Garbage Collection for Reactive Real-Time Systems

Martin Kero
Garbage Collection for
Reactive Real-Time Systems

Martin Kero

Dissertation presented in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Luleå University of Technology
Dept. of Computer Science and Electrical Engineering

Advisors:
Dr. Johan Nordlander
Prof. Per Lindgren
Dr. Pawel Pietrzak
To the apples of my eye

Malin, Eddie, Jeremiah, and Einar
“Of making many books there is no end; and much study is a weariness of the flesh.”

– Ecclesiastes 12:12
Predictable use of resources, such as processor time and memory, is a desirable property for virtually any computer system. In real-time computing, static predictability is of particular concern. A real-time system typically maintains an ongoing interaction with its environment, concurrently executing tasks at predefined intervals or as responses to sporadic external events. Its purpose is often safety-critical, where failures to meet deadlines may have severe consequences – even loss of life. The substantial body of research in real-time scheduling theory has demonstrated that a priori guarantees on schedulability are achievable. What is needed is some carefully chosen restrictions on how tasks may interact and what timing behavior external events may exhibit.

Safe use of shared resources in real-time systems are enabled by protocols for preserving mutually exclusive access to critical sections, free of deadlocks and priority inversions. The ability to achieve a priori schedulability guarantees can be preserved by taking the imposed restrictions into account. Nonetheless, allowing real-time tasks to share heap allocated data has far more complex consequences than just being a schedulability issue concerning shared resources. In order to guarantee failure free operation, the system must be free of memory related errors, such as memory leaks and dangling pointers. It is well-known that garbage collection can solve these problems. However, incorporating a garbage collector in a real-time system imposes a set of other requirements related to schedulability. This includes provably safe upper-bounds on required execution time and memory of the garbage collector in the presence of concurrently executing tasks, as well as preserved ability to achieve static schedulability guarantees of the real-time tasks.

The body of research work towards establishing a priori schedulability guarantees of garbage collected real-time systems has been growing for the past decade. However,
most of the existing work lacks a clear identification of the parameters required for and their impact on establishing a provably safe upper-bound on garbage collection execution time. The lack thereof has often imposed overly simplistic models for the cost of garbage collection; and – probably even more alarming – tractability of finding safe bounds of the involved parameters has not been established. Furthermore, most existing approaches require specialized scheduling policies and schedulability tests, as well as intrusive restrictions on the task model.

In this dissertation, we propose the following theses: (1) the key to successful real-time garbage collection is to preserve as much as possible of previous advances in real-time scheduling theory by imposing minimal restrictions on the task model, scheduler, and schedulability test; and (2) the keys to enabling a priori schedulability guarantees are clear identification and tractable analysis of the sources of execution time for the garbage collector. Proofs of our theses are based on the run-time behavior of the real-time programming language Timber and an incremental copying garbage collector running as the lowest priority (idle) process.

We identify all parameters needed for establishing a safe and tight upper bound on execution time of our collector, where the major ones (not surprisingly for a copying collector) are related to the amount of live heap space. We present a novel technique for establishing safe upper bounds of live heap space parameters for real-time systems. We rely on existing techniques for finding safe upper-bounds of parameters related to heap allocation of each task. Finally, we present the garbage collection demand analysis, decoupled from the schedulability analysis used for the real-time tasks, which determines if our collector is schedulable in the system.
## CONTENTS

**Chapter 1 – Introduction**  
1.1 Research scope and background .................................. 3  
1.2 Thesis statement and contributions .............................. 4  
1.3 Dissertation structure ........................................... 6

**Chapter 2 – Real-Time Systems**  
2.1 Basic task model .................................................. 10  
2.2 Real-time scheduling ............................................. 11  
2.3 Reactive systems ................................................. 20  
2.4 Timed automata .................................................... 29

**Chapter 3 – Garbage Collection**  
3.1 The mark-sweep garbage collector ................................ 40  
3.2 Fragmentation .................................................... 43  
3.3 The mark-compact garbage collector ............................. 45  
3.4 The copying garbage collector ................................... 46  
3.5 Generational Garbage Collection ................................. 53  
3.6 Incremental Garbage Collection .................................. 56  
3.7 Real-Time garbage collection .................................... 63  
3.8 Design choices .................................................... 66

**Chapter 4 – A Correct and Useful Garbage Collector**  
4.1 The Garbage Collector ............................................ 70  
4.2 The Mutator ....................................................... 72  
4.3 Correctness of the Garbage Collector ............................ 75
10.2 Further work ..................................... 154

Bibliography ........................................... 161
## List of Figures

2.1 Difference between hard and soft real-time systems. Value of the response as a function of time. ............................................. 10
2.2 Example schedule of a real-time system with three periodic tasks (dashed box indicates pending task). .................................................. 11
2.3 Sufficient RM-schedulability test based on $U_r$. .......................... 14
2.4 An exact RM-schedulability test for the example tasks in Table 2.2. ... 16
2.5 EDF-schedulability analysis example based on processor demand $h_r(t)$. 19
2.6 A typical event-loop structure. ........................................... 22
2.7 Connecting events directly to their corresponding handlers. ............. 23
2.8 Example showing the concurrency model of Timber. ...................... 25
2.9 Example Timber program, a Counter. ................................... 26
2.10 Time window of a Timber method execution. ................................ 27
2.11 Simple example of a periodic task in Timber. ............................. 27
2.12 Sonar example. A timed automaton with two clocks. ...................... 31
3.1 Example of a heap set up before a copying garbage collection cycle. ... 47
3.2 Begin by copying the first root. .......................................... 48
3.3 Continue with second one. ............................................... 48
3.4 Update a pointer with the new (forwarding) address. ...................... 49
3.5 Copy encountered objects to tospace. .................................. 49
3.6 When the garbage collection cycle is finished, the resulting heap is a de-fragmented and garbage collected version of the previous one. ....... 50
3.7 The Cheney algorithm’s breadth-first traversal .......................... 52
3.8 Moon’s approximately depth-first traversal based on a partial scan pointer of the latest virtual memory page. .......................... 53
3.9 The tri-color invariant is upheld during garbage collection.

3.10 The mutator introduces a violation of the tri-color invariant.

3.11 The tri-color invariant is preserved by coloring the white object (D) gray.

3.12 The tri-color invariant is preserved by coloring the black object (A) gray.

3.13 New allocations during garbage collection are made at the other end of tospace and considered black in Baker’s incremental copying garbage collector.

3.14 Typical schedule of a system based on Henriksson’s scheduling policy.

4.1 Scan transitions.

4.2 Copy transitions.

4.3 Mutator transitions.

4.4 Weight of a heap

5.1 The memory set up for our implementation.

6.1 Timed automaton $A_i$ capturing the execution points of task $t_i$

6.2 Timed automaton of the worked example.

6.3 Minimal FSM representation of the execution orders of the worked example.

6.4 Syntax of rule-based representation

6.5 The worked example in the rule-based representation

6.6 Operational semantics of rule-based programs

6.7 Abstract compilation to CLP

6.8 The worked example after abstract compilation

7.1 Example of a Timber system with only static objects.

8.1 Measurement data of GC time as a function of live memory.

8.2 Measurement data of GC time as a function of number of live references.

8.3 Measurement data of GC time as a function of the length of the live linked list.

8.4 Measurement data of GC time as a function of number of references allocated by an interrupting task.
8.5 Measurement data of GC time as a function of the length of the linked list allocated by an interrupting task. .......................... 141
8.6 EDF-schedulability analysis of our extended example. ...................... 143
8.7 Garbage collection demand analysis of our extended example. .......... 144
10.1 Timed Automata of a task with $D_i > \bar{P}_i^{\min}$. .................. 158
10.2 Example of SRP nested resource access by a task. ...................... 158
10.3 Example of sub-tasks conforming to SRP. ............................... 159
## List of Tables

2.1 Real-time scheduling: Notation ........................................ 13
2.2 Example task set .......................................................... 15
2.3 Example task set ($D_i \neq P_i$ for some $i$) ...................... 18
8.1 Zone graph and minimal FSM sizes of the example task sets .... 132
8.2 Task set of the extended example of live heap space analysis. . 133
8.3 Zone graph and minimal FSM sizes of our extended example. . . 135
8.4 Example transition times affecting $T_{gc}$ and $T_i^A$ .................. 142
10.1 Minimal FSM sizes of the example task sets for EDF and DM scheduling. 155
10.2 Minimal FSM sizes of our extended example for EDF and DM. . . 156
I would like to thank my advisors for their patience and guidance throughout my doctoral candiday at Luleå University of Technology. Dr. Johan Nordlander and Professor Per Lindgren, whom have been my advisors since the start and whose support are essential pediments of this dissertation. I would also like to give a special thank you to my advisor Dr. Paweł Pietrzak, whose acquaintance I consider a pure privilege. He is the mastermind behind the live heap space analysis presented in Chapter 6. This dissertation would not have been possible without you.

Thanks are also due to my colleagues and dear friends in the Timber group. Andrey Kruglyak, who helped me improve this dissertation through extensive proofreading as well as with ideas regarding the overall presentation. Simon Aittamaa, who made the measurements presented in Chapter 8 possible. Peter Jonsson and Viktor Leijon, with whom I have spent many hours in fruitful discussions about our research work.

Finally, to my wife Malin and our boys Eddie, Jeremiah, and Einar: your love, support, and great sacrifice, enabling this dissertation, are beyond what anyone can ask for. It is with great honor and love I dedicate this book to you.

Boden, October 2010

Parts of this dissertation are based on published research papers [48, 49, 50, 51, 58].
Predictable use of resources, such as processor time and memory, is a desirable property for virtually any computer system. Evidence of its importance can be discerned in most research areas in the field of computer science – ranging from computational complexity analysis of algorithms to implementation and performance evaluation of concrete systems. In safety-critical computer systems, where failures may have severe consequences (even loss of life), predictable behavior is utterly important. Such systems are commonly required to have a priori provable bounds on the use of particular resources in order to prevent failures due to excessive use.

In real-time systems – where timing properties play a vital role in the program specification – static predictability is of particular concern. Typically, such systems keep an ongoing interaction with its environment, concurrently executing fragments of code (tasks) at predefined intervals and in response to sporadic events. The substantial body of results in real-time scheduling theory has demonstrated that sporadic events and concurrent execution are not fundamentally at odds with static predictability [55]. What is required is some carefully chosen restrictions on how tasks may interact with each other, and what time patterns external events may exhibit. In the past decades, much research work has been invested in improving various aspects of real-time scheduling. These improvements range from extending task models with timing specifications closer to real-world needs, to liberating tasks to interact more freely. One of the most significant liberations is the possibility to let tasks share other resources, such as peripheral
hardware devices or dynamic memory, without disabling static schedulability analysis. However, enabling tasks to share dynamic memory has far more complex consequences than just being a schedulability issue concerning shared resources.

One of the most tedious and error-prone tasks of programming complex systems is that of managing dynamic memory. As features like dynamic multi-threading and object-orientation crave highly dynamic data storage capabilities, the task of reclaiming and reusing old storage space efficiently is often very detached from the primary purpose of the system. It would be very convenient if such low-level details of where and how dynamic storage is reclaimed could be dealt with automatically behind the scenes. The idea of integrating such a facility in the run-time system of a program’s environment has been around for a long time. Nowadays, with high-level object-oriented languages such as Java and C#, this idea has been spread to the mainstream programming community. Today it is well-known that garbage collection is a powerful tool to reduce memory related programming errors, such as memory leaks and dangling pointers. The basic idea is to automatically detect heap-allocated objects that are not used by the program anymore and recycle the memory they occupy by making it available for reuse. Despite the advantages, garbage collection of real-time systems has not yet reached the wide acceptance as it has in the object-oriented programming community. In the early days, garbage collectors typically required exclusive access to the heap while collecting, which made them directly unsuitable for real-time systems. However, much progress has been achieved since then and nowadays a broad spectrum of incremental garbage collectors exist [44]. Although real-time systems exhibit a perfect example of concurrent behavior that would benefit from garbage collection, there are still obstacles to overcome. Among other parameters, schedulability analyses require safe worst-case execution time (WCET) estimates for the real-time tasks in order to enable provable bounds on processor demand and response time. Similar safe estimate for the garbage collector would of course also be needed. However, establishing such an estimate is not trivial since it typically depends on a combination of memory and timing behaviors of the real-time tasks. Such behaviors range from each tasks individual allocation rate to their combined effect on live memory.
Other more subtle behaviors, such as fragmentation of memory, mutation patterns, and certain structural arrangements of objects in memory, may also contribute to the execution time of the garbage collector. On top of all this, a garbage collector must also be “schedulable” in the memory dimension, i.e., a provable bound on total heap memory usage must also be established.

1.1 Research scope and background

The purpose of the research work presented in this dissertation is to investigate the problem of enabling a priori provable bounds on both heap memory use and processor demand for a garbage collector in real-time systems. While such systems are described by fairly general task models in the real-time scheduling literature [55], a garbage collector is typically tightly coupled with the memory model of a run-time environment of a particular programming language. In order to remedy this modeling discrepancy, we target a real-time system model that connects fairly well to task models used in the literature, while drawing its concrete inspiration from the execution principles underlying the real-time programming language Timber [68].

Timber will serve as the basis for the research on real-time garbage collection presented in this dissertation. Although most of the results we present are applicable to real-time systems in general, some properties of Timber, such as pureness, will prove crucial when it comes to achieving predictable garbage collection. In order to limit our research scope to a reasonable size, we will only consider standard uniprocessor scheduling and fairly simple memory architectures (i.e., we assume near-uniform memory access time).

Real-time scheduling and garbage collection In real-time scheduling theory, parameters related to use of processor time have naturally received most of the attention. Schedulability is determined by feasibility tests crucially depending on parameters such as worst-case execution time. Although WCET analysis of real-time tasks is a research
topic in itself [88], it is typically safe, due to modest restrictions in the task model, to assume that such safe estimates can be determined independently of scheduling policy. Avoiding mutual dependencies between WCETs, timing assumptions, scheduling policies, and feasibility tests has in fact proved to be very successful in real-time scheduling theory. This has been one of the keys behind the substantial body of incremental enhancements of real-time scheduling in the past decades [55].

Due to the well-established foundation of real-time scheduling, it is very tempting to schedule a concurrent garbage collector by the same principles. It would be convenient to let it compete under similar conditions as the real-time tasks. As recent research in this area has shown, the collector task simply has to be assigned some appropriate timing assumptions [24]. However, the problems arise when a safe estimate of the collector WCET has to be established. Instead of following the spirit of being independent, its execution time commonly originates from a broad spectrum of sources, ranging from memory-related behaviors of the real-time tasks, to their timing assumptions (including the collector’s own artificial ones) and the scheduling policy used.

Instead of sharing a mission similar to the real-time tasks, a garbage collector is required to keep up with them by providing sufficient amounts of free heap space while not exhausting available memory resources. Allowing the garbage collector to compete under similar conditions as the real-time tasks inevitably leads to a highly specialized scheduling policy/feasibility test and an unnecessarily restricted task model. The latter may even require that tasks do not share dynamic memory; which, ironically, is one of the major motivations behind introducing garbage collection to these systems in the first place.

1.2 Thesis statement and contributions

In this dissertation, we propose the following theses: (1) the key to successful real-time garbage collection is to preserve as much as possible of previous advances in real-time scheduling theory by imposing minimal restrictions on the task model, scheduler, and
1.2. Thesis statement and contributions

schedulability test; and (2) the keys to enabling a priori schedulability guarantees are clear identification and tractable analysis of the sources of execution time for the garbage collector. While proving the above theses, we make the following major contributions:

1. We define an incremental copying garbage collector based on process terms in a labeled transition system (LTS). Garbage collector increments are defined as internal transitions and the mutators’ interface to the heap is captured by labeled transitions. We reason about the basic properties of the algorithm. We show that the garbage collector eventually terminates and that the meaning of the heap is preserved at all times. These properties constitute correctness of the model. We also show that the algorithm enables a priori memory requirement analysis by reasoning about usefulness of the garbage collector in terms of reduced memory occupancy.

2. We identify all parameters affecting execution time for the garbage collector, where the major ones are live heap space and heap allocations made by interrupting tasks. Analyzing heap allocation properties for each task corresponds quite well to execution time analysis (with a slightly different cost model). Both are monotonically increasing accumulative properties. The results of such program analyses are typically expressed as functions of the programs input data (e.g., see [3]). In a real-time system, where tasks maintain an ongoing interaction with the environment, input data are typically tightly coupled with the persistent state of the system. As a fortunate coincidence, such state-dependent properties correspond very well to the behavior of global live heap space.

3. We present a technique for computing upper bounds on live heap memory of real-time systems, which is safe even in the presence of state- and order-dependent tasks driven by external sporadic events. It 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 applying standard techniques used in reachability analysis to construct a finite state machine (FSM) representation of task execution orders.
We apply a standard variant of abstract interpretation to each task to infer linear input/output size relations 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. We also present a set of examples illustrating how the implemented analysis algorithm behaves in practice.

4. Since the behavior of a garbage collector is tightly coupled with the run-time environment of the programming language, understanding the interactions between mutators and the collector is a crucial component of predictable garbage collection. We identify the key properties of a real-time programming language that enable both predictable garbage collection and tractable analysis of the required task parameters involved. These properties involve type-safety, state integrity, and pureness, typically found in the functional programming paradigm.

5. We present the garbage collection demand analysis, which independently of scheduling policy applied to the real-time tasks, determines feasibility of our garbage collector. The analysis is based on the restriction that the garbage collector runs at the lowest priority, commonly referred to as background collection or idle time collection.

6. Our garbage collector has been implemented and deployed in the run-time environment of Timber, a real-time programming language exhibiting the required properties. We present experimental results confirming the behavior of our collector on a standard uniprocessor platform with a fairly simple memory architecture (with near-uniform access times).

1.3 Dissertation structure

- Chapter 2 is a background study of real-time systems (basic task model, real-time scheduling theory, reactive programming, Timber, and timed automata).
1.3. Dissertation structure

- Chapter 3 is a background study of various garbage collection techniques.
- Chapter 4 describes our garbage collection algorithm.
- Chapter 5 describes the deployment of our collector in the run-time environment of Timber.
- Chapter 6 describes our technique for determining global live heap space bounds for real-time systems.
- Chapter 7 describes the static schedulability analysis of our collector, i.e. the garbage collection demand analysis.
- Chapter 8 contains experimental results of our collector on a standard uniprocessor platform with a fairly simple memory architecture. It also contains experimental results of our live heap space analyzer.
- Chapter 9 is a survey of related work in the field of real-time garbage collection as well as static analysis of live heap space.
- Chapter 10 contains our conclusion and pointers to further work.
Any comprehensive definition of the term *real-time system* is deemed to be rather fuzzy. In order to sufficiently capture the intuition, such a definition would most probably end up covering virtually all computer systems present today. What we can say, though, is that the difference between a real-time system and a non-real-time system is that a real-time system has a set of explicit timing requirements as a *vital* part of its specification. This kind of requirements is typically found in safety-critical applications in embedded systems, where failures to meet timing requirements have severe consequences. In such a setting, a real-time system commonly maintains an ongoing interaction with its environment, concurrently executing fragments of code (called *tasks*) at predefined intervals or in response to external sporadic events.

In the literature (e.g., [55]), real-time systems are usually divided into two categories. On one hand, we have the *hard* real-time systems that are considered broken if they fail meeting a deadline, i.e. timing failures are as bad as computational errors. On the other hand, we have *soft* real-time systems for which the *value* of the response gradually degrades with its lateness. This difference is illustrated in Figure 2.1. In this dissertation we will only consider hard real-time systems, and whenever we refer to real-time systems, we implicitly mean hard real-time systems.

A real-time system is not necessarily *faster* than any other computer system (or slower for that matter). From a correctness point of view, it does not make much of a difference if a result is delivered early or just a mite before its deadline, as long as it is
delivered within the required time window. In order to enable a priori guarantees for such requirements, it is of particular concern that the system exhibits a predictable behavior. What is required are some carefully chosen restrictions on how tasks may interact with each other, and what time-patterns external events may exhibit.

2.1 Basic task model

In this section we will give a brief overview of the basic modeling paradigm used for real-time systems. A real-time system consists of a set of tasks, and each task – either independently or codependently with other tasks – has a mission. A task is distinguished by its timing behavior. A periodic task is released in a steady pace determined by its predefined period. A sporadic task, on the other hand, is released by an external event for which a minimum inter-arrival time must be assumed. If such an assumption cannot be made, the task is considered aperiodic. Each task has a predefined relative deadline. This deadline is used to compute the absolute deadline for each release of the task. For a sporadic task, the relative deadline is typically derived from requirements in the physical environment. For example, a nuclear reactor might risk over-heating if the temperature control unit does not respond in time. Another source of deadlines is due to over-load avoidance. For instance, the relative deadline of a periodic task might be equal to its
period, i.e. each release has to finish before the next one arrives. Furthermore, each task is required to have a safe worst-case execution time (WCET) estimate. In Figure 2.2, an example of a typical schedule of periodic tasks is shown.

\[
P = \text{Period}, \quad C = \text{Execution time}, \quad D = \text{Relative deadline}
\]

- task1 = \{ P=10, C=2, D=10 \}
- task2 = \{ P=5, C=3, D=5 \}
- task3 = \{ P=20, C=1, D=20 \}

Figure 2.2: Example schedule of a real-time system with three periodic tasks (dashed box indicates pending task).

In addition to this basic task model, typical extensions are concerned with precedence constraints, refined timing characteristics, and shared resources. In the next section we will look at how schedulability can be determined for the basic model, as well as for some of the extended ones.

### 2.2 Real-time scheduling

The main purpose of real-time computing is to always execute its released tasks within their corresponding deadlines. To this end, a scheduler is used to decide when a task
may execute and when it has to yield for something more urgent. In general, real-time
schedulers\footnote{Observe that we are only considering on-line schedulers.} are myopic and the problem of determining exactly which task that
is most urgent is typically undecidable. However, given certain restrictions, fairly good
approximations can be achieved efficiently. In this section we will survey the common
techniques for scheduling real-time task sets on uniprocessor platforms. First, we will go
through the basic concepts used in the rest of this section.

2.2.1 Scheduling basics and terminology

In general, each task has resource demands, often from a source shared among the other
tasks. The most obvious example of a resource demand is execution time on a processor;
but it can be objects in memory, peripheral hardware devices, etc. Tasks may temporarily
require exclusive access to certain resources, i.e. while a task holds access to a resource,
no other task may access it. Depending on what type of resource and action performed
on it, a resource may be classified as exclusive or nonexclusive.

The release time of a task is the time when all required input is available and the task
may start executing. The baseline, or earliest release time, of a task is the (theoretically)
earliest time when a task may be released. The absolute deadline of a task is the time by
which the task must complete its execution. The absolute deadline of a task is computed
as the baseline plus the specified relative deadline.

A schedule is feasible if all tasks start executing after their baselines and complete
before their deadlines. If a scheduling algorithm/policy, when run on a set of tasks,
results in a feasible schedule, the task set is considered schedulable by that policy. A
busy-period is when the processor is executing continuously for an extended period of
time. The longest busy-period in a schedule is when all tasks are released at exactly the
same time (called the critical instant) and then arrive at their maximum rates. However,
if periodic tasks are allowed to have different phasings (i.e., the first iterations may be
released at different points in time), then the longest busy-period may be less since the
release times of all tasks may never coincide. A typical assumption in most schedulability
tests is to assume that task phasings are all zero.

In order to enable sensible scheduling decisions, each task must have a priority. The way priorities are assigned to tasks is one of the key differences between different scheduling policies. On one hand we have static-priority schedulers, where task priorities are assigned a priori. On the other hand we have dynamic-priority schedulers, where priorities are allowed to change with time.

Preemptive scheduling means that a task can be interrupted by other tasks with higher priorities. Non-preemptive scheduling on the other hand requires that a task, once it starts to execute, must run to completion before another may start. We will only look at preemptive schedulers in this section. In Table 2.1, the notation used throughout this section is shown.

| $\tau = \{\tau_1, \ldots, \tau_n\}$ | Task set with $n$ tasks. |
| $C_i$ | Execution time of task $\tau_i$ |
| $D_i$ | Relative deadline of task $\tau_i$ |
| $P_i$ | Period of task $\tau_i$ |
| $U_i$ | Processor utilization of $\tau_i$ ($U_i = C_i/P_i$) |
| $U_\tau$ | Processor utilization of $\tau$ ($U_\tau = \sum_i U_i$) |

Table 2.1: Real-time scheduling: Notation.

2.2.2 Static-priority scheduling

One of the most well-known static-priority scheduling policies is the Rate Monotonic (RM) scheduler [60]. The RM scheduling policy is based on the assumption that all tasks are periodic$^2$ and the relative deadline of a task is equal to its period. The priority of each task is then statically determined by ordering the tasks by their periods, i.e. the

$^2$A safe approximation of a sporadic task as periodic is that the minimum inter-arrival time is its period.
task with the shortest period (highest arrival rate) has the highest priority.

If all tasks are independent (i.e., no shared resources besides the processor and no precedence constraints) and the penalty for preempting any task at any time is negligible, there is a simple sufficient schedulability test based on the total processor utilization.

\[ U_T \leq n(2^{1/n} - 1) \]  

(2.1)

Observe that this test is not a necessary condition; i.e., there might be task sets with greater processor utilization that are RM-schedulable. In Figure 2.3, the bound of (2.1) is shown.

![Figure 2.3: Sufficient RM-schedulability test based on \( U_T \).](image)

A sufficient and necessary schedulability test for RM under the given assumptions is based on computing the processor demand starting at a critical instant [56]. For notational simplicity, we assume that the task set is ordered in such way that \( P_1 \leq \ldots \leq P_n \). The following equation captures the total amount of processor time required by tasks \( \tau_1, \ldots, \tau_i \) initiated in the interval \([0, t]\).

\[ W_i(t) = \sum_{j=1}^{i} \left( \left\lceil \frac{t}{P_j} \right\rceil \ast C_j \right) \]

(2.2)
2.2. Real-time scheduling

The key observation is that if there exists a \( t' \) such that \( 0 < t' \leq P_i \) and \( W(t') \leq t' \), then \( \tau_i \) is schedulable with RM. Note that, even though there are infinite many points in time to check, it follows from (2.2) that \( W_i(t) \) will only change value at a finite number of points (when tasks are released) in the time interval \((0, P_i]\). For \( \tau_i \), it is sufficient to check the following set of points.

\[
T_i = \left\{ k \cdot P_j \mid k = 1, \ldots, \left\lfloor \frac{P_i}{P_j} \right\rfloor; j = 1, \ldots, i \right\}
\]  

(2.3)

<table>
<thead>
<tr>
<th>(i)</th>
<th>(P_i)</th>
<th>(C_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>45</td>
<td>8</td>
</tr>
</tbody>
</table>

*Table 2.2: Example task set.*

Let us look at how the analysis works through an example. Consider the set of periodic tasks in Table 2.2. We have the following sets of points in time to consider:

\[
T_1 = \{12\}
\]
\[
T_2 = \{12, 13\}
\]
\[
T_3 = \{12, 13, 24, 25\}
\]
\[
T_4 = \{12, 13, 24, 25, 26, 36, 39, 45\}
\]

In Figure 2.4, the resulting processor demands are plotted for all tasks. The dashed line show where the required and available processor times are equal.

2.2.3 Dynamic-priority scheduling

The basic idea behind dynamic-priority scheduling is that priorities change with time, typically in relation to tasks’ deadlines. The *Earliest Deadline First* (EDF) scheduler
assign priorities to released tasks based on the closeness of their absolute deadlines [60].

Given the same assumptions about the task set as for the simple RM-schedulability test
(2.1) EDF-schedulability can be determined by the following, *even simpler*, test based on total processor utilization.

\[ U_\tau \leq 1 \] (2.4)

Observe that (2.4) is a both sufficient and necessary condition for EDF-schedulability, in contrast to (2.1) for RM-schedulability whose result is not a necessary condition.
2.2. Real-time scheduling

2.2.4 Scheduling tasks with \( D_i \neq P_i \)

If, for some task \( \tau_i \), its relative deadline \( D_i \) is not equal to its period \( P_i \), the simple utilization test (2.4) is not sufficient. A sufficient and necessary EDF-schedulability test for such systems can be based on the following processor demand equation [13, 14].

\[
h_{\tau}(t) = \sum_{i=1}^{n} \max\left\{ 0, 1 + \left\lfloor \frac{t - D_i}{P_i} \right\rfloor \right\} \ast C_i \tag{2.5}\]

The schedulability test requires (2.4) and \( \forall t > 0 \cdot h_{\tau}(t) \leq t \). Similar to (2.2), (2.5) only changes value at some specific points in time, to wit at each absolute deadline. It is sufficient to check up to the latest relevant absolute deadline \( (L_a) \), which can be computed by the following formula [12, 71].

\[
L_a = \max\left\{ D_1, \ldots, D_n, \frac{\sum_{i=1}^{n} (P_i - D_i) \ast U_i}{1 - U_r} \right\} \tag{2.6}\]

This has been improved quite recently by Zhang and Burns [94], resulting in the following formula.

\[
L^*_{a} = \max\left\{ D_1 - P_1, \ldots, D_n - P_n, \frac{\sum_{i=1}^{n} (P_i - D_i) \ast U_i}{1 - U_r} \right\} \tag{2.7}\]

It is also sufficient to check only such points within the longest busy period \( (L_b) \), which can be computed by the following iterative formula [79].

\[
w^0 = \sum_{i=1}^{n} C_i \tag{2.8}\]

\[
w^{m+1} = \sum_{i=1}^{n} \left\lceil \frac{w^m}{P_i} \right\rceil \ast C_i\]

The process ends when \( w^{m+1} = w^m = L_b \). Combining (2.7), and (2.8) gives the following set points in time to check that \( h_{\tau}(t) \leq t \).

\[
T = \{ d_k \mid k \in N; i \in \{1..n\}; d_k = k \ast P_i + D_i < \min\{L^*_a, L_b\} \} \tag{2.9}\]

One key observation is that if \( \min\{L^*_a, L_b\} \) is large, the number of absolute deadlines to check in \( T \) can be large. This number can be reduced quite significantly due to the fact that \( t < t' \implies h_{\tau}(t) \leq h_{\tau}(t') \). Let \( L = \min\{L^*_a, L_b\} \) and \( d_{\min} \) be the earliest absolute
deadline in the interval \([0, L]\). Instead of constructing \(T\), it is sufficient to only check the time points defined by the following iterative formula \([94]\).

\[
\begin{align*}
t^0 &= h_\tau(L) \\
t^{m+1} &= h_\tau(t^m) & \text{if } h_\tau(t^m) < t^m \\
t^{m+1} &= \max\{d_k \mid d_k < t^m\} & \text{if } h_\tau(t^m) = t^m
\end{align*}
\]

Continue until, for some \(t^m\), either \(t^m < d_{\min}\) (\(\tau\) is schedulable) or \(t^m > h_\tau(t^m)\) (\(\tau\) is not schedulable).

<table>
<thead>
<tr>
<th>(i)</th>
<th>(P_i)</th>
<th>(D_i)</th>
<th>(C_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>45</td>
<td>40</td>
<td>9</td>
</tr>
</tbody>
</table>

\textit{Table 2.3: Example task set (\(D_i \neq P_i\) for some \(i\)).}

Let us look at how the EDF-schedulability analysis works through an example. Consider the task set shown in Table 2.3. First we need to check (2.4). From Table 2.3 it follows that:

\[
U_\tau = \frac{1}{3} + \frac{3}{13} + \frac{5}{25} + \frac{9}{45} \approx 0.964 \leq 1.
\]

We get the following \(L_\alpha^*, L_b, T\):

\[
\begin{align*}
L_\alpha^* &\approx 30.7 \\
L_b &= 89 \\
T &= \{2, 5, 8, 11, 14, 17, 20, 23, 25, 26, 27, 29\}
\end{align*}
\]

The processor demand \(h_\tau(t)\) is shown in Figure 2.5. The gray line shows the sufficient points in time to check according to (2.10).
2.2. Real-time scheduling

2.2.5 Shared resources

If real-time tasks are allowed to share resources, certain precautions must be taken in order to enable a priori schedulability analysis. In traditional computing, a shared resource is typically protected by a semaphore, which purpose is to guarantee mutually exclusive access to the shared resource. In order to incorporate such mechanisms in the schedulability analysis, one must avoid priority inversion [77]. Let us look at an example of priority inversion. Assume that we have three tasks $\tau_{hp}$ (high priority), $\tau_{mp}$ (mid priority), and $\tau_{lp}$ (low priority); and a resource shared between $\tau_{hp}$ and $\tau_{lp}$. Consider the following scenario: $\tau_{lp}$ executes and enters its critical section, then $\tau_{mp}$ is released and preempts $\tau_{lp}$, followed by a release of $\tau_{hp}$ who preempts $\tau_{mp}$ but must yield when it tries to enter its critical section (resource held by $\tau_{lp}$) and $\tau_{mp}$ continues to run. Now $\tau_{mp}$ is effectively blocking $\tau_{hp}$ even though they have no shared resources. One solution to this problem is to let $\tau_{hp}$ temporarily get the same priority as $\tau_{hp}$ when $\tau_{hp}$ enters its critical section of their shared resource. This solution is used in the priority ceiling protocol (PCP) where each shared resource is assigned a priority ceiling [77]. The ceiling is the highest priority among all tasks sharing the resource. For static-priority schedulers, such as RM, each ceiling can be determined a priori. For dynamic-priority schedulers, such
as EDF, there exist an extension to PCP called stack resource policy (SRP) [11]. The basic idea of EDF+SRP relies on the fact that, in EDF scheduling, a task \( \tau_j \) can only preempt \( \tau_i \) if \( D_j < D_i \). Thus, each resource can be assigned a preemption ceiling based on the relative deadlines of all tasks sharing the resource.

\[
\pi_r = \min \{ D_i \mid \tau_i \text{ may lock } r \} \tag{2.11}
\]

In addition to preemption ceilings of all resources, EDF+SRP needs to keep track of the current global system ceiling.

\[
\bar{\pi} = \min \{ \emptyset \cup \{ \pi_r \mid r \text{ is locked } \} \} \tag{2.12}
\]

Besides the scheduling decision based on earliest deadline, a preemption requirement is added, to wit in order to execute a released task \( \tau_i \), \( D_i < \bar{\pi} \). A schedulability test for EDF+SRP can be based on a straightforward addition to (2.5) by incorporating the cost of blocking,

\[
B_i(t) = \max \{ C_{i,j} \mid D_i > t \land D_j \leq t \} \tag{2.13}
\]

where \( C_{i,j} \) is the maximum time \( \tau_i \) may block \( \tau_j \) due to common shared resources [94]. Let \( B_{\text{max}} = \max_{d_i < D_{\text{max}}} \{ B_i(d_i) \} \) denote the maximum blocking time. Equation (2.7) and (2.9) needs to be modified to the following.

\[
L^B_i = \max \left\{ D_1 - P_1, \ldots, D_n - P_n, \frac{B_{\text{max}} + \sum_{i=1}^{n} (P_i - D_i) * U_i}{1 - U} \right\} \tag{2.14}
\]

\[
T^B = \{ d_k \mid k \in N; i \in \{1..n\}; d_k = k * P_i + D_i < \min \{ L^B_i, L_{\text{b}} \} \} \tag{2.15}
\]

The EDF+SRP-schedulability test is \( \forall t \in T : h_i(t) + B_i(t) \leq t \).

### 2.3 Reactive systems

The main idea of reactive programming is that the program should react to events in its environment. The traditional batch-oriented programming model, however, imposes an active master-slave relationship to the environment – the program controls the environment. Even though programming models have been evolving since the batch-oriented...
days, the view of the environment has persisted. Access to external devices is still made through block-and-wait-for-input procedures subtly hidden by regular subroutine calls. This may be appropriate (even today) when it comes to communication with a wide range of external devices dealing with data storage (hard disks, RAM, etc.). However, when these external "devices" become more difficult to control, this view of the world becomes highly inappropriate. For instance, consider a program including a graphical user interface may need to react to mouse clicks, key presses, and network packets in an order determined solely by the environment; or an embedded system, which usually has an even wider range of input sources. These types of complex environments cannot be controlled in a simple way such as the case of more controllable external devices. They require a facility for handling (in contrast to controlling) events in a robust manner.

2.3. Reactive on top of active

Block-waiting-for-input procedures (e.g. getChar()) are common in almost all modern programming languages, and event-driven programs are commonly encoded on top of this model. This is typically done by a so called event-loop. Events are captured by a block-and-wait call to an event delivery service offered by the operating system and then distributed to its appropriate handler. A typical structure of the event-loop is shown in Figure 2.6.

In order to preserve aliveness of the system, only one block-and-wait call to the event delivery service is allowed for each concurrent activity. This is done by putting it at the top-level. Utmost care must be taken to ensure that no one of the event handlers makes blocking calls.

Concurrency in such systems is usually achieved by using threads. Designing a multi-threaded system explicitly is a tedious and error-prone task. Shared data must be protected by semaphores, mutexes, etc. Special care must be taken to ensure that the system is free of race conditions and deadlocks, which usually are hidden very subtly.
Figure 2.6: A typical event-loop structure.
2.3.2 Pure reactive programming

A purely reactive system can be defined as a system without block-and-wait calls. Instead of that, the system is designed to react to input from its environment. In other words, instead of actively ask for input through a block-and-wait procedure, the default state of the system is idle, ready to react to whatever it is defined to react on.

![Diagram](image)

Figure 2.7: Connecting events directly to their corresponding handlers.

In contrast to an event-loop based structure as shown in Figure 2.6, a pure event-driven system may connect events directly to their corresponding handlers as shown in Figure 2.7. In order to make this structure more flexible, concurrency and mutual exclusion could be made implicit through the semantics of a programming model for pure reactive systems.

In the next section, we will describe such a programming model.

2.3.3 Timber – Reactive objects

Timber is a strongly typed, object-oriented programming language for reactive real-time systems. The basis of the language is the semantics of reactive objects. This section is just a brief description of the parts of the language that are relevant to the rest of this
dissertation. More in-depth descriptions can be found at the Timber homepage [68], in the draft language report [20], the formal semantics definition [23], and the description of the reactive object model [69]. The real-time properties of the language have been described from different point of views and in different contexts in several papers [59, 49, 82, 67, 58].

The reactive objects in Timber are implicitly concurrent. That means activity in one object can execute concurrently with activity in other objects. Furthermore, objects are the only stateful data structures. In addition, the only way to access the state of an object is through its methods, and methods of one particular object execute under mutual exclusion. That is how state integrity is preserved. In order to preserve aliveness, methods cannot block-and-wait for future events. Events are instead interpreted as method calls and each method will eventually terminate, leaving the object in an idle state. An example showing the concurrency model of Timber is shown in Figure 2.8. The arrows in the model represent message-based communication in both an object-oriented and concurrent programming sense.

Messages can either be asynchronous (equivalent to events in their own right) or synchronous rendezvous operations. In Figure 2.9, an example of a program in concrete Timber syntax is shown.

At the top-level, a Timber program is a set of bindings of names to expressions. In the example (Figure 2.9) an object generator expression (identified by the keyword class) is bound to the name counter. The statements within a class construct conceptually consist of state variable initializations and a result statement for the interface to the object. The interface is usually a record but could be any data structure. Anyhow, for a class construct to be practically useful, the returned interface should include at least one method. A method, in turn, can be of two kinds: either an asynchronous operation (identified by the keyword action), or a synchronous rendezvous (identified by the keyword request). An asynchronous method invocation is conceptually an event in its own right, no matter if it is due to an external event or an invocation by another method. Interrupts from the environment is connected to asynchronous methods. In the example, suppose the counter program was running on an interactive machine where one
2.3. Reactive systems

of the input sources is a button. Every time the user pushes the button, an interrupt connected to the incr method is generated. Or, incr might only be called from another method based on some internal calculations. Any way, every time incr is invoked, it is a unique event.

A statement within a method can primarily be one of three kinds: either an assignment to the state of its object, a class instantiation, or a method call. Local bindings as well as looping and branching constructs are of course also available. Observe the

Figure 2.8: Example showing the concurrency model of Timber.
counter = class
value := 0
result {
    read = request
    result value
    incr = action
    value := value + 1
}

Figure 2.9: Example Timber program, a Counter.

difference between the destructive update of state variables (denoted by :=) and the declarative bindings between names and expressions (denoted by =). Destructive updates are only available for state assignments within the scope of a class construct. In addition, in the spirit of separating different kinds of assignments, Timber also includes a third binding, which enables one to bind the result of a side-effecting command to a local name (denoted by a left-arrow <-), i.e. in contrast to the declarative binding, which binds the name to an expression, the left-arrow binds the name to the result of executing something that may have an externally observable effect.

Aside from the reactive objects, Timber has a pure functional layer based on a call-by-value lambda calculus. It is extended with constructors for representing primitive data types (such as integers, floating point numbers, etc.), as well as immutable data structures (lists, records, tuples, etc.).

2.3.4 Real-time in Timber

Timber offers a capability to express timing constraints directly in the model. Asynchronous methods (events) are associated with both an baseline (earliest release time) and a deadline (latest response time). If not explicitly set, the baseline of a method invocation is inherited from the method who called it. If the event is due to an interrupt, the baseline is set to the time-stamp of the interrupt. Deadlines are expressed relative
2.3. Reactive systems

to the current baseline. Similar to baselines, deadlines are inherited if not explicitly set. In Figure 2.10, a time window of a Timber method execution is shown.

![Figure 2.10: Time window of a Timber method execution.](image)

Correct timing behavior of a method is that it must start executing and finish within its time window. Baselines and deadlines can be adjusted by explicitly expressing it in the definition of an asynchronous method (or by the caller). The keyword `after` is used to denote an extension of the current baseline. Along the same line, `before` is used to extend the current deadline. In Figure 2.11, a Timber object with a periodic method is shown. The period and relative deadline is 50μs.

```plaintext
perobj = class
    ...
result {
    periodictask = before 50*us action
    ...
    after 50*us periodictask
}
```

*Figure 2.11: Simple example of a periodic task in Timber.*
2.3.5 The Timber run-time kernel

In order to meet the semantic specification of the language, a Timber program will depend on the infra-structure provided by the run-time kernel of the language. Conceptually, the kernel needs to offer the following services:

- Create an object,
- Send a synchronous message, and
- Send an asynchronous message (possibly delayed).

In reality, creating an object maps to allocation of heap storage. The only difference between allocating an object and an immutable data structure (record, tuple, cons cell, etc.) is that in order to enable mutual exclusion between methods of the same object, each object needs to have a lock field. A synchronous rendezvous is simply accomplished by locking the object before running the code of the method, and releasing the lock afterwards. An asynchronous method call on the other requires a new message (conceptually an execution thread) to be allocated and enqueued in the proper message queue (timer queue if delayed). Once this message gets to run, it simply makes a regular synchronous call.

The scheduling mechanism in the kernel is a preemptive priority scheduler based on deadlines. Delayed messages are scheduled based on baselines. The scheduler interacts with three types of data; objects (includes locks), messages (including executable code, possibly parameters, a baseline, and a deadline), and threads (messages extended with a stack and register file). Objects are the shared resources for which messages acquire and release locks. Asynchronous messages that are latent (either delayed or pending) are, when scheduled to run, promoted to a unique thread. The implicit concurrency of a Timber program is thus accomplished by the scheduler which, for each independent schedulable message, may allocate a new unique thread of execution.

The interface of the scheduler consists of four entry points.
2.4 Timed automata

1. Whenever a new asynchronous message is posted (either internally by another method or externally by an interrupt).

2. When a method calls lock or unlock for an object.

3. When a method terminates.

4. When a timer interrupt occurs.

The scheduler manages three main data structures: a stack of active threads (including each thread’s individual stored machine state), a priority queue of pending messages, and a priority queue of delayed messages.

There are a few very important properties of the kernel that are worth mentioning. All lock operations eventually return, either with the acquired lock or with a deadlock error. The lock of an object may only be owned by one message at a time. The highest priority thread, or the thread holding a lock wanted by the highest priority thread, will always be the one scheduled to run. All messages will run after their baselines. The aliveness property of a Timber program is crucially dependent upon the fact that Timber code always will follow a successful lock with an unlock and that all locks are acquired in a nested manner.

2.4 Timed automata

Timed automata is a formal notation to model the behavior of real-time systems. Through the concept of model checking it has proved to be a powerful model for reasoning about real-time systems, especially reachability analysis of system state. At its core, it is a rather simple way to annotate state transition systems with timing constraints through the use of real-valued clock variables. This section is a background study of the theory of timed automata based on [4, 5, 6, 7, 16, 28].
2.4.1 State transition system

A state transition system is an abstract machine consisting of a set of states and transitions (possibly labeled) between states. Formally, a state transition system is defined by a tuple \( \langle Q, Q_0, \Sigma, \rightarrow \rangle \) with the following meaning.

- \( Q \): Set of states
- \( Q_0 \): Set of initial states \( (Q_0 \subseteq Q) \)
- \( \Sigma \): Set of labels
- \( \rightarrow \): Set of transitions \( (\rightarrow \subseteq Q \times \Sigma \times Q) \)

A system starts in one of its initial states. If the current state is \( q \), the system may accept the label \( \sigma \) by changing its state to \( q' \) if and only if \( (q, \sigma, q') \in \rightarrow \) (typically, we write \( q \xrightarrow{\sigma} q' \)).

2.4.2 Timing constraints

Extending the state transition system to cope with timing constraints the system is enriched with a finite set of real-valued clocks. For timed automata, states are often referred to as locations. Time can elapse in a location, whereas transitions are instantaneous. An important property of the clocks is their unanimous pace in time. Clocks can be reset by transitions and the value of a clock at a particular instant is semantically defined by the elapsed time since its last reset. Each transition is guarded by a clock constraint, implying the requirement that a transition can only be taken if the current clock values satisfies the corresponding constraint. Similarly, each location is associated with a clock constraint (invariant), implying that time may elapse in a location as long as the clock values satisfies its corresponding invariant.

An example of a timed automata is shown in Figure 2.12. It is a model of a sonar that sends short sonar pulses (pings) in a steady pace. After a ping has been sent, it either receives an echo or timeout after the specified time. It uses two clocks \( x \) and \( y \) to ensure the timing properties of the sonar. In the initial location (start), a ping is started when \( x \) is exactly 1000. This is ensured by the combination of the invariant of
start and the guard on ping. The ping transition will reset both x and y to 0 and enter location on. The combination of the invariant of on and the guard on stop ensures that the stop transition will be taken when y is exactly 2. stop resets y to 0 and enters location off. In off, there are two possible transitions, either echo? or timeout. The invariant of off and the guard of timeout ensures that timeout can only be taken when y is 500. That also implies that echo can only be taken when y is at most 500.

![Timed automaton with two clocks](image)

Figure 2.12: Sonar example. A timed automaton with two clocks.

Clock constraints

Given a set of real-valued clock variables C, a clock constraint is a conjunction of atomic constraints of the form

\[ x \sim n \quad \text{or} \quad x - y \sim n \]

where \( x, y \in C \) \( \sim \in \{\leq, <, =, >, \geq\} \) \( \land n \in \mathbb{N} \).

Let \( B(C) \) denote the set of clock constraints.

2.4.3 Syntax and semantics

A timed automaton is defined by a tuple \( \langle L, l_0, \Sigma, C, I, E \rangle \) with the following meaning.
$L$ : Finite set of locations

$l_0$ : The initial location ($l_0 \in L$)

$\Sigma$ : Finite set of labels

$C$ : Finite set of clock variables

$I$ : Location invariants. Mapping locations to clock constraints ($I \subseteq L \times B(C)$)

$E$ : Set of transitions ($E \subseteq L \times B(C) \times \Sigma \times 2^C \times L$)

The operational semantics of a timed automaton $A = \langle L, l_0, \Sigma, C, I, E \rangle$ is described by the transition system $SA = \langle QA, \{\langle l_0, u_0 \rangle\}, \Sigma \cup R_+ \cup \rightarrow \rangle$ where each state is a pair ($\langle l, u \rangle$) of a location $l \in L$ and a clock valuation $u : C \mapsto \mathbb{R}_+$ mapping the clock variables to non-negative real numbers. $u_0$ is used to denote the function mapping all clock variables to 0. Given a clock valuation $u$ and a $\delta \in \mathbb{R}_+$, we write $u \oplus \delta$ to denote the clock valuation function $\{(x, u(x) + \delta) | x \in C\}$. The set of states $QA$ is defined as follows.

$$QA = \{ \langle l, u \rangle | l \in L \land u : C \mapsto \mathbb{R}_+ \land u \text{ satisfies } I(l) \}$$

Observe that since $\mathbb{R}_+$ is infinite, the set $QA$ is also infinite. The set of transitions $\rightarrow \subseteq QA \times \Sigma \cup R_+ \times QA$ is divided into two types, one capturing the corresponding delay of the timed automaton and the other one capturing change of location. Given a set of clock variables $\rho \subseteq C$ and a clock valuation $u$ we write $\rho(u)$ to denote the set $\{(x, 0) | x \in \rho\} \cup \{(x, v) | (x, v) \in u, x / \in \rho\}$. The transitions are defined as follows.

$$\langle l, u \rangle \xrightarrow{\delta} \langle l, u' \rangle \quad \text{iff} \quad \forall \delta' \leq \delta \in \mathbb{R}_+. \langle l, u \oplus \delta' \rangle \in QA$$

$$\langle l, u \rangle \xrightarrow{\varphi} \langle l', u' \rangle \quad \text{iff} \quad \exists (l, \varphi, \sigma, \rho, l') \in E. \langle u' = \rho(u) \land u \text{ satisfies } \varphi \rangle$$

A run of a timed automaton $A$ is captured by a sequence of transitions of $SA$ of the form:

$$\langle l_0, u_0 \rangle \xrightarrow{\delta_1 \sigma_1} \langle l_1, u_1 \rangle \xrightarrow{\delta_2 \sigma_2} \langle l_2, u_2 \rangle \xrightarrow{\delta_3 \sigma_3} \langle l_3, u_3 \rangle \ldots$$

which can be recorded by a sequence of timed-stamped labels $\xi = (t_1, \sigma_1), (t_2, \sigma_2), \ldots$ where $t_i = t_{i-1} + \delta_i$ for all $i > 1$ and $t_1 = \delta_1$. The timed language $L(A)$ is the set of all such $\xi$s of $A$. The time-stamps may be abstracted away forming an untimed language $untime(L(A))$ with the following definition.

$$\sigma_1, \sigma_2, \sigma_3, \ldots \in untime(L(A)) \text{ iff } \exists t_1, t_2, t_3, \ldots \text{ s.t. } (t_1, \sigma_1), (t_2, \sigma_2), (t_3, \sigma_3), \ldots \in L(A)$$
2.4. Timed automata

Let $U_A$ be the untimed (or time-abstract) transition system defined by hiding the $\delta$-transitions of $S_A$. Formally, $U_A = \langle Q_A, \{(l_0, u_0)\}, \Sigma, \Rightarrow \rangle$ where $\Rightarrow$ is defined as follows.

$$ q \xrightarrow{\delta} q' \quad \text{iff} \quad \exists q'' \in Q_A, \delta \in \mathbb{R}_+ . \ q \xrightarrow{\delta} q'' \xrightarrow{\sigma} q' $$

2.4.4 Composition of timed automata

Let $A = \langle L, l_0, \Sigma, C, I, E \rangle$, $A' = \langle L', l'_0, \Sigma', C', I', E' \rangle$, and $C \cap C' = \emptyset$, then $A \parallel A'$ is the composition of $A$ and $A'$ with the following resulting automaton.

$$ A \parallel A' = \langle L \times L', (l_0, l'_0), \Sigma \cup \Sigma', C \cup C', I'' \parallel E'' \rangle $$

For $l \in L$ and $l' \in L'$, $I''(l, l') = I(l) \land I'(l')$. The set of transitions $E''$ is defined as follows.

$$ \forall \sigma \in \Sigma \cap \Sigma' .$$

$$ \forall ((l_1, \varphi, \sigma, \rho, l_2), (l'_1, \varphi', \sigma', \rho', l'_2)) \in E' . ((l_1, l'_1), \varphi \land \varphi', \rho \cup \rho', (l_2, l'_2)) \in E'' $$

$$ \land $$

$$ \forall \sigma \in \Sigma \setminus \Sigma' .$$

$$ \forall ((l_1, \varphi, \sigma, \rho, l_2), (l', l'')) \in E, l' \in L' . ((l_1, l'), \varphi, \sigma, \rho, (l_2, l'')) \in E'' $$

$$ \land $$

$$ \forall \sigma' \in \Sigma' \setminus \Sigma .$$

$$ \forall ((l'_1, \varphi', \sigma', \rho', l'_2), (l, l')) \in E', l \in L . ((l'_1, l'), \varphi', \sigma', \rho', (l, l'')) \in E'' $$

2.4.5 Reachability

Given an automaton $A = \langle L, l_0, \Sigma, C, I, E \rangle$, a location $l \in L$ is reachable in $A$ if and only if there exists a clock valuation $u$ such that $(l, u)$ is reachable in $S_A$ (i.e., $(l_0, u_0) \Rightarrow (l, u)$).

For a clock constraint $\varphi \in B(C)$ a configuration $(l, \varphi)$ is reachable in $A$ if and only if there exists a clock valuation $u$ such that $(l_0, u_0) \Rightarrow (l, \varphi)$ and $u$ satisfies $\varphi$.

Region equivalence

For any $\delta \in \mathbb{R}_+$, let $[\delta]$ denote the integral part of $\delta$, and $fr(\delta)$ denote the fractional part. Let $k_x \in \mathbb{N}$ denote the largest constant appearing in the guards and invariants of
A constraining the clock \( x \in \mathcal{C} \). Region equivalence is an equivalence relation between clock valuations denoted by \( \cong \) and it is defined as follows. 
\[
u \cong u' \iff \forall x, y \in \mathcal{C}. \\
[u(x)] = [u'(x)] \lor (u(x) > k_x \land u'(x) > k_y) \\
\land \\
u(x) \leq k_x \land u(y) \leq k_y \land fr(u(x)) \leq fr(u(y)) \iff fr(u'(x)) \leq fr(u'(y)) \\
\land \\
u(x) \leq k_x \land fr(u(x)) = 0 \iff fr(u'(x)) = 0
\]

A clock region is an equivalence class of clock valuations induced by \( \cong \). The region equivalence is straightforwardly extended to an equivalence relation over \( Q_A \) forming equivalence classes called regions. The definition is as follows. 
\[
(l, u) \cong (l', u') \iff l = l' \land u \cong u'
\]

The key property of \( \cong \) is the strong bisimilarity (sometimes referred to as stability), i.e. if \( q \cong p \) and \( q \xrightarrow{a} q' \) then there exists a \( p' \) such that \( p \xrightarrow{a} p' \) and \( q' \cong p' \), and vice versa.

**Zone automata**

A clock region is either a corner point (intersection), line segment, or area. An efficient way to represent the state space is achieved by collapsing regions, considering convex unions of clock regions. A clock zone \( D \) is a convex set of clock valuations. Three key operations on clock zones are defined as follows.
\[
\text{Delay : } D^\tau = \bigcup_{\delta \in \mathbb{R}_+} \bigcup_{u \in D} \{(x, v + \delta) \mid (x, v) \in u\} \\
\text{Reset : } \rho(D) = \bigcup_{u \in D} \{(x, 0) \mid x \in \rho\} \cup \{(x, v) \mid (x, v) \in u, x \notin \rho\} \\
\text{Normalization : } [D] = \{u \mid u \cong_g u', u' \in D\}
\]

Clock zones are closed under all three operations above, i.e. the resulting sets of the operations are also clock zones. We will write \( D_\varphi \) to denote the clock zone defined by a clock constraint \( \varphi \) containing lower and upper bounds on clocks and differences of two clocks. We will implicitly assume that \( \varphi \) is in canonical form, meaning that no atomic
constraint in $\varphi$ can be strengthened without reducing the corresponding solution set (i.e., $\varphi$ is closed under entailment). Normalization relies on the extended equivalence relation $\equiv_G$ between clock valuations with the following definition ($\mathcal{G}$ is the set of all guards and invariants in $\mathcal{A}$).

$$u \equiv_G u' \quad \text{if } u \equiv u' \land \forall \varphi \in \mathcal{G}. \ u \text{ satisfies } \varphi \iff u' \text{ satisfies } \varphi$$

Clock zones extend to zones (forming pairs $\langle l, D_\varphi \rangle$) when the class is extended to an equivalence relation over $Q_A$. For a timed automaton $\mathcal{A} = \langle L, l_0, \Sigma, C, I, E \rangle$, the transitions (denoted by $\rightsquigarrow$) of the corresponding zone automaton $Z_A$ are defined as follows ($\langle l, D_\varphi \rangle \rightsquigarrow \langle l', D_\psi \rangle \implies D_\psi \neq \emptyset$ is implicit).

$$\langle l, D_\varphi \rangle \rightsquigarrow \langle l, [D_\varphi \cap D_{I(l_0)}] \rangle$$

$$\langle l, D_\varphi \rangle \rightsquigarrow \langle l', [\rho(D_\varphi \cap D_\psi) \cap D_{I(l')} \rangle \quad \text{if } (l, \psi, \sigma, \rho, l') \in E$$

$Z_A$ is a correct and full characterization of $\mathcal{S}_A$.

### 2.4.6 Implementation

The formal operational semantics of timed automata is expressed in terms of infinite sets of clock valuations and operations on such sets. In practice, representing such sets is, as already hinted, possible with a finite set of clock constraints (given that $C$ is finite). However, it is not obvious how the corresponding operations should be carried out on the constraints, preserving the properties of a clock zone.

#### Clock zones as difference bound matrices

A clock zone can be represented by a difference bound matrix (DBM) if the set of clocks $C$ is extended with a reference clock 0 with the constant value 0. Let $C_0 = C \cup \{0\}$. Then, any clock zone $D_\varphi$ such that $\varphi \in \mathcal{B}(C)$ can be captured by a conjunction of atomic constraints of the restricted form

$$x - y \preceq n \quad \text{where } x, y \in C_0, \ \preceq \in \{<, \leq\}, \text{ and } n \in \mathbb{Z}.$$
Let \( C_0 = \{0, x_1, \ldots, x_m\} \), then a zone can be represented by a \((m+1) \times (m+1)\) DBM \( D \). The element \((i, j)\) of \( D \) is denoted by \( D_{ij} \in \mathbb{Z} \times \{<, \leq\} \). Given a clock zone \( \mathcal{D}_\varphi \) where \( \varphi \) is in the restricted form, a corresponding DBM \( D \) can be computed as follows.

1. For each \( x_i - x_j \preceq n \) of \( \varphi \) let \( D_{ij} = (n, \leq) \)
2. For each \( x_i - x_j \) that is not bounded in \( \varphi \) let \( D_{ij} = (\infty, <) \)
3. For each \( x_i \in C \) let \( D_{ii} = (0, \leq) \) and \( D_{0i} = (0, \leq) \)

Canonical DBMs  In a canonical DBM of a clock zone, each clock difference \( D_{ij} \) must be as tight as possible (i.e., without reducing the set of solutions). The key observation needed to compute this is that the sum of the bounds of \( D_{ij} \) and \( D_{jk} \) is also a bound for \( D_{ik} \). In other words, in order to make a DBM zone canonical, the bound of each element \( D_{ik} \) must be compared to all sums of elements \( D_{ij} \) and \( D_{jk} \) and updated when such a sum is smaller than the current bound. It is clearly beneficial if the operations needed to construct the zone graph (intersection, delay, etc.) preserve the zones in canonical form.

Intersection  The intersection of two clock zones \( \mathcal{D}_\varphi \cap \mathcal{D}_\psi \) can be captured by the conjunction \( \varphi \land \psi \) (which preserves the convex form). However, the resulting constraint is not in canonical form.

Delay  The delay operation on a DBM zone is computed by removing the individual upper-bounds on all clocks. That means for all clocks \( x_i \in C \) the corresponding bound \( D_{ii} \) is set to \((\infty, <)\). The delay operation preserves the DBM zone in canonical form.

Reset  Resetting a specific clock in a DBM zone can be done by recomputing all difference bounds on that clock with respect to all other clocks. The reset operation preserves the DBM zone in canonical form.

Normalization  If the timed automaton contains no difference constraints the normalization of a DBM zone can be done by first removing all bounds \( x - y \preceq m \) such
that $(m, \preceq) < (k_x, \preceq)$ and then replace all bounds $x - y \preceq m$ with $x - y < -k_y$ if $(m, \preceq) < (-k_y, <)$. In the case where difference constraints may appear in the timed automaton, it is more complicated to perform the normalization operation. We refer to [16] for further details on this. The normalization operation does not preserve the zone in canonical form.
Chapter 3

Garbage Collection

The basic principle of automatic memory reclamation, or garbage collection, is the process of automatically reclaiming memory space holding dynamic data that will not be used by the program anymore. The reclaimed memory space is instead made available for future use by the program. In terms of correctness, the garbage collector must not reclaim memory that will be used, nor should it fail to reclaim memory that will not be used. Non-free memory space that will not be used in the future is referred to as garbage. When and how garbage is collected depends on the actual algorithm used. We will survey some of the basic techniques used for garbage collection and then look at what makes a garbage collector work for real-time systems.

There exist (at least) two different approaches to garbage collection. On one hand we have the reference counting garbage collectors, which means that every dynamically allocated node on the heap holds a counter that keeps track of how many references there exists to that particular node. In other words, at every destructive update and new allocation (with initialization) of heap nodes, the reference counters of the new (and old) immediate descendants must be updated. In addition, references from outside the heap (i.e. from run-time stacks, CPU registers, static/global references, etc.) must also be accounted for. The two biggest disadvantages of this approach are, firstly, the amount of overhead due to incrementing/decrementing reference counters tends to be large for many applications, and secondly, storing aliveness information locally in each node fails to collect cyclic garbage structures. The two biggest advantages of reference counting
are that it is quite straightforward to implement and it is naturally incremental (i.e. the work can "easily" be spread out during the execution of the program without inferring too long pause times).

The second approach takes a global perspective on the aliveness property. It is based on a set of root references (run-time stack, CPU registers, static/global references, etc.), from which all live nodes on the heap are reachable. The reachability property is an over-approximation of the aliveness property, since a live node is always reachable, while a reachable node might be garbage due to the fact that the program might never use it anymore. Anyhow, the approach is based on traversing the graph of references reachable from the roots, distinguishing the reachable data from the garbage. This approach is called tracing garbage collection. How garbage is collected after the reachable nodes have been found depends on which tracing technique that is used. We will look into a few of these techniques in the following sections.

For readers interested in a more comprehensive description of different garbage collection algorithms and their intricate details are recommended to consult the book *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*, by Richard Jones and Rafael Lins [44]; as well as the garbage collection survey by Paul R. Wilson [89].

### 3.1 The mark-sweep garbage collector

One of the earliest tracing garbage collectors is the *mark and sweep* garbage collector [61]. The basic idea of the algorithm is to mark all live (reachable) nodes on the heap by traversing the graph of references beginning in the roots. Then the heap is swept and everything that is unmarked is reclaimed. The advantages of this approach in comparison to reference counting are that it can detect and collect cyclic structures without any special precautions and it does not cause any overhead upon pointer manipulation. The downside is that it is naturally non-incremental; i.e., in order to collect any garbage, it has to run uninterruptedly for a relatively long time.
3.1.1 The marking stack

The marking phase of mark-sweep collector is naturally recursive. In order to improve performance, mark-sweep collectors commonly utilize an auxiliary marking stack that holds pointers to nodes that are known to be alive but have not been visited yet. Using a stack for marking naturally leads to a depth-first traversal. The procedure can be described as follows. At the beginning of a garbage collection cycle, the marking stack holds only the root pointer (i.e. it is marked). The procedure then repeatedly pops off the pointer at the top the stack, scans the node for unmarked children and pushes them onto the stack. When the stack is empty, the marking phase is finished.

One issue that is commonly encountered when implementing the marking stack is that of minimizing the depth of the stack to reduce the risk of overflow. As an example of stack depth minimizing actions, we can mention the Boehm-Demers-Weiser mark-sweeping conservative garbage collector for C and C++ [21]. Their marking stack holds pairs of pointers, pointing at the start and the end of the object. In order to reduce the risk of overflow, they examine the object on the top of the stack before they push its descendants onto it. If the object is small (less than 128 words), all its descendants are pushed onto the stack. If the object is big (more than 128 words) only descendants within the first 128 words are pushed onto the stack (marked) and the stack entry is updated so as to correspond to what is left to examine. In order to handle, the less likely, but still possible, stack overflows, nodes are simply not pushed onto the stack whenever the stack is about to overflow (nodes are simply dropped). When the stack eventually becomes empty, the old stack is replaced by a new one of twice the size of the old one. The collector then scans the heap for marked objects with unmarked children and continues the marking phase from there.

3.1.2 Pointer reversal

In this section we will look at a technique for storing marking stack information in-place, while traversing the graph of reachable objects on the heap. The technique of pointer
Garbage Collection

reversal was independently developed by Schorr and Waite [75] and Deutsch [54]. The idea is to, while the marking process traverses down the graph, reverse the pointers it follows, i.e. the address of the parent is stored in one of the pointer fields of the object that is currently under examination. As the process retreats after visiting a branch, the pointer fields are restored to their original contents. Thus, the information used for marking is stored in-place rather than in a possibly unbounded marking stack. This was first developed for binary-branched objects [75, 54], but later also extended to variable sized objects, e.g. by Thorelli [84].

The biggest disadvantage of pointer reversal in terms of efficiency is that objects need to be visited many times in order to traverse the graph (at least $n + 1$ times, where $n$ is the number of pointers in the object). On systems with more complex memory architectures, this might be extremely undesirable due to possibly increased number of cache misses and page faults. Due to this, pointer reversal is suggested to be used only as a last resort, e.g., after a detected stack overflow [75]. Still, some systems rely on this technique as their primary strategy [85, 92].

3.1.3 Bitmap marking

In order to determine if an object is marked, the most intuitive solution is to store a mark bit in a header field of the object. For many systems, this is not a problem. For example, some systems store type information in a header field, and the mark bit can probably be squeezed in at some free position in this field without imposing larger overhead. However, for some systems this free space might be impossible to find. A solution is to store the mark bits in a separate bitmap table. The size of the table is inversely proportional to the size of the smallest object that can be allocated on the heap. In order to make it more sophisticated, separate bitmaps can be used for each different kind of object. Accesses can then typically be done via a hash table or search tree. The Boehm-Demers-Weiser collector is an example of where these separate bitmaps are used [21]. In addition, bitmaps can conveniently also be used for the sweep phase, as we will see in the next subsection.
The disadvantage of using bitmaps is that mapping an address to a mark bit in the table is generally more expensive than if the information would be stored in the object. Some results indicate that it can cost as much as twelve times as much [96].

3.1.4 The sweep phase

Commonly, the main case made against mark-sweep garbage collection is that it has to sweep the whole heap in order to complete a garbage collection cycle. However, this can be done in parallel with the mutator execution since mark bits, as well as garbage (unmarked) objects, are inaccessible for the mutator. The most common way to interleave the mutator and the sweeper is to do a certain amount of sweeping at each allocation (e.g. [39]). In addition, this approach does not require any explicit free-list manipulations since garbage objects are recycled directly to the mutator instead of via a free-list buffer. However, if a bitmap is used, it is more efficient to sweep at least a few words of the bitmap at the same time in order to avoid the need of having to reload and save the bitmap (indexes and bit-masks) at every allocation.

3.2 Fragmentation

Both reference counting and mark sweep garbage collectors suffer from fragmenting the heap. This is caused when objects of different sizes are repeatedly allocated and reclaimed. Fragmentation can be a serious problem due to the fact that the free space is not contiguous, and a big enough allocation request may fail even though the total amount of free memory is sufficient. There is, at least, a couple of ways to deal with this problem.

3.2.1 Fixed block size(s)

In order to handle the fragmentation problem the memory can be divided into a set of blocks of fixed size(s). The simplest and easiest way is to use one block size, which is determined by the biggest possible allocation request the application may make. This
will of course cause a lot of overhead if the allocation needs of the application vary a lot in size. There are other, more sophisticated, ways to do this, where differently sized blocks may be used. The ideas are based on splitting and coalescing memory blocks for a more accurate match between the block sizes and the actual requests of storage space. As examples we have segregated free lists [27], and buddy systems [53, 70]. However, even though these systems make the worst case scenario more unlikely, they do not prohibit it.

Commonly, the problem of fragmentation is divided into two kinds: internal and external fragmentation. External fragmentation is what we normally refer to as just fragmentation; i.e., the fragments of free memory in between objects on the heap. The term internal fragmentation refers to the overhead induced by using fixed block sizes, where some objects might not fit exactly within one block and thus causing more memory to be allocated than needed. An example that segregates objects of different sizes is the two-level allocator of Boehm-Demers-Weiser collector [21]. The low level allocator allocates big chunks of memory (typically four kilobytes) through a standard operating system service (e.g. `malloc`). Each one of these bigger chunks contains only objects of the same size, and a free-list for each object size is maintained. If the free-list is empty for a certain size when a allocation request is made (and the garbage collector is unsuccessful in reclaiming any objects of that particular size), a new chunk for the particular object size is allocated at the lower level, i.e. the free-list for a certain object size may be threaded through several bigger chunks. If a chunk is detected as empty (which is cheaply detected in the Boehm-Demers-Weiser collector) it can be returned to the operating system (e.g. via `free`).

In the worst case scenario, the segregated free-lists of the Boehm-Demers-Weiser collector may cause extreme memory overheads if objects of the same size with long lifetimes are spread out on different chunks. In addition, when new allocations of that size are made, objects with possibly very different lifetime behavior may reside in the same chunk, causing the chunk to never become empty. This, in turn, means that the total amount of allocated memory (at the low level) may be several times more than the
3.3. The mark-compact garbage collector

actual need. However, in the common case, objects that are active at the same time tend
to share the same lifetime behavior, i.e. they are allocated closely, both temporarily and
spatially, and also likely to be reclaimed at the same time [34].

3.2.2 Defragmentation

The more robust, but harder, way to deal with fragmentation is to actually defragment
the memory by means of moving and compacting live data into a contiguous memory
space, which ultimately leads to a contiguous free space. This is of course a more di-
rect way to deal with fragmentation but it comes with a huge load of complications.
Defragmenting a big memory may take a long time and allowing the application to run
during this process requires a very precautionary synchronization scheme. In addition,
uncooperative systems may not function correctly if objects are moved (e.g. systems im-
plemented in C). For these systems, garbage collectors that move live data are simply not
an option. In the next sections, we will look at two different approaches to defragmenting
the heap.

3.3 The mark-compact garbage collector

The mark-compact garbage collector commonly traverses the live data on the heap many
times. The process can be described as consisting of three phases; firstly the live data is
marked (similarly to the mark-sweep collector), secondly the graph is compacted by relo-
cating the objects, and thirdly pointers to relocated cells are updated. The way objects
are ordered in the resulting heap may differ depending on the actual mark-compact algo-
rithm that is used. There are two ways (three if one counts arbitrary ordering) in which
objects commonly are ordered upon compaction. First we have linearizing compaction,
which orders objects according to the order they appear in the graph (e.g. depth first or
breadth first). The second way to order objects is based on the order in which they were
allocated; i.e., the live objects are slid together. We will look at two different techniques
that are used by mark-compact garbage collectors.
3.3.1 The two-finger algorithms

The first technique we will look at is the two-finger algorithm [18]. It is based on two pointers: one of them points to the next free spot in memory (free) and the second to the next live object to be moved (live). They start at each end of the heap and work their way towards each other. free sweeps the heap for free spots while live sweeps from the other end for live objects. Objects discovered by live are moved to the empty spots found by free. A forwarding pointer is written in the old location of the object. The procedure is finished when the two pointers meet. Then the heap is swept again to update pointers that are pointing to old copies.

The downside with this technique is that a fixed block size is required. However, similarly to the Boehm-Demers-Weiser allocator, objects of different sizes can be segregated into different regions of the heap, and compacted separately.

3.3.2 The forwarding address algorithms

The forwarding address technique reverses the order in which objects are moved and pointers are updated. After the mark phase, new locations of all live objects are calculated and stored in a header field of each object. Then pointers are updated to the new calculated address and, finally, live objects are slid together.

The greatest advantage of this technique is that it can handle differently sized objects with no restriction. On the downside it has to make three passes over the heap, although the amount of work at each pass is quite small. In addition, it requires every object to have an extra field for the forwarding address, although it can be combined with the mark bit.

3.4 The copying garbage collector

The main idea behind the copying garbage collector is to, while traversing the graph, copy reachable objects into a large enough contiguous free space. When all reachable
3.4. The Copying Garbage Collector

objects have been copied, the old space can then be freed altogether. The resulting heap is both defragmented and garbage collected. The basic steps can be described as follows. The heap is split in two parts; tospace is the active one and fromspace the inactive one (see example in Figure 3.1). The first thing the copying garbage collector does is to flip the labels so fromspace becomes tospace and vice versa. The objects pointed to by the root-set is then copied and forwarding pointers are installed in their old copies (Figure 3.2 and 3.3). While traversing the graph, if an object is found in fromspace and it is already copied (determined by the existence of a forwarding pointer) the pointer through which it was found is updated with the value of the forwarding address (Figure 3.4). Otherwise the object is copied (Figure 3.5). When the collector has finished traversing the graph the garbage collection cycle is done. The resulting heap is a defragmented and garbage collected version of the previous one (3.6).

![Figure 3.1: Example of a heap set up before a copying garbage collection cycle.](image)

The first copying collector, presented in 1963, was the Minsky collector for LISP [64].
Figure 3.2: Begin by copying the first root.

Figure 3.3: Continue with second one.
3.4. The copying garbage collector

Figure 3.4: Update a pointer with the new (forwarding) address.

Figure 3.5: Copy encountered objects to tospace.
When the garbage collection cycle is finished, the resulting heap is a defragmented and garbage collected version of the previous one.

It used secondary storage, i.e. a file on disk, as target space for copying. Today this approach would not be very efficient, due to the several orders of magnitude slower file operations in comparison to memory operations. Fenichel’s and Yochelson’s [32] collector divides the heap into two semispaces where only one of them is active at the same time.

A collection cycle is performed by copying all live blocks from the active space (fromspace) into the inactive space (tospace). Then the labels are switched, i.e. the previously active space becomes inactive and vice versa. To avoid copying the same block more than once, a forwarding pointer is installed in the old copy.

3.4.1 Cheney’s copying garbage collector

In [25], Cheney presents a non-recursive copying traversal algorithm using a scavenging scan pointer. It utilizes an in-place breadth first traversal in contrast to the recursive approach of Fenichel’s and Yochelson’s collector. The basic idea is to scan the copied
blocks for references into fromspace, and if such a reference is encountered, copy it into tospace (see Figure 3.7). The process can be described as follows. The algorithm uses two pointers (scan and top). Initially, both of them point to the beginning of tospace. The algorithm then copies the roots to tospace, which moves top accordingly. These objects in between scan and top are then scanned (from left to right) for pointers into fromspace. When (if) such a pointer is found, the object is copied into tospace (pushing top). Thus, while scan chases top it will push top forward as long as there are more objects to copy. The algorithm is finished when scan catches up with top (i.e., the whole graph has been traversed and copied).

The biggest advantage of the Cheney copying collector is its simplicity. No mark bitmaps or free-lists need to be maintained, and allocation is simply done by pushing a pointer (top). Only live data has to be visited (i.e. no sweeping the heap). This is especially good for systems that produce a lot of garbage (e.g. functional programs).

The main drawback of a copying garbage collector is that it requires twice as much memory than the program actually needs in order to be able to copy all live objects from fromspace into tospace.

In [8], Appel argues that copying garbage collection can be made arbitrary efficient if sufficient space is available. Since the heap is a push-only stack, Appel argues that heap allocation can be made faster than stack allocation. However, this requires the heap size to be several times larger than the average volume of live data.

In addition, the spatial locality issue is still a matter of discussion. However, in contrast to most other tracing garbage collectors, Cheney’s copying collector arranges the graph of live objects in a breadth-first manner, in contrast to the forwarding address mark-compact algorithms (Section 3.3.2), which arrange objects in the same order they were allocated.

In [65], Moon presents a modified version of Cheney’s collector, which is approximately depth-first. Instead of always moving scan towards top, Moon uses a third pointer, scan_partial, which scans the latest virtual memory page [65] (see Figure 3.8).

The downside of Moon’s algorithm is that objects will be scanned twice. However,
Figure 3.7: The Cheney algorithm’s breadth-first traversal
it is argued that scanning an object one extra time is cheap, since it cannot cause any more copying.

Wilson et al. remove the re-scanning of Moon’s algorithm by modifying it into a two-level version of Cheney’s original breadth-first traversal [90]. In addition to the two pointers, scan and top, each page has its own local two pointers. The major scan pointer points at the first incompletely scanned page in tospace. The minor scan pointer of that page scans the page until it reaches the end (or top) of it, then the major scan pointer is moved to the next page. If the minor scan pointer of the first incompletely scanned page causes an object to be copied it triggers a local scan of that page and continues until it is finished. Then the process continues from where it was before. Already scanned pages of tospace is easily detected and skipped by the major scan pointer, since each page has its own local scan pointer. That is, if the local scan pointer is at the end of the page it is already scanned.

### 3.5 Generational Garbage Collection

A significant drawback of a copying collector scheme is the possibility of copying long-lived blocks back and forth between the spaces when only short-lived blocks are collected. To remedy this problem one could use more than two spaces and separate the blocks
between the spaces by their age. Due to the distinction in age, these spaces are commonly referred to as generations. Examples of early copying collectors using generations are the Lieberman and Hewitt collector [57] and the Ungar collector [86]. The main difficulty in designing a copying collector based on generations, which was identified even in the early days, is the detection and treatment of inter-generation pointers. To avoid traversing a generation that is not subject for garbage collection, these pointers need to be stored as part of the root set. For example, the root set for garbage collecting generation A has to include all pointers from generation B to generation A. If these pointers are not known a priori, generation B must be traversed in order to find them.

3.5.1 Promotion schemes

The basic idea of a promotion scheme is to determine when objects should be promoted from a younger generation to an older one. Promoting objects too soon will cause the older generation to be filled with garbage and increase the need for garbage collecting it in a frequency close to that of the younger one. On the other hand, promoting objects too late causes the generational collector to behave like a regular one. For example, for a copying collector, long-lived objects would be copied back and forward several times.

The common way to record age is to count how many times an object has been copied. The question then is; how many times does an object need to be copied before it is considered old enough to be promoted? Ungar claims that increasing the threshold beyond two will likely only reduce the number of surviving objects very little [86]. In addition, according to Wilson and Moher [91], one commonly needs to increase the threshold by a factor of four or more in order to reduce the number of survivors by fifty percent.

Counting the number of times an object has been copied is not a pure measure of age. If the size of the younger generation is small, the number of times an object has been copied will result in a younger age than if the size is big. One of the driving forces behind generational (non-incremental though) garbage collection is that smaller pieces of the heap are collected, which in turn means shorter pause times. In addition, for generational copying garbage collectors, pause times are reduced even more since younger generations
3.5. Generational Garbage Collection

(with lower survivor rates) are garbage collected more often. Anyhow, increasing the size of the young generation will not only let objects grow older, but also induce longer pause times. Tuning this is a difficult and complex task, which for example is shown in [40].

A downside with generational garbage collectors is that age of objects needs to be recorded somehow. In [78], Shaw suggests that each generation should be divided into regions, called buckets. The young generation is divided into newspace and agingspace. New allocations are made in newspace. Every \( n \)th garbage collection cycle, objects in newspace are moved to agingspace and objects in agingspace are promoted to the next generation. Thus, no age is needed to be recorded in the object, instead the age is detected by where in the generation it resides.

Lieberman and Hewitt [57] realized that if the expected lifetime of an object is known when it is allocated, generational garbage collection would be much more efficient. The necessary bookkeeping in order to enable objects to grow old, as well as the work needed to promote an object to an older generation, both carry significant overhead.

3.5.2 Inter-generational pointers

Since generational garbage collectors only garbage collect parts of the heap each time (that is after all the main idea of it), pointers into the current part from other parts of the heap must be accounted for, i.e. some objects in the part that we garbage collect might only be alive through pointers from outside that part. We refer to this kind of pointers as inter-generational pointers. Inter-generational pointers have to be accurately recorded and maintained in order to garbage-collect each generation correctly. The simplest and probably most naive way would be to traverse (or even linearly scan) other generations at collection time. However, that would remove the basic idea of generations, which, all in all, aims at reducing the amount of objects to touch. Even though this technique has been shown to improve performance in comparison to the basic two-semispace copying collector [83, 78], there are more precise and delicate ways to deal with this kind of pointers. We will, for the sake of simplicity, from now on only consider pointers from older to younger generations, since most generational garbage collectors either collect
only the younger generation (which happens often) or collect the whole heap (which happens more seldom). Thus, pointers from younger to older generations do not cause any problems.

### 3.5.3 The write-barrier

There are two ways inter-generational pointers are created; either through a destructive update of a pointer field in an object in the older generation, or the pointers simply reside in a young object that is promoted to an older generation. The second case is easily detected by the garbage collector, i.e. every time an object is promoted to an older generation the object is scanned for pointers to the younger generations.

The problem lies in the first case, the mutator updating pointer fields of objects in older generations. This is commonly solved by means of a write-barrier, which traps writes to heap objects, checks if the written value is a pointer to a younger generation, and if so, takes proper action. As examples of this write-barrier we can mention entry tables used by Lieberman and Hewitt [57] and remembered sets by Ungar [86]. The basic idea behind entry tables is that pointers from an older generation into a younger one have to go through an indirection table that is stored in the younger generation. Thus, when the garbage collector runs it only has to use the entry table as part of the root set in order to capture the inter-generational pointers. The idea behind remembered sets is that when the mutator writes an old-to-young pointer, a pointer to the old object is stored in a remembered set. When the garbage collector runs, it scans the objects stored in the remembered set.

### 3.6 Incremental Garbage Collection

One of the major motivations behind generational garbage collection was to reduce the imposed pause times due to garbage collection. Since even generational garbage collectors occasionally will collect the whole heap, they still have the same worst-case scenario as their non-generational dittos. In order to reduce the worst-case pause time it was early
discovered that it would be desirable to let mutators interleave the garbage collecting process in a pseudo-concurrent manner. However, this gives rise to a need for rigorous synchronization schemes between the memory management system and its mutators.

### 3.6.1 Coherence, consistency, and conservatism

In general, the coherence problem is basically due to multiple processes sharing mutable data. Preserving coherence in an incrementally garbage collected system generally boils down to actions to notify the collector about mutations made by the mutators. Incremental copying collectors generally suffer from an even worse coherence problem, as the mutators must also account for mutations (re-locations) made by the collector. This situation is commonly referred to as a *multiple readers, multiple writers* problem. However, the required protection is not that severe. The reason is that collector and mutators do not necessarily need to have a consistent view of the heap. For instance, the garbage collector may very well, for a limited period of time, treat some dead objects as being alive (which only results in some unnecessary occupied memory), as long as it never treats live objects as being dead (which could have severe consequences).

All garbage collectors are more or less *conservative*, in such way that some dead objects may be treated as alive for a limited time because the cost, at a certain moment, of determining if it really is dead is too high.

### 3.6.2 The tricolor invariant

Dijkstra et. al. [30] presents the *tricolor marking* abstraction. This is useful in understanding how garbage collection can be made incremental. During a garbage collection cycle, an object in the heap can be in three different states. It can be undetected (possibly garbage), detected (alive), or processed. The tricoloring marking is based upon the idea of painting objects in different colors. An undetected object is colored white, detected (alive), or processed. At the beginning of a collection cycle, all objects are white. When the collector traverses the graph, it will color white
objects gray and gray objects black. The order in which this is done is dependent on the type of tracing collector that is used, but for an incremental copying collector, it is suitable to interleave the detection and copying in a \textit{breadth-first} manner; that is, before a gray object is colored black, all its white descendants are colored gray. This will lead to a criterion that is the essence of the \textit{tricolor invariant}; to wit, a black object can never point to a white object. It always has to be a gray object in between them. A violation of this invariant may cause an object that is alive to be reclaimed as garbage.

In Figure 3.9 and 3.10, we see an example of how mutations may cause violations of the tricolor invariant. Figure 3.9 shows the heap before the mutation, and in Figure 3.10 we see the heap after the mutation. The collector thinks that since A is already scanned (i.e., it is black) all its descendants have been detected. Thus, D will be collected even though it is still alive.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3.9.png}
\caption{The tri-color invariant is upheld during garbage collection.}
\end{figure}

The solution is to either color D gray (Figure 3.11), or revert the color of A back to gray (Figure 3.12), when A is mutated. Either way, the tricolor invariant is preserved.
3.6. Incremental Garbage Collection

3.6.3 Conservatism and allocation

Another choice needed to be made is where new allocations should reside. One could allocate them as white, which means that new allocations are able to die and be collected within the same garbage collection cycle they were allocated. This is the least...
conservative allocation strategy. New allocations that survive, must be marked/copied and scanned. Allocating new object gray means that the collector does not need to mark/copy newly allocated objects but still needs to scan them. The last possibility is allocating new objects black. This means that allocations made during garbage collection do not impose any more work needed to be done by the collector. However, initialization of these objects must ensure no black to white pointers are created.

### 3.6.4 Barrier methods

Preserving the tricolor invariant is done through so called *barriers*. We will look at these barrier methods in the context of an incremental copying garbage collector. We can prohibit the mutator from accessing fromspace (white) objects by using a *read barrier* that, whenever the mutator tries to access a white object, the collector interleaves to copy the object (color it gray), as shown in Figure 3.11. Alternatively, whenever the mutator writes a white pointer into a black object, the color of the mutated object is reverted back to gray.
3.6. Incremental Garbage Collection

The read barrier

Baker’s collector [10] is an extension of Cheney’s collector. It is made incremental with a tospace invariant; that is, the mutator can only see objects in tospace. That means, creating or dereferencing a pointer to a white object causes the object to be copied.

This is accomplished by a read barrier that checks if the mutator dereferences a pointer to a white object, and a write barrier that checks if the written value is a pointer to a white object. If the mutator dereferences a pointer to a white object, the object is copied first.

The intrinsic tricolor marking in Cheney’s collector is shown in Figure 3.7, where everything behind the scan pointer is black and everything in front of it is gray. In Baker’s algorithm, new allocations during garbage collection are made at the other end of tospace and considered black (see Figure 3.13).

![Figure 3.13: New allocations during garbage collection are made at the other end of tospace and considered black in Baker’s incremental copying garbage collector.](image)

In contrast to Baker’s choice of allocating objects during garbage collection as black, one could allocate them as either gray or white. However, since Baker’s collector is based on a tospace invariant, allocating new objects as white would not make any sense. Furthermore, since these newly allocated objects cannot contain pointers to white objects (i.e., the mutator is not allowed to see white objects and thus cannot write pointers to white objects into the new object), allocating new objects as gray would just cause
unnecessary scanning.

Brooks [22] extends Baker’s collector by always initializing the forwarding pointer field of a new allocation to point to the new object itself. Accesses can then always be indirected through the forwarding pointer by the read barrier; instead of checking if a forwarding pointer exists.

Although enforcing the tricolor invariant through a read barrier has its advantages, the main drawback of read barriers is its inefficiency. Zorn’s evaluation of different barrier methods shows that read barriers are expensive on conventional machines, mainly because reads tend to occur very frequently in comparison to writes [95].

The write barrier

There is one way an incremental garbage collector can behave correctly without preserving the tricolor invariant. Yuasa’s snapshot-at-the-beginning incremental mark-sweep garbage collector does not prohibit black to white pointers [93]. Instead, the write barrier traps pointer updates and marks the object pointed to previously if the object is white. In effect, it preserves the aliveness of the heap as it was when the garbage collector started (hence the name snapshot-at-the-beginning). New allocations during garbage collection are made non-white. Thus, Yuasa’s collector will not fail to mark all live objects. However, on the other hand, Yuasa’s collector is very conservative since white objects that die during garbage collection will not be collected before the next cycle.

A more common technique for ensuring that all live objects will be marked/copied is the incremental update methods, which are not as conservative as Yuasa’s algorithm. They all preserve the tricolor invariant (no black to white pointers) through the write-barrier which, depending on the algorithm, colors one of the objects involved gray.

The garbage collector presented by Dijkstra et al. [30] is an example of the more conservative incremental update techniques. It colors a white object gray regardless of the color of its new parent. For example, when a white to white pointer is written, the pointee is colored gray. A less conservative technique is the one due to Steele [80, 81], which reverts the color of a black object back to gray upon mutation.
3.6.5 Distinguishing mutable and immutable data

Doligez and Leroy [31], as well as Huelsbergen and Larus [38], identify the advantage of distinguishing mutable and immutable objects from a garbage collectors perspective. This is especially an advantage for a copying collector, due to its consistency issues. Basically, in terms of consistency, mutable and immutable data differ in such way that access to two different copies of the same immutable data structure does not result in incoherence, whereas mutator access to multiple copies of a mutable data structure does.

3.7 Real-Time garbage collection

A real-time garbage collector is a garbage collector that (1) does not cause any real-time task to miss its deadline at the same time as it (2) reclaims garbage memory in a sufficient pace. Both of these requirements originate in the schedulability problem; i.e., is the system schedulable or not? However, in order to make such assessments, one needs to concretize what there is to schedule. Commonly, scheduling garbage collection is often reasoned about in terms of induced pause times. Since non-incremental garbage collectors induce too long pause times, they are not suitable for real-time systems. Reducing the worst-case pause time means reducing the length of the longest (worst-case) garbage collector increment. One thing is certain; the execution time of each increment of the garbage collector must be small and bounded (preferably constant).

3.7.1 Scheduling garbage collection

The problem of scheduling the garbage collector increments within a real-time system is two-dimensional. First of all, one needs be able to guarantee that the timing requirements of the real-time tasks are not violated. At the same time the garbage collector must also be scheduled in such way that sufficient free memory is available at all times. The ultimate question is then, is there a schedule where both of these requirements are fulfilled?
Scheduling policies

Finding an optimal scheduling policy (i.e. being able to schedule every system for which there exists a schedule) is probably practically impossible due to the NP nature of its corresponding decision problem. However, from a pragmatic point of view, it is still good if we may schedule a big subset of them.

In his dissertation, Henriksson presents a policy for scheduling garbage collection in embedded systems with both soft and hard real-time tasks [36]. The main idea of his policy is that the garbage collector should never interfere with the hard real-time tasks. However, in order to sufficiently reclaim garbage produced by the hard real-time tasks, necessary garbage collection work due to their actions has to be scheduled with a higher priority than the soft real-time tasks. In Figure 3.14, a typical example of Henriksson’s schedule is shown.

Figure 3.14: Typical schedule of a system based on Henriksson’s scheduling policy.
3.7. Real-Time garbage collection

3.7.2 Predicting garbage collection pause times

Before we go into details about how pause times due to garbage collection can be predicted, we need to decide how the garbage collector may be scheduled in the system. We assume a concurrent garbage collector with sporadic behavior. That means, even though the garbage collector has a lower priority than the hard real-time tasks, it may be running when a hard real-time job is released.

There are two kinds of induced pause times due to garbage collection. The first kind is due to executing a garbage collection increment. That is, if a real-time job is released while a garbage collector increment is being executed, the job has to be postponed until the increment has finished. The WCET of real-time tasks must include the longest WCET of the garbage collector increments. The second kind is due to the read barrier and the write barrier. The WCET of heap reads and writes has to include the WCET of the corresponding barrier methods.

3.7.3 Guaranteeing sufficient memory reclamation

Guaranteeing that enough memory is reclaimed at all times is a more difficult task than predicting the pause times due to garbage collection. Let us first look at a system with only hard real-time tasks\(^1\). We assume that the only pause time the garbage collector may induce is as defined in the previous subsection. In other words, the garbage collector increments may only be started when no hard real-time tasks are pending. The question is then: is there enough idle time and is it distributed in such way that garbage memory can be reclaimed sufficiently fast? In order to assess this, we need to know the memory consumption behavior of each task.

Real-time garbage collection is more than just an algorithm; it includes schedulability, memory requirements, etc.. An incremental garbage collection algorithm is just a good starting point. The main requirements on the algorithm are typically as follows. Bounded

\(^{1}\)It can be argued that it is always possible to suspend a soft real-time task in order to do sufficient garbage collection.
pause times are achieved by predictable worst-case execution time of all increments. Worst-case memory usage overhead is derivable from the algorithm. And finally, the most important aspect: the collector must be possible to schedule in the real-time system. It should neither cause violations of timing constraints, nor should it cause the system to consume more memory than available.

3.7.4 Predictability and efficiency of copying garbage collection

If we would be able to determine the amount of live data at any given point in time, the copying garbage collector would be almost one hundred percent predictable. The execution time of the collector is directly proportional to the amount of data that currently is alive. Due to the fact that the copying garbage collector is actually a defragmentation process and garbage collection is a side effect, it would be unfair to evaluate its efficiency only in terms of collected garbage. The copying collector suffers from inefficiency when it performs unnecessary defragmentation. In the worst case, it would copy half the heap (whole fromspace), accomplishing neither defragmentation nor garbage collection.

On the other hand, when the amount of live data is small and there is a lot of garbage produced, the collector would be very efficient. In the best case, it would collect half a heap worth of garbage almost without doing anything at all, at least not in terms of copying work.

3.8 Design choices

In his dissertation, Ritzau presents a real-time reference counting garbage collector where cyclic structures have to be broken manually [72]. Even though reference counting is naturally incremental, the lack of native support for collecting circular data structures removes the technique from our list of choices. Furthermore, the behavior of non-compacting algorithms is generally very unpredictable in terms of free memory availability due to fragmentation. Sophisticated methods exist for solving this problem, but they commonly induce too much overhead on the allocator. So, we need an algorithm that is incremen-
tal, compacts the heap, and enables small and predictable allocation time. In addition, it should be able to run together with an arbitrary number of mutator threads; i.e., it should be possible to integrate into a typical real-time model as described in Chapter 2. Most importantly, the algorithm should enable decoupling of the WCET of the real-time tasks and the garbage collector. That is, assuming that enough free memory is available at all times, it should be possible to reason about the schedulability of the system without knowing anything about the garbage collector. Furthermore, adding garbage collection should not invalidate that analysis. In other words, the algorithm should enable incremental analyses without mutual dependencies.

Based on the aforementioned reasons, we have chosen an incremental copying garbage collector that is scheduled to only run in idle periods. In the next chapter we describe the memory model including the garbage collector, which enables reasoning about the garbage collector in the context of parallel mutator processes. In Chapter 5, we describe how scheduling garbage collection in idle periods opens up possibilities for incremental reasoning about the real-time capabilities of the whole system, including garbage collection.
In this chapter we present a formal model of an incremental copying garbage collector (Section 4.1). Our collector is defined in terms of very small atomic increments, and we use the process calculus techniques of a labeled transition system and bisimulation to model mutator interaction and prove correctness (Sections 4.2 and 4.3). We furthermore show that our garbage collector is useful, in the sense that it actually recovers unreachable portions of a heap (Section 4.4).

Apart from taking the first step towards the overall goal of designing a formally proved correct real-time garbage collection system (which in the end should include both a correct algorithm and statically analyzable properties such as schedulability and memory usage, etc.), the main contribution of this chapter is that we show that it is possible to reason about the garbage collector and the mutator(s) as process terms. This has, to the best of our knowledge, not been done before. Another significant contribution is that formal correctness of incremental copying garbage collection has not been shown until very recently [62]. A third contribution is that we demonstrate that it is also possible to formally reason about the usefulness of an incremental copying garbage collector.
4.1 The Garbage Collector

Before we go into details of our model we would like to spend some time describing the algorithm more informally. Even though the way we present the algorithm is rather non-conventional, the algorithm itself is quiet well-known. We use Cheney’s in-place breadth-first traversal of gray objects [25], and we deploy a read barrier similar to that of Brooks [22]; i.e., reads to old copies in white space are forwarded to their corresponding new copies in the gray/black heap. Furthermore, we use a write barrier in the style of Steele [80, 81], where the tri-color invariant is upheld by reverting black objects to gray upon mutation. The driving force behind these choices is that we strive to defer as much of the garbage collection work into scheduled garbage collection time instead of taking the cost when a real-time task allocates, reads, or writes to the heap. These choices may need to be reconsidered depending on the mutator and the mission of the application. However, the algorithm and proofs can easily be reworked for this purpose.

We will now continue by describing the garbage collector in more detail. Let $x, y, z$ range over heap addresses, and let $n$ range over integers. Let $u, v$ range over values and be either a heap address or an integer. Let $U, V$ range over sequences of such values.

A heap node can be either a sequence of values (enclosed by angle brackets $⟨V⟩$) or a single forwarding address (denoted $•x$). A heap $H$ is a finite mapping from addresses to nodes, as captured by the following grammar.

\[
(\text{heap}) \quad H ::= \{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\}
\]

\[
(\text{node}) \quad o ::= ⟨V⟩ | •x
\]

\[
(\text{value}) \quad v ::= x | n
\]

The domain $\text{dom}(H)$ of a heap $H = \{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\}$ is the set $\{x_1, \ldots, x_n\}$. A heap look-up is defined as $H(x) = o$ if $x \mapsto o \in H$. We will write $U, V$ to denote the concatenation of the value sequences $U$ and $V$. Along the same line, we will write $H, G$ for the concatenation of heaps $H$ and $G$, provided their domains are disjoint.

For garbage collection purposes, a heap can be described as a triple of subheaps separated by heap borders (|). A heap border has the same meaning as the regular
4.1. The Garbage Collector

concatenation operator for heaps, but it also provides necessary bookkeeping information. The three subheaps captures the white, gray, and black part of the heap as in the tricolor abstraction [30].

The algorithm is based on a labeled transition system (LTS), where garbage collection transitions are so called internal (τ) transitions. Each individual τ transition constitutes an atomic increment by the garbage collector. In Figure 4.1 and 4.2, all possible internal transitions are shown. Determinism between different internal transitions are achieved by pattern matching, as each configuration matches only one single clause. The clauses are furthermore divided into two groups, which we call scan and copy transitions. A garbage collection cycle is a sequence of such transitions beginning with a Start transition and ending with a Done transition.

\[ H_0 \xrightarrow{\text{START}} H_1 \xrightarrow{} \ldots \xrightarrow{} H_{n-1} \xrightarrow{\text{DONE}} H_n \]

Notice that an active garbage collection cycle is identified by a non-empty white subheap.

In contrast, the external triggered transitions, such as mutations and allocations are labeled, denoted by \( H \xrightarrow{\text{mutations}} H' \). We will look into these in more detail in the next section. We use a single root pointer \( r \) to capture the root-set. Even though a real system most likely will contain more than one root, this can easily be captured in our model by adjusting the content of the node pointed to by \( r \), i.e. the actual root-set is the content of the node labeled \( r \).

At the beginning of a cycle the heap has the form \( \emptyset | G, r \mapsto \langle V \rangle | \emptyset \), and initiating a garbage collection cycle (START) invalidates the whole heap except for the root node. That is, all nodes but \( r \) are made white by placing them to the left of the white-gray heap border. The algorithm then proceeds by scanning gray nodes (SCANSTART) and takes proper actions when embedded addresses are encountered. This is accomplished by a scan pointer that traverses the gray nodes. The scan pointer is denoted by the symbol \( \downarrow \) and has similar function and purpose as the heap borders, i.e. regular concatenation as
well as bookkeeping information. When a whole node has been scanned it is promoted from gray to black (ScanDone). When there are no more gray nodes to scan the garbage collector is finished (Done).

During scanning of a gray node, encountering an address may result in one of three possible actions. If the address found is not in the white heap, the algorithm just goes on to examine the next gray node field (ScanAddr). If the address is in the white heap, and the corresponding node is a forwarding node, the forwarding address replaces the encountered address (Forward). If, on the other hand, the node found is a regular white node, copying is initiated (CopyStart). This is done by allocating a new empty node in the gray heap and locking the scan pointer, which we denote by the alternative concatenation symbol \( \uparrow_z \) (where the index is the address of the new empty node). The white node is then copied word by word (CopyWord) until the whole node has been copied (CopyDone). At this point, the address of the newly allocated node replaces the old encountered address, and the original white node is converted into a forwarding node.

In addition to these scenarios, two other transitions are also defined: ScanRestart and CopyRestart. These transitions are taken when mutations occur during garbage collection, and will be described in more detail in the next sections.

### 4.2 The Mutator

In order to capture the behavior of an interacting mutator we begin by defining a recursive function \( \text{read} \) that is based on the notion of an abstract \( \text{path} \) beginning in a \( \text{root} \).

**Definition 4.2.1**

\[
\begin{align*}
(p,q) &::= \varepsilon \mid i : p \quad \text{where } i \text{ is an index}
\end{align*}
\]
4.2. The Mutator

Definition 4.2.2 For some heap $H$ and root $x$,

\[
\begin{align*}
\text{read}(H, x, n : p) &= \text{read}(H, V[i], p) & \text{if } H(x) = \langle V \rangle \\
\text{read}(H, x, \varepsilon) &= n & \\
\text{read}(H, x, \varepsilon) &= x & \text{if } H(x) \neq \bullet y \\
\text{read}(H, x, p) &= \text{read}(H, y, p) & \text{if } H(x) = \bullet y
\end{align*}
\]

The mutator activities can now be defined as a set of labeled transitions $H \xrightarrow{l} H'$ (see Figure 4.3) where the label $l$ captures the different means a mutator can interact with a heap.

\[
(l) : = r(p = n) \mid w(p = n) \mid w(p = q) \mid a(p)
\]

Here $r(p = n)$ means reading the integer $n$ through the path $p$, $w(p = n)$ means writing the integer $n$ at the end of path $p$, $w(p = q)$ means writing the value found at the end of path $q$ at the end of path $p$, and $a(p)$ means allocate a fresh node and write the address of it at the end of path $p$. The behavior of $\xrightarrow{l}$ transitions is shown in Figure 4.3.

Two important details of Figure 4.3 require special mentioning. Firstly, clauses \textsc{Mut}/\textsc{AllocMut} mark the mutated node as dirty, indicated by a dot over the node’s
address (\( \hat{x} \)). Secondly, if a black node is mutated (\texttt{MutB} / \texttt{AllocMutB}), the node is reverted back to gray.
4.3 Correctness of the Garbage Collector

The tri-color invariant [30] is captured by a property we denote as well-formedness.

Figure 4.3: Mutator transitions.
Definition 4.3.1 (Well-formedness). A heap $H = W | G | B$ is well-formed iff

1. $B(x) = \langle V \rangle$ implies $V \cap \text{dom}(W) = \emptyset$
   
2. $H(x) = \bullet y$ implies $(x \in \text{dom}(W) \land y \notin \text{dom}(W))$
   
3. $r \in \text{dom}(G, B)$
   
4. $H = W \upharpoonright G, x \mapsto \langle V \downarrow U \rangle \upharpoonright B$ implies $V \cap \text{dom}(W) = \emptyset$
   
5. $H = W \upharpoonright G, x \mapsto \langle V \uparrow z, y, U \rangle \upharpoonright B$ implies $W(y) = \langle V, V' \rangle$ and $G(z) = \langle V \rangle$

Informally, a heap is well-formed if and only if no black nodes contain addresses to any white nodes (1), a forwarding node can only be white and its forwarding address cannot be to a white node (2), the root node $r$ is either in $G$ or $B$ (3), and while scanning a node in the gray heap nothing left of the scan pointer can be pointers to the white heap (4).

Lemma 4.3.1 states that a garbage collection transition of the heap preserves well-formedness.

Lemma 4.3.1 If $H$ is well-formed and $H \xrightarrow{\text{---}} H'$ then $H'$ is also well-formed.

Proof By case study on the clauses defining $\xrightarrow{\text{---}}$. For convenience, let $H = W | G | B$ and $H' = W' | G' | B'$. Observe that since $H$ is well-formed, the root node is in $G$ or $B$. Furthermore, since only transition Start removes nodes from $G \cup B$, and there makes an explicit exception for $r$, (3) is upheld throughout.

Start:

Since $B' = \emptyset$, (1) holds vacuously. Since $H$ is well-formed and $W = \emptyset$, it cannot include any forwarding nodes. Thus, since no new forwarding node is introduced by the transition, $H'$ does not include any forwarding nodes, i.e. (2) holds vacuously. (4) and (5) also hold vacuously.
4.3. Correctness of the Garbage Collector

**ScanStart, ScanInt, ScanAddr, ScanRestart, Forward, CopyStart, CopyWord, CopyRestart, CopyDone:**

Since neither $B$ nor $W$ is modified, and no forwarding node is created, both (1) and (2) are upheld by the transition. Since none of the transitions extend the sequence of values left of the scan pointer with pointers to $W$, (4) is also upheld. Furthermore, since CopyStart requires that $y \in \text{dom}(W)$ and $z$ is a fresh node allocated in $G$, (5) is upheld. For CopyWord and CopyRestart, it is immediate from the structure of $y$ and $z$ that (5) is upheld.

**ScanDone:**

From Definition 4.3.1 follows that $V \cap \text{dom}(W) = \emptyset$. Thus (1) holds for $H'$. ScanDone does not create any forwarding nodes and no new node is introduced to $W$ (i.e. $W = W'$). Thus, (2) is upheld by the transition. (4) and (5) hold vacuously.

**Done:**

Since $H$ is well-formed, we know that $B$ does not contain any forwarding nodes. Thus, both (1) and (2) hold vacuously.

**MutW, MutG, Read, AllocMutW, AllocMutG:**

Since $B$ is not modified (i.e. $B = B'$) and $\text{dom}(W) = \text{dom}(W')$, (1) is upheld by the transitions. Since no forwarding nodes are introduced and, again, $\text{dom}(W) = \text{dom}(W')$, (2) is upheld by the transitions. Since the mutated node is made dirty, (4) and (5) hold vacuously.

**MutB, AllocMutB:**

Since $B' \subset B$, i.e. no new node is introduced, and $\text{dom}(W) = \text{dom}(W')$, (1) is upheld by the transition. Since no new forwarding node is introduced and, again, $\text{dom}(W) = \text{dom}(W')$, (2) is also upheld by the transition. (4) and (5) hold vacuously.
Lemma 4.3.2 captures the property of determinism.

**Lemma 4.3.2** If $H$ is well-formed then the structure of $H$ matches the pattern of exactly one $\tau$ transition.

**Proof** By case study on all possible structures of a well-formed $H$.

$H = W | \emptyset | B$ where $W \neq \emptyset$, $B \neq \emptyset$:

Only one clause matches this pattern (DONE), and it will succeed no matter what internal structure $W$ and $B$ have.

$H = W | G | \emptyset$ where $W \neq \emptyset$, $G \neq \emptyset$,

$H = W | G | B$ where $W \neq \emptyset$, $G \neq \emptyset$, $B \neq \emptyset$:

Since $G$ is not empty, there is at least one node in it. Thus, $G$ must match the pattern $G = G', x \mapsto o$. We now need to look at the possible structures of $x \mapsto o$.

From the definition of a node, $o$ is either a regular node ($\langle V \rangle$) or a forwarding address ($\bullet z$). However, from Definition 4.3.1 (well-formedness) follows that $x \mapsto o$ cannot be a forwarding node. We now need to look at the possible structures of $\langle V \rangle$. We have three cases.

$x \mapsto \langle V \rangle$:

Matches **ScanStart**.

$x \mapsto \langle V \downarrow V' \rangle$:

Matches **ScanRestart**.

$x \mapsto \langle V \downarrow V' \rangle$:

$V' = \emptyset$:

Matches **ScanDone**.

$V' = v, V''$:

$v = n$:

Matches **ScanInt**.

$v = y, y \not\in \text{dom}(W)$:

Matches **ScanAddr**.
4.3. Correctness of the Garbage Collector

\[ v = y, y \mapsto (U) \in W: \]

Matches CopyStart.

\[ v = y, y \mapsto \bullet z \in W: \]

Matches Forward.

\[ x \mapsto \langle \lfloor \ell_z, V' \rfloor \rangle: \]

From definition 4.3.1 follows that \( V' = y, V'' \), \( y \mapsto (U) \in W \), and \( z \mapsto (U') \in G \). Furthermore, it follows that \( H(z) \subseteq H(y) \) if \( y \) is not dirty.

Thus we have the following cases.

\[ \dot{y} \mapsto \langle U \rangle: \]

Matches CopyRestart.

\[ y \mapsto \langle U \rangle, z \mapsto \langle U' \rangle, U' \subset U: \]

Matches CopyWord.

\[ y \mapsto \langle U \rangle, z \mapsto \langle U' \rangle, U' = U: \]

Matches CopyDone.

\[ H = \emptyset | G | \emptyset \] where \( G \neq \emptyset \):

Since \( H \) is well-formed it includes at least the root node. I.e. \( G = G', r \mapsto \langle V \rangle \). START will match whatever structure \( G' \) has.

\[ H = W | \emptyset | \emptyset \]

Since \( H \) is required to be well-formed, \( r \in \text{dom}(G, B) \) which means that this structure is not well-formed.

\[ 4.3.1 \text{ } \text{Termination} \]

Theorem 4.3.3 captures a very important property: termination. It states that for every well-formed heap that is within a garbage collection cycle (the Start transition has been taken), a state where the Done transition can be taken may be reached after a finite number of garbage collection steps.

**Theorem 4.3.3 (Termination)** For every well-formed \( H = W | G | B \) \((W \neq \emptyset)\) there is some \( H' = W' | \emptyset | B' \) such that \( H \xrightarrow{*} H' \).
Proof We first need to define a notion of weight for a heap. Since the algorithm is incremental down to single word copy actions and allows mutations in between, we need a fine-grained metric for weight in order to capture the small weight-losses in each increment. We do this by a triple metric, where the first element is the dominant weight factor while as the second and the third elements capture the progress of scanning and copying a single node, respectively. The weight of a heap is defined in Figure 4.4.

Heap weights are ordered lexicographically; i.e., the first element is the most significant and the third (last) element is the least significant. For example, suppose we have two heaps $H$ and $H'$, and $weight(H) = (a, b, c)$ and $weight(H') = (d, e, f)$. Then,

$$(a, b, c) < (d, e, f)$$

if $(a < d)$ or $(a = d \land b < e)$

or $(a = d \land b = e \land c < f)$.

The proof relies on the fact that each $\tau$ transition possible when $W \neq \emptyset$ and $G \neq \emptyset$ reduces the weight of the heap.

ScanStart:

$weight(W \mid G, x \mapsto \langle V \rangle \mid B) > weight(W \mid G, x \mapsto \langle \downarrow V \rangle \mid B)$

$\iff$

$(w(W \mid G, x \mapsto \langle V \rangle \mid B), \infty, \infty) > (w(W \mid G, x \mapsto \langle \downarrow V \rangle \mid B), |V|, \infty)$

$\iff$

Let $a = w(W \mid G, x \mapsto \langle V \rangle \mid B)$, then,

$(a, \infty, \infty) > (a, |V|, \infty)$, which is true since $\infty > |V|$.

ScanInt,ScanAddr:

$weight(W \mid G, x \mapsto \langle V \downarrow v, V' \rangle \mid B) > weight(W \mid G, x \mapsto \langle V, v \downarrow V' \rangle \mid B)$

$\iff$

$(w(W \mid G, x \mapsto \langle V \downarrow v, V' \rangle \mid B), |n, V'|, \infty) > (w(W \mid G, x \mapsto \langle V, v \downarrow V' \rangle \mid B), |V'|, \infty)$

$\iff$

Let $a = w(W \mid G, x \mapsto \langle V, v \downarrow V' \rangle \mid B)$, then,

$(a, |v, V'|, \infty) > (a, |V'|, \infty)$, which is true since $|v, V'| > |V'|$. 
4.3. Correctness of the Garbage Collector

ScanRestart:

\[ \text{weight}(W \mid G, \dot{x} \mapsto (V \downarrow V') \mid B) > \text{weight}(W \mid G, x \mapsto (\downarrow V, V') \mid B) \]

\[ \Leftarrow \]

\[ (w(W \mid G, \dot{x} \mapsto (V \downarrow V') \mid B), \infty, \infty) > (w(W \mid G, x \mapsto (\downarrow V, V') \mid B), |V, V'|, \infty) \]

\[ \Leftarrow \]

Let \( a = w(W \mid G, x \mapsto (V, V') \mid B) \), then,

\[ (a, \infty, \infty) > (a, |V, V'|, \infty) \]

which is true since \( \infty > |V, V'| \).

ScanDone:

\[ \text{weight}(W \mid G, x \mapsto (V \downarrow) \mid B) > \text{weight}(W \mid G \mid x \mapsto (V), B) \]

\[ \Leftarrow \]

\[ (w(W \mid G, x \mapsto (V \downarrow) \mid B), 0, \infty) > (w(W \mid G \mid x \mapsto (V), B), \infty, \infty) \]

\[ \Leftarrow \]

Let \( a = w(W \mid G \mid B), H = W \mid G, x \mapsto (V \downarrow) \mid B \), and \( H' = W \mid G \mid x \mapsto (V), B \), then,

\[ (a + w(x \in H) + w((V)), 0, \infty) > (a + w(x \in H') + w((V)), \infty, \infty) \]

which is true since \( w(x \in H) > w(x \in H') \).

Forward:

\[ \text{weight}(W, y \mapsto \bullet z, W' \mid G, x \mapsto (V \downarrow y, V') \mid B) > \text{weight}(W, y \mapsto \bullet z, W' \mid G, x \mapsto (V, z \downarrow V') \mid B) \]

\[ \Leftarrow \]

Let \( a = w(W, y \mapsto \bullet z, W' \mid G, x \mapsto (V, V') \mid B) \) then,

\[ (a + w(y), |y, V'|, \infty) > (a + w(z), |V|, \infty) \]

From Definition 4.3.1 follows that \( z \notin \text{dom}(W, W') \). Thus,

\[ (a + w(y), |y, V'|, \infty) > (a + w(z), |V|, \infty) \]

is true since \( w(y) > w(z) \).
CopyStart:

\[ \text{weight}(W, y \mapsto \langle U \rangle, W', G, x \mapsto \langle V \downarrow y, V' \rangle | B) > \text{weight}(W, y \mapsto \langle U \rangle, W', G, x \mapsto \langle V \uparrow z, V' \rangle | B) \]

\[ \Leftarrow \]

Let \( a = w(W, y \mapsto \langle U \rangle, W', G, x \mapsto \langle V, y, V' \rangle | B) \), then

\( (a, |y, V'|, \infty) > (a + 0, |y, V'|, |U| - 0) \) which is true since \( \infty > |U| \).

CopyWord:

\[ \text{weight}(W, y \mapsto \langle U, u, U' \rangle, W', G, z \mapsto \langle U' \rangle, G', x \mapsto \langle V \downarrow z, y, V' \rangle | B) > \]

\[ \text{weight}(W, y \mapsto \langle U, u, U' \rangle, W', G, z \mapsto \langle \rangle, G', x \mapsto \langle V \downarrow z, y, V' \rangle | B) \]

\[ \Leftarrow \]

Let \( a = w(W, y \mapsto \langle U, u, U' \rangle, W', G, z \mapsto \langle \rangle, G', x \mapsto \langle V, y, V' \rangle | B) \), then

\( (a, |y, V'|, |U, u, U'| - |U|) > (a, |y, V'|, |U, u, U'| - |U, u|) \)

\[ \Leftarrow \]

\( (a, |y, V'|, |a, U'|) > (a, |y, V'|, |U'|) \) which is true since \( |a, U'| > |U'| \).

CopyRestart:

\[ \text{weight}(W, \hat{y} \mapsto \langle U \rangle, W', G, z \mapsto \langle U' \rangle, G', x \mapsto \langle V \downarrow z, y, V' \rangle | B) > \]

\[ \text{weight}(W, y \mapsto \langle U \rangle, W', G, z \mapsto \langle \rangle, G', x \mapsto \langle V \downarrow z, y, V' \rangle | B) \]

\[ \Leftarrow \]

Let \( a = w(W, y \mapsto \langle U \rangle, W', G, z \mapsto \langle \rangle, G', x \mapsto \langle V \downarrow z, y, V' \rangle | B) \) then,

\( (a, |y, V'|, \infty) > (a, |y, V'|, |U|) \) which is true since \( \infty > |U| \).

CopyDone:

\[ \text{weight}(W, y \mapsto \langle U \rangle, W', G, z \mapsto \langle U \rangle, G', x \mapsto \langle V \downarrow z, y, V' \rangle | B) > \]

\[ \text{weight}(W, y \mapsto \bullet z, W', G, z \mapsto \langle U \rangle, G', x \mapsto \langle V, z \downarrow V' \rangle | B) \]

\[ \Leftarrow \]

Let \( a = w(W, W' | G, z \mapsto \langle U \rangle, G', x \mapsto \langle V, V' \rangle | B) \) then,

\( (a + 2 * w(y) + w(\langle U \rangle), |y, V'|, 0) > (a + w(y) + w(z) + w(\langle U \rangle), |V'|, \infty) \)

which is true since \( w(y) > w(z) \).
4.3. Correctness of the Garbage Collector

4.3.2 Preservation

In this section, we show that our garbage collector preserves the meaning of the heap. In order to do that we need to define what it means for two heaps to be equivalent. The first attempt is to define a read-equivalence; that is, two heaps are equivalent if and only if a non-address read at a legal path gives the same result in both heaps. However, since a heap may contain shared paths, we may get identical read-behavior in two structurally different heaps that might be distinguished after further mutation. In order to capture equivalence even after mutation we need to assure that, for two heaps to be equivalent, they are both read-equivalent and contain exactly the same shared paths.

We begin by defining the predicate join, which is a boolean function that takes a heap, two starting addresses, and two paths as parameters, and returns True if the two paths starting at each address reach the same node. We also define a variant for the special case when the two paths both start at the same address.

**Definition 4.3.2**

\[
\text{join}(H, x, y, p, q) \overset{\text{def}}{=} \\begin{array}{ll}
\text{read}(H, x, p) = z & \\
\text{read}(H, y, q) = z & \\
\end{array} \\
\text{join}(H, x, x, p, q) \overset{\text{def}}{=} \text{join}(H, x, x, p, q)
\]

We can now define the equivalence relation between heaps.

**Definition 4.3.3** Two heaps \( H \) and \( H' \) are structurally equivalent, written \( H \equiv H' \), if and only if,

- \( H \) and \( H' \) are both well-formed, and
- \( \forall p . \text{read}(H, r, p) = n \iff \text{read}(H', r, p) = n \)
  
  and \( \forall q . \text{join}(H, r, p, q) \iff \text{join}(H', r, p, q) \)

This equivalence relation is preserved by mutations. In fact it is preserved by all possible mutator activities, as captured by the following lemma.
Lemma 4.3.4 If $H \equiv H'$ and $H \xrightarrow{l} \hat{H}$, then $H' \xrightarrow{l} \hat{H}'$ and $\hat{H} \equiv \hat{H}'$.

Proof By case study on $H \xrightarrow{l} \hat{H}$. We only show the case for MutW; all other cases are either similar or trivial.

\[ H = W, x \mapsto (U), W' | G | B \quad \text{read}(H, r, p) = x \]

\[ \hat{H} = W, \hat{x} \mapsto (U[i] : = y), W' | G | B \]

From the definition of $H \equiv H'$ (Def. 4.3.3) follows that:

1. $\text{join}(H, r, p, p) = \text{True}$ (i.e. $x$) $\iff \text{join}(H', r, p, p) = \text{True}$ (let it be $x'$). (I)
2. $\text{join}(H, r, q, q) = \text{True}$ (i.e. $y$) $\iff \text{join}(H', r, q, q) = \text{True}$ (let it be $y'$). (II)
3. $\text{join}(H, x, r, p', q')$ $\iff \text{join}(H', x, r, p', q')$ (III)
4. $\text{read}(H, r, p : i) = n$ $\iff \text{read}(H', r, p : i) = n$ (IV)

\[ \forall p', q'. \]

\[ \text{read}(H, x, p') = n \iff \text{read}(H', x', p') = n \quad \text{(V)} \]
\[ \text{read}(H, y, p') = n \iff \text{read}(H', y', p') = n \quad \text{(VI)} \]
\[ \text{join}(H, x, r, p', q') \iff \text{join}(H', x', r, p', q') \quad \text{(VII)} \]
\[ \text{join}(H, y, r, p', q') \iff \text{join}(H', y', r, p', q') \quad \text{(VIII)} \]
\[ \text{join}(H, x, p', q') \iff \text{join}(H', x', p', q') \quad \text{(IX)} \]
\[ \text{join}(H, y, p', q') \iff \text{join}(H', y', p', q') \quad \text{(X)} \]
\[ \text{join}(H, x, y, p', q') \iff \text{join}(H', x', y', p', q') \quad \text{(XI)} \]

From (I), (II), (III), and (IV) follows that $H' \xrightarrow{w(p \mapsto q)} \hat{H}'$.

We now have to show that $\hat{H} \equiv \hat{H}'$.

From the definition of $\equiv$ follows that:

\[ \forall p'. \]

\[ \text{read}(\hat{H}, r, p') = n \iff \text{read}(\hat{H}', r, p') = n \quad \text{(1)} \]

\[ \wedge \]

\[ \forall q'. \text{join}(\hat{H}, r, p', q') \iff \text{join}(\hat{H}', r, p', q') \quad \text{(2)} \]

We proceed by induction on the length of $p'$. 
Base case: \(p' = \varepsilon\)

(1) holds vacuously.

(2) by induction on the length of \(q'\)

Base case: \(q' = \varepsilon\)

\[
\text{join}(\hat{H}, r, \varepsilon, \varepsilon) = \text{join}(\hat{H}', r, \varepsilon, \varepsilon) = \text{True} \quad \text{(i.e. both for } r)\]

Otherwise (i.e. \(q' \neq \varepsilon\)):

Let \(q' = q'_1 : j : q'_2\) and \(\text{read}(\hat{H}, r, q'_1 : j) = z\) \((\text{read}(\hat{H}', r, q'_1 : j) = z')\)

Induction hypothesis:

\[
\text{join}(\hat{H}, r, \varepsilon, q'_1) \iff \text{join}(\hat{H}', r, \varepsilon, q'_1)\]
\[
\text{join}(\hat{H}, r, z, \varepsilon, q'_2) \iff \text{join}(\hat{H}, r, z, \varepsilon, q'_2)\]

Case study on \(q'_1\):

\[
\text{read}(\hat{H}, r, q'_1) = x \quad (\text{read}(\hat{H}', r, q'_1) = x')\]

\(j = i\): From the induction hypothesis and (IX) follows that

\[
\text{join}(\hat{H}, r, \varepsilon, q'_1) \iff \text{join}(\hat{H}, r, \varepsilon, q'_1)\]

\(j \neq i\): From the induction hypothesis and (VII) follows that

\[
\text{join}(\hat{H}, r, \varepsilon, q'_1) \iff \text{join}(\hat{H}, r, \varepsilon, q'_1)
\]
\[
\text{read}(\hat{H}, r, q'_1) \neq x: (\text{read}(\hat{H}', r, q'_1) \neq x')\]

It follows directly from the induction hypothesis that

\[
\text{join}(\hat{H}, r, \varepsilon, q'_1) \iff \text{join}(\hat{H}, r, \varepsilon, q'_1)\]

Otherwise (\(p' \neq \varepsilon\)):

Let \(p' = p'_1 : j : p'_2\) and \(\text{read}(\hat{H}, r, p'_1 : j) = z\) \((\text{read}(\hat{H}', r, p'_1 : j) = z')\)

Induction hypothesis:

\[
\text{read}(\hat{H}, r, p'_1) = n \iff \text{read}(\hat{H}', r, p'_1) = n\land\]
\[
(\forall q' \text{ join}(\hat{H}, r, p'_1, q') \iff \text{join}(\hat{H}', r, p'_1, q'))\land\]
\[
\text{read}(\hat{H}, z, p'_2) = n \iff \text{read}(\hat{H}', z', p'_2) = n\land\]
\[
(\forall q' \text{ join}(\hat{H}, z, r, p'_2, q') \iff \text{join}(\hat{H}', z', r, p'_2, q'))\]

Case study on \(p'_1\):

\[
\text{read}(\hat{H}, r, p'_1) = x \quad (\text{read}(\hat{H}', r, p'_1) = x')\]

\(j = i\):
(1):

From the induction hypothesis and (VI) follows that

\[ \text{read}(\hat{H}, r, p') = n \iff \text{read}(\hat{H}', r, p') = n \]

(2):

By induction on the length of \( q' \)

Base case: \( q' = \varepsilon \)

Similar to (2) in base case since \( \text{join}(\hat{H}, r, p', \varepsilon) = \text{join}(\hat{H}, r, \varepsilon, p') \)

Otherwise (i.e. \( q' \neq \varepsilon \)):

Let \( q' = q'_1 : k : q'_2 \) and \( \text{read}(\hat{H}, r, q'_1 : k) = \hat{x} \cdot (\text{read}(\hat{H}', r, q'_1 : k) = \hat{x}') \)

Induction hypothesis:

\[ \text{join}(\hat{H}, y, r, p'_2, q'_1) \iff \text{join}(\hat{H}', y', r, p'_2, q'_1) \]
\[ \text{join}(\hat{H}, y', \hat{x}, p'_2, q'_1) \iff \text{join}(\hat{H}', y', \hat{x}', p'_2, q'_1) \]

Case study on \( q'_1 \)

\[ \text{read}(\hat{H}, r, q'_1) = x \cdot (\text{read}(\hat{H}', r, q'_1) = x') \]

\( k = i \):

From the induction hypotheses and (X) follows that

\[ \text{join}(\hat{H}, r, p', q') \iff \text{join}(\hat{H}', r, p', q') \]

\( k \neq i \):

From the induction hypotheses and (XI) follows that

\[ \text{join}(\hat{H}, r, p', q') \iff \text{join}(\hat{H}', r, p', q') \]
\[ \text{read}(\hat{H}, r, q'_1) \neq x \iff \text{read}(\hat{H}', r, q'_1) \neq x' \]

It follows from the induction hypotheses and (IX) that

\[ \text{join}(\hat{H}, r, p', q') \iff \text{join}(\hat{H}', r, p', q') \]

\( j \neq i \):

(1):

From the induction hypothesis and (V) follows that

\[ \text{read}(\hat{H}, r, p') = n \iff \text{read}(\hat{H}', r, p') = n \]

(2):

Similar to (2) in previous case (i.e. \( j = i \)).
4.3. Correctness of the Garbage Collector

\[ read(\hat{H}, r, p') \neq x \land (read(\hat{H}', r, p'_1) \neq x') \]

(1):

From the induction hypothesis follows that

\[ read(\hat{H}, r, p') = n \iff read(\hat{H}', r, p') = n \]

(2): Similar to (2) in previous cases.

Bisimulating Garbage Collection.

In order to show that our garbage collector preserves the meaning of a heap, we adopt the notion of weak bisimulation between processes [63]. In our context, a process is simply a heap that may take one \( \tau \rightarrow \) transition or one of all possible \( l \rightarrow \) transitions. We define two processes, one with garbage collection and one without.

\[
\begin{align*}
P_{GC}(H) & \overset{\text{def}}{=} \sum_{H \xrightarrow{l} H'} l.P(H') + \sum_{H \xrightarrow{\tau} H''} \tau.P(H'') \\
P(H) & \overset{\text{def}}{=} \sum_{H \xrightarrow{l} H'} l.P(H') \\
\mathcal{P} & \overset{\text{def}}{=} \{ P(H) \mid H \text{ is well-formed} \} \cup \\
& \quad \{ P_{GC}(H) \mid H \text{ is well-formed} \}
\end{align*}
\]

A heap process is simply the sum of all \( l \) transitions and (for the garbage collecting case) the \( \tau \) transitions possible for a certain heap \( H \). Note that we use same notation for transitions made by a heap and a heap process. This should however not lead to any confusion since the actual meaning of a transition is still the same. In order to introduce bisimilarity we need to recapitulate some standard notions from the process calculus literature [63].
Definition 4.3.4 The relations $\Rightarrow$ and $\downarrow$ are defined as follows:

$\Rightarrow \overset{\text{def}}{=} \tau \rightarrow^*$

$\downarrow \overset{\text{def}}{=} \Rightarrow \lor \downarrow \Rightarrow$

Definition 4.3.5 Let $S$ be a binary relation over $\mathcal{P}$, then $S$ is a weak simulation if and only if, whenever $P SQ$,

- if $P \xrightarrow{\tau} P'$ then there exists $Q' \in \mathcal{P}$ such that $Q \Rightarrow Q'$ and $P' SQ'$.
- if $P \xrightarrow{l} P'$ then there exists $Q' \in \mathcal{P}$ such that $Q \Rightarrow Q'$ and $P' SQ'$.

A binary relation $S$ over $\mathcal{P}$ is said to be a weak bisimulation if both $S$ and its converse are weak simulations. $P$ and $Q$ are weakly bisimilar, weakly equivalent, or observation equivalent (written $P \approx Q$) if there exists a weak bisimulation $S$ such that $PSQ$.

We now want to show that our two definitions of heap processes are weakly equivalent. This is captured by the following theorem.

Theorem 4.3.5 If $H \equiv H'$, then $P_{GC}(H) \approx P(H')$

Proof We will show that the following is a weak bisimulation:

$S \overset{\text{def}}{=} \{(P_{GC}(H), P(H')) \mid H \equiv H'\}$

Since $H$ and $H'$ are well-formed, we show that each transition $\xrightarrow{\tau}$ or $\xrightarrow{l}$ of $P_{GC}(H)$ corresponds a matching transition $\Rightarrow$ or $\Rightarrow l$, respectively, of $P(H')$ (1), and vice versa (2).

We can make the proposition more precise. We already know that $\rightarrow$ preserves well-formedness (Lemma 4.3.1). For the two directions we have:

(1) If $H \equiv H'$ and $P_{GC}(H) \xrightarrow{\tau} P_{GC}(H')$ then $P(H') \Rightarrow P(H')$ (i.e. $\Rightarrow$ is empty). Thus, what we need to show is that if $H \xrightarrow{\tau} H'$ then $H \equiv H'$.

On the other hand, if $H \equiv H'$ and $P_{GC}(H) \xrightarrow{l} P_{GC}(\hat{H})$, then $P(H') \xrightarrow{l} P(\hat{H})$. I.e., what we will show is that, if $H \equiv H'$ and $H \xrightarrow{l} \hat{H}$ then $H' \xrightarrow{l} \hat{H}$ and $\hat{H} \equiv \hat{H}'$. 
4.3. Correctness of the Garbage Collector

(2) Since \( P(H) \) cannot take \( \tau \) transitions we only need to show that, if \( H \equiv H' \) and \( P(H) \xrightarrow{\tau} P(\hat{H}) \) then \( \hat{P}_\text{GC}(H') \Rightarrow P_\text{GC}(H') \) and \( \hat{H} \equiv \hat{H}' \). This is similar to what we have in case (1).

Thus, if \( H \) and \( H' \) are well-formed and \( H \equiv H' \):

(i) if \( H \xrightarrow{\tau} H'' \) then \( H'' \equiv H \).

(ii) if \( H \xrightarrow{l} \hat{H} \) then \( H \xrightarrow{l} \hat{H}' \) and \( \hat{H} \equiv \hat{H}' \). However, from Lemma 4.3.4 follows that this is true for all possible \( l \xrightarrow{} \) transitions.

What remains is to show (i).

We proceed by case study on the clauses defining \( \tau \xrightarrow{} \).

**START, SCANSTART, SCANINT, SCANINT, SCANRESTART, SCANDONE:**

Since new paths neither are created nor removed, (i) is true.

**DONE:**

Since \( H \) is well-formed, we know that \( B \) does not contain any addresses to \( W \). Thus, since the root is in \( B \), there does not exist any path such that one may reach \( W \). Hence, paths are neither created nor removed, i.e. (i) is true.

**FORWARD:**

Since \( y \mapsto \bullet \) we have \( \forall p \cdot \text{read}(H,y,p) = \text{read}(H,z,p) \). Thus, no path is created (nor removed), i.e. (i) is true.

**COPYSTART, COPYWORD, COPYRESTART:**

Since \( \#p \cdot \text{read}(H,r,p) = z \), paths are neither created nor removed, i.e. (i) is true.

**COPYDONE:**

Let \( \text{read}(H,r,p) = x, \text{read}(H,r,p') = y, \) and \( i = |V| \). Then, \( \text{join}(H,r,i : p,p') \) is true. For \( H' \), if \( \text{read}(H',r,p') = y \) and \( \text{read}(H',r,i : p) = z \) then since \( H'(y) = \bullet z \), \( \text{join}(H',r,i : p,p') \) is true. Since for any path \( p'' \) we have \( \text{read}(H',y,p'') = \text{read}(H',z,p'') \) and \( H(y) = H(z) = H'(z) \), read equivalence is upheld by the transition. Thus, (i) is true.

The termination and soundness proofs of the garbage collector are now complete.
4.4 Usefulness

In this section we show that our algorithm actually performs garbage collection. Definition 4.4.1 defines, for a root address $r$, the live part of a heap. We consider a node as live if it is reachable from the root, or if there exists a node containing the locked scan pointer with the address of the node as its index.

Definition 4.4.1 *(Live heap).* The live portion $L(H)$ of a heap $H$ is defined as:

$$L(H) = \{ y \mid \exists p. \text{read}(H, r, p) = y \lor \exists x, V, V'. H(x) = (V \uparrow_y V') \}$$

As the dual of the live portion of the heap we also define the dead portion.

Definition 4.4.2 *(Dead heap).* The dead portion $D(H)$ of a heap $H$ is defined as:

$$D(H) = \text{dom}(H) - L(H)$$

The first and crucial step towards proving usefulness of the garbage collector is to show that a dead node can never become live again.

Lemma 4.4.1 If $H \rightarrow^* H'$ then $D(H) \cap L(H') = \emptyset$.

Proof By induction on the sequence $H \rightarrow^* H'$.

Base case: $H \rightarrow^* H$. From Definition 4.4.1 and 4.4.2 follows that $L(H) \cap D(H) = \emptyset$. Furthermore, looking at the last transition in the sequence $(\hat{H} \rightarrow H')$, it is clear that for all cases $L(H') \cap D(H) = \emptyset$.

We can now proceed to the concluding theorem (Theorem 4.4.2), which states that if a node has been copied (i.e. there exists a forwarding node to the new copy), the address of its corresponding forwarding node is not in the dead part of the original heap (i.e. when the garbage collector started). We begin by defining the set of forwarding nodes (Definition 4.4.3).

Definition 4.4.3 *(Forwarding nodes).* $F(H) \overset{def}{=} \{ x \mid H(x) = \bullet y \}$
4.4. Usefulness

**Theorem 4.4.2 (Usefulness).** If $H_0 \longrightarrow^* H_n$ forms a GC cycle, then
\[ \forall H_i, 0 \leq i \leq n. \, F(H_i) \cap D(H_0) = \emptyset \]

**Proof** By induction on the sequence $H_0 \longrightarrow^* H_i$.

Base case: $F(H_0) \cap D(H_0)$, since $F(H_0) = \emptyset$, $F(H_0) \cap D(H_0) = \emptyset$.

Induction hypothesis: $F(H_{i-1}) \cap D(H_0) = \emptyset$

We proceed by case study on the last transition in the sequence, i.e. $H_{i-1} \longrightarrow H_i$.

**CopyDone:**

$F$ is extended with $y$ in this transition, so we need to show that $y \notin D(H_0)$.

Since $y$ is reachable from $x$ in $H_{i-1}$ we proceed by looking at $x$. $x$ can either be a new allocation or $\exists x', H_{i-1}(x') = x$.

For the first case:

Since $y$ has been introduced to $x$ by a mutation it follows from Lemma 4.4.1 that $y \notin D(H_0)$.

For the latter case:

From the induction hypothesis follows that $x' \notin D(H_0)$. Since $y$ is reachable from $x'$ in $H_{i-1}$, it follows from Lemma 4.4.1 that $y \notin D(H_0)$.

For all other cases:

Since $F(H_i) = F(H_{i-1})$ (or $F(H_i) = \emptyset$ for DONE) it follows from the induction hypothesis that $F(H_i) \cap D(H_0) = \emptyset$.

From Theorem 4.4.2 follows that if a node is dead when the garbage collector starts, it will not get copied. Thus, we can conclude the usefulness property in Corollary 4.4.3.

We first define what we mean by the size of a heap.
Definition 4.4.4 (Size of a heap). The size of a heap is defined as follows:

$$\text{size}(H) = \sum_{o \in \text{rng}(H)} |o|$$

where $|o|$ is the number of values in $o$

(zero for a forwarding node).

Let $\text{live}(H) \overset{\text{def}}{=} \{x \mapsto o \mid x \in L(H) \text{ and } x \mapsto o \in H\}$.

Corollary 4.4.3 Suppose $H \rightarrow^* H'$ is a GC cycle then,

$$\text{size}(H') \leq \text{size(\text{live}(H) \cup \text{new allocations})}$$

4.5 Summary

We have presented an incremental copying garbage collector model based on a set of atomic transitions (increments), and we have shown that it is both deterministic and terminating. We have also proved that a mutator-oriented model based on a process calculus with labeled transitions behaves the same with and without the internal garbage collector transitions (soundness). Finally, we have shown that our garbage collector actually does recover unreachable portions of the heap, a property we call usefulness.
4.5. Summary

\[ \text{weight}(H) \overset{\text{def}}{=} (w(H), w'(H), w''(H)) \]

\[ w(H) \overset{\text{def}}{=} \sum_{x \in \text{dom}'(H)} (w(H, x) + w(H, H(x))) \]

\[ \text{dom}'(H) \overset{\text{def}}{=} \{ x \mid x \in \text{dom}(H) \land \exists y, V, V'. H(y) = \{ V \uparrow_x V' \} \} \]

\[ w(H, \langle V \rangle) \overset{\text{def}}{=} \sum_{v \in V} w(H, v) \]

\[ w(H, \bullet x) \overset{\text{def}}{=} 0 \]

\[ w(H, n) \overset{\text{def}}{=} 1 \]

\[ w(W|G|B, x) \overset{\text{def}}{=} \begin{cases} 
2 & \text{if } x \in \text{dom}(B) \\
3 & \text{else if } x \in \text{dom}(G) \\
4 & \text{else if } x \in \text{dom}(W) 
\end{cases} \]

\[ w'(H) \overset{\text{def}}{=} \begin{cases} 
\text{if } \exists x, V, V'. H(x) = \{ V \downarrow V' \} \lor H(x) = \{ V \uparrow V' \} \text{ then} \\
\infty & \text{if } \dot{x} \text{ then} \\
|V'| & \text{else} \\
\infty & \text{else} 
\end{cases} \]

\[ w''(H) \overset{\text{def}}{=} \begin{cases} 
\text{if } \exists x, V, V', y, z. H(x) = \{ V \downarrow_x y, V' \} \text{ then} \\
\infty & \text{if } \dot{y} \text{ then} \\
|H(y)| - |H(z)| & \text{else} \\
\infty & \text{else} 
\end{cases} \]

Figure 4.4: Weight of a heap
In Chapter 4, we described a model of an incremental copying garbage collector based on process terms in a labeled transition system. Garbage collector increments were represented by internal transitions and mutator interactions were represented by labeled transition. A labeled transition was a read, a write, or an allocation. In this chapter we describe how this abstract process model is implemented as part of the Timber run-time kernel.

From the garbage collector’s point of perspective, it really does not matter what kind of mutator it is set out to garbage collect for, as long as it follows the rules of the labeled transitions. Even though mutator access is defined by the path notion, it is clear that, in reality, mutator access will commonly be made from pointer fields directly accessible at the place in the execution of the program. Nonetheless, each such access must correspond to a path-based labeled transition beginning in the root-set of the system. Otherwise, the pointer field would not have been locally accessible by the program in the first place. In other words, we assume the program to be well-behaved in terms of aliveness (i.e., \( \text{live} \subseteq \text{reachable} \)).

**Definition 5.0.1** Let a mutator be well-behaved iff it can be described by a relation \( R \) on heaps such that for each \( HRH' \) there exists at least one labeled transition \( H \xrightarrow{l} H' \).
5.1 The memory set up

Since the algorithm is a copying garbage collector, we need to divide the heap into two parts, tospace (which is the active part of the heap), and fromspace (which is the inactive one). It is not entirely true that these spaces need to be of the exact same size even though it simplifies the underlying intuition. Copying objects from one space into another space of exactly the same size will always succeed. If the objects do not fit into the new space it would not have fit in the old space either. However, the actual need of space to copy to is not dependent on the size of the old space but rather on how much memory the live objects in the old space will require. A survival rate of ten percent would require the new space to be at least a tenth of the size of the old space. Anyhow, allowing the border between tospace and fromspace to slide requires a more rigorous (read cumbersome) scheme for both allocations and free memory calculations. We will use a fixed, equally sized, partitioning of the heap. In Figure 5.1, the memory set up of our implementation is shown.

![Figure 5.1: The memory set up for our implementation.](image)
5.2 Implementation details

In this section we will look at the interesting details of the implementation. Most of the internal transitions, defined in Figure 4.1 and 4.2, are straightforwardly implemented (e.g. by adjusting a pointer or index). However, some of the details deserve some extra attention.

5.2.1 The info-field

Since Timber is a strictly typed language the locations of pointers can safely be determined a priori. Thus, whenever a new allocation is requested, the positions within the new node that can hold pointers are known, and the necessary type information can be attached to each node\(^1\). This is done by statically create an array of pointer offsets for each unique type, and store a pointer to the corresponding array as a header in each node when it is allocated.

5.2.2 The write barrier

The write-barrier needs to do two things. First of all it needs to set the dirty bit of the mutated node (we will see how this is implemented efficiently in Section 5.2.3). The second thing is to; if the node is black, revert its color back to gray. We do not want to actually move the node just to be on the right side of the border. That would cost too much, and even more alarming, the pause time of the write-barrier would be dependent upon the size of the node. What we do instead is that we allocate a certain node in the gray heap that, when the scanning process reaches the node, notifies the scanner that a black node has been mutated. The scanner then temporarily jumps back to the black node, scans it, and returns back to where it was. These notifier nodes can easily be detected by using a unique type identifier (instead of the array pointer used for regular nodes).

\(^1\)We use the term node instead of object in order to avoid confusion between objects on the heap and Timber objects
5.2.3 The dirty bit

Another feature that could induce unnecessary overhead is the dirty bit. However, the only nodes where the dirty bit matters is the two nodes currently being scanned and currently being copied. Whenever the mutator needs to set the dirty bit of a node it can check if the node is the one currently being scanned or the one currently being copied. If the node is the one that is currently being scanned, a static dirty bit field for the current node being scanned is set. At each iteration of the scanning process, this field is checked and if set take the ScanRestart transition. Similarly, a static dirty bit for the node currently being copied is checked at each iteration of the copy process and if set take the CopyRestart transition. This requires two extra comparisons to be done by the write-barrier. In the next section we will look at how the frequency of write-barrier calls may be reduced significantly due to the concurrency model of Timber.

5.2.4 Reducing the frequency of write-barrier calls

We know that the only mutable data structure in Timber is the object. We further know that the only way to gain access to these mutable state fields is through the methods of the object. Since we assumed that the garbage collector has the lowest priority in the system we can safely say that, at all times, if a method successfully acquires a lock, it will return the very same lock before the garbage collector is started or resumed. This is immediate from the fact that messages in Timber are non-blocking. Thus, when a message acquires a lock within a garbage collection cycle then, and only then, is the write-barrier for that object called. In other words, since the garbage collector never may interleave the execution of a message, only one write-barrier call for the whole message is necessary.

5.2.5 The read barrier

The read transition assumes that forwarding pointers always are followed. However, a read barrier is only necessary for data structures that might suffer from inconsistencies
due to mutations. Read only data does not need to be protected by a read barrier. Similarly to the write barrier, the read barrier can also be called only once for every method invocation instead of upon every single memory access. The efficiency gained here is probably even higher than for the write barrier since reads tend to occur much more frequently than writes.

5.3 Garbage collection in a real-time context

A Timber program may create reactive objects dynamically at run-time, just like any other data structure. In schedulability theory terms, this feature corresponds to the ability to dynamically create both tasks and shared resources. In order to achieve a priori schedulability guarantees, we need to limit the expressiveness of Timber somewhat. A sufficient (but probably not absolutely necessary) way to support static schedulability analyses would be to restrict the program from dynamically creating objects. That is, all shared mutable data (objects) and all tasks (methods) would be statically known. If this restriction is set, which all in all is not that controversial in the context of hard real-time systems, it puts the garbage collector in a much more favorable situation. If objects are statically allocated outside the heap, it means that the heap will contain no mutable data structures at run-time. That, in turn, makes both the read barrier and the write barrier superfluous. The same holds for the dirty bits. We will return to this in Chapter 7.
As we have seen in the previous chapters, amount of live (reachable) heap space is the key contributor to running times of tracing garbage collectors, in particular for a copying collector. In contrast to cumulative behaviors – such as execution time and heap allocation – live heap space exhibits a behavior strongly connected to the current state of the program execution.

Recent years have seen a respectable development in techniques for analysis of live heap space usage of traditional programs [3, 46, 26, 87]. 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, real-time systems (as those described in Chapter 2) do not fit such a traditional programming model. Instead of terminating with a result, a real-time task typically maintains an ongoing interaction with its environment at periodic intervals or as responses to sporadic events. Moreover, tasks 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 amount of live heap space in the system.
In this chapter we present a technique for lifting live heap space analysis to the real-time programming model, sufficiently restricted to make static analysis feasible, but still expressive enough to fit a large class of real-world systems.

The key observation for predicting live heap space in real-time systems is that the possible orders in which tasks may execute plays a vital role. 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 live heap space 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, live heap space analysis cannot ignore the actual interleaving of tasks that are allowed to run concurrently, unless the effect each task has on live heap space can be considered atomic.

The main contribution presented in this chapter is a technique for calculating upper bounds on live heap memory of real-time systems, which 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 6.1). This is arguably a stronger restriction than necessary to guarantee atomicity, but it is appealingly simple and "obviously" correct for our purpose.

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 [6] 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 6.2).

3. We apply a standard variant of abstract interpretation to each task for inferring size relations [29], which capture how each individual task affects an abstract notion of size for every persistent state variable (Section 6.4). 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 6.3).

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 6.5).

6.1 Our refined real-time system model

Here we define the model of execution we will work with in the rest of the dissertation. Our model connects fairly well to task models used in the real-time scheduling literature (as described in Chapter 2), while drawing its concrete inspiration from the execution principles of Timber.

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 ($P_{i}^{\text{min}}, P_{i}^{\text{max}} \in \mathbb{N}$). 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 6.1.1 A task set $\tau$ is well-formed iff $\forall t_i \in \tau$ . $0 < P_i^{\text{min}} \leq P_i^{\text{max}}$ and $0 \leq D_i \leq P_i^{\text{min}}$.

In other words, *aperiodic* tasks are excluded from our model (i.e., tasks for which $P_i^{\text{min}} = 0$), 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 $D_i > P_i^{\text{min}}$). We will return to this in Chapter 10 and what is required to enable tasks where $D_i > P_i^{\text{min}}$.

Periodic tasks are captured in this model by letting $P_i^{\text{min}} = P_i^{\text{max}}$, and fully sporadic tasks by $P_i^{\text{max}} = \infty$. 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.

### 6.1.1 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

$^1$Or arbitrarily slow, provided that all deadlines are still met.
6.1. Our refined real-time system model

any task execution times and scheduling policies. In Section 6.2 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 analysis we propose, 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 [11]: 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 our model effectively adds to SRP 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 burdensing in practice; in the Timber language it simply corresponds to limiting the use of synchronous inter-object method calls to at most one per method. 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. We will return this in Chapter 10.
6.1.2 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 the analysis presented in this chapter, external ports 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 \( t_i(\vec{x}, \vec{y}) \) that maps an input state vector \( \vec{x} \) to an output state vector \( \vec{y} \). 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 6.3.

6.1.3 Worked example

Throughout this chapter 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:

<table>
<thead>
<tr>
<th>task</th>
<th>( p_{\text{min}} )</th>
<th>( p_{\text{max}} )</th>
<th>( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>10</td>
<td>( \infty )</td>
<td>10</td>
</tr>
<tr>
<td>( b )</td>
<td>10</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

Our underlying analysis question is: what is the maximum sum of the sizes of \( x_1 \) and \( x_2 \) that ever may occur?
6.2 FSM representation

As a core technical idea of our approach we choose to express the behavior 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.

6.2.1 Real-time systems as Timed Automata

Recalling the definition in Chapter 2 (Section 2.4.3), a timed automaton is defined by a tuple $A = (L, l_0, A, C, I, E)$, 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 $A_i = (L_i, l_{0i}, A_i, C_i, I_i, E_i)$ defined as follows:

- $L_i = \{\text{idle, released}\}$
- $l_{0i} = \text{idle}$
- $A_i = \{t_i, \varepsilon\}$
- $C_i = \{c_i\}$
- $I_i = \{(\text{idle}, c_i \leq P_{i}^{\text{max}}, \text{released}, c_i \leq D_i)\}$
- $E_i = \{(\text{idle}, c_i \geq P_{i}^{\text{min}}, \varepsilon, \{c_i\}, \text{released}), (\text{released}, \text{true}, t_i, \emptyset, \text{idle})\}$

Fig. 6.1 shows the definition above in a graphical notation.

The transitions of $A_i$ 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 \( (c_i \leq P_i^{\text{max}}) \) ensures that the time between two triggering events for \( t_i \) never exceeds \( P_i^{\text{max}} \). Moreover, the guard \( (c_i \geq P_i^{\text{min}}) \) on the event transition forces the inter-arrival time to be at least \( P_i^{\text{min}} \). The invariant in the \textit{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 \( D_i \leq P_i^{\text{min}} \), 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 \( c_i \leq D_i \leq P_i^{\text{min}} \)).

The timed automaton of the whole task set \( \tau \) is then constructed by parallel composition \( A_\tau = \|_{i \in \tau} A_i \). The resulting automaton is entirely straightforward, with locations and invariants being conjunctions of the individual automata counterparts (see Chapter 2, Section 2.4.4 for further details on parallel automata composition). Fig. 6.2 shows the timed automaton of our worked example.
6.3. Rule-based representation

6.2.2 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 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. 6.3.

However, as we will see in Section 8.1, it is not necessary an improvement of performance of our analysis to make these quite costly transformations.

![Figure 6.3: Minimal FSM representation of the execution orders of the worked example.](image)

6.3 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. 6.4.

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 \(\text{p}(\overline{x}, \overline{y})\) 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 \(\text{type}(x, c)\), 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.

With the rule-based representation, the code of the tasks of our worked example looks as shown in Fig. 6.5 (assuming \(value\) is some suitable integer element). We use constructors \(\text{cons}\) and \(\text{nil}\) for building lists, where \(\text{new nil} \{\}\) denotes a "null pointer" (i.e., a zero-arity constructed value requiring no additional heap space).
6.3. Rule-based representation

\[
a((x_1, x_2), (x'_1, x'_2)) \leftarrow x'_1 := \textbf{new} \{ \text{head} := \text{value}, \text{tail} := x_1 \}, x'_2 := x_2
\]

\[
b((x_1, x_2), (x'_1, x'_2)) \leftarrow x'_2 := x_1, x'_1 := \textbf{new} \text{ nil} \}
\]

Figure 6.5: The worked example in the rule-based representation

In addition to rules defining tasks, we always include a predefined procedure \textit{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:

\[
\text{init}((x_1, x_2)) \leftarrow x_1 := \textbf{new} \text{ nil} \}, x_2 := \textbf{new} \text{ nil} \}
\]

From [2] we also adopt an operational semantics for our rule-based programs, as depicted in Fig. 6.6. A value \( v \) is either an integer constant or a tagged heap reference \( r^c \), where \( c \) is a constructor name. A heap \( \hat{A} \cdot h \) maps references to objects \( o \), which in turn are mappings from field names to values. Execution steps are described as transitions \( S \leadsto S' \), 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 (6.1) deals with evaluating expressions and storing the resulting value in the environment. We assume that function \( \text{eval}(e, \rho) \) evaluates \( exp \) in the context of \( \rho \). Rule (6.2) 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 (6.3). Rules (6.4) and (6.5) 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 \( p \).

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 \( S_0 \leadsto S_1 \leadsto \cdots \leadsto S_m \). Let \( \leadsto^* \) denote a transitive
\[ b \equiv x := e \quad v = \text{eval}(e, \rho) \]  
\[ (p, (b, bs), \rho) \cdot A : h \leadsto (p, bs, \rho[x \mapsto v]) : A : h \]  
(6.1)

\[ b \equiv x := \text{new} \{ f := e \} \quad v = \text{eval}(e, \rho) \]  
\[ \{ p, (b, bs), \rho \cdot A : h \leadsto (p, bs, \rho[x \mapsto v]) : A : h[v^\rho \mapsto [f \mapsto \emptyset]] \} \]  
(6.2)

\[ b \equiv x := y.f \quad h(\rho(x)) = o \]  
\[ (p, (b, bs), \rho) \cdot A : h \leadsto (q, bs, \rho[x \mapsto o(\rho)]) : A : h \]  
(6.3)

\[ b \equiv q(\tau, \gamma) \quad q(\tau', \gamma') \rightarrow q, bs' \text{ is a rule} \quad \rho'(\pi) = \rho(\pi) \quad \text{eval}(g, \rho') = \text{true} \]  
\[ (p, (b, bs), \rho) \cdot A : h \leadsto (q, bs', \rho') \cdot (p[\tau, \gamma], bs, \rho) : A : h \]  
(6.4)

\[ (q, \epsilon, \rho) \cdot (p[\tau, \gamma], bs', \rho') \cdot A : h \leadsto (p, bs', \rho' \mapsto \rho(\gamma')) : A : h \]  
(6.5)

Figure 6.6: Operational semantics of rule-based programs

closure of \( \leadsto \). Complete execution of a single task \( t_i \) corresponds to the trace (called complete trace) \( \langle \bot, t_i(\sigma, \sigma), \rho \rangle : h \leadsto (\bot, \epsilon, \rho') : h', \) 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.

### 6.4 Inferring size relations

The notion of size of a heap allocated object can vary depending on what exact purpose our analysis will serve. Let \( \text{size}_X(o) \) denote size of a heap allocated object \( o \), where the size is determined by the cost model \( X \) and may denote

- \( M \) memory size occupied by \( o \)
- \( R \) number of reference fields in \( o \)
- \( O \) value 1, if we are interested in finding an upper bound of a total number of objects.

The above notation also extends over sequences: \( \text{size}_X([\pi]) = [\text{size}_X(o_1), \ldots, \text{size}_X(o_n)] \).

Inferring size relations, similarly to e.g. [17, 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 [17], 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.

### 6.4.1 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 [29]. Size of a compound structure \( c \{\ldots\} \) is a sum of sizes of its elements, plus a size \( k_c^x \) of a single node, suitable for a cost model \( \mathcal{X} \). Abstract compilation proceeds over rules in the program as depicted in Fig. 6.7.

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

Given a rule-based program \( P \), its compiled version \( \text{Abs}_P[P] \) is a CLP program over real numbers (CLP(R)). We refer, for instance, to [43] for further reading on CLP. Let us assume the *model-theoretic (or algebraic) semantics*\(^2\) of CLP(R), where semantics of programs is given by means of *models over \( \mathbb{R} \) (R-models), and standard interpretation of arithmetic functions (see e.g. [42] for details). The following lemma shows the soundness of the abstract compilation. It is shown that the resulting CLP program correctly captures relation between sizes of input and output parameters of a given procedure.

\(^2\)This is an arbitrary choice made for an illustrative purpose only. All kinds of semantics of CLP coincide in some well-defined sense, so choosing any other semantics would be equally valid.
\[
\begin{align*}
\text{Abs}_P[R_1, \ldots, R_n] &= \text{Abs}_P[R_1], \ldots, \text{Abs}_P[R_n] \\
\text{Abs}_R[p(\overline{x}, \overline{y}) \leftarrow g, b_1, \ldots, b_n] &= p(\overline{x}, \overline{y}) \leftarrow \overline{x} \geq 0, \overline{y} \geq 0, \text{Abs}_{b_1}, \ldots, \text{Abs}_{b_n} \\
\text{Abs}_b[true] &= true \\
\text{Abs}_b[e_1 \text{ op } e_2] &= \text{if } e_1 \text{ and } e_2 \text{ are linear then } e_1 \text{ op } e_2 \text{ else true} \\
\text{Abs}_b[\text{type}(x, c)] &= true \\
\text{Abs}_b[x := e] &= \text{if } e \text{ is linear then } x = e \text{ else true} \\
\text{Abs}_b[x := \text{new } c \{f_i := e_i\}] &= x \leq k^X + \sum_i \text{norm}(c, f_i, e_i) \\
\text{Abs}_b[q(\overline{x}, \overline{y})] &= q(\overline{x}, \overline{y}) \\
\text{norm}(c, f, e) &= \text{if the type of field } f \text{ of a } c \text{ is integer then 0 else } e
\end{align*}
\]

**Figure 6.7: Abstract compilation to CLP**

**Lemma 6.4.1** Given a program \( P \) and procedure \( p \), assume the trace \( (q, (p(\overline{x}, \overline{y}), bs), \rho) \) : \( h \sim^* (q, bs, \rho') : h' \). The atomic formula \( p(k, l) \) where \( k = \text{size}_X(h(\rho(\overline{y}))) \) and \( l = \text{size}_X(h(\rho'(\overline{y}))) \) belongs to the least \( R \)-model of \( \text{Abs}_P[P] \).

**PROOF:** By induction over depth of recursion in \( P \).

Assume that \( k_{\text{cons}}^M = 3 \) and \( k_{\text{null}}^M = 0 \). The abstractly compiled worked example is shown in Fig. 6.8

**Figure 6.8: 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 \( \phi_p[\mathbf{x}, \mathbf{y}] \) if we want to make the involved variables explicit). See [17] for the details of the fixpoint iteration algorithm.

The following theorem states soundness of size relation inference.

**Theorem 6.4.2** Given a trace \( \langle \bot, t_i(\sigma, \sigma), \rho \rangle : h \sim^* \langle \bot, \epsilon, \rho' \rangle : h' \), the vector pair 
\[ \text{size}_X(h(\rho(\sigma))), \text{size}_X(h'(\rho'(\sigma))) \]
is true.

**PROOF:** By combining Lemma 6.4.1 and the soundness of size relation analysis of [17].

### 6.4.2 Example

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

\[
app(\langle x, y \rangle, \langle \rangle) \leftarrow x = \text{null}(), z := y
\]

\[
app(\langle x, y \rangle, \langle z \rangle) \leftarrow x \neq \text{null}(), x' := x.\text{tail}, \ app(\langle x', y \rangle, z'),
\]
\[
z := \textbf{new cons}(x.\text{head}, z')
\]

Abstract compilation of the above two rules, with respect to the cost model \( O \), results in

\[
app(\langle x, y \rangle, \langle z \rangle) \leftarrow x \geq 0, y \geq 0, z \geq 0, y = z
\]

\[
app(\langle x, y \rangle, \langle z \rangle) \leftarrow x \geq 0, y \geq 0, z \geq 0, x' \geq 0, z' \geq 0, x' \leq x - 1,
\]
\[
app((x', y), z'), z \leq z' + 1
\]

Observe that \( z := \textbf{new cons}(x.\text{head}, z') \) has been compiled to \( z \leq z' + 1 \) rather than \( z = z' + 1 \), due to possible sharing. Computing bottom-up fixpoint over convex polyhedra domain, as described in [17], gives the final size relations:

\[
app(\langle x, y \rangle, \langle z \rangle) \leftarrow x \geq 0, y \geq 0, z \geq 0, z \leq x + y
\]
6.5 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 \(m(x, y)\) we infer size relations \(\phi_m\) which in the matrix form can be written as

\[
Y \leq A_m X + C_m, \quad X \geq 0
\]  

(6.6)

where \(X = [x_1, \ldots, x_n]\), \(Y = [y_1, \ldots, y_n]\), \(n\) is a number of states variables. For an initialization method \(\text{init}\) (which has no input parameters) the constraints take form:

\[
X_0 \leq C_{\text{init}}, \quad X_0 \geq 0
\]  

(6.7)

Thus the vector \(C_{\text{init}}\) describes sizes of initial values of the state variables. The size relation matrices with respect to cost model \(M\) for the worked example look like the following:

\[
A_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad C_a = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \quad A_b = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \quad C_b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

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

\[
\hat{X}_j \geq A_m \hat{X}_i + C_m
\]  

(6.8)
and for the initialization
\[ \hat{X}_0 \geq C_{\text{init}} \quad (6.9) \]

For the \( \varepsilon \)-transitions we have \( A_\varepsilon = I_n \) (the \( n \times n \) unit matrix) and \( C_\varepsilon = 0 \).

We require the size relation matrices \((A's)\) 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 \( A \) is a relaxation and always a safe step to do. The reason for this requirement is that \( \hat{X}_i \) and \( \hat{X}_j \) represent upper bounds, which means that the inferred constraints for \( \hat{X}_j \) must be safe for any sizes between 0 and \( \hat{X}_i \).

Let \( i_0 \xrightarrow{m_0} i_1 \xrightarrow{m_1} i_2 \xrightarrow{m_2} \cdots \) be a run of the state machine. With every step \( k \) we assign variables \( X_k \) denoting sizes of state variables in \( k \), in according to (6.6) and (6.7).

**Lemma 6.5.1** 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 (6.8) + (6.9), any \( k \) we have \( \hat{X}_{i_k} \geq X_k \).

**PROOF:** Inductive over \( k \).

**Base case:** Trivially holds by combining (6.7) and (6.9).

**Inductive step:** By inductive assumption we know that \( \hat{X}_{i_{k-1}} \geq X_{k-1} \), and by the fact that \( A_{m_{k-1}} \) contains only non-negative values we conclude that
\[ A_{m_{k-1}} \hat{X}_{i_{k-1}} + C_{m_{k-1}} \geq A_{m_{k-1}} X_{k-1} + C_{m_{k-1}} \]

By combining the above with (6.6) and (6.8) we can observe that \( \hat{X}_{i_k} \geq X_k \), which concludes the proof.

Lemma 6.5.1 suggests the way to compute upper bounds of state variable sizes. In addition to (6.9) and (6.8) we add \( \hat{X} \geq H \cdot \hat{X}_i \) for every state \( i \); where \( H = [h_1, \ldots, h_n] \) and \( h_i = 1 \) if \( s_i \) is heap allocated; \( h_i = 0 \) otherwise. Let the cost function \( c = \hat{X} \) and \( c^* \) denote its minimum value. The following theorem states soundness of the analysis.
Theorem 6.5.2 Let \( T \) be a set of all complete traces, over all feasible (possibly infinite) schedules. The following holds:

\[
\max \left\{ \sum \text{size}_X(h'(\rho'(y))) \mid (\perp, p(\overline{\tau}, \overline{y}), \rho) : h \xrightarrow{\tau} (\perp, \epsilon, \rho') : h' \in T \right\} \leq c^* \\
\]

PROOF: Follows directly from Theorem 6.4.2 and construction of (6.8) and (6.9), and Lemma 6.5.1.

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

\[
\begin{align*}
)x_1 & \geq x_{1_0} + 3 & x_2 & \geq x_{2_0} & x_2 & \geq x_{1_0} & x_1 & \geq x_{1_1} + 3 \\
x_2 & \geq x_{2_1} & x_3 & \geq x_{1_1} & x_4 & \geq x_{1_2} + 3 & x_2 & \geq x_{2_2} \\
x_2 & \geq x_{1_2} & x_1 & \geq x_{1_3} + 3 & x_2 & \geq x_{2_3} & x_2 & \geq x_{1_3} \\
x_1 & \geq x_{1_4} + 3 & x_2 & \geq x_{2_4} & x_2 & \geq x_{1_4} & x_2 & \geq x_{1_5} \\
\end{align*}
\]

\[
\begin{align*}
x_1 & \geq 0 & x_1 & \geq 0 & x_2 & \geq 0 & x_2 & \geq 0 \\
x_1 & \geq 0 & x_2 & \geq 0 & x_2 & \geq 0 & x_2 & \geq 0 \\
x_2 & \geq 0 & x_2 & \geq 0 & x_2 & \geq 0 & x_2 & \geq 0 \\
\hat{X} & \geq x_{1_0} + x_{2_0} & \hat{X} & \geq x_{1_1} + x_{2_1} & \hat{X} & \geq x_{1_2} + x_{2_2} \\
\hat{X} & \geq x_{1_3} + x_{2_3} & \hat{X} & \geq x_{1_4} + x_{2_4} & \hat{X} & \geq x_{1_5} + x_{2_5}
\end{align*}
\]

6.6 Summary

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 achieved by representing the task set as a timed automaton and then applying 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.
7.1 Scheduling the GC

The problem of scheduling garbage collection in realtime systems is twofold. The requirements put on the garbage collector scheduler are:

1. the garbage collector must not cause any task to miss its deadline, and

2. the system must not run out of memory.

One can easily see that fulfilling one of the requirements may cause a failure to meet the other. For example, to avoid running out of memory, the garbage collector needs to run, which in turn may cause a task to fail meeting its deadline. This scheduling problem is not easily solved and it becomes even more difficult in our case because we use a copying collector, which, conceptually, has to finish before any garbage memory can be reused.

7.1.1 Idle time GC

In the general case, finding and scanning the root-set incrementally is a very difficult task. The problem is that the root-set is not constant. Finding all roots requires scanning of,
Garbage Collection Demand Analysis

not only static fields and CPU registers, but also the run-time stacks. Since the depth and content of the run-time stacks are not constant, the cost of scanning them is bound to be unpredictable. Scanning them incrementally is also deemed to be notoriously hard due to their volatile nature. In the worst case, the scanning process would need to be restarted at each increment since the content of the stacks may have been altered altogether. The idea of possibly restart the scanning process of an arbitrary sized root-set at each increment is not very appealing, especially not when it comes to real-time systems.

In order to remedy this problem, we will restrict the garbage collector to only run at idle time. Although this policy has been considered inferior to other approaches (in terms of performance), it comes with a couple of substantial advantages.

**Constant time root-set scanning**

In a reactive system, such as Timber, the root-set during idle time is very small. The queue of pending messages is empty and no run-time stacks exist. In fact, the root-set consists of only two elements, the interrupt vector and the queue of delayed messages. The sizes of these data structures are directly connected to the number of tasks defined. Since periodic task releases are achieved by posting a delayed message for the next instance of the task, each periodic task will contribute with one message in the queue of delayed messages. Tasks that are connected to external events (hardware interrupts) have their corresponding messages in the interrupt vector. Thus, if the set of tasks is statically known, the size of the root-set is also statically known.

**Schedulability analysis is preserved**

Since the garbage collector is the lowest priority process in the system it will never compete for CPU time with the real-time tasks. Nonetheless, if the garbage collector is running (which must be assumed in the worst case) some extra overhead is put on the tasks due to synchronization issues. Furthermore, even though the garbage collector is incremental, the longest atomic increment must be taken into account as an execution
7.2 GC overhead

Restricting the garbage collector to only run at idle time reduces the problem of determining the time overhead due to garbage collection significantly. The induced overhead can be divided into two parts, the cost of actually interrupting the garbage collector while it is running, and the cost of synchronizing with garbage collector (read and write barriers).

7.2.1 Longest atomic increment of the GC

The longest atomic increment of the garbage collector is easily determined directly from the algorithm definition (Fig. 4.1 and 4.2). Since none of the transitions are dependent on any variable-sized data the cost is constant in terms of number of instructions to execute.

7.2.2 Synchronization

Generally, the need of synchronization for an incremental garbage collector is to preserve the tri-color invariant [30]. However, for a copying garbage collector the situation is a little bit trickier. Not only is the synchronization mechanism needed to preserve the liveness view of the heap but also for assuring that mutators access the new copies (if they exist) of heap objects. The synchronization between mutators and the garbage collector typically boils down to so called barrier methods: one for reading and one for writing heap data.

The reactive objects of Timber enable us to reduce these synchronizations substantially. The reason is that the only mutable data structures are the objects. Immutable
data, as pointed out already by Doligez and Leroy [31] as well as Huelsbergen and Larus [38], requires no synchronization actions.

Since access to objects must be made under mutual exclusion, a combined read/write barrier can be used that is invoked as part of the lock operation. That is, instead of calling a barrier method for every heap access, we now only need to invoke the barrier when an object is locked. Since most methods only lock one object, the combined read/write barrier is called only once per method invocation.

### 7.2.3 Hard realtime systems

Since objects should contain at least one method in order to be meaningful, the concept of dynamic object creation ultimately leads to dynamic creation of tasks. However, generally speaking, in order to determine schedulability, dynamic creation of tasks cannot be allowed. All tasks must be statically known. Thus creating objects dynamically cannot be allowed. These mutable objects can then be allocated static. What is left in the dynamic heap is now only immutable data that does not require any synchronization. Synchronization overhead is hence eliminated altogether. In Fig. 7.1, an example of how such a system is arranged in memory is shown.

![Figure 7.1: Example of a Timber system with only static objects.](image)
7.3 Schedulability of the GC

We have so far shown that the first requirement of garbage collection scheduling is fulfilled by our collector. We have accomplished this by restricting the garbage collector to only run at idle time. In this section we will look at what is required in order to fulfill the second requirement.

7.3.1 Garbage collection demand analysis

Baruah et al. presents a feasibility test for realtime tasks called processor demand analysis [14, 13]. We present a feasibility test for garbage collection in realtime systems called garbage collection demand analysis. In contrast to checking that the processor demand for all possible time windows are less than the size of the window, our analysis determines the required size of the window such that the garbage collection demand is less than the size of the window. If such a window size can be found and the required memory demands of that window size can be met, our garbage collector can feasibly be schedule in the system.

We will only look at the hard realtime case for the analysis, which means no mutable data will be present on the heap and the transitions ScanRestart and CopyRestart will never be taken.

We begin by looking at the execution time of the garbage collector if it is allowed to run without interruptions, and then we look at the cost contributed by tasks interrupting the collector.

The garbage collector transitions are defined in Fig. 4.1 and 4.2. The Start and Done transition is only taken once per garbage collection cycle. The number of transitions of each kind can be derived from three basic parameters of the heap to be collected:

\[ M \]: Amount of reachable memory to copy (in words).

\[ O \]: Number of reachable nodes on the heap.

\[ R \]: Number of reachable references on the heap.
These parameters correspond to the three cost models for live heap objects presented in Chapter 6 (Section 6.4).

We can now formulate the execution time of garbage collection ($T_{gc}$) in terms of these parameters and execution times of each transition (denoted by $T_{<\text{name of transition}>}$). Since, in the real implementation, we only store information about offsets to reference fields in heap nodes, the cost of $\text{SCANINT}$ is not present.

$$T_{gc} =$$

$$M \times T_{\text{COPYWORD}} +$$

$$O \times (T_{\text{SCANSTART}} + T_{\text{SCANDONE}} + T_{\text{COPYSTART}} +$$

$$T_{\text{COPYDONE}})$$

$$+ R \times T_{\text{SCANADDR}}$$

$$+ (R - O) \times T_{\text{FORWARD}}$$

$$+ T_{\text{START}} + T_{\text{DONE}}$$

$$= M \times T_{\text{COPYWORD}} +$$

$$O \times (T_{\text{SCANSTART}} + T_{\text{SCANDONE}} + T_{\text{COPYSTART}} +$$

$$T_{\text{COPYDONE}} - T_{\text{FORWARD}}) +$$

$$R \times (T_{\text{SCANADDR}} + T_{\text{FORWARD}})$$

$$+ T_{\text{START}} + T_{\text{DONE}}$$

Due to the fact that the tasks never can cause anymore copying work for the garbage
7.3. Schedulability of the GC

collector\(^1\) and we only have immutable data on the heap, the only extra garbage collection cost of interruptions is due to new allocations. Similarly to the heap parameters above we have three parameters due to new allocations during garbage collection:

\( A^M_i \): Amount of new memory allocated by task \( i \) (in words).

\( A^O_i \): Number of nodes allocated by task \( i \).

\( A^R_i \): Number of references allocated by task \( i \).

We can now formulate the garbage collector execution time due to new allocations made by task \( i \) (\( T^A_i \)).

\[
T^A_i =
\]

\[
A^O_i \ast \left( T^\text{ScanStart} + T^\text{ScanDone} \right) +
\]

\[
A^R_i \ast T^\text{ScanAddr} +
\]

\[
\left( A^R_i - A^O_i \right) \ast T^\text{Forward}
\]

\( (7.2) \)

Now we will look at the total garbage collection demand for a time window of size \( t \). For the sake of simplicity we will assume that the heap and allocation parameters are constants. Even though this is not true in reality (reachable memory, allocation rates, etc. tend to vary over time) we can always assume the worst case. We will return to the validity of this assumption at the end of this section.

\(^1\)This is because new data is allocated in tospace and garbage nodes can never become reachable again (see Lemma 4.4.1).
Let $n$ be the number of tasks, $T_i^A$ be the garbage collection execution time due to allocations made by task $i$, $\left\lceil \frac{t}{P_{i_{\text{min}}}} \right\rceil$ the maximum number of releases of task $i$ in a time window of size $t$, $C_i$ the execution time of task $i$, and $P_{i_{\text{max}}}$ the minimum inter-arrival time of task $i$. We can now formulate the total garbage collection demand ($C_{gc}$) in a time window of size $t$ as follows.

$$C_{gc}(t) = T_{gc} + \sum_{i=1}^{n} \left\lceil \frac{t}{P_{i_{\text{min}}}} \right\rceil \ast (C_i + T_i^A)$$ (7.3)

Informally, $C_{gc}$ consists of two independent parts. One accounting for the execution time required to garbage collect the heap, as if undisturbed. The second part accounts for the extra time induced and consumed by tasks interrupting the garbage collector.

**Theorem 7.3.1** For any $t$ such that $C_{gc}(t) \leq t$ the garbage collector can be scheduled feasibly (w.r.t. time) with a period of $t$.

**Proof** By contradiction.

The memory needs of the system is formulated as a function of the period $t$ of the garbage collector.

$$M_{tot}(t) = 2 \ast \left( M + \sum_{i=1}^{n} \left\lceil \frac{t}{P_{i_{\text{min}}}} \right\rceil \ast A_i^M \right)$$ (7.4)

In order to complete the proof of correctness of the feasibility test, we need to show that it is sufficient to test feasibility of the worst case of heap and allocation parameters.

**Lemma 7.3.2** If the garbage collector can feasibly be scheduled with a period of $t$ for a system with heap parameters $M$, $O$, and $R$, and for all tasks $i$, allocation parameters $A_i^M$, $A_i^O$, and $A_i^R$. Then, for any $M' \leq M$, $O' \leq O$, $R' \leq R$, and for all tasks $i$, $A_i^{M'} \leq A_i^M$, $A_i^{O'} \leq A_i^O$, and $A_i^{R'} \leq A_i^R$, the garbage collector can still be feasibly scheduled with a period of $t$.

**Proof** Follows directly from the fact that Equation 7.1 and 7.2 are monotonic.
7.4 Summary

We have shown how the reactive object model of Timber enables us to decouple the schedulability analysis of the real-time tasks from the cost of garbage collection. We have, based on the incremental copying garbage collector presented in Chapter 4 and the real-time programming language Timber [68], developed a feasibility test for the collector. We call this test *garbage collection demand analysis*, which contains only clearly identified parameters of the real-time system and their corresponding effect on garbage collection time. In chapter 8, we will confirm the validity of the model through an experimental study run on the LPC2468 Developer’s Kit from Embedded Artists.
8.1 Live heap space analysis

Our live heap space analysis (presented in Chapter 6) 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.

It is well-known that the number of zones is exponential to the number of clocks present in the timed automaton [28]. That is, in our case, we have an exponential growth of zones with respect to number of tasks. In Figure 8.1, the zone graph and minimal FSM sizes for the following example task sets are shown.
### Experimental results

#### Table 8.1: Zone graph and minimal FSM sizes of the example task sets.

<table>
<thead>
<tr>
<th>Task set</th>
<th># states in zone graph</th>
<th># states in minimized FSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>1255</td>
<td>1</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>200</td>
<td>699</td>
</tr>
<tr>
<td>( \tau_3 )</td>
<td>10368</td>
<td>3393</td>
</tr>
<tr>
<td>( \tau_4 )</td>
<td>12968</td>
<td>6343</td>
</tr>
</tbody>
</table>

*Task set \( \tau_1 \) :*

<table>
<thead>
<tr>
<th>task</th>
<th>( P_{\text{min}} )</th>
<th>( P_{\text{max}} )</th>
<th>( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>17</td>
<td>( \infty )</td>
<td>17</td>
</tr>
<tr>
<td>( b )</td>
<td>19</td>
<td>( \infty )</td>
<td>19</td>
</tr>
<tr>
<td>( b )</td>
<td>23</td>
<td>( \infty )</td>
<td>23</td>
</tr>
</tbody>
</table>

*Task set \( \tau_2 \) :*

<table>
<thead>
<tr>
<th>task</th>
<th>( P_{\text{min}} )</th>
<th>( P_{\text{max}} )</th>
<th>( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>10</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>( b )</td>
<td>300</td>
<td>350</td>
<td>300</td>
</tr>
</tbody>
</table>

*Task set \( \tau_3 \) :*

<table>
<thead>
<tr>
<th>task</th>
<th>( P_{\text{min}} )</th>
<th>( P_{\text{max}} )</th>
<th>( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( b )</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>( c )</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>( d )</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>( e )</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

*Task set \( \tau_4 \) :*

<table>
<thead>
<tr>
<th>task</th>
<th>( P_{\text{min}} )</th>
<th>( P_{\text{max}} )</th>
<th>( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>( b )</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>( b )</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
</tbody>
</table>
8.1. Live heap space analysis

Observe that, for $\tau_2$, the number of states is less in the zone graph than in the determined 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. In the following subsection, we show the analysis results of a full-blown task set (with task bodies and state variables).

8.1.1 Extended example

<table>
<thead>
<tr>
<th>task</th>
<th>$P_{\text{min}}$</th>
<th>$P_{\text{max}}$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample$_1$</td>
<td>10</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>sample$_2$</td>
<td>20</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>lphigh</td>
<td>100</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>acquire</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 8.2: Task set of the extended example of live heap space analysis.

\[ \text{sample}_1(\langle gval, gbuf, buf_1, buf_2 \rangle, \langle gval', gbuf', buf'_1, buf'_2 \rangle) \leftarrow \]

\[ gval' := gval, \]
\[ gbuf' := gbuf, \]
\[ buf'_2 := buf_2, \]
\[ val_1 := buf_1.\text{head}, \]
\[ buf'_1 := \text{new cons} \{\text{head} := (\text{sensor}_1 + 99 \cdot \text{val}_1)/100, \text{tail} := buf_1\}. \]
Experimental results

\[ \text{sample}_2((gval, gbuf, buf_1, buf_2), (gval', gbuf', buf'_1, buf'_2)) \leftarrow \\
\quad gval' := gval, \\
\quad gbuf' := gbuf, \\
\quad buf'_1 := buf_1, \\
\quad val_2 := \text{buf}_2.\text{head}, \\
\quad \text{buf}'_2 := \text{new cons} \{\text{head} := (\text{sensor}_2 + 99 \cdot \text{val}_2)/100, \text{tail} := \text{buf}_2\}.
\]

\[ \text{lphigh}((gval, gbuf, buf_1, buf_2), (gval', gbuf', buf'_1, buf'_2)) \leftarrow \\
\quad gval' := gval, \\
\quad \text{mean}((\text{buf}_1), (m_1)), \\
\quad \text{mean}((\text{buf}_2), (m_2)), \\
\quad buf'_1 := \text{new nil} \{\}, \\
\quad \text{buf}'_2 := \text{new nil} \{\}, \\
\quad gbuf' := \text{new cons} \{\text{head} := (m_1 + m_2)/2, \text{tail} := gbuf\}.
\]

\[ \text{acquire}((gval, gbuf, buf_1, buf_2), (gval', gbuf', buf'_1, buf'_2)) \leftarrow \\
\quad \text{mean}((gbuf), (gval'')), \\
\quad gval' := (gval + gval'')/2, \\
\quad gbuf' := \text{new nil} \{\}, \\
\quad buf'_1 := buf_1, \\
\quad \text{buf}'_2 := \text{buf}_2.
\]

\[ \text{init}((gval, gbuf, buf_1, buf_2)) \leftarrow \\
\quad gval := 0, \\
\quad gbuf := \text{new nil} \{\}, \\
\quad buf_1 := \text{new nil} \{\}, \\
\quad \text{buf}_2 := \text{new nil} \{\}.
\]

For cost models \( \mathcal{M}, \mathcal{O}, \) and \( \mathcal{R} \) \((k^M_{\text{cons}} = 3, k^O_{\text{cons}} = 1, k^R_{\text{cons}} = 1; \) and \( k_{\text{null}} = 0 \) for all three cost models) we get the following matrices:
8.1. Live heap space analysis

<table>
<thead>
<tr>
<th># states</th>
<th># arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>zone graph</td>
<td>6100 14072</td>
</tr>
<tr>
<td>minimal FSM</td>
<td>3510 8428</td>
</tr>
</tbody>
</table>

Table 8.3: Zone graph and minimal FSM sizes of our extended example.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

The minimum solution to \(c_M\) is \(c_M^* = 111\); to \(c_O\) and \(c_R\) is \(c_O^* = c_R^* = 37\). Solving
the ILP problem took about 25 seconds, using \texttt{lp\_solve} version 5.5.2.0.\footnote{Platform: 3.06GHz Intel Core 2 Duo, 4 GB RAM, Mac OS X 10.6.4}

\section*{8.2 Garbage collection demand analysis}

The objective of the experimental study is to confirm that the model conceptually captures the execution time of the garbage collector. In other words, for a particular platform, there shall exist constants (e.g. $T_{CopyWord}$) such that we can construct the platform specific model of worst-case garbage collection execution time by simply replacing the constants in the general model by appropriate values. Naturally, experimental results are by no means guaranteed to reveal the worst-case behavior (i.e. the constants associated with the actual worst-case behavior). However, if set up appropriately, it will inevitably expose flaws of the model (non-existence of constants).

In the model, we have five parameters ($M$, $O$, $R$, $A^O_i$, and $A^R_i$) that affect the execution time of the garbage collector ($T_{gc}$ and $T_{A^i}$). The general approach we pursue is to measure the effect of each parameter in isolation whilst keeping the other parameters constant.

\subsection*{8.2.1 Hardware platform}

The experimental platform we use is the LPC2468 Developers’s Kit from Embedded Artists \cite{9}, with the standard LPC2468-16 OEM board replaced by an LPC2468-32 OEM board. The LPC2468-32 OEM board contains a standard NXP ARM7TDMI-S LPC2468 microcontroller \cite{76} along with 32 MB of SDRAM from Micron \cite{41}. The reason we choose this platform is simply because it has a sufficient amount of memory (i.e. enables sufficiently wide ranges of input parameter values for the measurements), a JTAG interface, and a fairly simple memory hierarchy. The microcontroller has no cache, but there exists a primitive buffering mechanism within the External Memory Controller (EMC).
8.2.2 Software platform

In order to make it easy to run the measurements, we use two different code-bases. The first code-base is provided by Embedded Artists [9] and takes care of the initialization of the LPC2468, setting the CPU clock to 57.6 MHz and initializes the External Memory Controller (EMC). When the initialization of the LPC2468 is complete the microcontroller will be stuck in an infinite-loop. Once the infinite-loop is reached we connect to the microcontroller using the JTAG interface and upload the second code-base into the SDRAM. The second code-base consists of the test-code along with the Timber Run-Time System (Timber RTS). The Timber RTS used in the experiments is based on the standard ARM RTS available in the Timber darcs repository [68]. The ARM RTS has then been modified to support logging of several events within the RTS, among other things the GC-time and execution time of messages. Special care is taken to minimize the effects of the logging. While it is impossible to completely remove the effects of logging the overhead is very small, constant, and predictable. After the test-run is complete we download the log from the internal SRAM of the microcontroller via the serial port.

8.2.3 Measurements

Parameters affecting $T_{gc}$

We have three heap parameters affecting the resulting $T_{gc}$. We setup the test runs in such way that each parameter can be varied in isolation (except when varying number of live nodes). The general configuration is a time-triggered scheduling of the garbage collector. The live data on the heap is a constant structure that is relocated 100 times by the garbage collector for each parameter value. We collect the total running time of each garbage collection cycle. During these measurements, we do not have any tasks running concurrently with garbage collector. We do this procedure for 1000 different values of the parameter in question.

Varying amount of live memory We use one live node, which we vary the size from 1 to 1000 words containing no references. The results of this are shown in Figure 8.1.
Experimental results

**Varying amount of live references** We use one live node with a constant size of 1000 words, for which we vary the number of self references from 1 to 1000. The results of this are shown in Figure 8.2.

**Varying number of live nodes** We use a linked list where each node is of a constant size (16 words) and containing 1 reference (the next field). We vary the length of the list (i.e. the number of live nodes) from 1 to 1000. The results of this are shown in Figure 8.3. Due to the fact that we cannot easily generate equivalent data structures with amount of live memory and number live references constant and vary the number of nodes with sought granularity, we actually vary all three parameters in a controlled way (i.e. 16 words live memory, 1 live reference, and 1 live object per data point).

![Figure 8.1: Measurement data of GC time as a function of live memory.](image)
8.2. Garbage collection demand analysis

![Figure 8.2: Measurement data of GC time as a function of number of live references.](image)

![Figure 8.3: Measurement data of GC time as a function of the length of the live linked list.](image)

**Parameters affecting** $T^A_i$

We have two heap parameters affecting the resulting $T^A_i$. We setup the test runs in such way that each parameter can be varied in isolation. The general configuration is, again,
a time-triggered scheduling of the garbage collector, but now accompanied with a task interrupting the GC once at every run. The amount of live data on the heap is kept constant over all runs as a constant payload of work to be done. We collect the total running time of each garbage collection cycle. We do this procedure for 1000 different values of the parameter in question.

**Varying amount of references allocated** We allocate one node with a constant size 1000, for which we vary the number of self references from 1 to 1000. The results of this are shown in Figure 8.4.

**Varying number of nodes allocated** We allocate a linked list where each node is of a constant size (16 words) and containing 1 reference (the next field). We vary the length of the list (i.e. the number of live nodes) from 1 to 1000. The results of this are shown in Figure 8.5.

![Figure 8.4: Measurement data of GC time as a function of number of references allocated by an interrupting task.](image)
8.2. Garbage collection demand analysis

Figure 8.5: Measurement data of GC time as a function of the length of the linked list allocated by an interrupting task.

From the measurements we can see that the behavior appears to be linear in all parameters. The scattered clustering of data points for some of the measurements can be explained by the, although relatively simple, non-flat behavior of the EMC. The reason why, for some measurements, the clustering forms distinct lines instead of being more evenly distributed in an area is the discrete behavior of the EMC (i.e., either hit or miss in the pre-fetch cache) together with the way the heap parameters were controlled (keeping all but one constant). Nonetheless, the overall tendency is linear which supports the validity of the model.

8.2.4 Full GCDA example

Given the transition times in Table 8.4, we get the following formulas for $T_{gc}$ and $T_i^A$:

\[
T_{gc} = 0.001 \times M + 0.026 \times O + 0.002 \times R + 0.210
\]

\[
T_i^A = 0.015 \times A_i^O + 0.002 \times A_i^R
\]
Experimental results

\[
\begin{align*}
T_{\text{COPYWORD}} &= 0.001 \\
T_{\text{SCANSTART}} + T_{\text{SCANDONE}} + T_{\text{COPYSTART}} + T_{\text{COPYDONE}} - T_{\text{FORWARD}} &= 0.026 \\
T_{\text{SCANADDR}} + T_{\text{FORWARD}} &= 0.002 \\
T_{\text{START}} + T_{\text{DONE}} &= 0.210 \\
T_{\text{SCANSTART}} + T_{\text{SCANDONE}} - T_{\text{FORWARD}} &= 0.015
\end{align*}
\]

Table 8.4: Example transition times affecting \( T_{gc} \) and \( T_{A_i} \).

Let us extend the example presented in Section 8.1.1 with WCETs:

<table>
<thead>
<tr>
<th>task</th>
<th>( p_{\text{min}} )</th>
<th>( p_{\text{max}} )</th>
<th>( D )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample1</td>
<td>10</td>
<td>15</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>sample2</td>
<td>20</td>
<td>20</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>lphigh</td>
<td>100</td>
<td>100</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>acquire</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>400</td>
</tr>
</tbody>
</table>

We get the following values for \( L_a^*, L_b, T \) (see Chapter 2 for details on their definitions):

\[
\begin{align*}
U_r &\approx 0.76 \\
L_a^* &\approx 32.9 \\
L_b &= 634 \\
T &= \{5, 10, 15, 25\}
\end{align*}
\]

The processor demand \( h_r(t) \) of our extended example is shown in Figure 8.6. From Section 8.1.1 follows that \( M = 111 \), \( O = 37 \), and \( R = 37 \). Furthermore; \( \text{sample}_1, \text{sample}_2 \), and \( \text{lphigh} \) each allocate one \( \text{cons} \), which gives us \( A_{\text{sample}_1}^O = A_{\text{sample}_2}^O = A_{\text{lphigh}}^O = 1 \), \( A_{\text{sample}_1}^R = A_{\text{sample}_2}^R = A_{\text{lphigh}}^R = 1 \), and \( A_{\text{sample}_1}^M = A_{\text{sample}_2}^M = A_{\text{lphigh}}^M = 3 \). Since \( \text{acquire} \) does not allocate any new heap nodes, we have \( A_{\text{acquire}}^M = A_{\text{acquire}}^O = A_{\text{acquire}}^R = 0 \). We get...
8.2. Garbage collection demand analysis

the following values for (8.1):

\[ T_{gc} = 0.001 \times 111 + 0.026 \times 37 + 0.002 \times 37 + 0.21 = 1.357 \]
\[ T_{\text{sample}_1}^A = 0.015 \times 1 + 0.002 \times 1 = 0.017 \]
\[ T_{\text{sample}_2}^A = 0.015 \times 1 + 0.002 \times 1 = 0.017 \]
\[ T_{\text{high}}^A = 0.015 \times 1 + 0.002 \times 1 = 0.017 \]
\[ T_{\text{acquire}}^A = 0 \]

In Figure 8.7, the garbage collection demand \( C_{gc}(t) \) is shown. The gray line shows one possible period for feasible garbage collection \( C_{gc}(638) = 637.108 \leq 638 \). From (7.4) it follows that the total heap memory need \( M_{t_{\text{old}}}(638) = 840 \) words.

Figure 8.6: EDF-schedulability analysis of our extended example.
Figure 8.7: Garbage collection demand analysis of our extended example.
Chapter 9

Related work

9.1 Garbage collection correctness

McCreight et al. [62] introduces a general framework for reasoning about garbage collecting algorithms in Coq theorem prover, and also includes a mechanized proof of correctness for Baker’s copying garbage collector. Central to this approach is a logical specification of the abstract properties a garbage collected heap must expose to its mutator client, a notion somewhat similar to the transition labels we use to model mutator-collector interaction. In contrast to our work, they require the actual algorithm to be expressed in a low level assembly language, and their garbage collector steps must furthermore be externally invoked in a strictly sequential fashion.

Another mechanized study can be found in [74], where Russinoff presents a correctness proof in Nqthm theorem prover of Ben-Ari’s incremental mark and sweep collector [15]. He shows that it is safe (i.e. nothing but garbage is collected) and that it will eventually collect all garbage.

In [19], Birkedal et al. prove correctness of Cheney’s classical stop-and-copy collector [25] using separation logic with the extension of local reasoning. This elegantly enables them to reason about both the specification and the proof in a manageable way. At the end, they predict a promising future of this approach that would enable one to reason about more complex algorithms such as garbage collectors of a different type than stop-and-copy. However, this track of future work has, to the best of our knowledge, not been
In [66], Morrisett et al. present a garbage collection calculus and specify a garbage collection rule based on \textit{free variables}, which models tracing garbage collectors. They present two implementations of it, which correspond one to a copying collector and one to a generational collector. They furthermore show that Milner-style type inference can be used to show that, even though a node is reachable, it can still be garbage, semantically. In [37], Hosoya and Yonezawa extend the idea of using type reconstruction for garbage collection, and they present a garbage collection algorithm based on dynamic type inference. In contrast to tracing garbage collection, where only unreachable garbage is collected, their scheme collects nodes that are semantically garbage.

A huge volume of work on informal descriptions of real-time garbage collection has been presented. One of the earliest incremental copying garbage collectors presented is that of Baker [10]. It is an extension of Cheney’s collector into an incremental collector. He utilizes a read-barrier that disallows accesses to fromspace (the white heap), i.e. when the mutator tries to access the white heap, the barrier enforces either forwarding (if the node has been copied already) or copying of the node. In the same year (1978) Dijkstra et al. presented the ideas behind the tri-color invariant [30], which has been extensively used by others for reasoning (mostly informally though) about correctness of incremental copying garbage collection. In [22], Brooks presents a variation of Baker’s collector. Among other differences, one is that the mutator always follows the indirection via the forwarding pointer, and if the node has not been copied (or the mutator accesses a node in tospace) the forwarding pointer points to the node itself. Instead of checking if the node is in fromspace upon every access, the mutator is always redirected.

9.2 Live heap space analysis

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 6.1. 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 [1, 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. [46] present a 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. [26] present 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. [87] present 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.

9.3 Garbage collection schedulability

The key issue in scheduling concurrent garbage collection in real-time systems is undoubtedly how the garbage collector should be able to compete with the real-time tasks. Typically, this amounts to finding appropriate timing assumptions for the collector task. Although the garbage collector does not really have such timing properties, more or less artificial ones are necessary in order to determine schedulability of the whole system. Due to the quite complex dependency between properties of the real-time tasks and the required execution time for garbage collection, much of the focus in this field has unfortunately diverged from pure schedulability analysis towards improving measured performance.

In [33], Fu and Hauser present a framework for describing a broad class of real-time garbage collectors and their corresponding scheduling premises. They also accurately identify the key to enabling provable schedulability guarantees of a garbage collected
system, to wit comprehensive knowledge about the memory behavior of the real-time tasks as well as the inter-dependencies between them and the garbage collector.

In [35], Henriksson presents a feasibility test for garbage collection based on response time analysis [45]. In contrast to our work, he assumes a per task parameter for worst-case garbage collection time required after one invocation ($G_i$). This parameter slightly corresponds to our execution time parameter due to new allocations ($T_{i\Delta}$). However, he does not present the connection between $G_i$ and the actual garbage collection algorithm used. This work was later extended by Gestegård-Robertz and Henriksson to compute an upper-bound on the cycle time of the garbage collector in order to meet total heap memory limits [73], which corresponds to a rearrangement of our memory needs formula (Equation 7.4).

In [52], Kim et al. present upper-bound estimates on the execution time of a garbage collector based on Brook’s [22] evacuation strategy. They present a schedulability test for the whole system based on a worst-case response time analysis of a sporadic server. In addition, they also present a live memory analysis to determine the worst-case local live heap memory of each task. In contrast to our work, they do not present a detailed connection between the parameters used in the execution time estimate and the actual garbage collection algorithm used. They are also limited to using rate monotonic priorities to enable the sporadic server schedulability test.

In [24], Chang presents a hybrid approach based on a lazy freeing reference counting collector and a backup mark-sweep collector. External fragmentation is avoided by using a fixed block size. He also presents a schedulability test based on a dual-priority scheduling scheme including the worst-case cost for both the reference counting collector and the mark-sweep collector. This is achieved by integrating the two garbage collectors into the real-time scheduling framework as tasks (i.e. deriving appropriate timing assumptions for them). The main difference between his approach and ours is that he integrates the cost of garbage collection in the regular schedulability analysis. This makes it more difficult to extend the regular schedulability test with more features (e.g., shared resources) without affecting the schedulability test of the garbage collector.
Recently, Kalibera et al. developed schedulability tests for both time-based and slack-based scheduling of time-triggered garbage collection [47]. They show that none of them is superior to the other (in terms of schedulability) and they draw the conclusion that the choice of scheduling policy is a key part of designing garbage collected real-time systems.
150

Related work
Throughout the history of real-time scheduling theory, successful static schedulability analyses have relied on safe and tight WCET estimates for each individual task. One of the keys for achieving such estimates is to require them to be independent, with respect to both other tasks and scheduling policy. Incorporating garbage collection in a real-time system as a task of its own inevitably violates that requirement. The workload and memory overhead for a garbage collector depends on both timing and memory behaviors of the real-time tasks, as well as the principles governing how those tasks are scheduled. Typically, in order to give a provably safe estimate on the WCET of the garbage collector, one needs to account for both the individual effect of each task and their combined effect with respect to various properties. To this end, depending on collector strategy, several task properties would have to be quantified, e.g.:

- live heap space
- new allocations
- mutation patterns
- memory fragmentation
- amount of heap memory forming circular structures

Although such properties might be hard (or even impossible) to determine, the consequences of these dependencies reach even further. If the garbage collector is allowed to
Conclusions and Further Work

compete with the real-time tasks (i.e., either be prioritized over pending tasks or impose significant overhead onto their execution time), the scheduling policy and corresponding feasibility test must be tailored for the particular collector, often imposing further restrictions onto the task model.

In this dissertation we have proposed an approach to remedy these problems. Recalling our theses from Chapter 1:

_In this dissertation, we propose the following theses: (1) the key to successful real-time garbage collection is to preserve as much as possible of previous advances in real-time scheduling theory by imposing minimal restrictions on the task model, scheduler, and schedulability test; and (2) the keys to enabling a priori schedulability guarantees are clear identification and tractable analysis of the sources of execution time for the garbage collector._

10.1 Contributions

Based on an incremental copying garbage collector deployed in the run-time system of the real-time programming language Timber, we have demonstrated that our first thesis (1) is achievable. The proof relies on disallowing the garbage collector to compete with the real-time tasks with respect to processor time, i.e., it is scheduled as the lowest priority process. The execution time of each increment of our collector is bounded by a small constant, which guarantees that the cost of preempting our collector at any time is completely predictable. The mutator-collector synchronization cost (i.e., for read/write barriers) is completely eliminated by restricting tasks (asynchronous methods in Timber) to be known and created statically. By means of these modest restrictions, we have achieved a complete decoupling between the schedulability test for the real-time tasks and the cost of garbage collection. Instead, we have proposed another schedulability analysis that solely determines the schedulability of our garbage collector – the _garbage collection demand analysis_.

We have proved our second thesis (2) through the identification of all parameters
affecting execution time of our garbage collector, to wit global live heap space and heap allocations made by interrupting tasks. Analyzing heap allocation properties for each task corresponds quite well to execution time analysis (with a slightly different cost model). Both are monotonically increasing accumulative properties. The results of such program analyses are typically expressed as functions of the programs input data (e.g., see [3]). In a real-time system, where tasks maintain an ongoing interaction with the environment, input data are typically tightly coupled with the persistent state of the system. As a fortunate coincidence, such state-dependent properties correspond very well to the behavior of global live heap space.

We have proposed a novel technique for determining safe upper bounds on live heap memory of real-time systems, which is safe even in the presence of state- and order-dependent tasks driven by external sporadic events. It 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 applying standard techniques used in reachability analysis to construct a finite state machine (FSM) representation of task execution orders. We apply a standard variant of abstract interpretation to each task for inferring linear input/output size relations 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.

To this end, we have imposed two restrictions onto the task model. We require the relative deadline of each task to be bounded by its minimum inter-arrival time. However, in Section 10.2.2, we will address the modifications to our model necessary to incorporate tasks where relative deadlines and minimum inter-arrival times are unrelated. In order to sufficiently capture the order of task executions, or more precisely, the order in which they affect live heap space, we restrict the task model to only incorporate strictly nested resource locking (similar to SRP). The only restriction our model effectively adds to SRP is that we prohibit non-nested sequential resource access: new resources may not be
locked once a previously held resource has been released. Although our experience shows that this restriction is not very burdening in practice, we sketch a possible solution to allow sequential locking in Section 10.2.3.

10.2 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 occur at the very same instant. Adding a safe lower bound on execution time for each task will reduce the time window 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.

We will now address some of these tracks for further work in detail and present initial ideas on solutions.
10.2. Further work

10.2.1 Specific scheduling policies

In order to improve both performance and precision of our live heap space analysis, we can modify the construction of the timed automaton so that it strictly follows the behavior of a specific scheduling policy. To this end, what is needed is to restrict each zone in the zone graph to have at most one outgoing labeled transition, i.e. of all released tasks in a zone, the scheduler will always deterministically choose only one task to execute.

For deadline monotonic (DM) scheduling, if more than one task is in the released state, we discard all other labelled transitions but the one for the task with smallest relative deadline. For EDF scheduling, the absolute deadline of each pair of released tasks can be compared by trying the corresponding difference constraint for solutions with respect to the current clock zone. For example, given a clock zone $D_\phi$ and two released tasks $t_i$ and $t_j$, the conjunction $\varphi \land c_j - c_i \leq D_j - D_i$ is nonempty iff the absolute deadline of $t_i$ may be less than or equal to the absolute deadline of $t_j$. In Table 10.1, the minimal FSM sizes of the example task sets from Chapter 8 are shown for each scheduler (compared with the original size). For $\tau_3$ we get a minimal FSM of 25% of the original size.

<table>
<thead>
<tr>
<th>Task set</th>
<th>EDF</th>
<th>DM</th>
<th>Not spec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_2$</td>
<td>652</td>
<td>668</td>
<td>699</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>847</td>
<td>1610</td>
<td>3393</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>2221</td>
<td>3891</td>
<td>6343</td>
</tr>
</tbody>
</table>

*Table 10.1: Minimal FSM sizes of the example task sets for EDF and DM scheduling.*

The results of applying the schedulers to our extended example (Table 8.2 and 8.3) are shown in Table 10.2. Furthermore, the minimum solutions to the corresponding cost functions are reduced by the size of one cons cell for EDF scheduling but are the same for DM scheduling. However, the execution time of lp_solve for solving the ILP problem is reduced to about 20% of the original problem for both EDF and DM.
10.2.2 Tasks with $D_i > P_{i}^{\text{min}}$

Since one of our theses is to keep the set of restrictions on the task model to a minimum we need to address the possibility to incorporate tasks where $D_i > P_{i}^{\text{min}}$. The main problem with doing this is that we would need more than one clock in the timed automaton of such a task. We must allow more than one instance of the same task to be in the released state at the same time. Let us first define the maximum number of concurrent instances of the same task.

$$x_i = \left\lceil \frac{D_i}{P_{i}^{\text{min}}} \right\rceil$$  \hspace{1cm} (10.1)

We can now construct one timed automaton for each potential concurrent release of task $i$, using one unique clock each. They all synchronize (listen) on the same binary channel $\text{release}_i$ which is triggered by a releaser timed automaton for task $i$. Observe that this approach is general in the sense that it works for $x_i = 1$, but requires two clocks instead of one (one for the releaser and one for the part being released). Note that the binary synchronization channel $\text{release}_i$ needs to be replaced by $\varepsilon$ (the empty label) when constructing the corresponding untimed automaton. For each $j \in \{1..x_i\}$ we have

\[
\begin{array}{|c|c|c|}
\hline
 & \text{# states} & \text{# arcs} \\
 & (\text{ILP unknowns}) & (\text{ILP constraints}) \\
\hline
\text{EDF} & 1732 & 3738 \\
\text{DM} & 1789 & 3927 \\
\text{Not spec.} & 3510 & 8428 \\
\hline
\end{array}
\]

*Table 10.2: Minimal FSM sizes of our extended example for EDF and DM.*
the following timed automaton:

\[
\begin{align*}
L_{ij} &= \{\text{idle, released}\} \\
\ell_{0ij} &= \text{idle} \\
A_{ij} &= \{t_i, \text{release}_i\} \\
C_{ij} &= \{c_{ij}\} \\
I_{ij} &= \{(\text{released}, c_{ij} \leq D_i)\} \\
E_{ij} &= \{(\text{idle, true, release}_i!, \{c_{ij}\}.\text{released}), (\text{released, true, } t_i, \emptyset, \text{idle})\}
\end{align*}
\]

The releaser timed automaton looks as follows:

\[
\begin{align*}
L_i &= \{\text{idle}\} \\
\ell_{0i} &= \text{idle} \\
A_i &= \{\text{release}_i\} \\
C_i &= \{c_i\} \\
I_i &= \{(\text{idle}, c_i \leq P_{i}^{\max})\} \\
E_i &= \{(\text{idle}, c_i \geq P_{i}^{\min}.\text{release}_i!, \{c_i\}, \text{idle})\}
\end{align*}
\]

These timed automata are shown graphically in Figure 10.1. The binary synchronization channel requires one sender ! and one listener ? to be able to take the transition simultaneously. If more than one combination exist, one of them is chosen non-deterministically.

Observe that, if EDF scheduling is not applied in the construction of the zone graph, there is no guarantee that the precedence ordering between simultaneous releases of the same task is preserved.

### 10.2.3 Tasks where full SRP is allowed

As we pointed out in Chapter 6, the run-time behavior of Timber closely follows the principles of SRP. However, in order to lift the restriction on multiple non-nested sequential resource accesses, we need to solve two problems. First of all, we need to find all required sub-tasks, whose locking behavior conforms to the restricted form of nested resource access.

Let us look at an example of how such a behavior might look like. In Figure 10.2, a task that uses five different resources in SRP fashion. We may divide the task into
two sub-tasks, where the first one has an effect on \( r_4 \) and the second one has an effect on \( r_1, r_2, r_3, \) and \( r_5 \). To this end, we would need to construct two different size relation matrices, accordingly. The timed automaton of such a task can be captured by adding a location for the intermediate state (see Figure 10.3) and instead of the label \( t_i \), we would have \( t_{i,1} \) that corresponds to the one affecting \( r_4 \) and \( t_{i,2} \) the other one.

However, there is another, more severe, problem related to this: in order to interleave
10.2. Further work

159

![Diagram showing states and transitions](image)

Figure 10.3: Example of sub-tasks conforming to SRP.

the task, i.e. execute something else (say \( t_j \)) in between \( t_{i,1} \) and \( t_{i,2} \), \( t_j \) cannot lock \( r_1 \), \( r_2 \), or \( r_3 \). In fact, this property is tightly coupled with the scheduling policy EDF+SRP (as described in Chapter 2). Thus what we need to do is to keep track of \( \pi \), see (2.11) and (2.12).

Although the restrictions we have imposed on the task model can be lifted by the ideas sketched above, we still have an exponential growth of states and transitions in the resulting FSM. Initial experiments have shown that only a very small part of ILP constraints generated from the FSM contributes to the worst case live heap space behavior. A typical scenario is that some tasks increase the size of the persistent state, while others reduce it. The timing properties and release patterns of a particular task are independent of the properties and patterns of other tasks. Thus, one may be able to neglect some of the possible transitions while constructing the zone graph (similarly to the scheduling decision described in Section 10.2.1) and still acquire a provably safe upper-bound on live heap space.
Bibliography


[85] David A. Turner. Miranda – a non-strict functional language with polymorphic
types. In Record of the 1985 Conference on functional programming and com-
puter architecture (FPCA ’85), pages 1–16, Portland, Oregon, USA, September 1985.
Springer-Verlag.

reclamation algorithm. In Proceedings of the first ACM SIGSOFT/SIGPLAN soft-
ware engineering symposium on Practical software development environments, pages

[87] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Optimized Live Heap
Bound Analysis. In Fourth International Conference on Verification, Model Checking
and Abstract Interpretation (VMCAI), volume 2575 of Lecture Notes in Computer

[88] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan
Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann,
Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and
Per Stenström. The worst-case execution-time problem—overview of methods and

the International Workshop on Memory Management, Saint-Malo (France), 1992.
Springer-Verlag.

[90] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph
reorganization to improve locality in garbage collected systems. ACM SIGPLAN

[91] Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling
intergenerational references in generation-based garbage collection on stock hardware.


