Rate-Monotonic Analysis in the C++ Type System [*]

Morgan Deters Christopher Gill Ron Cytron
{mdeters,cdgill,cytron}@cs.wustl.edu

Distributed Object Computing Laboratory
Department of Computer Science and Engineering
Washington University in St. Louis
One Brookings Drive, Box #1045
St. Louis, MO 63130 USA

Abstract

We describe an implementation of Rate-Monotonic Analysis (RMA) within the C++ parametric type system that provides C++ real-time software developers a good way to reason with types at the source level about recurrent tasks and deadlines. Using our approach, a program can be considered *incorrect*, raising type errors at compile time, if a given set of tasks is not statically schedulable. Similarly, this compile-time "metaprogram" can adjust a task set so as to become feasible; we perform this analysis inside the C++ type system, which allows a very natural integration into C++ programs. We discuss our approaches and the applicability of our work to the model-driven development of real-time embedded systems.

1 Introduction

Real-time embedded systems have specific timeliness requirements that result in the necessity of *scheduling* tasks' access to scarce resources. Rate-Monotonic Scheduling (RMS) is a well-known static scheduling technique in which periodic tasks are assigned priorities in accordance with their period: more frequent tasks receive a higher priority. A runtime schedule honoring RMS-assigned priorities is known to be an optimal schedule for the fixed-priority scheduling problem [7]; that is, if any assignment of fixed priorities yields a feasible schedule, the RMS assignment will.[1] Rate-Monotonic Analysis (RMA) refers to the computation performed on a set of periodic tasks to determine whether they may be statically assigned fixed priorities with RMS (or indeed with any such scheme, since RMS leads to an optimal schedule with respect to feasibility) and meet all deadlines.

As originally stated by Liu and Layland [7], a set of m periodic tasks has utilization:

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i}$$

where C_i is the execution time budget, or *cost*, of task i on some machine and T_i is the execution *period* of task i. A task set is feasibly schedulable with RMS *if*

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} \leq m\left(2^{1/m} - 1\right) . \qquad (1)$$

This is a computationally simple test, and can easily be performed (even manually) for a given set of tasks. However, this test is pessimistic, disqualifying task sets that are, in fact, feasible. Lehoczky, Ding, and Sha offer a stricter test [6, 9].[2] A set of m periodic tasks is feasibly schedulable *if and only if*

$$\forall\, i,\ 1 \leq i \leq m,$$
$$\exists\, t \in \left\{ l \cdot T_k \;\middle|\; 1 \leq k \leq i,\ 1 \leq l \leq \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\}$$
$$\textit{s.t.} \quad \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil \leq t . \qquad (2)$$

When engineering a real-time system that makes use of static scheduling, such tests are typically performed on a set of proposed tasks ahead of time, often long before compilation—even in the design phase, *e.g.*, through model-integrated computing tools—to secure a guarantee that they will meet their deadlines. This may be acceptable if the task set is known in advance and does not change through the software development process. However, for purposes of debugging and design flexibility, a solution that integrates compilation with RMA task set verification is desired so that the task set can easily be modified. Further, for retargetable, reconfigurable real-time systems, software development teams often wish to provide similar systems meeting slightly different design requirements and manage all such configurations using a modeling tool. Clearly, this goal is unnecessarily complicated if the software is designed in a rigid manner for a specific set of tasks.

One solution to this problem would be to compute feasibility of the task set at runtime. Indeed, this approach is taken by some systems [4]. However, the main benefit of static

[*]Sponsored by DARPA under contract F33615-00-C-1697.

[1]In this paper, we intend "feasible" to mean that all tasks are guaranteed to meet all deadlines, over all possible task phasings. The deadline of a task in classical RMS is the start of its next execution period.

[2]The proof is found in [6]; a useful discussion appears in [9].

scheduling over dynamic scheduling is its simplicity and low overhead. At worst, the only computation required at runtime for a fixed-priority periodic scheduling mechanism is the comparison of eligible tasks' priorities; at best, the processor is scheduled in a sequential fashion and scheduling and context switches are free.[3]

Because runtime feasibility checks are not required for many real-time systems, we do not seek to require them in a new system for real-time software development. At the same time, we wish to ease the development process by allowing the task set to change with each compilation, yet require that compiled programs are indeed feasible. In Section 2 we propose a system that uses the C++ compiler to perform feasibility testing as part of program translation. We extend the basic idea in Section 3 to show that our technique can be used to enforce that every correct program is feasible—that is, a semantic error is flagged by a standard-compliant C++ compiler when infeasible task sets are specified by the program—and to search a parameter space of different task rates for feasibility.

This paper is organized as follows. Section 2 explains our approach, Section 3 discusses some useful extensions to our base technique, Section 4 points to some related work, and Section 5 offers some conclusions and our thoughts on future research directions in this area.

2 Approach

We are prototyping a template metaprogramming framework, coded in C++, that performs rate-monotonic analysis at compile-time and enables code to reflect at compile-time upon its task sets and reason about their feasibility. Generally, we believe compile-time "reflection" of this sort—which does not require runtime support—to be valuable in C++ real-time software development. We use the technique to achieve the following specific requirements:

- Real-time tasks can be specified as optional,
- "Cheap" task sets that have the critical features of standard task sets can be linked to their more "expensive" versions,
- The "best-fit" versions of expensive services can be automatically selected and compiled in with no user intervention or runtime penalty in time or space or the size of the executable, and
- Truly infeasible task sets can be automatically rejected; if there is no guarantee that a task set can be scheduled, the compiler can be used signal an error.

We provide details on these particular aspects of our approach in the rest of Section 2 and in Section 3, but the above list is not an exhaustive one. First, we specify the base of our approach, which allows us to construct task sets and perform basic queries of them.

[3]Task sets are scheduled most easily when the task rates are harmonic; such task sets also have the benefi t of achieving 100% utilization.

```
struct my_task {
  enum { cost       = 100,
         period     = 600,
         phasing    = 50,

         droppable  = 0,
         importance = 1000 };

  static void do_task(const context& c) {
    cout << "my_task::do_task()" << endl;
  }
};
```

Figure 1: A sample Task.

2.1 Specification

We define a generic-programming *concept* [3] Task, implemented in C++ as a `struct`, which, along with zero or more associated TaskTraits providing additional, optional information (discussed in Section 3), fully specifies a *periodic real-time task*. A Typelist [1, 2] of Tasks then describes a *task set*. In addition to the standard parameters that we need to perform RMA for each periodic task (*i.e.,* task cost and period), we include other useful information for scheduling the task. A sample Task definition is shown in Figure 1.[4] Its elements are:

cost specifies the logical *cost* of the Task. This may be a measurement on a particular platform or a theoretical upper-bound, calibrated to agree with the other time-based parameters below.

period specifies the logical *period* of the Task.

phasing specifies the logical *phasing* of the Task. This is the offset of the logical clock at which its logical period begins.

droppable is a boolean value indicating whether or not a task can be dropped if necessary to make its task set feasible—this value, in effect, declares whether or not the task is optional.

importance is an integer value specifying the relative willingness of the compile-time scheduling analysis to drop the task. Tasks with lower importance are dropped before higher-importance tasks.

do_task is a Functor [3] that specifies the work to be performed by the Task.

Once the basic structures defining tasks have been built, task sets can be constructed using `typedef`:

```
typedef TYPELIST_2(taskA, taskB) my_tasks;
```

In this case, a task set type (called `my_tasks`) of two independent task types is constructed: `taskA` and `taskB`.

2.2 Operation

We then wish to perform basic operations on this task set. These operations include:

[4]*Note:* All C++ code examples in this paper have been tested and compile properly on the GNU C++ compiler v3.2.2 [5].

- Sorting the task set by period,
- Determining the schedulability of such a task set,
- Generating code to schedule the task set at runtime, and
- Querying on the task set regarding its constituent tasks, its feasibility, and its utilization.

Further, we wish to perform these operations at compile-time to the fullest extent possible. Obviously, the tasks will actually *execute* only at runtime, but we wish to perform the queries and other operations above at compile-time. We also wish to expand and inline a specialized `start()` routine specifically for this task set so that starting the tasks has as little overhead as possible. Finally, we want the associated structures and queries to be reasonably easy and intuitive to use. By offering an interface to user code in the metaprogram, we introduce a mechanism similar to compile-time *structural reflection* into a real-time program. Using this facility, a real-time programmer can write code that is easy to read and reconfigure despite being tailored for a particular task set. In effect, the task set introduces various *constraints* onto the program, and the C++ compiler (by evaluating the template metaprogram) is able to resolve these constraints and generate a specialized executable, even though the source code remains modular and generic.

Fortunately, these operations can all be performed by manipulating the task set with a template metaprogram. In this paper, we focus on the last operation: determining the feasibility and expected utilization of a task set and integrating this with the program. We define a `Schedule` template, shown in Figure 2. This template calls an `RMA_Feasible` template metaprogram shown in Figure 3. This metaprogram solves inequality (2) directly, for each i, by trying different values of t as necessary. It utilizes the support templates of Figure 4, which compute the set of all $l \cdot T_k$ and the j-summation.

```
template <class TaskSet> struct Schedule;

template <class Head, class Tail>
struct Schedule<Typelist<Head, Tail> > {
  typedef Typelist<Head, Tail> TL;
  enum { feasible=RMA_Feasible<TL>::Result };
  static const double utilization =
         Schedule<Tail>::utilization +
         double(Head::cost) / Head::period;
  static void schedule(void) {
    /* (not shown) */
  }
};

template <>
struct Schedule<NullType> {
  static const bool Result = true;
  static const double utilization = 0.0;
  static void schedule(void) {
    // no action necessary
  }
};
```

Figure 2: The `Schedule` template.

```
template <class TL, int m, int i>
struct check_i;

template <class Head, class Tail,
          int m, int i>
struct check_i<Typelist<Head, Tail>, m, i> {
  enum { task_result =
           task_feasible<Typelist<Head,Tail>,
                              i>::Result,
         Result = check_i<Typelist<Head,Tail>,
                               m, i+1>::Result
                  && task_result };
};

template <class Head, class Tail, int m>
struct check_i<Typelist<Head, Tail>, m, m> {
  enum { Result =
           task_feasible<Typelist<Head,Tail>,
                              m>::Result };
};

template <class TaskSet>
struct RMA_Feasible {
  enum { m = Length<TaskSet>::value,
         Result = check_i<TaskSet,
                               m, 1>::Result };
};
```

Figure 3: The main "loop" of the `RMA_Feasible` template metaprogram.

Given this metaprogramming mechanism, client code using our framework can then be specified in a very straightforward manner (Figure 5). The `schedule()` method of the `Schedule` template (implementation not shown in this paper) is used to set up the proper threading mechanism for a specified compilation target and invokes the `do_task` routines of the task set's constituent task types as appropriate. Because this can be inlined, no runtime overhead need exist for permitting this flexibility of task types as template parameterization, as this is sorted out by the C++ compiler at compilation time. Providing the task-invocation capability in a parameterized fashion (which could automate the choice of threading model, for example) is the subject of ongoing work and is not described in this paper. In Section 2.5, we describe a way to cause a compiler error if an infeasible schedule is encountered.

2.3 A Walkthrough Example

As an example of how this template expansion works,[5] consider the task set of Figure 6. In this case, tasks `taskA` and `taskB` have only costs and periods—for simplicity of the example, other parameters have been omitted from the listing.

In evaluating the `RMA_Feasible<my_tasks>` template instantiation (at the bottom of Figure 6), we must direct the C++ compiler to check that inequality (2) holds for each task i in our example task set. We do this by first counting the

[5]The discussion of this section is by necessity abbreviated and imprecise. The reader is referred to [10] for a more careful treatment of this material.

```
template <class TL, int i, int t, int j = 0>
struct sum_j {
  typedef typename TypeAt<TL, j>::Result J;
  enum { Cj = J::cost,
         Tj = J::period,
         my_result = Cj * ((t%Tj > 0 ? 1 : 0)
                           + (t / Tj)),
         Result = sum_j<TL,i,t,j+1>::Result
                  + my_result };
};

template <class TL, int i, int t>
struct sum_j<TL, i, t, i> {
  enum { Result = 0 };
};

template <class TL, int i, int t_ix, int k=0>
struct get_t {
  enum { Ti = TypeAt<TL,i-1>::Result::period,
         Tk = TypeAt<TL,k>::Result::period,
         num_l = Ti/Tk,
         Result = (t_ix >= num_l)
           ? get_t<TL, i,
               t_ix - num_l, k+1>::Result
           : (t_ix + 1) * Tk };
};

template <class TL, int i, int t_ix>
struct get_t<TL, i, t_ix, i> {
  enum { Result = 0 };
};

template <class TL, int i, int t_ix = 0>
struct task_feasible {
  typedef get_t<TL, i, t_ix> t_type;
  enum { t = t_type::Result,
         Result = (t > 0) &&
           ( sum_j<TL, i, t>::Result <= t
             || task_feasible<TL, i,
                  t_ix + 1>::Result ) };
};

template <class TL, int i>
struct task_feasible<TL, i, i> {
  enum { Result = 0 };
};
```

Figure 4: Supporting templates for the `RMA_Feasible` template metaprogram of Figure 3.

```
typedef Schedule<TYPELIST_3(
        taskA, taskB, taskC)> my_schedule;
if(! my_schedule::feasible)
  cerr << "WARNING: infeasible!" << endl;
my_schedule::schedule();
```

Figure 5: Instantiating and using the `Schedule` template.

number of tasks in the set, then instantiating another template (`check_i`) to perform these checks individually. The `check_i` instantiation is parameterized by the value of i it is to check; but `check_i` recursively makes another instantiation of `check_i` with the next value of i, so `RMA_Feasible` only needs to instantiate a single `check_i`. The result of these checks are composed together with logical *and* ($\wedge$), since each sub-check must be satisfied for the task set to be feasible. In this way, the final computed feasibility of the task set is dependent upon the feasibility of each sub-check.

```
struct taskA { enum { cost      = 5,
                      period    = 10 }; };
struct taskB { enum { cost      = 5,
                      period    = 15 }; };
typedef TYPELIST_2(taskA, taskB) my_tasks;
const int isFeasible =
        RMA_Feasible<my_tasks>::Result;
```

Figure 6: An example task set specification and its feasibility test.

In our example, the "size" of the task set is calculated to be 2, and `check_i<my_tasks,2,1>` is instantiated. This instantiation does two things: it computes the check for $i = 1$ (by instantiating `task_feasible<my_tasks,1>`), and, later, it will compose its result with that of the *next* `check_i`.

The `task_feasible` template's job is to find, given a fixed i and task set, a value of t for which inequality (2) holds. To do this, it must try successive values of t, chosen from the appropriate set, and compute the summation over $1 \leq j \leq i$. It uses two other templates to accomplish this—`get_t<my_tasks,1,0>`, which gets the "first" value of t (subject to an arbitrary ordering we impose on the set, discussed below), and `sum_j<my_tasks,1,t>` to compute the summation (once t has been computed).

Therefore, in our running example, we have at this point `RMA_Feasible<my_tasks>` instantiating `check_i<my_tasks,2,1>` instantiating `task_feasible<my_tasks,1>` instantiating `get_t<my_tasks,1,0>`.

`get_t`'s purpose is to compute and return a value of t based on an index (the `t_ix` parameter). This indexing scheme is arbitrary—we choose it to start with ($k = 1, l = 1$) and increase to the maximal (k,l) value pair in the set.[6] The code of `get_t` (which has an implicit $k = 0$ parameter if unspecified) first gets the period of tasks i and k and computes the maximum value permitted for l for the given k (see inequality (2)). The value of t is then computed by instantiating `get_t` to service the next-larger value of k, or, if this instantiation has a sufficient k to service index `t_ix`, then it returns the value directly (which corresponds to $l \cdot T_k$ in inequality (2)).

In our running example, `get_t<my_tasks,1,0>` computes `Ti` $= 10$, `Tk` $= 10$, `num_l` $= 1$, and `Result` $= 10$. Therefore, `task_feasible<my_tasks,1>` uses $t = 10$, and thus instantiates `sum_j<my_tasks,1,10>`.

The `sum_j<my_tasks,1,10>` instantiation is straightforward. First, notice that such an instantiation uses the default parameter $j = 0$—the summation will be recursively computed by recursively instantiating `sum_j`, and $j = 0$ serves

[6]The implementation actually uses zero-based indexes.

as the entry to this recursion. The jth task (A in our example) is given an alias J, and Cj and Tj get the values for its cost and period, respectively. my_result is computed (this is $C_j \cdot \lceil t/T_j \rceil$), and the result is summed together with further instantiations of sum_j.

Finally, task_feasible<my_tasks,1>, instantiated so long ago, performs its computation by checking to see if this sum is less than or equal to t, as required in inequality (2); if this test fails, it creates another task_feasible for another value of t. The computation continues along similar lines, and the task set is ultimately determined feasible by the compiler.

2.4 Tasks as Types

Our system models tasks as C++ types. Type systems are typically used in high-level languages to help ensure that the logical intent of the programmer matches the code as written. Generally, developers have types in mind when designing and writing programs, and making this explicit in a language can flag logical errors that are difficult to track down otherwise. We provide something analogous for real-time developers; with our constructs, various nonfunctional aspects of the program (in this case, task schedulability guarantees) become part of the structure of the program. The next section demonstrates how to signal type errors for infeasible task sets.

2.5 Feasibility and Program Correctness

Using techniques developed by Alexandrescu [1] and embodied in the Loki C++ library [2], we can easily require that a particular task set declared in a program is always feasible. We do this using the STATIC_CHECK macro of Loki, which conditionally raises a C++ type error:

```
typedef Schedule<TYPELIST_2(taskA, taskB)>
    my_schedule;
STATIC_CHECK(my_schedule::feasible,
             Schedule_Infeasible);
```

The Schedule_Infeasible macro parameter is a description string—typically, compiler output indicates this description in its error listing. The GNU C++ compiler v3.2.2 [5], for example, gives the following useful output if my_schedule is infeasible:

```
mysched.cc: In function ‘int main(void)’:
mysched.cc:20: aggregate
   ‘Loki::CompileTimeError<0>
     ERROR_Schedule_Infeasible’ has
   incomplete type and cannot be defined
```

Using this technique, a global policy can be enforced that requires every task set to be feasible. In this case, the use of the STATIC_CHECK macro is placed in the Schedule::schedule() method.[7] This will verify that every task set that could be scheduled at runtime is feasible.

[7]An additional template parameter to the Schedule template can be used to achieve maximal flexibility in specifying such a policy.

3 Extensions to the Base Model

It is possible to add a number of useful extensions to our base model. We discuss here our ideas regarding enhanced tasks. *Enhanced tasks* are used to specify task dependence and task alternation.[8] By using *traits* we can make such enhancements without changing our base code or the requirements of the Task generic-programming concept as specified in Section 2.1.

Task dependence refers to interdependence of tasks within a task set. It is important to note that RMA assumes independent tasks. We do not break that assumption here because our notion of dependence is not a dependence on a particular computational result; rather, a dependence of task A on task B is merely a requirement that any task set including task A must also include task B. This can be flexibly used to group tasks into common configurations, or to model execution dependence loosely. However, since synchronization is not taken into account in classical RMA, any computational dependence should only be a dependence upon the generated value guaranteed to complete before the start of the task performing the computation. Dependence is easily represented as a trait (Figure 7). For each type T modelling the Task concept that has

```
// default case, no dependencies
template <class Task>
struct task_dependencies {
  typedef NullType dependencies;
};

// sample specialization for My_Task
template <>
struct task_dependencies<My_Task>
  typedef TYPELIST_2(
              My_Second_Task,
              My_Third_Task) dependencies;
};
```

Figure 7: Specifying traits for task dependence.

one or more task dependencies, a task_dependencies template specialization is written for the type specifying as a Typelist the tasks upon which T depends.

Task alternation allows one task to be readily “swapped out” for another, cheaper task. This can be quite useful, especially for optional, debugging, or logging tasks that are not critical but are nice to include when other tasks do not “starve them out” of feasibility. Basically, the idea is to check the programmer-specified task set for feasibility; if the task set is infeasible, the least important task in the task set is exchanged for a cheaper alternative or dropped (if the task concept is specified as *droppable*). This process continues until either the task set becomes feasible or an infeasible task set is reached in which no constituent task can be exchanged or dropped. Alternation is traited simply (Figure 8).

[8]For brevity, implementation details are not given here.

```
// default case
template <class Task>
struct task_alternative {
  // "NullType" means no alternative
  typedef NullType alternative;
  enum { importance = 0 };
};

// sample specialization for My_Task
template <>
struct task_alternative<My_Task> {
  typedef My_Cheaper_Task alternative;
  // importance relative to other tasks
  enum { importance = 100 };
};
```

Figure 8: Specifying traits for task alternation.

4 Related Work

Template metaprogramming has been used for fast Fourier transforms [13], prime computation [12], and many other computations. Our work is similar but brings metaprogramming techniques to the compilation of real-time programs.

Other analysis tools are commercially available for real-time applications using RMS. TimeWiz [11] from TimeSys supports graphical modelling, analysis, and simulation of real-time software, and RapidSched [14] performs its real-time task analysis in a front-end for TriPacific Software's PERTS [8]. Our approach is oriented toward analysis rather than graphical modelling or simulation; however, it does the analysis inside the language itself and requires no additional tools. Further, our approach automatically stays in-sync with the program: it is *part* of the program. We do not dictate a way of arriving at an estimate for the execution budget of a task; such an estimate could certainly be reached using the analysis or simulation modes of such tools, or by other means.

5 Conclusions

We have described and presented code for a compile-time Rate-Monotonic Analysis (RMA) computation performed within the parametric type system of standard C++. We specify tasks and task sets as types to gain flexibility, and we leverage template metaprogramming mechanisms to compute feasibility of these task sets and to perform additional functions. Because our approach is entirely within the C++ language itself, we achieve complete integration with the language without a requirement of preprocessing or translation from a higher-level language. Thus, with minimal effort, real-time software developers implementing periodic task sets in C++ can apply our techniques to gain flexibility and retargetability, organize tasks into groups, easily specify task dependence and alternation, and reason metaprogrammatically about processor utilization and schedule feasibility, all within the language.

As future work, we intend to further consider compile-time precomputation of certain values that could aid in dynamic scheduling, as well as a compile-time determination as to whether static or dynamic scheduling is appropriate for the program being compiled.

Acknowledgments

We would like to thank the anonymous reviewers for their invaluable comments and suggestions.

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, 2001.

[2] A. Alexandrescu. Loki C++ library. `sourceforge.net/projects/loki-lib/`, 2003.

[3] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard*. Addison-Wesley, Reading, MA, 1999.

[4] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[5] Free Software Foundation. GCC Home Page. `gcc.gnu.org`, 2003.

[6] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm–exact characterization and average-case behaviour. *IEEE Real-Time Systems Symposium*, Dec. 1989.

[7] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, Jan. 1973.

[8] J. W. S. Liu, J.-L. Redondo, Z. Deng, T.-S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W.-K. Shih. PERTS: A prototyping environment for real-time systems. Technical Report UIUCDCS-R-93-1802, May 1993.

[9] L. Sha and J. B. Goodenough. Real-time scheduling theory and Ada. *IEEE Computer*, 23(4):53–62, 1990.

[10] B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, Boston, 2000.

[11] TimeSys Corporation. TimeSys TimeWiz. `www.timesys.com/index.cfm?hdr=tools_header.cfm&bdy=tools_bdy_model.cfm`, 2003.

[12] E. Unruh. Primzahlen. `www.erwin-unruh.de/primorig.html`, 2003.

[13] T. Veldhuizen. Techniques for scientific C++. Technical Report TR542, Indiana University, Department of Computer Science, Aug. 2000.

[14] V. F. Wolfe, R. Johnston, P. Kortmann, B. Watson, S. Wohlever, L. C. DiPippo, R. Bethmagalkar, and G. Cooper. RapidSched: Static scheduling and analysis for Real-Time CORBA. In *Proceedings of the 5th International Workshop on Real-Time Object-Oriented Dependable Systems*. IEEE, Jan. 1999.