

Interactive Execution Time Predictions using Reference Attributed Grammars

Patrik Persson and Görel Hedin

*Department of Computer Science, Lund University
Box 118, SE-221 00 Lund, Sweden
{Patrik.Persson|Gorel.Hedin}@cs.lth.se*

Abstract

A central problem for real-time scheduling is to acquire tight but conservative upper bounds on task execution times. We present a prototype for an environment where such bounds are interactively presented in terms of source code constructs to the programmer during development. The prototype is based on the language development tool APPLAB and uses an extended attribute grammar formalism, reference attributed grammars (RAGs), which overcomes some drawbacks of conventional attribute grammars in this context (e.g. description of non-local dependencies). In this paper we show how timing schemata can be implemented as RAGs. Our experience is that the RAG approach allows timing schemata to be implemented in a clear, concise, and modular manner.

1. Introduction

Real-time scheduling algorithms, such as [9], generally assume the worst-case execution time (WCET) of tasks to be known. However, few tools to predict the WCET are available. Without prediction tools, the only way to estimate the WCET is to perform actual measurements of execution times with what is believed to be the worst-case input data. Such measurements are hard to make and generally not reliable — there is typically no way of telling whether the longest observed time is in fact the longest time possible.

Our research is concerned with techniques and tools for predicting the WCET of real-time programs. We base our WCET analysis on a conceptual framework known as timing schema [12]. Since the timing schema is a set of equations for the execution times of various language constructs, it would appear suitable to implement using attribute grammars (AGs). As we will demonstrate, however, conventional AGs have a few drawbacks in this context, and we instead use an extended variant known as reference attributed grammars (RAGs). RAGs allow non-local dependencies (for example, between procedure definition and procedure call sites) to be expressed concisely [5].

Our WCET analysis techniques are designed for integration with a compiler, allowing the analysis to be based on both source level information and generated code. We base our prototype tool on APPLAB [1], an environment for interactive development of domain-specific languages. The environment integrates structure-oriented editing with semantic analysis and code generation. Our tool provides WCET predictions of the program (both in parts and as a whole) continuously as the program is developed.

APPLAB uses a specification language based on the RAG formalism. The timing schema is described as a module of its own, textually separate from the remaining semantic analysis. Other modules include name analysis, type analysis, and code generation.

In this paper, we show how the timing schema formulation can be used for the actual implementation of a WCET analysis tool, rather than as a reasoning methodology as was originally intended. We show the drawbacks of using traditional AGs for this purpose, and how these drawbacks are overcome using RAGs. We also report from other experiences from implementing our WCET prediction tool in APPLAB.

1.1. Paper outline

In Section 2, we motivate our requirements for a WCET prediction tool. We also present some related work, including the original timing schema concept and its relation to attribute grammars. Section 3 is an overview of our prototype implementation using RAGs. Section 4 concludes the paper.

2. Timing properties of real-time systems

Stankovic [13] characterizes real-time systems as follows:

A real-time system is one in which the correctness of the system depends not only on the logical results, but also on the time at which the results are produced.

In other words, a real-time system performs its computations within some timing constraints. This manifests itself in the scheduling of such a system, where the system's tasks are scheduled in time to meet the system's timing constraints (task deadlines). Depending on the nature of the system, a missed deadline may be more or less critical: in a multimedia application it may result in a transient distortion in a video stream, whereas in an engine control system it may result in an engine explosion.

2.1. Predicting the worst-case execution time

An important area of research is the prediction of the WCET of real-time tasks. A tool for this purpose should provide a prediction that is a tight upper bound of the actual WCET. That is, WCET estimations should be close to, but no lower than, the actual WCET.

The timing schema approach (presented in more detail in Section 2.2) was originally developed by Shaw [12] and was the basis for an experimental timing tool targeted at a subset of the C language [11]. However, in that work the timing schema concept was used as a “reasoning methodology for deterministic timing” rather than as an actual implementation.

As suggested in [11], the problem of bounding the WCET is twofold:

Hardware level analysis: determining the WCET (in cycles or nanoseconds) of a piece of object code on a particular hardware platform. Modern processors employ a variety of techniques to enhance performance, such as caching, pipelining, and speculative execution [6]. Although these techniques enhance average-case performance, they also make it considerably more difficult to predict the worst-case performance. Existing hardware level analysis techniques include low-level data-flow analysis [14] and the extended timing schema [8].

Structure level analysis: determining the execution time of the longest possible path in a program (or part of it) based on the execution times of individual pieces of the program. This analysis may be performed at the object code level [10] or the source code level [3].

The research we describe in this paper is focused on structure level analysis. We do, however, intend to integrate some kind of hardware level analysis in the future.

A related and interesting problem is to provide the user with feedback about the program at hand. For this information to be useful and understandable, we believe it should be expressed in terms of the source code. Consequently, we work with source code level concepts; more specifically, abstract syntax trees (ASTs).

2.2. Timing schemata and attribute grammars

In the timing schema approach, a WCET prediction with each "atomic block", where an atomic block is essentially any piece of sequential source code. (The original timing schema approach is concerned with source code rather than object code.)

The granularity of atomic blocks is a design parameter of the timing schema. In our present prototype, we have chosen to associate a constant WCET prediction with each terminal symbol. We concentrate on combining those predictions into composite predictions for non-terminals such as loops, procedures, and tasks.

Once timing predictions have been produced for the atomic blocks in a program, predictions can be calculated for composite constructions using their constituents. For example, the time of the assignment statement

```
a = b * c;
```

may be described as $T(b)+T(c)+T(*)+T(a)+T(=)$, where $T(X)$ denotes the worst-case execution (or evaluation) time of the node X in the abstract syntax tree. Similarly, the time of the if statement

```
if (exp) stmt1; else stmt2;
```

may be described as $T(exp) + T(if) + \max(T(stmt1), T(stmt2))$, where the function $\max(a, b)$ is defined to return the larger of the numbers a and b .

The timing schema formalism is clear and intuitive, and it seems an attractive option is to implement a schema using an attribute grammar. The execution time bounds of the AST nodes can be represented by synthesized attributes, and the timing schema by semantic rules. However, some important drawbacks arising in practice will be addressed in Section 3.1.

2.3. The role of timing assertions

The timing constraints mentioned in the beginning of Section 2 are typically known at the time of system design. Software engineers developing a real-time system must be aware of these requirements throughout the development, including during the implementation phase. A while loop, for example, must never loop more than some fixed number of times. If no such bound exists, the WCET of the algorithm is unknown and the system cannot be safely scheduled. Although this information could in some special cases be extracted by an analysis tool, it is in fact an important part of the design.

We allow timing information to be explicitly expressed in the code using what we call *timing assertions*. These represent system design parameters and facilitate a more accurate analysis.

Assertions can also play a slightly different role. In some applications (in particular, multimedia applications), it is desirable to schedule some tasks within tighter bounds than the WCET (giving higher performance/throughput at the cost of sporadic missed deadlines). Timing assertions complemented with time-out exceptions allow real-time tasks to fall back on simpler algorithms when deadlines are missed.

A third use of timing assertions relates to object-oriented languages. The actual code to execute at a virtual method call is determined at run-time based on object type information. Except in some

special cases, it is not possible to statically determine which code is actually called. In cases where a global analysis of all possible implementations of a virtual method is not possible, a timing assertion can be associated with the top-level declaration of a method. This way each implementation of the virtual method can be checked to ensure that its execution time does not exceed the execution time specified by the assertion.

2.4. Desired tool support

To conclude this section, we propose a WCET prediction tool with the following properties:

- *Interactive, source code oriented user feedback.* Tools and methods for non-real-time systems do not generally address the execution time; it is considered a low-level property of the finished implementation. For real-time systems, however, the timing properties (in particular, the execution time) are an inherent part of the system's behaviour. Execution time requirements are specified at design time. A tool should allow the user to monitor the execution time throughout development, at different granularity levels such as loops, procedures, and entire tasks.
- *Possible integration with low-level hardware specific WCET prediction techniques.* In Section 2.1, the WCET prediction was separated into two subproblems. While we focus on the structure level aspects of WCET analysis, it is desirable to provide for integration with hardware level prediction.
- *Support for assertions in the WCET analysis.* A system should allow the user to specify timing constraints (maximum bounds for the execution time/number of loop iterations) for loops, procedures, and tasks.

3. An interactive WCET prediction tool based on reference attributed grammars

We have implemented a prototype for a WCET prediction tool. The prototype tool operates on a simple experiment language supporting some advanced object-oriented language concepts such as classes, inheritance, and qualified access. The language bears some similarities to Java [4], which we intend to support in the near future.

In the rest of this section, we present some of the considerations that have to be made when implementing timing schemata using attribute grammars. We also show how timing schemata may be implemented using RAGs.

3.1. Implementing timing schemata using conventional attribute grammars

In Section 2.2, attribute grammars were suggested as an attractive technique for implementing timing schemata. Although viable for small examples in simple languages, this approach has a number of drawbacks when applied to a typical procedure- or object-oriented programming language. To see why, consider a function call:

$$f(a, b)$$

A reasonable approach to describing the execution time of the call might be $T(a) + T(b) + T(f_{call}) + T(f_{body})$, where $T(a)$ and $T(b)$ refer to the time to evaluate the arguments, $T(f_{call})$ refers to the time

for the function call itself, and $T(f_{body})$ refers to the execution time of the function body. The problem is the term $T(f_{body})$. While the other terms are directly available as attributes of either the call itself or its immediate children, the body of the called function is, in general, not automatically available at the call site. Instead it may be located arbitrarily far away in the AST.

One possible approach to making the timing information available is to integrate the timing analysis with the name analysis. The usual approach to name analysis is to make use of an inherited attribute *env* containing a mapping from visible identifier names to declaration information (such as variable types and function signatures). The declaration information in this mapping could be extended with an execution time value, i.e. a relatively modest modification of the specification. However, modularity is lost since WCET analysis is integrated with the conceptually quite unrelated name analysis.

To keep WCET analysis separate from name analysis, a table could be associated with each scope in the program. Such a table would hold the execution time of the body of every procedure declared in the scope. In a conventional procedural language, the execution times of function bodies would be available at the call sites by making the table an inherited attribute in the son nodes of the scope. In other words, this approach involves another name analysis.

Name analysis for object-oriented languages with inheritance and qualified access is more complex, since the dependencies between declarations and uses of identifiers do not follow the block structure of programs. Regardless of which of the two approaches above is chosen, this complexity is reflected in the WCET analysis. Either modularity is lost or a substantial amount of administration is required to describe what is essentially a simple relationship between a function call and the corresponding function body.

3.2. The RAG implementation in the APPLAB system

We have implemented WCET analysis for a simple object-oriented language in our interactive language tool APPLAB (APPLication language LABoratory), an environment originally designed for interactive development of domain-specific languages [1, 2]. APPLAB is based on reference attributed grammars (RAGs) which is an extension to attribute grammars that allows attributes to be references to syntax nodes, supporting specification of non-local dependencies. For example, a use of an identifier may have a reference attribute denoting the corresponding declaration node.

The APPLAB system allows language specifications (RAGs) and programs to be edited simultaneously. A program is edited using a language based editor controlled by the language specification, allowing attributes of the syntax tree for the program to be displayed. We used this feature to interactively monitor the WCET for the edited program. Figure 1 shows a screendump from the system.

In our prototype, WCET analysis is implemented as a separate grammar module. That module is concerned solely with the WCET analysis (thus corresponding directly to a timing schema) but uses results (attributes) computed by the name analysis module, as will be shown below.

We focus on the structure level of WCET prediction as described in Section 2.1, that is, computation of WCET predictions for programs, loops, and methods based on given WCET predictions of individual atomic blocks. (In our case, these atomic blocks correspond to terminal symbols.) In the present prototype, we have assigned constant WCET predictions to all basic operations, such as assignment, arithmetic operations, and so on.

The remainder of this section is based on the small object-oriented language whose context-free syntax is given in Table 1. The language supports variables, classes, non-virtual functions, qualified access, and statements such as assignments and while statements. The grammar notation in APPLAB makes use of an object-oriented extension of RAGs where the nonterminals may be organized in a class hierarchy, and where the productions are leaves in this hierarchy. In Table 1 a superclass is shown to the left of its subclasses (e. g. *Use* is a subclass of *Exp*). Attributes and semantic rules are

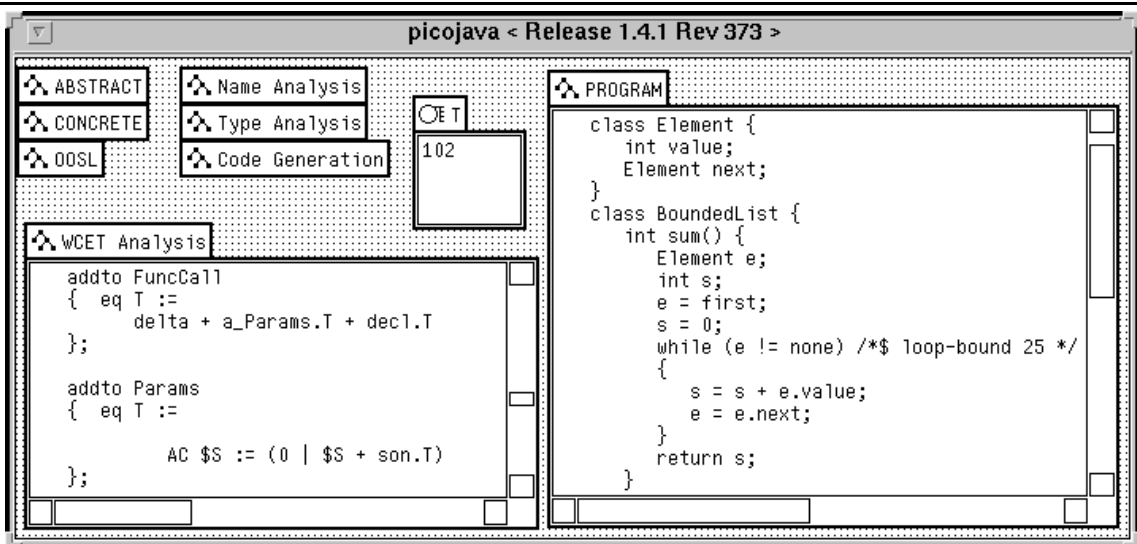


Figure 1: Example WCET analysis in APPLAB. The window labelled T shows the WCET prediction (the attribute T) of a portion of the program in the PROGRAM window.

inherited (in the object-oriented sense) along this hierarchy. For example, an attribute declared in the *Exp* nonterminal will be present in any node which is a subclass of *Exp*, e.g. *QualUse*, *SimpleUse*, or *FuncCall*.

The name analysis attribution module connects uses of identifiers to their declarations. The details of this attribution module are described in [5] and results in a synthesized attribute *decl* of the *Use* nonterminal denoting the corresponding *Decl* node. Table 2 shows this brief interface to the name analysis module.

The WCET analysis module is shown in Tables 3, 4, and 5. In Table 3, a synthesized attribute T (modelling the WCET) is declared for the nonterminals *Stmt* and *Exp*, and semantic rules in *AssignStmt*, *SimpleUse*, and *QualUse* define values for this attribute. (We have denoted the WCET of the actual assignment by α , the WCET of a variable access by β , and the WCET of dereferencing a pointer by γ .) These equations could be expressed in most attribute grammar formalisms, since the *Use* and *Exp* instances associated with an *AssignStmt* instance are immediately available.

Function calls are described as a special use of identifiers, that is, we have a production *FuncCall* which is a subclass of *Use* (refer to Table 3). Table 4 shows how function calls are described in our timing schema. The WCET of a function call is represented as a sum of the time for the actual call (represented by the constant δ), the time to evaluate the parameters, and the time to execute the function.

The WCET analysis uses non-local references to analyze function calls. The *decl* attribute defined in the name analysis module is a reference to the declaration of the called function, a non-local reference. The reference attribute allows the T attribute of the declaration (a *FuncDecl*) to be accessed directly in the *FuncCall* production, without the need for introducing auxiliary attributes on the path from the *FuncDecl* to the *FuncCall* in the syntax tree. The WCET analysis module is thus completely independent on the scope rules of the language: the rule for *FuncCall* is the same regardless of if the language is a procedural language with nested functions, or an object-oriented language (like here) where functions are inherited from superclasses to subclasses and can be accessed via a qualifying reference, or if any other scope rules are applied. However, to support also virtual functions (where the function body is not decided until runtime) the specification has to be extended.

Nonterminals		Productions
<i>Program</i>		$\rightarrow \text{program } \{ \text{Block} \}$
<i>Block</i>		$\rightarrow \text{Decls } \text{Stmts}$
<i>Decls</i>		$\rightarrow \text{Decl}^*$
<i>Stmts</i>		$\rightarrow \text{Stmt}^*$
<i>DeclType</i>		<i>IntDeclType</i> : $\rightarrow \text{int}$ <i>BoolDeclType</i> : $\rightarrow \text{boolean}$ <i>RefDeclType</i> : $\rightarrow \text{SimpleUse}$
<i>Decl</i>		<i>ClassDecl</i> : $\rightarrow \text{class } \text{ID } \text{SuperOpt } \{ \text{Block} \}$ <i>VarDecl</i> : $\rightarrow \text{DeclType } \text{ID}$ <i>FuncDecl</i> : $\rightarrow \text{DeclType } \text{ID } (\text{FormalParams}) \{ \text{Block} \}$
<i>SuperOpt</i>		<i>Super</i> : $\rightarrow \text{extends } \text{Use}$ <i>NoSuper</i> : \rightarrow
<i>FormalParams</i>		$\rightarrow \text{VarDecl}^*$
<i>Stmt</i>		<i>AssignStmt</i> : $\rightarrow \text{Use} = \text{Exp}$ <i>WhileStmt</i> : $\rightarrow \text{while } (\text{Exp}) \text{LoopBound } \text{Stmt}$ <i>CompoundStmt</i> : $\rightarrow \{ \text{Block} \}$
<i>Exp</i>	<i>Use</i>	<i>UnQualUse</i>
		<i>QualUse</i> : $\rightarrow \text{Use} . \text{UnQualUse}$
		<i>SimpleUse</i> : $\rightarrow \text{ID}$ <i>FuncCall</i> : $\rightarrow \text{ID } (\text{Params})$
<i>LoopBound</i>		$\rightarrow /*\$ \text{loop-bound } \text{INT} */$
<i>Params</i>		$\rightarrow \text{Exp}^*$

Table 1: Context-free grammar. (Some minor details, mainly syntactic sugar, have been omitted.)

Nonterminals	Attributes
<i>Use</i>	$\uparrow \text{decl} : \text{ref } \text{Decl}$

Table 2: Interface to name analysis module.

Nonterminals	Attributes and Semantic Rules
<i>Exp</i>	$\uparrow T : \text{integer}$
<i>Stmt</i>	$\uparrow T : \text{integer}$
<i>AssignStmt</i>	$T := \alpha + \text{Use}.T + \text{Exp}.T$
<i>SimpleUse</i>	$T := \beta$
<i>QualUse</i>	$T := \gamma + \text{Use}.T + \text{UnQualUse}.T$

Table 3: Excerpt from the timing schema module.

Nonterminals	Attributes and Semantic Rules
<i>Decl</i>	$\uparrow T : \mathbf{integer}$
<i>FuncDecl</i>	$T := \mathit{Block}.T$
<i>FuncCall</i>	$T := \delta + \mathit{Params}.T + \mathit{decl}.T$
<i>Params</i>	$\uparrow T : \mathbf{integer}$ $T := \sum_{e \in \mathit{Exp}^*} e.T$
<i>Block</i>	$\uparrow T : \mathbf{integer}$ $T := \mathit{Stmts}.T$
<i>Stmts</i>	$\uparrow T : \mathbf{integer}$ $T := \sum_{s \in \mathit{Stmt}^*} s.T$

Table 4: Timing schema for function declarations and calls.

Nonterminals	Semantic Rules
<i>WhileStmt</i>	$T := \epsilon + \mathit{Exp}.T + \mathit{LoopBound}.INT.val * (\zeta + \mathit{Exp}.T + \mathit{Stmt}.T)$

Table 5: Timing schema for while statements with timing assertions. The expression $INT.val$ denotes the numeric value of the integer constant INT .

A use of timing assertions is given in the while statement in Table 5. The statement has an associated loop bound (an integer constant) that states the maximal number of loop iterations. (INT is a predefined non-terminal modelling an integer constant.) The WCET of the while statement also includes the constants ϵ and ζ . The former accounts for any administration associated with the start and end of the while statement, and the latter for the overhead associated with each iteration.

A while statement in the language may, for example, look as follows (the compound statement is executed at most 10 times):

```
while (flag) /*$ loop-bound 10 */ {
    ...
}
```

The assertion represents the programmer’s knowledge about the maximal number of iterations. (As discussed in Section 2.3, such bounds must exist and be known in hard real-time systems.) Compatibility with traditional compilers is ensured by expressing assertions in terms of special ‘tagged’ comments, similar in concept to the ones used by Javadoc [4]. This technique allows other compilers to parse the code without problems, while our tool can still predict the WCET.

3.3. Discussion

Implementation of timing schemata using RAGs in APPLAB suggests at least three major advantages:

- *RAGs provide reference attributes.* These permit timing schemata to be implemented in a clear and concise manner. While the traditional AG formalism resembles the timing schema concept, the non-local dependencies that frequently occur in WCET analysis are difficult to describe in traditional AGs. RAGs allow these non-local dependencies to be described in a concise way.
- *RAG modules provide modularity.* The timing schema of a language is described as a module of its own, separate from the name analysis module. The WCET module is concerned solely

with the timing properties of the language at hand, and strongly resembles the timing schema. Although the timing schema is textually separate from other modules, it can exploit results from those modules. The timing schema module uses, for instance, the *decl* reference from a procedure call site to the corresponding procedure declaration. That reference is defined in the name analysis module. This makes the timing schema module independent of the scope rules of the language.

- *APPLAB provides interactivity.* For reasons given in Section 2.4, we want WCET predictions to be readily available throughout the development of real-time software. APPLAB allows an attribute, such as the WCET prediction of an arbitrary subtree of the AST, to be viewed at any time without noticeable delay.

4. Conclusions

We have presented an approach to predicting the worst-case execution time of real-time tasks. In our tool these execution time predictions are associated with source level constructs and continuously updated as the program is developed. We suggest this interactivity to be especially useful in the development of real-time software, where the timing properties must be kept in mind throughout the development process.

We base our approach on the timing schema concept [12], an intuitive way of expressing timing properties of programming languages. Although attribute grammars seem superficially suitable for implementing timing schemata, the frequently occurring non-local references are not easily described. The reference attributed grammar (RAG) formalism [5] overcomes this problem and allows timing schemata to be described as equations in a straightforward manner.

Rather than just using the timing schema as a specification, we use it for the actual implementation. The RAG based specification language of APPLAB allows attributes and equations for a particular purpose to be encapsulated into a separate module. In our prototype, one module is concerned solely with the timing schema and contains only the attributes and equations associated with that analysis. That module bears a striking resemblance to the timing schema formulation and holds all information that is required for the structure level WCET analysis.

Although our work is concerned with structure level analysis, we acknowledge the need to predict the low-level hardware timing behaviour. For example, the extended timing schema approach [8] (developed to predict timing behaviour of RISC processors utilizing caches and pipelining) seems suitable for integration with our approach.

4.1. Future work

The initial prototype presented in this paper deals with a fictive programming language. We intend to extend our environment to handle the Java programming language [4] in the near future. This would require us to address some WCET prediction problems specific to object-oriented languages:

- *Predicting the WCET of virtual method invocations.* It is not possible (in general) to statically determine which code is executed upon an invocation of a virtual method. We plan to make use of timing assertions (as briefly mentioned in Section 2.3) to handle this problem.
- *Predicting the overhead of automatic memory management.* Java employs garbage collection, which may impose unpredictable interruptions of the execution unless special action is taken. The problem of predicting execution time overhead recently has been addressed by integrating garbage collection work with task scheduling [7]. However, some work remains, such as predicting

the maximal allocation rate and the maximal amount of live memory possible in a task. We plan to investigate a technique similar to timing schema for this purpose.

Acknowledgments

We wish to thank Boris Magnusson and Klas Nilsson for numerous ideas and suggestions for our work, and Elizabeth Bjarnason for developing most of APPLAB. The anonymous reviewers and Margaret Newman-Nowicka provided several helpful comments.

The work of Patrik Persson is financed by ARTES (A network for Real-Time research and graduate Education in Sweden). The work of Görel Hedin is financed by NUTEK (the Swedish National Board for Industrial and Technical Development).

Bibliography

- [1] E. Bjarnason: *Interactive Tool Support for Domain-Specific Languages*. Licentiate Thesis, Dept. of Computer Science, Lund University, December 1997.
- [2] E. Bjarnason, G. Hedin, K. Nilsson: Interactive Language Development for Embedded Systems. To appear in *Nordic Journal of Computing*.
- [3] A. Ermedahl, J. Gustafsson: Deriving Annotations for Tight Calculations of Execution Time. *Proceedings of EuroPar '97*, LNCS 1300, Springer, August 1997.
- [4] J. Gosling, B. Joy, G. Steele: *The Java Language Language Specification*. Addison-Wesley, 1996.
- [5] G. Hedin: Reference Attributed Grammars. To be presented at *The Second International Workshop on Attribute Grammars and their Applications (WAGA '99)*, Amsterdam, the Netherlands, March 1999.
- [6] J. L. Hennessy, D. A. Patterson: *Computer Architecture: A Quantitative Approach (Second Edition)*. Morgan Kaufmann, 1996.
- [7] R. Henriksson: *Scheduling Garbage Collection for Embedded Systems*. Ph. D. Thesis, Dept. of Computer Science, Lund University, Sweden, September 1998.
- [8] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, C. S. Kim: An Accurate Worst Case Timing Analysis Technique for RISC Processors. *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.
- [9] C. L. Liu, J. W. Layland: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1, 1973.
- [10] T. Lundqvist, P. Stenström: Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. Technical Report No. 93-3, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, February 1998.
- [11] C. Y. Park, A. C. Shaw: Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, Vol. 24, No. 5, 1991.
- [12] A. C. Shaw: Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, 1989.
- [13] J. A. Stankovic et al.: Strategic Directions in Real-Time and Embedded Systems. *ACM Computing Surveys*, Vol. 28, No. 4, 1996.

- [14] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, M. G. Harmon: Timing Analysis for Data Caches and Set-Associative Caches. *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1997.

