Towards Formally Verifiable WCET Analysis for a Functional Programming Language

Kevin Hammond* Christian Ferdinand † Reinhold Heckmann† Roy Dyckhoff*
Martin Hofmann‡ Steffen Jost* Hans-Wolfgang Loidl‡ Greg Michaelson§
Robert Pointon§ Norman Scaife¶ Jocelyn Sérot¶ Andy Wallace§

Abstract

This paper describes ongoing work aimed at the construction of formal cost models and analyses to yield verifiable guarantees of resource usage in the context of real-time embedded systems. Our work is conducted in terms of the domain-specific language *Hume*, a language that combines *functional programming* for computations with *finite-state automata* for specifying reactive systems. We outline an approach in which high-level information derived from source-code analysis can be combined with worst-case execution time information obtained from high quality abstract interpretation of low-level binary code.

1 Introduction

The EU Framework VI EmBounded project (IST-2004-510255) aims to automatically determine strong resource bounds for high-level programming language features. We aim to obtain formally verifiable certificates of bounds on resource usage from a source program through automatic analysis.

1.1 The Hume Language

Our work is undertaken in the context of Hume [13], a functionally-based domain-specific high-level programming language for real-time embedded systems. Hume is designed as a layered language where the *coordination layer* is used to construct reactive systems using a finite-state-automata based notation; while the *expression layer* is used to structure computations using a strict purely functional rule-based notation that maps patterns to expressions. The coordination layer expresses reactive Hume programs as a static system of interconnecting *boxes*. If each box has bounded space cost internally, it follows that the system as a whole also has bounded space cost. Similarly, if each box has bounded time cost, a simple schedulability analysis can be used to determine reaction times to specific inputs, rates of reaction and other important real-time properties. Expressions can be classified according to a number of levels where lower levels lose abstraction/expressibility, but gain in terms of the properties that can be inferred. For example, the bounds on costs inferred for recursive functions will usually be less accurate than those for non-recursive programs, and cannot always be deduced.

Previous papers have considered the Hume language design in the general context of programming languages for real-time systems [13], and specifically functional notations for bounded computations [12], described a heap and stack analysis for FSM-Hume [14], and considered the relationship of Hume with classical finite-state machines [17]. The main contribution of this paper is to outline a worst-case execution time analysis for Hume combining high- and low-level information, where source-level information on the costs of recursive functions, conditional expressions etc. is combined with machine-level information on cache behaviour, pipelines etc.

We will illustrate the design of Hume with a simple control example for a reactive system: the controller for a drinks vending machine. Figure 1 shows the Hume box diagram for this system, and the code is shown below. Note that `*` in an input or output position is used to indicate that that position is ignored, and that all inputs and outputs are matched asynchronously.

```
type Cash = int 8;

data Coins =  Nickel | Dime;
data Drinks = Coffee | Tea;
data Buttons = BCoffee | BTea | BCancel;

-- vending machine control box
```

*School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9SX.
email: {kh,rd,jost}@dcs.st-and.ac.uk.
†AbsInt GmbH, Saarbrücken, Germany.
email: {cf,heckmann}@absint.com
‡Ludwig-Maximilians Universität, München.
email: {mhofmann,hwloidl}@informatik.uni-muenchen.de
§Depts. of Comp. Sci. and Elec. Eng., Heriot-Watt University, Riccarton, Edinburgh, Scotland.
email: {G.Michaelson, A.M.Wallace}@hw.ac.uk
¶LASMEA, Université Blaise-Pascal, Clermont-Ferrand, France.
email: Jocelyn.SEROT@univ-bpclermont.fr
This work has been supported by EU Framework VI grant IST-2004-510255 and by EPSRC Grant EPC/0001346.

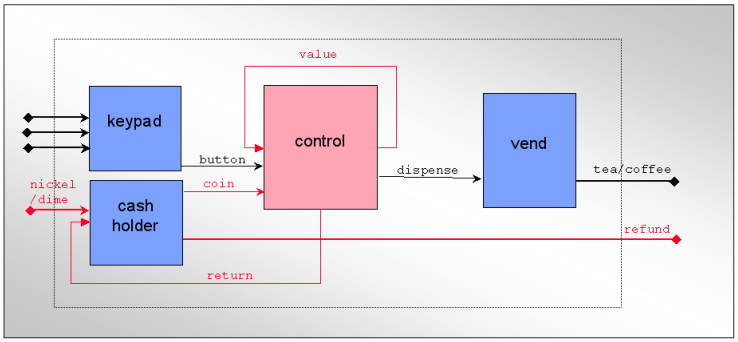

Figure 1: Hume example: vending machine box diagram

```
box control
in  ( coin :: Coins, button :: Buttons,
                           value :: Cash )
out ( drink :: Drinks, value' :: Cash,
                           return :: Cash )
match
    ( Nickel, *,  v )  ->  ( *, v +  5, * )
  | ( Dime,   *,  v )  ->  ( *, v + 10, * )
  | ( *, BCoffee, v )  ->  vend Coffee 10 v
  | ( *, BTea,    v )  ->  vend Tea 5 v
  | ( *, BCancel, v )  ->  ( *, 0, v )
  ;

vend drink cost v =
   if v >= cost then ( drink, v-cost, * )
                else ( *,     v,      * );
```

Functional languages have rarely been applied to hard real-time systems, partly because they are perceived as hard to cost. The most widely used *soft* real-time functional language is the impure, strict language Erlang [1], which has been succesfully used in many large commercial applications. However, there have also been attempts to apply pure functional languages to soft real-time settings (e.g. [24, ?, 25]). Few, if any, of these approaches provide strong cost models, however. Synchronous dataflow languages such as Lustre [6] or Signal [11] have strong similarities with a functional approach, being similarly declarative. The primary difference from our approach is that Hume also supports asynchronicity, is built around state machines, and provides a highly-expressive programming environment including rich data structures, recursion, and higher-order functions, while still providing a strong cost model.

2 A Source-Level Cost Model for Hume Expressions

Our approach involves producing a formally verifiable upper bound cost model for Hume programs that is related to actual execution costs. We can use this model to produce high-level static analyses for determining bounds on recursive calls, iterative loops etc. In this paper, we use an abstract machine approach, where execution costs are associated with abstract machine instructions, and where these costs are related to Hume source forms through formal translation. In this way, the mathematical cost model can be insulated from changes in the concrete architecture, being effectively parameterised by cost information for each abstract machine operation. A *correspondence proof* (omitted here, for brevity) formally relates the high-level model to costs expressed in terms of the abstract machine. In this way, we are able to prove that the source level timing information we give here is an upper bound on actual exection costs, provided only that the timing information for each abstract machine instruction is a true upper bound on the execution cost of that instruction, including effects of pipelining and cache behaviour. The use of purely functional expressions within Hume boxes simplifies both the construction of the cost model and the corresponding proofs, by avoiding the need for dataflow analysis, alias analysis and other consequences of the use of side-effects. This also improves the accuracy of the result.

As a proof-of-concept, we have chosen a simple, high level stack-based machine, the Hume Abstract Machine (or HAM). The approach can, however, be generalised to other abstract machine designs or to direct compilation, as required.

The formal statement $\mathcal{V}, \eta \vdash^{t\,|\,p\,|\,m}_{t'\,|\,p'\,|\,m'} e \rightsquigarrow \ell, \eta'$ may be read as follows: given the value environment $\mathcal{V}$ and initial heap η, expression e evaluates in a finite number of steps to a result value stored at location ℓ in the modified heap η', provided that there were t time, p stack and m heap units available before computation. Furthermore, at least t' time, p' stack and m' heap units are unused after evaluation. We illustrate the approach by showing a few sample rules covering key expression forms.

Variables are simply looked up from the environment and the corresponding value pushed on

the stack. The time cost of this is the cost of the `PushVar` instruction, shown here as `Tpushvar`. Concrete values for this constant can be obtained using either measurement-based approaches [2] or abstract interpretation (see Section 3). There is no heap cost.

$$\frac{\mathcal{V}(x) = \ell}{\mathcal{V}, \eta \vdash \frac{t' + \texttt{Tpushvar}}{t'} \mid \frac{p'+1}{p'} \mid \frac{m}{m}\ x \rightsquigarrow \ell, \eta} \text{ (Variable)}$$

There are three rules for conditionals: two symmetric cases where the condition is true or false, respectively; and a third case to deal with exceptions (omitted here). In the case of a true/false condition the time cost is the cost of evaluating the conditional expression, plus the cost of evaluating an `If` instruction `Tiftrue`/`Tiffalse` plus the cost of executing the true/false branch, plus the cost of a goto if the condition is false.

$$\frac{\begin{array}{c}\mathcal{V}, \eta \vdash \frac{t_1}{t_1'} \mid \frac{p}{p'} \mid \frac{m}{m'}\ e_1 \rightsquigarrow \ell, \eta' \\ \eta'(\ell) = (\mathsf{bool}, \mathsf{ff}) \\ \mathcal{V}, \eta' \vdash \frac{t_1' - \texttt{Tiffalse}}{t_3'} \mid \frac{p'+1}{p''} \mid \frac{m'}{m''}\ e_3 \rightsquigarrow \ell'', \eta''\end{array}}{\mathcal{V}, \eta \vdash \frac{t_1}{t_3' - \texttt{Tgoto}} \mid \frac{p}{p''} \mid \frac{m}{m''}\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \ell'', \eta''} \text{ (Conditional False)}$$

The Call rule deals with calls to some function fid, whether or not this is recursive. Each argument to the call is evaluated, and then the function is applied used App. The cost of the call is `Tcall` and the cost of completing the call is `Tslide`, where the underlying `Slide` instruction removes function arguments from the stack, whilst preserving the return value.

$$\frac{\begin{array}{c}\mathcal{V}, \eta_{(i-1)} \vdash \frac{t_{(i-1)}}{t_i} \mid \frac{p_{(i-1)}}{p_i} \mid \frac{m_{(i-1)}}{m_i}\ e_i \rightsquigarrow \ell_i, \eta_i \\ \mathcal{V}, \eta_k \vdash \frac{t_k - \texttt{Tcall}}{t_a'} \mid \frac{p_k}{p'} \mid \frac{m_k}{m'}\ \textsc{App}(fid, [\ell_k, \ldots, \ell_1]) \rightsquigarrow \ell, \eta'\end{array}}{\mathcal{V}, \eta_0 \vdash \frac{t_0}{t_a' - \texttt{Tslide}} \mid \frac{p_0}{p'+k} \mid \frac{m_0}{m'}\ fid\ e_k \cdots e_1 \rightsquigarrow \ell, \eta'} \text{ (Call)}$$

These rules can be easily extended to cover other expression forms and boxes, so giving a complete cost model for Hume. From this cost model, it is possible to derive a number of behavioural properties. The most important are that the cost model correctly captures the potential change in time and memory usage and that the result of execution is always left as an extra value on the stack. In order to produce this proof, we construct a formal translation from Hume to HAM, and prove for each case that the costs of the HAM translation are precisely captured in the cost model for the Hume source.

We have produced a prototype implementation of an analysis for space usage with non-recursive functions, based on this cost model [14], and calibrated against our abstract machine implementation. We are now working on extending the analysis to recursive functions and to include time information.

3 WCET Analysis using Abstract Interpretation

Our objective is to develop a combined high- and low- level analysis for worst-case execution time. We will achieve this by extending the stack and heap cost model presented above with the addition of parameters representing actual timing costs. Our ultimate aim is to produce accurate worst-case cost information from source level programs.

The AbsInt **aiT** tool (described below) uses abstract interpretation to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a given program point. These results can be combined with ILP (Integer Linear Programming) techniques to safely predict the worst-case execution time and a corresponding worst-case execution path.

The AbsInt analysis works at a code snippet level, analyzing imperative C-style code snippets to derive safe upper bounds on the worst-case time behavior. Whilst the AbsInt analysis works at a level that is more abstract than simple basic blocks, providing analyses for loops, conditionals and non-recursive subroutines, it is not presently capable of managing the complex forms of recursion which occur in functional languages such as Hume. We are thus motivated to link the two levels of analysis, combining information on recursion bounds and other high-level constructs from the Hume source analysis with the low-level worst-case execution time analysis from the AbsInt analysis.

3.1 WCET Prediction

Static determination of *worst-case execution time* (WCET) in real-time systems is an essential part of the analyses of overall response time and of quality of service [4, 18]. However, WCET analysis is a challenging issue, as the complexity of interaction between the software and hardware system components often results in very pessimistic WCET estimates. For modern architectures such as the Motorola PPC755, for example, WCET prediction based on simple weighted instruction counts may result in an over-estimate of time usage by a factor of 250. Obtaining high-quality WCET results is important to avoid seriously over-engineering real-time embedded systems, which would result in con-

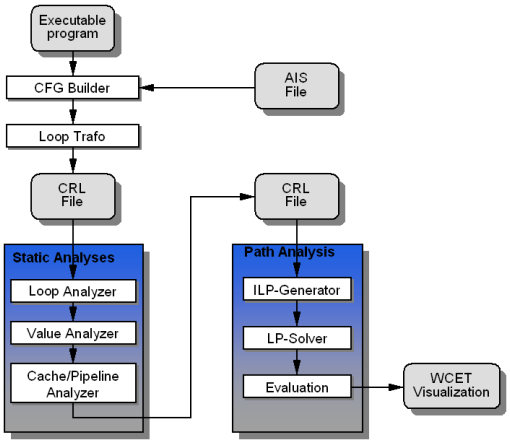

Figure 2: Phases of WCET computation

siderable and unnecessary hardware costs for the large production runs that are often required.

Three competing technologies can be used for worst-case execution time analysis: *experimental* (or testing-based) approaches [26], *probabilistic measurement* [2, 3] and *static analysis*. Experimental approaches determine worst-case execution costs by (repeated and careful) measurement of real executions, using either software or hardware monitoring. However, they cannot guarantee upper bounds on execution cost. Probabilistic approaches similarly do not provide absolute guaranteed upper bounds, but are cheap to construct and deliver more accurate costs than simple experimental approaches [2].

Motivated by the problems of measurement-based methods for WCET estimation, AbsInt GmbH has investigated a new approach based on static program analysis [16, 15]. The approach relies on the computation of abstract cache and pipeline states for every program point and execution context using *abstract interpretation*. These abstract states provide safe approximations for all possible concrete cache and pipeline states, and provide the basis for an accurate timing of hardware instructions, which leads to safe and precise WCET calculations that are valid for all executions of the application.

3.2 Phases of WCET Computation

In AbsInt's approach [9] the WCET of a program task is determined in several phases (see Figure 2):

- **CFG Building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from an executable binary program;
- **Value Analysis** computes address ranges for instructions accessing memory;
- **Cache Analysis** classifies memory references as cache misses or hits [10];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [16];
- **Path Analysis** determines a worst-case execution path of the program [22].

The cache analysis phase uses the results of the value analysis phase to predict the behavior of the (data) cache based on the range of values that can occur in the program. The results of the cache analysis are then used within the pipeline analysis to allow prediction of those pipeline stalls that may be due to cache misses. The combined results of the cache and pipeline analyses are used to compute the execution times of specific program paths. By separating the WCET determination into several phases, it becomes possible to use different analysis methods that are tailored to the specific subtasks. Value analysis, cache analysis, and pipeline analysis are all implemented using abstract interpretation [7], a semantics-based method for static program analysis. Integer linear programming is then used for the final path analysis phase.

The techniques described above have been incorporated into AbsInt's **aiT** WCET analyzer tools, that are widely used in industry [20, 5, 8, 27, 19]. For example, they have been used to demonstrate the correct timing behavior of the new Airbus A380 fly-by-wire computer software in a certification process according to DO178B level A [23, 21]. For this purpose, **aiT** for MPC755 and **aiT** for TMS320C33 will be qualified as verification tools according to DO178B.

3.3 Linking the Analyses

In order to link the two levels of analysis, we must base the costs for time potentials in the cost model (`Tpushvar` etc) on actual times for execution on the Hume Abstract Machine using information obtained from the **aiT** tool. In this way, we will have constructed a complete time cost model and analysis from Hume source to actual machine code.

Pragmatically, in order to obtain timing information from the **aiT** tool, our high level analysis must be adapted to output information on the limits on recursion bounds and other high-level constraints derived from the program source that can be fed to the **aiT** tool using its native system specification language (**aiS**). This information must be provided in terms of the compiled executable code that has been produced from the Hume source rather than directly from the source itself. It will

Cost	**aiT** bound for M32 (cycles)	Prob. bound for PPC (μs)
Tiftrue	30	0.051
Tiffalse	30	0.051
Tpushvar	109	0.110
Tmatchint	30. . .32	0.047
Tmatchedrule	11	0.039
Tmatchrule	22	0.053
Tmatchnone	11	0.040
Tconsumeset	82	—
Tmkint	220 . . . 223	0.046
Tcopyarg	110	0.045

Figure 3: WCET bounds on HAM instructions

therefore also be necessary to provide details of the compilation process in an appropriate form.

3.3.1 Preliminary WCET Results

This section reports timing results obtained using the **aiT** tool using the IAR C-compiler for the Renesas M32C. The M32C is a 32-bit architecture designed for typical automotive applications. It has a complex instruction set and a three-stage pipeline, but neither data nor instruction cache. Instruction cache analysis is therefore disabled. Figure 3 gives timings obtained from **aiT** for some sample HAM abstract machine instructions on the M32, and the corresponding costs on a 1.25GHz PowerPC G4 obtained using a probabilistic approach over 1,000,000 executions of each instruction (we have not yet been able to obtain probabilistic cost information for the M32, though we expect to be able to achieve this soon). While there are some clear differences in the underlying implementation of the instructions on the two architectures (notably for `Tmkint`, which allocates heap in external memory), there are also broad similarities.

An interesting observation is that combining the WCET costs of individual HAM instructions gives a result that is usually within 1-2% of the WCET bound of the complete sequence of instructions. While this observation certainly holds as long as the code on the M32C is executed from internal memory with single cycle access time, for slower, external memory, the internal instruction buffer of the M32C might have a bigger influence. This means that for "simple" architectures, WCET bounds for Hume-like languages can be computed by considering WCET bounds of individual abstract machine instructions.

4 Conclusions

We have introduced Hume and shown how a cost model can be constructed to expose time, stack and heap cost information. We have also outlined how our work can be extended in order to synthesise worst-case execution time costs using a combination of source- and binary-based analysis. Our work is formally based and motivated: we aim to construct formal models of behaviour at source program and abstract machine levels; have provided elsewhere a formal translation between these levels; and will synthesise actual worst-case execution time costs using abstract interpretation of binary programs.

References

[1] J. Armstrong, S.R. Virding, and M.C. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[2] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proc. 12th Euromicro Intl. Conf. on Real-Time Systems*, Stockholm, June 2000.

[3] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc 23rd IEEE Real-Time Systems Symp. (RTSS 2002)*, Dec 2002.

[4] A. Burns and A.J. Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley Longman, 2001.

[5] Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Applying static WCET analysis to automotive communication software. In *17th Euromicro Conf. on Real-Time Systems, (ECRTS'05)*, July 2005.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Place. Lustre: a Declarative Language for Programming Synchronous Systems. In *Proc. ACM Symp. on Princ. of Prog. Langs. (POPL '87)*, 1987.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. on Princ. of Prog. Langs. (POPL '77)*, pages 238–252, 1977.

[8] Ola Eriksson. Evaluation of static time analysis for CC systems. Technical report, Mälardalen University, August 2005.

[9] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. EMSOFT 2001, First Workshop on Embedded Software, Springer-Verlag LNCS 2211*, pages 469–485, 2001.

[10] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems, Saarland University, Saarbrücken, Germany.* PhD thesis, 1997.

[11] T. Gautier, P. Le Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lect Notes in Computer Science*, pages 257–277. Springer-Verlag, 1987.

[12] K. Hammond. Is it Time for Real-Time Functional Programming? In *Trends in Functional Programming, volume 4*. Intellect, 2004.

[13] K. Hammond and G.J. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[14] K. Hammond and G.J. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. Implementation of Functional Langs. (IFL '02), Madrid, Spain*, volume 2670 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[15] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003. Special Issue on Real-Time Systems.

[16] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proc. Intl. Static Analysis Symp. SAS 2002, Springer-Verlag LNCS 2477*.

[17] G. Michaelson, K. Hammond, and J. Sérot. The Finite State-ness of Finite State Hume. In *Trends in Functional Programming, Volume 4*. Intellect, 2004.

[18] Johan Nordlander, Magnus Carlsson, and Mark Jones. Programming with Time-Constrained Reactions. http://www.cse.ogi.edu/pacsoft/projects/Timber/publications.htm. 2006.

[19] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[20] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Static timing analysis of real-time operating system code. In *1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, Cyprus, October 2004.

[21] Daniel Sehlberg. Static WCET analysis of task-oriented code for construction vehicles. Master's thesis, Mälardalen University, October 2005.

[22] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[23] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.

[24] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. 2003 IEEE Intl. Conf. on Dependable Systems and Networks (DSN 2003)*, pages 625–632, 2003.

[25] M. Wallace and C. Runciman. Extending a Functional Programming System for Embedded Applications. *Software: Practice & Experience*, 25(1), January 1995.

[26] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ACM Intl. Conf. on Functional Programming (ICFP '01)*, Sep 2001.

[27] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Proc. IEEE Workshop on Software Tech. for Future Embedded and Ubiquitous Systs. (SEUS'05)*, pages 7–10, 2005.

[28] Yina Zhang. Evaluation of methods for dynamic time analysis for CC-systems AB. Technical report, Mälardalen University, 2005.