend on this value, like worst-case execution time estimation for an idle time garbage collector, for example. The related problem of finding a size-bound on the total memory that must be set aside for a system's heap is not directly addressed, but we will return to the question of how our analysis fits this larger picture in Section 9.

## 2 Real-time system model

Here we define the model of execution we will work with in the rest of the paper. Our model connects fairly well to task models used in the real-time scheduling literature [16], while drawing its concrete inspiration from the execution principles underlying the real-time programming language Timber [17].

We consider a real-time system to consist of a finite set $\tau = \{t_1, \ldots, t_m\}$ of *tasks*, and a finite set $\sigma = \{s_1, \ldots, s_n\}$ of *shared state variables*. Each task is supposed to be triggered by a recurring event whose origin we know nothing about, but for which we can make timing assumptions. To this end we assume that each task $t_i \in \tau$ is characterized by a minimum and a maximum inter-arrival time between activation events ($P_i^{min}, P_i^{max} \in \mathbb{N}$). Furthermore, we assume that there is a deadline ($D_i \in \mathbb{N}$) associated with each task, and that every task is scheduled correctly (that is, every task will execute to completion within $D_i$ time units after each triggering event). A task set is *well-formed* if the following is true:

**Definition 1.** *A task set $\tau$ is well-formed iff $\forall t_i \in \tau \;.\; 0 < P_i^{min} \leq P_i^{max}$ and $0 \leq D_i \leq P_i^{min}$.*

In other words, *aperiodic* tasks are excluded from our model (i.e., tasks for which $P_i^{min} = 0$), motivated by the unbounded load such tasks can place on the processor as well as on the heap. For technical reasons we also exclude tasks for which the permissible execution window of one instance is allowed to overlap with the next one (i.e., where $D_i > P_i^{min}$).

Periodic tasks are captured in this model by letting $P_i^{min} = P_i^{max}$, and fully sporadic tasks by $P_i^{max} = \infty$. Note that the model allows a continuum of behaviors between these extremes, even though the typical cases will be found at either end of the scale.

**Shared state.** Each shared state variable $s_j$ is assumed to be protected by some mutual exclusion mechanism, and we furthermore require every task that either

reads or writes to $s_j$ to maintain exclusive access to $s_j$ throughout its whole execution. This way every pair of tasks with any state variables in common will be forced to execute in some sequential order rather than in a potentially interleaved fashion. Tasks which do not share any state variables are allowed to execute under arbitrary interleaving, but the effect such tasks have on the global state is consequently independent of the interleaving pattern, and thus equivalent to their sequential execution in some arbitrary order.

Furthermore, we make our analysis independent of the actual processing power of the chosen execution platform by assuming that tasks may run arbitrarily fast[1]; that is, task execution can be associated with a point in time rather than a time interval. What we achieve under these hypotheses is that we may approximate the concurrent execution of a real-time system by a *set of sequential task orderings*, strictly governed by the underlying inter-arrival time assumptions and deadline requirements, and notably independent of any task execution times and scheduling policies. In Section 3 we will show how to concretely represent this set of task orderings in the form of a non-deterministic finite state machine.

Keeping all accessed state variables locked for the duration of full task executions is of course detrimental to the concurrent schedulability of a system, and thus not a very realistic model of concrete real-time software. However, we argue that for the purpose of the specific analysis of this paper, our simplistic model is an accurate description of a much more general class of concurrent systems that actually do occur in practice. Indeed, the Timber language that we target in our analysis implementation uses a run-time model that closely follows the principles of Baker's Stack Resource Policy [6]: state variables are partitioned into logical units called resources (or objects), each resource uses a common lock for its set of variables, and tasks (or methods) are required to lock and unlock resources in a stack-like fashion according to a total resource order (a resource may only be acquired if it is of less rank than those already held).

The only restriction this paper effectively adds to the SRP model is that we prohibit non-nested sequential resource access: new resources may not be locked once a previously held resource has been released. Under this assumption we are able to describe all relevant state update sequences of a system in terms of its possible task orderings, which is a key to the tractability of our technique and from an analysis point of view equivalent to locking all resources at once. In our experience, this additional restriction is not very burdening in practice; in the Timber language it simply corresponds to limiting the use of synchronous inter-object method calls to at most one per metod. Nevertheless, we do consider lifting the nesting requirement as an important topic for future work, and one approach we have been pondering is to automatically split tasks not conforming to the restriction into smaller parts until they do. This approach does however require that the FSM construction algorithm can be modified to take the implied sequential order of such sub-tasks into account.

**Task bodies.** The sole purpose of a task is to modify the contents of the system state variables $\sigma = \{s_1, \ldots, s_n\}$. For the purpose of this paper, external ports

[1] Or arbitrarily slow, provided that all deadlines are still met.

and other observable state containers such as operating system services also count as state variables. Apart from these global state variables, we require that variables and data structures are *immutable* and thus never change their values once they are assigned. To better capture the freedom from arbitrary side-effects during task runs, we make threading of the system state through each task $t_i$ explicit by representing it as a procedure $t_i(\overline{x}, \overline{y})$ that maps an input state vector $\overline{x}$ to an output state vector $\overline{y}$. The intention is then that the global scheduling mechanism of a system uses the output state vector to destructively update the system state, which we henceforth never need to make explicit. The exact format of each task body is further explained in Section 4.

**Worked example.** Throughout the paper we will work with the following example through the steps of our analysis. Suppose we have two tasks, $a$ and $b$, sharing two lists $x_1$ and $x_2$ in the following manner:

- $a$ extends $x_1$ with one element, leaving $x_2$ as is.
- $b$ sets $x_2 := x_1$ and $x_1 := []$ (empty list), i.e. $x_2$ becomes the list that $x_1$ was, and $x_1$ becomes empty.
- Initially, $x_1$ and $x_2$ are both empty.

For the purpose of the example, let $a$ and $b$ have the following timing characteristics:

| task | $P^{min}$ | $P^{max}$ | $D$ |
|---|---|---|---|
| $a$ | 10 | $\infty$ | 10 |
| $b$ | 10 | 20 | 10 |

Our underlying analysis question is: what is the maximum sum of the sizes of $x_1$ and $x_2$ that ever may occur?

## 3 FSM representation

As a core technical idea of our approach we choose to express the behaviour of a given task set as a *Timed Automaton*, itself constructed as the parallel composition of timed automata representing every individual task. The observable transitions of this automaton are the execution points of the tasks, i.e., the momentary points in time where we consider a task to perform its mutation of the system state. We then apply standard techniques for obtaining a finite *untimed* representation of the timed automaton, representing all possible task execution orders of the system in a compact form.

### 3.1 Real-time systems as Timed Automata

We follow a notation similar to Bengtsson and Yi [7] for representing the legal orders of execution for a real-time task set. A timed automaton is defined by a tuple $\mathcal{A} = \langle L, l_0, A, C, I, E \rangle$, where $L$ is a set of locations, $l_0$ an initial location, $A$ a set of labels (including the silent label $\varepsilon$), $C$ a set of clock variables, $I$ a mapping from locations to clock variable constraints, and $E$ a set of transitions

(characterized by a label, a transition guard, and a set of clock variables to reset as a side-effect).

For a well-formed task set $\tau$, let each task $t_i \in \tau$ be represented by a timed automaton $\mathcal{A}_i = \langle L_i, l_{0i}, A_i, C_i, I_i, E_i \rangle$ defined as follows:

$$
\begin{aligned}
L_i &= \{\texttt{idle}, \texttt{released}\} \\
l_{0i} &= \texttt{idle} \\
A_i &= \{t_i, \varepsilon\} \\
C_i &= \{c_i\} \\
I_i &= \{(\texttt{idle}, c_i \leq P_i^{max}), (\texttt{released}, c_i \leq D_i)\} \\
E_i &= \{(\texttt{idle}, c_i \geq P_i^{min}, \varepsilon, \{c_i\}, \texttt{released}), (\texttt{released}, \mathit{true}, t_i, \emptyset, \texttt{idle})\}
\end{aligned}
$$

Fig. 1 shows the the definition above in a graphical notation.

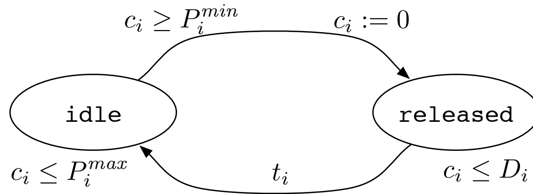


**Fig. 1.** Timed automaton $\mathcal{A}_i$ capturing the execution points of task $t_i$

The transitions of $\mathcal{A}_i$ capture the execution points of $t_i$: either the silent arrival of a triggering event for $t_i$, or the observable execution of $t_i$. Location `idle` denotes the state when the task is neither executing nor pending. Clock variable $c_i$ increases in synchrony with real time and is reset whenever an event transition is taken, thus the invariant $(c_i \leq P_i^{max})$ ensures that the time between two triggering events for $t_i$ never exceeds $P_i^{max}$. Moreover, the guard $(c_i \geq P_i^{min})$ on the event transition forces the inter-arrival time to be at least $P_i^{min}$. The invariant in the `released` location guarantees that execution must take place within the deadline $(c_i \leq D_i)$. Because of the well-formedness assumption we know that $D_i \leq P_i^{min}$, which means that we can capture the timing constraint of the deadline with the same clock used for inter-arrival times (i.e., whenever the execution transition is taken, we know that $c_i \leq D_i \leq P_i^{min}$).

The timed automaton of the whole task set $\tau$ is then constructed by parallel composition $\mathcal{A}_\tau = \|_{t_i \in \tau} \mathcal{A}_i$. The resulting automaton is entirely straightforward, with locations and invariants being conjunctions of the individual automata counterparts (see [4] for further details on parallel automata composition).

### 3.2 Untimed automata

Our goal is to construct a compact FSM that accurately captures all legal orders of task executions. In reachability analysis for timed automata, such faithful constructions of FSMs are usually referred to as *untimed* automata.

The operational semantics of timed automata is described as a transition system of which states are pairs $\langle l, u \rangle$ where $l$ is a location in the original timed automaton and $u$ is a *clock valuation* mapping clock variables to real values [5].

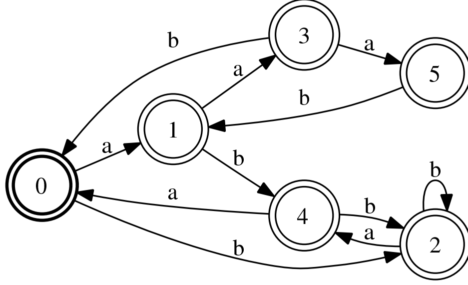


**Fig. 2.** Minimal FSM representation of the execution orders of the worked example.

State space and transitions are defined based on the locations and transitions in the timed automaton, combined with the condition that $u$ satisfies the implied clock constraints from invariants, guards, and resets. The state space of this transition system can efficiently (and finitely) be quotiented by configurations $\langle l, D_\varphi \rangle$, called *zones*, where $D_\varphi$ is a convex set of clock valuations (called *clock zone*) defined by the clock constraint $\varphi$. The transitions of the *zone graph* is defined as follows.

$$\begin{array}{ll} \langle l, D_\varphi \rangle \longrightarrow \langle l, [D_\varphi{}^{\Uparrow} \cap D_{I(l)}] \rangle & \\ \langle l, D_\varphi \rangle \xrightarrow{\alpha} \langle l', [\delta(D_\varphi \cap D_\psi) \cap D_{I(l')}] \rangle & \quad \text{if } (l, \psi, \alpha, \delta, l') \in E \end{array}$$

The operations on clock zones in the definition are: *delay* ($D_\varphi{}^{\Uparrow}$) which computes the strongest post condition of $\varphi$ (i.e., the clock zone containing all valuations after an arbitrary delay); *reset* ($\delta(D_\varphi)$) which computes the new clock zone to capture the resets of $\delta$; and *normalization* ($[D_\varphi]$) which widens the clock zone based on the maximum constants used in clock constraints of the timed automaton.

The zone graph accurately accepts all legal sequences of untimed labels as of the original timed automaton, while faithfully keeping track of both locations and clock valuations. However, our analysis only requires a faithful representation of the untimed language accepted by the zone graph (i.e., reachability of particular locations and clock valuations are irrelevant), we may apply standard FSM transformation techniques (such as determinization and minimization) to possibly improve the automaton. The *raw* zone graph of our worked example contains 31 zones and 59 transitions of which 34 are $\varepsilon$-transitions (delays and events). After determinization and minimization we get a FSM of 6 states and 11 transitions. The minimized FSM is shown in Fig. 2.

However, as we will see in Section 7, it is not necessary an improvement of performance of our analysis to make these quite costly transformations.

## 4 Rule-based representation

Following [1, 2] we express the actual code a task executes in a *rule-based representation*, or RBR for short. Our RBR is essentially a slight simplification of the format used in [2]. We assume four distinct name-spaces: ordinary variables ranged over by $x$ and $y$, procedure names ranged over by $p$ and $q$, as well as

constructor names and field names (ranged over by $c$ and $f$, respectively). The syntax of our rule-based language is given in Fig. 3.

Variables can take atomic values (we limit ourselves to integers), or compound values (lists, trees, etc.), in which case the variable is a reference to a *constructed* and possibly heap-allocated object. A program consists of a set of procedures, of which some are designated as *tasks*. A procedure with a head $p(\overline{x}, \overline{y})$ has input $(\overline{x})$ and output $(\overline{y})$ parameters, and is defined by one or more *rules*. Each rule is guarded by a boolean applicability condition $g$, which may be either the unconditional constant *true*, a simple arithmetic comparison, or the special form $\mathsf{type}(x, c)$, which tests whether $x$ is a reference to a constructed $c$ object. The guard is followed by the procedure *body*, which might contain a variable assignment, an object creation instruction (with a vector of field initializations within braces), an assignment with field selection, or a (possibly recursive) procedure call. Since we do not support mutation of any other data than the system state vector, we make it an implicit condition that all variables in a procedure body are assigned only once.

$$
\begin{aligned}
P &::= R_1, \ldots, R_n \\
R &::= p(\overline{x}, \overline{y}) \leftarrow g, b_1, \ldots, b_n \\
g &::= \mathit{true} \mid e \; \mathit{op} \; e \mid \mathsf{type}(x, c) \\
b &::= x := e \mid x := \textbf{new} \; c \, \{\overline{f} := \overline{e}\} \mid x := y.f \mid q(\overline{x}, \overline{y}) \\
e &::= x \mid n \mid e - e \mid e + e \mid e * e \mid e/e \\
\mathit{op} &::= \; > \mid \; < \mid \; \geq \mid \; \leq \mid \; = \mid \; \neq
\end{aligned}
$$

**Fig. 3.** Syntax of rule-based representation

With the rule-based representation, the code of the tasks of our worked example looks as shown in Fig. 4 (assuming *value* is some suitable integer element). We use constructors *cons* and *nil* for building lists, where **new** *nil* {} denotes a "null pointer" (i.e., a zero-arity constructed value requiring no additional heap space).

$$
\begin{aligned}
&a(\langle x_1, x_2\rangle, \langle x_1', x_2'\rangle) \leftarrow x_1' := \textbf{new} \; \mathit{cons} \, \{\mathit{head} := \mathit{value}, \mathit{tail} := x_1\}, x_2' := x_2 \\
&b(\langle x_1, x_2\rangle, \langle x_1', x_2'\rangle) \leftarrow x_2' := x_1, x_1' := \textbf{new} \; \mathit{nil} \, \{\}
\end{aligned}
$$

**Fig. 4.** The worked example in the rule-based representation

In addition to rules defining tasks, we always include a predefined procedure *init* with just a single vector of output parameters carrying the initial values for a system's state variables. For the worked example we have:

$$
\mathit{init}(\langle x_1, x_2\rangle) \leftarrow x_1 := \textbf{new} \; \mathit{nil} \, \{\}, x_2 := \textbf{new} \; \mathit{nil} \, \{\}
$$

From [2] we also adopt an operational semantics for our rule-based programs, as depicted in Fig. 5. A value $v$ is either an integer constant or a tagged heap reference $r^c$, where $c$ is a constructor name. A heap $h$ maps references to objects $o$, which in turn are mappings from field names to values. Execution steps are described as transitions $S \rightsquigarrow S'$, where $S$ is a configuration $A : h$ containing

a stack $A$ of activation records (of the form $\langle p, bs, \rho\rangle$, where $p$ is a procedure name, $bs$ a sequence of instructions, and $\rho$ an environment mapping variables to values), and a heap. We write $\rho(x)$ for the value referred to by $x$ in $\rho$, and $\rho[x \mapsto v]$ for the mapping identical to $\rho$ except that $x$ maps to $v$. Both notations extend to vectors of variables and values, and also apply to mappings $o$ and $h$ in a similar manner.

Rule (1) deals with evaluating expressions and storing the resulting value in the environment. We assume that function $\mathsf{eval}(e, \rho)$ evaluates $exp$ in the context of $\rho$. Rule (2) shows extension of the heap with a new tagged object reference $r^c$, mapped to an object associating each field with its evaluated value. Field access is shown in rule (3). Rules (4) and (5) illustrate calling and returning from a procedure, respectively. The notation $p[\overline{y}, \overline{y}']$ stands for a saved association between the formal and actual output parameters of $p$.

$$\frac{b \equiv x := e \quad v = \mathsf{eval}(e, \rho)}{\langle p, (b, bs), \rho\rangle \cdot A : h \rightsquigarrow \langle p, bs, \rho[x \mapsto v]\rangle \cdot A : h} \tag{1}$$

$$\frac{b \equiv x := \textbf{new}\ c\,\{\overline{f} := \overline{e}\} \quad \overline{v} = \mathsf{eval}(\overline{e}, \rho) \quad r^c \notin dom(h)}{\langle p, (b, bs), \rho\rangle \cdot A : h \rightsquigarrow \langle p, bs, \rho[x \mapsto r^c]\rangle \cdot A : h[r^c \mapsto [\overline{f} \mapsto \overline{v}]\,]} \tag{2}$$

$$\frac{b \equiv x := y.f \quad h(\rho(x)) = o}{\langle p, (b, bs), \rho\rangle \cdot A : h \rightsquigarrow \langle p, bs, \rho[x \mapsto o(f)]\rangle \cdot A : h} \tag{3}$$

$$\frac{b \equiv q(\overline{x}, \overline{y}) \quad q(\overline{x}', \overline{y}') \leftarrow g, bs' \text{ is a rule} \quad \rho'(\overline{x}') = \rho(\overline{x}) \quad \mathsf{eval}(g, \rho') = true}{\langle p, (b, bs), \rho\rangle \cdot A : h \rightsquigarrow \langle q, bs', \rho'\rangle \cdot \langle p[\overline{y}, \overline{y}'], bs, \rho\rangle \cdot A : h} \tag{4}$$

$$\frac{}{\langle q, \epsilon, \rho\rangle \cdot \langle p[\overline{y}, \overline{y}'], bs), \rho'\rangle \cdot A : h \rightsquigarrow \langle p, bs, \rho'[\overline{y} \mapsto \rho(\overline{y}')]\rangle \cdot A : h} \tag{5}$$

**Fig. 5.** Operational semantics of rule-based programs

Note that since we model state variables as explicit input and output parameters, we can avoid mutation of the heap altogether in the formal semantics.

Executions can be seen as traces $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_m$. Let $\overset{*}{\rightsquigarrow}$ denote a transitive closure of $\rightsquigarrow$. Complete execution of a single task $t_i$ corresponds to the trace (called *complete trace*) $\langle \bot, t_i(\sigma, \sigma), \rho\rangle : h \overset{*}{\rightsquigarrow} \langle \bot, \epsilon, \rho'\rangle : h'$, where $\bot$ stands for the "scheduler" procedure, $\sigma$ contains the names of the global state variables, and $\rho$ and $\rho'$ hold the state variable values before and after executing $t_i$, respectively.

## 5 Inferring size relations

The notion of size of a heap allocated object can vary depending on what exact purpose our analysis will serve. Let $size_{\mathcal{X}}(o)$ denote *size* of a heap-allocated object $o$, where the size is determined by the cost model $\mathcal{X}$ and may denote $\mathcal{M}$ (memory size occupied by $o$), $\mathcal{R}$ (number of reference fields in $o$), and $\mathcal{O}$ (number of objects in $o$ – i.e., 1).

The above notation also extends over sequences of objects: $size_{\mathcal{X}}(\overline{o}) = [size_{\mathcal{X}}(o_1), \ldots, size_{\mathcal{X}}(o_n)]$. Inferring size relations, similarly to e.g. [8, 1, 2] is

performed in two steps. The first one is *abstract compilation* of rules into linear constraints capturing relations between sizes of program variables. In the second step the fixpoint of the linear constraints system is computed in a bottom-up fashion. We apply the approach and implementation of [8], which originally was designed to compute size relations in logic programs. Since we represent the tasks as rules rather than logic programs, we do compile our rules to constraint logic programs (CLP), but we use a different abstract compilation scheme, as described in the following section.

**Abstract compilation of rules.** In the abstractly compiled version of a program we keep the original variable names, possibly with scripts or overlines, and use boldface to denote their sizes, with respect to a given cost model $\mathcal{X}$. For example, $\mathbf{x}$ denotes size of $x$. We shall extend the notation to expressions, writing $\mathbf{e}$ for a size of an expression $e$ in which every variable $x$ has been replaced by $\mathbf{x}$. A size of an integer number is its value [11]. Size of a compound structure $c\,\{...\}$ is a sum of sizes of its elements, plus a size $k_c^{\mathcal{X}}$ of a single node, suitable for a cost model $\mathcal{X}$. Abstract compilation proceeds over rules in the program as depicted in Fig. 6.

$$
\begin{aligned}
\mathsf{Abs}_P[\![R_1,\ldots,R_n]\!] &= \mathsf{Abs}_R[\![R_1]\!],\ldots,\mathsf{Abs}_R[\![R_n]\!] \\
\mathsf{Abs}_R[\![p(\overline{x},\overline{y}) \leftarrow g, b_1,\ldots,b_n]\!] &= p(\overline{\mathbf{x}},\overline{\mathbf{y}}) \leftarrow \overline{\mathbf{x}} \geq 0, \overline{\mathbf{y}} \geq 0, \mathsf{Abs}_g[\![g]\!], \mathsf{Abs}_b[\![b_1]\!],\ldots,\mathsf{Abs}_b[\![b_n]\!] \\
\\
\mathsf{Abs}_g[\![true]\!] &= true \\
\mathsf{Abs}_g[\![e_1\ op\ e_2]\!] &= \text{if } e_1 \text{ and } e_2 \text{ are linear then } \mathbf{e}_1\ op\ \mathbf{e}_2 \text{ else } true \\
\mathsf{Abs}_g[\![\mathsf{type}(x,c)]\!] &= true \\
\\
\mathsf{Abs}_b[\![x := e]\!] &= \text{if } e \text{ is linear then } \mathbf{x} = \mathbf{e} \text{ else } true \\
\mathsf{Abs}_b[\![x := \textbf{new}\ c\,\{\overline{f_i := e_i}\}]\!] &= \mathbf{x} \leq k_c^{\mathcal{X}} + \textstyle\sum_i \mathsf{norm}(c, f_i, e_i) \\
\mathsf{Abs}_b[\![x := y.f]\!] &= \mathbf{x} < \mathbf{y} \\
\mathsf{Abs}_b[\![q(\overline{x},\overline{y})]\!] &= q(\overline{\mathbf{x}},\overline{\mathbf{y}}) \\
\\
\mathsf{norm}(c,f,e) &= \text{if the type of field } f \text{ of a } c \text{ is integer then } 0 \text{ else } \mathbf{e}
\end{aligned}
$$

**Fig. 6.** Abstract compilation to CLP

Note that for compiling an object creation instruction ($x :=$ **new** $c\,\{...\}$) does not result in an equality, but rather in an inequality. This is the effect of possible sharing between fields of $c\,\{...\}$ which we do not try to detect.

Given a rule-based program $P$, its compiled version $\mathsf{Abs}_P[\![P]\!]$ is a CLP program over real numbers (CLP($\mathbb{R}$)). We refer, for instance, to [13] for further reading on CLP. Let us assume the *model-theoretic (or algebraic)* semantics[2] of CLP($\mathbb{R}$), where semantics of programs is given by means of *models* over $\mathbb{R}$

[2] This is an arbitrary choice made for an illustrative purpose only. All kinds of semantics of CLP coincide is some well-defined sense, so choosing any other semantics would be equally valid.

($\mathbb{R}$-models), and standard interpretation of arithmetic functions (see e.g. [12] for details). The following lemma shows the soundness of the abstract compilation. It is shown that relation between sizes of input and output parameters of a given procedure is correctly captured by the resulting CLP program.

**Lemma 1.** *Given a program $P$ and procedure $p$, assume the trace $\langle q, (p(\overline{x}, \overline{y}), bs), \rho\rangle : h \overset{*}{\rightsquigarrow} \langle q, bs, \rho'\rangle : h'$. The atomic formula $p(k, l)$ where $k = size_{\mathcal{X}}(h(\rho(\overline{x})))$ and $l = size_{\mathcal{X}}(h(\rho'(\overline{y})))$ belongs to the least $\mathbb{R}$- model of $\mathsf{Abs}_P[\![P]\!]$.*

*PROOF: By induction over depth of recursion in $P$.*

Assume that $k^{\mathcal{M}}_{cons} = 3$ and $k^{\mathcal{M}}_{null} = 0$. The abstractly compiled worked example is shown in Fig. 7

$$\begin{aligned}
init(\langle \mathbf{x}_1, \mathbf{x}_2\rangle) &\leftarrow \mathbf{x}_1 \geq 0, \mathbf{x}_2 \geq 0, \mathbf{x}_1 \leq 0, \mathbf{x}_2 \leq 0\\
a(\langle \mathbf{x}_1, \mathbf{x}_2\rangle, \langle \mathbf{x}'_1, \mathbf{x}'_2\rangle) &\leftarrow \mathbf{x}_1 \geq 0, \mathbf{x}_2 \geq 0, \mathbf{x}'_1 \geq 0, \mathbf{x}'_2 \geq 0, \mathbf{x}'_1 \leq \mathbf{x}_1 + 3, \mathbf{x}'_2 = \mathbf{x}_2\\
b(\langle \mathbf{x}_1, \mathbf{x}_2\rangle, \langle \mathbf{x}'_1, \mathbf{x}'_2\rangle) &\leftarrow \mathbf{x}_1 \geq 0, \mathbf{x}_2 \geq 0, \mathbf{x}'_1 \geq 0, \mathbf{x}'_2 \geq 0, \mathbf{x}'_2 = \mathbf{x}_1, \mathbf{x}'_1 \leq 0
\end{aligned}$$

**Fig. 7.** The worked example after abstract compilation

In general, right hand sides of abstractly compiled rules might contain recursive calls. In this case a bottom-up fixpoint algorithm is applied to infer, for each procedure $p$, a set linear constraints $\phi_p$ (or $\phi_p[\overline{\mathbf{x}}, \overline{\mathbf{y}}]$ if we want to make the involved variables explicit). See [8] for the details of the fixpoint iteration algorithm. Theorem 1 states soundness of size relation inference.

**Theorem 1.** *Given a trace $\langle \bot, t_i(\sigma, \sigma), \rho\rangle : h \overset{*}{\rightsquigarrow} \langle \bot, \epsilon, \rho'\rangle : h'$, the vector pair $size_{\mathcal{X}}(h(\rho(\sigma))), size_{\mathcal{X}}(h'(\rho'(\sigma)))$ satisfies $\phi_{t_i}$; that is, the formula $\phi_{t_i}[size_{\mathcal{X}}(h(\rho(\sigma))), size_{\mathcal{X}}(h'(\rho'(\sigma)))]$ is true.*

*PROOF: By Lemma 1 and the soundness of size relation analysis of [8].*

**Example.** Let us illustrate the behaviour of the size relation analyzer by means of the following list concatenation procedure:

$$\begin{aligned}
app(\langle x, y\rangle, \langle z\rangle) &\leftarrow x = null(), z := y\\
app(\langle x, y\rangle, \langle z\rangle) &\leftarrow x \neq null(),\ x' := x.tail,\ app(\langle x', y\rangle, z'),\\
&\qquad z := \textbf{new}\ cons(x.head, z')
\end{aligned}$$

Abstract compilation of the above two rules, with respect to the cost model $\mathcal{O}$, results in

$$\begin{aligned}
app(\langle \mathbf{x}, \mathbf{y}\rangle, \langle \mathbf{z}\rangle) &\leftarrow \mathbf{x} \geq 0, \mathbf{y} \geq 0, \mathbf{z} \geq 0, \mathbf{y} = \mathbf{z}\\
app(\langle \mathbf{x}, \mathbf{y}\rangle, \langle \mathbf{z}\rangle) &\leftarrow \mathbf{x} \geq 0, \mathbf{y} \geq 0, \mathbf{z} \geq 0, \mathbf{x}' \geq 0, \mathbf{z}' \geq 0, \mathbf{x}' \leq \mathbf{x} - 1,\\
&\qquad app(\langle \mathbf{x}', \mathbf{y}\rangle, \mathbf{z}'), \mathbf{z} \leq \mathbf{z}' + 1
\end{aligned}$$

Observe that $z := \textsf{new}\ cons(x.head, z')$ has been compiled to $\mathbf{z} \leq \mathbf{z}' + 1$ rather than $\mathbf{z} = \mathbf{z}' + 1$, due to possible sharing. Computing bottom-up fixpoint over convex polyhedra domain, as described in [8], gives the final size relations:

$$app(\langle \mathbf{x}, \mathbf{y} \rangle, \langle \mathbf{z} \rangle) \leftarrow \mathbf{x} \geq 0, \mathbf{y} \geq 0, \mathbf{z} \geq 0, \mathbf{z} \leq \mathbf{x} + \mathbf{y}$$

# 6 Upper bounds

The crucial observation is that the value we are looking for is the upper bound of live memory size occupied by state variables after any possible completion of any task executed in the concurrent environment, that is in every possible schedule. The size value is not accumulated over recursive calls that might take place while executing the tasks. Therefore, for our purpose we do not need cost relations in the form of [1, 2], but rather than that we work directly with the size relations introduced in the previous section. Based on the FSM representation of task execution orders and size relations for each task, we set up a system of linear constraints which is essentially an ILP (integer linear programming) problem that can be solved by any standard solver. The ILP problem, whose construction is shown below, captures the upper bounds of live memory usage.

Assume there are $n$ state (shared) variables $s_1, \ldots, s_n$. In previous steps, for every task $m(\overline{x}, \overline{y})$ we infer size relations $\phi_m$ which in the matrix form can be written as

$$\mathbf{Y} \leq \mathbf{A}_m \mathbf{X} + \mathbf{C}_m, \quad \mathbf{X} \geq \mathbf{0} \tag{6}$$

where $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$, $\mathbf{Y} = [\mathbf{y}_1, \ldots, \mathbf{y}_n]$, $n$ is a number of states variables. For an initialization method *init* (which has no input parameters) the constraints take form:

$$\mathbf{X}_0 \leq \mathbf{C}_{init}, \quad \mathbf{X}_0 \geq \mathbf{0} \tag{7}$$

Thus the vector $\mathbf{C}_{init}$ describes sizes of initial values of the state variables. The size relation matrices with respect to cost model $\mathcal{M}$ for the worked example look like the following:

$$\mathbf{A}_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{C}_a = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \quad \mathbf{A}_b = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \quad \mathbf{C}_b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

In order to find an upper bound of $\mathbf{X}$'s, for every state $i$ in the FSM we assign a vector of sizes of the state variables, written $\hat{\mathbf{X}}_i$. For a transition $i \xrightarrow{m} j$ we set up a set of constraints

$$\hat{\mathbf{X}}_j \geq \mathbf{A}_m \hat{\mathbf{X}}_i + \mathbf{C}_m \tag{8}$$

and for the initialization

$$\hat{\mathbf{X}}_0 \geq \mathbf{C}_{init} \tag{9}$$

For the $\varepsilon$-transitions we have $\mathbf{A}_\varepsilon = \mathbf{I}_n$ (the $n \times n$ unit matrix) and $\mathbf{C}_\varepsilon = \mathbf{0}$.

We require the size relation matrices ($\mathbf{A}$'s) to only contain non-negative coefficients. If, for some task, a size relation matrix with negative coefficients is inferred (this might occur if, for instance, the task definition is incomplete), we

simply replace those coefficients by 0's. For our purpose, which is finding upper bounds, increasing coefficients in $\mathbf{A}$ is a relaxation and always a safe step to do. The reason for this requirement is that $\hat{\mathbf{X}}_i$ and $\hat{\mathbf{X}}_j$ represent upper bounds, which means that the inferred constraints for $\hat{\mathbf{X}}_j$ must be safe for any sizes between $\mathbf{0}$ and $\hat{\mathbf{X}}_i$.

Let $i_0 \overset{m_0}{\rightarrow} i_1 \overset{m_1}{\rightarrow} i_2 \overset{m_2}{\rightarrow} \cdots$ be a run of the state machine. With every step $k$ we assign variables $\mathbf{X}_k$ denoting sizes of state variables in $k$, in according to (6) and (7).

**Lemma 2.** *Consider a run of the state machine and its $k$-th step. Let $i_k$ denote the state in step $k$. For any solution of (8) + (9), any $k$ we have $\hat{\mathbf{X}}_{i_k} \geq \mathbf{X}_k$.*

*PROOF: Inductive over $k$.*

**Base case:** *Trivially holds by combining (7) and (9).*
**Inductive step:** *By inductive assumption we know that $\hat{\mathbf{X}}_{i_{k-1}} \geq \mathbf{X}_{k-1}$, and by the fact that $\mathbf{A}_{m_{k-1}}$ contains only non-negative values we conclude that $\mathbf{A}_{m_{k-1}} \hat{\mathbf{X}}_{i_{k-1}} + \mathbf{C}_{m_{k-1}} \geq \mathbf{A}_{m_{k-1}} \mathbf{X}_{k-1} + \mathbf{C}_{m_{k-1}}$ By combining the above with (6) and (8) we can observe that $\hat{\mathbf{X}}_{i_k} \geq \mathbf{X}_k$, which concludes the proof.*

Lemma 2 suggests the way to compute upper bounds of state variable sizes. In addition to (9) and (8) we add $\hat{X} \geq \mathbf{H} \cdot \hat{\mathbf{X}}_l$ for every state $l$; where $\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_n]$ and $\mathbf{h}_i = 1$ if $s_i$ is heap allocated, $\mathbf{h}_i = 0$ otherwise. Let the cost function $c = \hat{X}$ and $c^*$ denote its minimum value. The following theorem states soundness of the analysis.

**Theorem 2.** *Let $T$ be a set of all complete traces, over all feasible (possibly infinite) schedules. The following holds:*

$$max\{\sum size_{\mathcal{X}}(h'(\rho'(y))) \mid \langle \bot, p(\overline{x}, \overline{y}), \rho\rangle : h \overset{*}{\rightsquigarrow} \langle \bot, \epsilon, \rho'\rangle : h' \in T\} \leq c^*$$

*PROOF: Follows directly from Theorem 1 and construction of (8) and (9), and Lemma 2.*

The constraints (wrt cost model $\mathcal{M}$) for our worked example are shown below. The minimum solution to its corresponding cost function is $c^* = 30$.

| | | | |
|---|---|---|---|
| $\mathbf{x1}_1 \geq \mathbf{x1}_0 + 3$ | $\mathbf{x2}_1 \geq \mathbf{x2}_0$ | $\mathbf{x2}_2 \geq \mathbf{x1}_0$ | $\mathbf{x1}_3 \geq \mathbf{x1}_1 + 3$ |
| $\mathbf{x2}_3 \geq \mathbf{x2}_1$ | $\mathbf{x2}_4 \geq \mathbf{x1}_1$ | $\mathbf{x1}_4 \geq \mathbf{x1}_2 + 3$ | $\mathbf{x2}_4 \geq \mathbf{x2}_2$ |
| $\mathbf{x2}_2 \geq \mathbf{x1}_2$ | $\mathbf{x1}_5 \geq \mathbf{x1}_3 + 3$ | $\mathbf{x2}_5 \geq \mathbf{x2}_3$ | $\mathbf{x2}_0 \geq \mathbf{x1}_3$ |
| $\mathbf{x1}_0 \geq \mathbf{x1}_4 + 3$ | $\mathbf{x2}_0 \geq \mathbf{x2}_4$ | $\mathbf{x2}_2 \geq \mathbf{x1}_4$ | $\mathbf{x2}_1 \geq \mathbf{x1}_5$ |
| | | | |
| $\mathbf{x1}_0 \geq 0$ | $\mathbf{x1}_1 \geq 0$ | $\mathbf{x1}_2 \geq 0$ | $\mathbf{x1}_3 \geq 0$ |
| $\mathbf{x1}_4 \geq 0$ | $\mathbf{x1}_5 \geq 0$ | $\mathbf{x2}_0 \geq 0$ | $\mathbf{x2}_1 \geq 0$ |
| $\mathbf{x2}_2 \geq 0$ | $\mathbf{x2}_3 \geq 0$ | $\mathbf{x2}_4 \geq 0$ | $\mathbf{x2}_5 \geq 0$ |
| $\hat{X} \geq \mathbf{x1}_0 + \mathbf{x2}_0$ | $\hat{X} \geq \mathbf{x1}_1 + \mathbf{x2}_1$ | $\hat{X} \geq \mathbf{x1}_2 + \mathbf{x2}_2$ | |
| $\hat{X} \geq \mathbf{x1}_3 + \mathbf{x2}_3$ | $\hat{X} \geq \mathbf{x1}_4 + \mathbf{x2}_4$ | $\hat{X} \geq \mathbf{x1}_5 + \mathbf{x2}_5$ | |

## 7 Examples

Our analysis relies on constructing an integer linear programming problem, whose solution includes a provably safe upper-bound on the live heap size observable between all possible task executions. Solving such problems can be done by standard solvers. However, the complexity of solving such problems depends on both the number of unknowns and the number of constraints. In our case, the number of unknowns is determined by the number of states in the FSM. Similarly, the number of constraints is dependent on the number of transitions in the FSM. Both these multiplied by the number of shared state variables.

| Task set | $P^{min}$ | $P^{max}$ | $D$ | #states zone graph | #states FSM |
|---|---|---|---|---|---|
| $\tau_1$ | {17,19,23} | $\{\infty, \infty, \infty\}$ | {17,19,23} | 1255 | 1 |
| $\tau_2$ | {10,300} | {20,350} | {10,300} | 200 | 699 |
| $\tau_3$ | {10,20,30,40,50} | {10,20,30,40,50} | {10,20,30,40,50} | 10368 | 3393 |
| $\tau_4$ | {17,23,29} | {17,23,29} | {17,23,29} | 12968 | 6343 |

**Fig. 8.** Zone graph and minimal FSM sizes of four different example task sets.

It is well-known that the number of zones is exponential to the number of clocks present in the timed automaton [10]. I.e., in our case, we have an exponential growth of zones w.r.t. number of tasks. In Figure 8, the zone graph and minimal FSM sizes for four different example task sets are shown. Observe that, for $\tau_2$, the number of states is less in the zone graph than in the determinized and minimized FSM. However, as $\tau_1$ shows, the minimized FSM can be as small as 1 state (the order between fully sporadic tasks is in fact completely arbitrary), even though the original zone graph contains many more states. Appendix A contains an extended example of our analysis. The required times by our prototype implementation for constructing the zone graphs of the task sets in Figure 8 and Appendix A are neglectable ($< 1$ s). Solving the ILP problem of the example in Appendix A took about 25 seconds, using `lp_solve version 5.5.2.0`.[3]

## 8 Related work

To the best of our knowledge, there is no existing work on predicting global live heap space for real-time systems similar to those we describe in Section 2. Nonetheless, a substantial body of work has been presented for analyzing live heap space bounds for standard sequential programs. In this section we briefly describe some of the more recent contributions in this line of research.

As already mentioned, for each task we borrow from [2, 3], the rule-based representation of programs along with semantics, which we could however simplify due to special treatment of state variables and lack of mutation. We also adopt from their work the step of inferring size relations. Jost et al. [14] presents a

[3] Platform: 3.06GHz Intel Core 2 Duo, 4 GB RAM, Mac OS X 10.6.4

generic type-based resource analysis for inferring linear bounds on resource consumption for higher-order polymorphic programs. The corresponding type inference is based on a standard linear programming solver. Chin et al. [9] presents a memory resource analysis for low-level assembly programs. They infer both net usage and a high watermark bound for each computation unit based on explicit allocation and deallocation of heap space. Unnikrishnan et al. [18] presents a live heap space analysis based on program transformation and symbolic evaluation. The transformed program mimics the memory behavior and essentially keeps the same computational complexity as of the original program.

## 9 Conclusion and further work

We have proposed a technique for computing upper bounds on live heap memory of real-time systems, that is safe even in the presence of state- and order-dependent tasks driven by external sporadic events.

Our key contribution is based on the derivation of an accurate prediction of task execution orders according to timing assumptions of each task (inter-arrival times and deadlines). This is done by representing the task set as a timed automaton and apply standard techniques used in reachability analysis to construct an FSM representation of task execution orders. We infer linear input/output size relations for each task on the persistent state of the system, which is then combined with the execution order FSM to obtain an integer linear programming problem, whose solution includes a provably safe upper bound on the total live heap size observable between all possible task executions.

**Heap space usage and schedulability.** In real-time systems where tasks share heap data (as we describe in Section 2) it is in general impossible to manage heap memory manually. If such systems are to be memory managed by a concurrent garbage collector, the key question is how it affects schedulability. In fact, the problem is twofold; (1) will all tasks meet their deadlines, and (2) how much heap memory will be needed? These two interests are obviously in conflict since running the garbage collector will reduce the memory needs while it may cause tasks to miss their deadlines. On the other hand, avoiding to run the collector might keep tasks meeting deadlines but at the same time cause the system to exhaust memory resources.

A tracing garbage collector recycles the dead (non-reachable) part of the heap and the running time of such collectors is directly dependent on the amount of live (reachable) memory. Thus, finding bounds on the global live heap space of such systems are crucial for both determining schedulability of the task set as well as predicting the total heap space usage.

In [15], Kero and Aittamaa presents a schedulability analysis, called *garbage collection demand analysis*, for a concurrent copying garbage collector in a reactive real-time system. Their garbage collector is restricted to run only during idle time, which enables them to rely on regular schedulability analysis of the task set to ensure (1). The analysis determines an upper bound on the start to

finish time of the garbage collector as well as the amount of memory consumed during that time.

**Further work.** One key observation is that the execution order FSM accepts traces of task executions that are *legal* according to the timing assumptions of each task. In our case, we have left those timing assumptions as open as possible, containing only inter-arrival times and deadlines. Generally, the schedulability requirement leaves the choice of order in which released tasks are executed open as long as all individual deadlines are met. In reality, schedulability is typically reached by a myopic scheduling policy (e.g., EDF, RM, etc.), which has a fully deterministic outcome. Thus, from any zone in the zone graph, if assuming a particular scheduling policy, one can reduce the number of labelled transitions to a maximum of one. Apart from tighter bounds, preliminary experimental results show significant improvements in FSM sizes (down to 25 % of the original size). Along the same line, the zone graph accepts traces where the release of a task and its execution point occurs at the very same instant. Adding a safe lower bound on execution time for each task will reduce the time windows in which task execution points may occur, ultimately reducing the number of possible execution orders. Although standard solvers of ILP problems are quite efficient nowadays, the complexity of finding the optimal solution is still exponential. However, suboptimal solutions to our ILP problems are still safe bounds (although less precise), which opens up the possibility to use heuristics to reduce complexity.

## References


1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *16th European Symposium on Programming*, 2007.
2. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *ISMM*, 2009.
3. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In *ISMM*, 2010.
4. R. Alur. Timed automata. *Computer Aided Verification*, LNCS 1633, 1999.
5. R. Alur and D. Dill. A theory of timed automata. *Journal of Theoretical Computer Science*, 126(2), 1994.
6. T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1):67–99, 1991.
7. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, 2003.
8. F. Benoy and A. King. Inferring argument size relationships with CLP(R). In *Workshop on Logic-based Program Synthesis and Transformation*, 1997.
9. W. N. Chin, H. H. Nguyen, C. Popeea, and S. Qin. Analysing memory resource bounds for low-level programs. In *ISMM*, 2008.
10. C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1(4), 1992.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

12. J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, (37)1–3, 1998.
13. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19 & 20, 1994.
14. S. Jost, H. W. Loid, K. Hammond, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL*, 2010.
15. M. Kero and S. Aittamaa. Scheduling garbage collection in real-time systems. In *CODES*, 2010.
16. C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.
17. J. Nordlander, M. Carlsson, A. Gill, P. Lindgren, and B. von Sydow. The Timber home page. http://timber-lang.org, 2008.
18. L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In *VMCAI*, 2003.


## Appendix A – Extended example

| task | $P^{min}$ | $P^{max}$ | $D$ |
|---|---|---|---|
| $sample_1$ | 10 | 15 | 5 |
| $sample_2$ | 20 | 20 | 5 |
| $lphigh$ | 100 | 100 | 10 |
| $acquire$ | 1000 | 1000 | 1000 |

| | #states | #arcs |
|---|---|---|
| zone graph: | 6100 | 14072 |
| minimal FSM: | 3510 | 8428 |

$$\begin{aligned}
&sample_1(\langle gval, gbuf, buf_1, buf_2\rangle, \langle gval', gbuf', buf_1', buf_2'\rangle) \leftarrow\\
&\quad gval' := gval, \quad gbuf' := gbuf, \quad buf_2' := buf_2, \quad val_1 := buf_1.head,\\
&\quad buf_1' := \textbf{new}\ cons\,\{head := (sensor_1 + 99 * val_1)/100, tail := buf_1\}.
\end{aligned}$$

$$\begin{aligned}
&sample_2(\langle gval, gbuf, buf_1, buf_2\rangle, \langle gval', gbuf', buf_1', buf_2'\rangle) \leftarrow\\
&\quad gval' := gval, \quad gbuf' := gbuf, \quad buf_1' := buf_1, \quad val_2 := buf_2.head,\\
&\quad buf_2' := \textbf{new}\ cons\,\{head := (sensor_2 + 99 * val_2)/100, tail := buf_2\}.
\end{aligned}$$

$$\begin{aligned}
&lphigh(\langle gval, gbuf, buf_1, buf_2\rangle, \langle gval', gbuf', buf_1', buf_2'\rangle) \leftarrow\\
&\quad gval' := gval, \quad mean(\langle buf_1\rangle, \langle m_1\rangle), \quad mean(\langle buf_2\rangle, \langle m_2\rangle), \quad buf_1' := \textbf{new}\ nil\,\{\},\\
&\quad buf_2' := \textbf{new}\ nil\,\{\}, \quad gbuf' := \textbf{new}\ cons\,\{head := (m_1 + m_2)/2, tail := gbuf\}.
\end{aligned}$$

$$\begin{aligned}
&acquire(\langle gval, gbuf, buf_1, buf_2\rangle, \langle gval', gbuf', buf_1', buf_2'\rangle) \leftarrow\\
&\quad mean(\langle gbuf\rangle, \langle gval''\rangle), \quad gval' := (gval + gval'')/2, \quad gbuf' := \textbf{new}\ nil\,\{\},\\
&\quad buf_1' := buf_1, \quad buf_2' := buf_2.
\end{aligned}$$

$$\begin{aligned}
&init(\langle gval, gbuf, buf_1, buf_2\rangle) \leftarrow\\
&\quad gval := 0, \quad gbuf := \textbf{new}\ nil\,\{\}, \quad buf_1 := \textbf{new}\ nil\,\{\}, \quad buf_2 := \textbf{new}\ nil\,\{\}.
\end{aligned}$$

For cost model $\mathcal{M}$ ($k^{\mathcal{M}}_{cons} = 3$ and $k^{\mathcal{M}}_{null} = 0$) we get the following matrices:

$$\mathbf{A}_{sample_1} = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \quad \mathbf{A}_{sample_2} = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \quad \mathbf{A}_{lphigh} = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix} \quad \mathbf{A}_{acquire} = \begin{bmatrix}1&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

$$\mathbf{C}_{sample_1} = \begin{bmatrix}0\\0\\3\\0\end{bmatrix} \quad \mathbf{C}_{sample_2} = \begin{bmatrix}0\\0\\0\\3\end{bmatrix} \quad \mathbf{C}_{lphigh} = \begin{bmatrix}0\\3\\0\\0\end{bmatrix} \quad \mathbf{C}_{acquire} = \begin{bmatrix}0\\0\\0\\0\end{bmatrix}$$

The minimum solution to $c$ is $c^* = 111$.