Chapter 1

The EmBounded Project

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

Abstract This paper introduces the EU Framework VI **EmBounded** project, a €1.3M project that will develop static analyses for resource-bounded computations (both space and time) in real-time embedded systems using the domain-specific language Hume,a language that combines functional programming for computations with finite-state automata for specifying reactive systems.

> **embound**, *v.*
> *poet. arch.*
>
> *trans.* To set bounds to; to confine, contain, hem in.
>
> Hence **embounded** *ppl. a.*
>
> 1595 SHAKESPEARE *The Life and Death of King John* IV. iii. 137
> That sweete breath which was *embounded* in this beauteous clay.

*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,Norman.Scaife}@univ-bpclermont.fr

This work is partially funded by EU Framework VI grant IST-2004-510255, EmBounded.

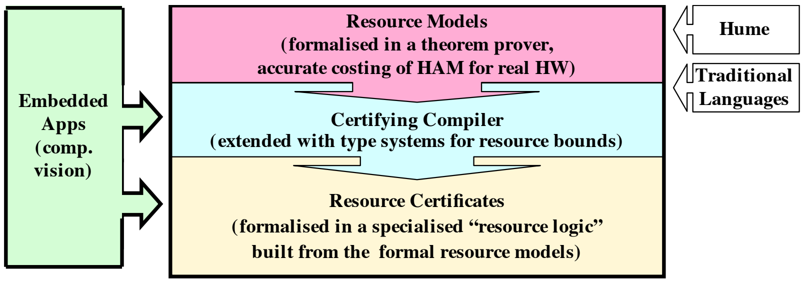

FIGURE 1.1. Schematic Diagram of the Embounded Project Objectives

1.1 PROJECT OVERVIEW

EmBounded is a 3-year Specific Targeted Research Project (STREP) funded by the European Commission under the Framework VI Future and Emerging Technology Open (FET-OPEN) programme. It commenced in June 2005 and involves 5 partners from 3 European countries, providing expertise in high-level resource prediction (Ludgwig-Maximilians-Universität, Germany and St Andrews, UK); precise costing of low-level hardware instructions (AbsInt GmbH, Germany); domain-specific languages and implementation (Heriot-Watt University, UK and St Andrews); and the design and implementation of real-time embedded systems applications, in particular in the area of computer vision algorithms for autonomous vehicles (LASMEA, France and Heriot-Watt). Further details of the project may be found at `http://www.embounded.org`.

The Embounded Vision

We envisage future real-time embedded system software engineers programming in very high-level functionally-based programming notations, whilst being supported by automatic tools for analysing time and space behaviour. These tools will provide *automatically verifiable certificates* of resource usage that will allow software to be built in a modular and compositional way, whilst providing strong guarantees of overall system cost. In this way, we will progress towards the strong standards of mathematically-based engineering that are present in other, more mature, industries, whilst simultaneously enhancing engineering productivity and reducing time-to-market for embedded systems.

Project Objectives

The primary technical *objectives* of the EmBounded project are (Figure 1.1):

a) to produce *formal models of resource consumption* in real-time embedded systems for functional programming language constructs;

b) to develop *static analyses* of upper bounds for these resources based on the formal models of resource consumption;

c) to provide independently and cheaply verifiable automatically generated *resource certificates* for the space and time behaviour of software/firmware components that can be used to construct embedded software/firmware in a compositional manner;

d) to validate our analyses against complex real-time embedded *applications* taken from computer vision systems for autonomous vehicle control;

e) to investigate how these technologies can be applied in the short-to-medium term in more *conventional language frameworks* for embedded systems.

Overall Research Methodology

Our work is undertaken in the context of Hume [12], a functionally-based domain-specific high-level programming language for real-time embedded systems. The project will combine and extend our existing work on source-level static analyses for space [18, 16] and time [26] with machine-code level analyses for time [20]. This will yield static analyses capable of deriving generic time and space resource bounds from source-level programs that can be accurately targeted to concrete machine architectures. Our source-level analyses will exploit a standard *type-and-effect systems* approach [2] and will model bounds on resource consumption for higher-order, polymorphic and recursive expressions. The analyses will be combined with the generation of resource certificates that can be checked against concrete resource prediction models using standard automatic theorem-proving techniques. We will also prove the correctness of our analyses for the same theorem-proving technology by extending the proofs we have developed as part of an earlier EU-funded project (IST-2001-33149, Mobile Resource Guarantees – MRG). Our resource model will be phrased in terms of the Hume abstract machine architecture, HAM; will extend our earlier work by considering time and other resources in addition to space usage and by handling advanced features of the expression language including timeouts and exceptions; and will be related to a concrete architecture specifically designed for real-time embedded systems used, the Renesas M32C. The work will be evaluated in the context of a number of applications taken from the embedded systems sphere, primarily real-time computer vision.

Novelty and Progress Beyond the State-of-the-art

EmBounded is novel in attempting to i) construct *formal upper bounds for space and time* on recursive, polymorphic and higher-order functions; ii) bring automatic memory management techniques to a hard real-time, real-space domain; iii) apply functional programming design to hard real-time and tightly bounded space settings; and iv) produce formally verifiable and compositional *certificates of resource usage* for real-time embedded programs. These are all open research problems, for which at best partial solutions have so far been found. Novelty also comes from the combination of static analyses at both high and low levels; from the integration of hard real-time program analyses with certificate verification; and from the applications domain. Finally, we anticipate developing new cost analyses that will allow the analysis of more forms of recursive program and/or the production of more accurate cost information than can presently be obtained.

If successful, we anticipate that the EmBounded project will enable several research advances to be made:

- it will develop compositional resource certificates for embedded systems;
- it will allow safe use of features such as recursion, polymorphism and automatic memory management in real-time systems, so allowing the in-principle use of functional programming technology under real-time conditions;
- it will synthesise resource cost models from both source and machine levels, so enabling more accurate modelling than is possible individually;
- it will extend theoretical cost modelling technology to recursive, higher-order and polymorphic functions;
- it will characterise software development using constructs with well defined formal and analytic properties in the context of realistic applications;
- it will represent the first serious attempt to apply modern functional programming language technology to hard real-time systems, including complex industrially-based applications.

As a minimum outcome, we expect to produce a set of certified models and analyses that will determine upper bounds on time and space costs for a range of useful primitive recursive function forms. We should also have determined the accuracy of these models both against some representative computer vision algorithms that have been adapted to the analyses, and against some representative, simple real-time control applications that have been written in Hume. In this way we will have made a step towards ensuring the practical application of functional programming technology in a real-time, hard-space setting.

1.2 THE HUME LANGUAGE

Our research uses Hume as a "virtual laboratory" for studying issues related to time and space cost modelling. 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 purely functional rule-based notation that maps patterns to expressions. Expressions can be classified according to a number of levels (Figure 1.2),

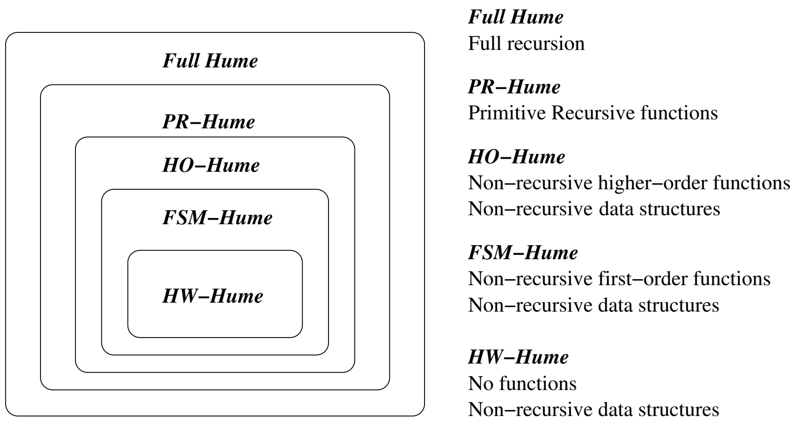

FIGURE 1.2. Expression Levels in the Hume Language

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 primitive recursive functions (PR-Hume) will usually be less accurate than those for non-recursive programs, while cost inference for Full Hume programs is undecidable in general (and we therefore restrict our attention in the **EmBounded** project to PR-Hume and below). A previous paper has considered the Hume language design in the general context of programming languages for real-time systems [11].

We have previously developed prototype stack and heap cost models for FSM-Hume [13], based on a simple formal operational semantics derived from the Hume Abstract Machine, and have also developed a prototype stack and heap analysis for a subset of PR-Hume. During the course of the **EmBounded** project, these analyses will be extended to cover time issues and the full range of Hume language constructs. We must also explore issues of quality, compositionality and the cost of the analysis in order to reach a good balance between theoretical coverage and practicality.

1.3 PROJECT WORK PLAN

Formal Models of Resource Consumption

Our first technical objective is to produce formal models of the exact time and space consumption of Hume programs. Space properties of interest include both dynamic stack and heap allocations and static global data allocations. Time must be measured in real, absolute time units at the granularity of the hardware clock for each target architecture. In order to ensure accurate modelling of time consumption, the models will reflect domain-specific compiler optimisations and important architectural characteristics such as cache behaviour.

Conceptually the formal cost models will be based on a formal operational se-

	Semantics		Cost Model
Hume	code	$\xrightarrow{\text{evaluation}}$	cost
	$\downarrow$ translation		$\Vert$ correspondence
HAM	code	$\xrightarrow{\text{evaluation } \mathbf{D4}}$	cost

FIGURE 1.3. Cost Modelling Methodology

mantics, extended in order to make explicit the intensional properties of program execution, such as time and space consumption. So as to achieve the desired level of accuracy, low-level architectural issues will be integrated in the description of state in the operational semantics. Accurate modelling of the compilation process is also required, in order to retain a close relationship between information that can be obtained from the concrete architecture and the results from the static analyses. This will link the resource consumption models with the static analyses. Finally, a *correspondance proof* determines the soundness of the Hume cost model against the HAM seamtics. Our approach is shown in Figure 1.3.

The resulting formal models will form the basis for defining and automatically verifying resource certificates. They will be novel in their accurate and rigorous modelling of time and space. In particular, they will model low-level processor characteristics such as cache behaviour and instruction-level-parallelism (ILP) using the techniques developed by Absint.

At the time of writing, we have largely completed this objective, having constructed formal operational semantics for both Hume and the HAM that have been extended to expose explicit stack, heap and time information. We are now proceeding to incorporate time information derived from abstract interpretation of binary code fragments using the AbsInt tool.

A Cost Model for Hume Expressions We illustrate our approach by showing how a cost model can be constructed to expose time, heap and stack costs for Hume expressions. The statement

$$\mathcal{V},\eta \;\vdash^{t}_{t'}\vdash^{p}_{p'}\vdash^{m}_{m'}\; e \rightsquigarrow \ell,\eta'$$

may be read as follows: expression e evaluates under the environment, heap configuration $\mathcal{V},\eta$ in a finite number of steps to a result value stored at location ℓ in 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 the evaluation is finished. We illustrate the approach by showing a few sample rules. Integers are constructed as boxed values, and a pointer to the new value saved on the stack. The time cost is given by `Tmkint`.

$$\frac{n \in \mathbb{Z} \qquad \textsc{New}(\eta) = \ell \qquad w = (\mathsf{int},\mathsf{n})}{\mathcal{V},\eta \;\vdash^{t'+\texttt{Tmkint}}_{t'}\vdash^{p'+1}_{p'}\vdash^{m'+\textsc{Size}(w)}_{m'}\; n \rightsquigarrow \ell,\eta[\ell \mapsto w]} \qquad (\textsc{Const Int})$$

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`. There is no heap cost.

$$\frac{\mathcal{V}(x) = \ell}{\mathcal{V}, \eta \vdash^{t' + \texttt{Tpushvar}}_{t'} {}^{p'+1}_{p'} {}^{m}_{m} \; x \rightsquigarrow \ell, \eta} \qquad \text{(VARIABLE)}$$

There are three cases for conditionals: two symmetric cases where the condition is true or false, respectively; and a third case to deal with exceptions. We show here only the false case. 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.

$$\frac{\begin{array}{c} \mathcal{V}, \eta \vdash^{t_1}_{t'_1} {}^{p}_{p'} {}^{m}_{m'} \; e_1 \rightsquigarrow \ell, \eta' \qquad 0 \notin \mathrm{dom}(\eta') \\ \eta'(\ell) = (\mathsf{bool}, \mathsf{ff}) \\ \mathcal{V}, \eta' \vdash^{t'_1 - \texttt{Tiffalse}}_{t'_3} {}^{p'+1}_{p''} {}^{m'}_{m''} \; e_3 \rightsquigarrow \ell'', \eta'' \end{array}}{\begin{array}{c} \mathcal{V}, \eta \vdash^{t_1}_{t'_3 - \texttt{Tgoto}} {}^{p}_{p''} {}^{m}_{m''} \\ \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \ell'', \eta'' \end{array}} \quad \text{(CONDITIONAL FALSE)}$$

The remaining rules are constructed similarly. The main technical difficulties are dealing correctly with higher-order functions and exceptions and capturing the costs of pattern-matching and scheduling. For reasons of brevity we will not consider these issues here.

Static Analyses

The cost models we have outlined above can now be used as the basis for static analyses. Our second objective is the development of static analyses corresponding to these formal models. The analyses will predict upper bounds on both worst-case execution time (WCET) and maximum space (both static and dynamic memory) usage for (a subset of) Hume programs as previously identified. They will work on the Hume source level to produce conservative estimates of worst-case behaviour based on the target architecture (whether abstract machine or concrete hardware implementation).

Our analyses will build on our theoretical work on costing higher-order and recursive definitions [26, 33, 16, 18], applied work on first-order programs [14, 24], and the static analyses of low-level code developed by AbsInt [10, 22, 15]. Combining these analyses will lead to a hybrid analysis that should yield considerably more accurate results than can be obtained using either kind of analysis alone, and that should be capable of analysing very high-level language constructs.

Our high-level analyses will be constructed using a type-and-effect system approach. This approach allows our analyses to be scaled to consider higher-order functions and complex data structures in a common framework. In order to support automatic memory management, we will include mechanisms to support limited forms of compile-time garbage collection based on Tofte-style *memory regions* [32] and/or *usage annotations* [4]. This will enable effective and accurate prediction of run-time memory usage without compromising the required real-time program properties. We will also investigate the application of our analyses to implicit memory allocation.

Our low-level analyses use abstract interpretation of machine-code instructions to provide time and space analyses for a complete program. They exploit detailed models of the hardware architecture including cache logic and the instruction scheduler, pipeline, and branch prediction.

At the time of writing, construction of the static analyses is the main focus of work at St Andrews and LMU.

Formal, Verifiable Resource Certificates

Our third objective is the automatic generation of certificates of bounded resource consumption. Such certificates can be attached to code fragments for the target machines, and composed to provide overall guarantees of bounded resource consumption. In an embedded system context, once a program is linked and the resource bounds verified, there is no further need for a certificate and it may be discarded. An additional benefit from certificate generation is the enhancement of confidence in the behavioural correctness of the program.

Formally defining the structure of certificates will amount to first defining an assertion language that defines which statements can be made for HAM programs. The structure of certificates will be a suitably simplified representation of a formal proof of statements in the assertion language. The proof will be relative to the resource-aware program logic for the HAM. This program logic has to accurately model resources, but still be simple enough to enable automated reasoning on these certificates. We will draw on our program logic, the Grail Logic [3], for a JVM-like low-level language, in deciding on the style of the logic and the embedding of the assertion language into the logic. In contrast to the Grail Logic, the HAM Logic will have to model costs incurred at assembler level, for the particular hardware. Bridging this gap in abstraction levels on the low level will be a major focus of this work, and we will investigate methods of reflecting this level of detail without making the program logic prohibitively expensive.

At the time of writing, we have started work on encoding the formal cost models we have now developed in a form that can be used by the Isabelle theorem prover. This will form the basis for our subsequent work on certification.

Embedded Applications

Our fourth objective is the development of testbed applications in Hume that can be costed using our new analyses. We need to develop three kinds of applications: simple exemplars, isolating single issues; more complex *cost benchmarks*; and realistic applications. The simple exemplars will provide underpinning components for the subsequent applications. They will also enable us to explore principled approaches to developing embedded software that exploit program constructs with well characterised properties and analyses. The more complex cost benchmarks will build on the simple exemplars and enable exploration of integration of different analyses. The realistic applications will serve as proofs of concept, demonstrating that our approach can deal with complex real-time applications with hard space requirements.

EmBounded will build on Heriot-Watt and LASMEA expertise in formally motivated development of vision and control software using functional languages, through a series of closely linked stages of application software development. Initially, we will revisit classic vision algorithms for low-, intermediate- and high-level vision, focusing on the Hume expression layer. We will investigate the degree to which such algorithms can be formulated using strongly finite-state, higher-order or primitive recursive constructs. We will empirically evaluate these algorithms for direct comparison with predictions from the analyses, embodying the cost models developed above. We will then look at composing classic vision algorithms to form a complete mono-source vision system and a high-level stereoscopic vision system. Again we will empirically measure these systems to enable evaluation of compositional cost-model based analyses. Next, we will explore real-time tracking, again using composed components developed at earlier stages. This is where we will first introduce concurrency at the Hume coordination layer, enabling initial evaluation of cost modelling of full Hume programs. Finally, we intend to develop a real-time control system for the *CyCab* autonomous vehicle, incorporating real-time tracking and multiple sensor monitoring. Whilst the focus will be on evaluation of cost models and analyses applied to a substantive, complex system, we would also seek to incorporate the control system in a CyCab vehicle for on-road trials.

At the time of writing, we have produced some simple exemplars of computer vision algorithms that exploit recursion and dynamic data structures, and are considering how these can best be analysed. We have also successfully produced and analysed space usage for a simple real-time computer game, based on the commercial Simple Simon system. This application runs on a simple Renesas M32C development board in less than 2KB of dynamic memory for Hume stack and heap, plus 7KB of flash memory for the Hume program code and runtime system, including interfaces to the physical buttons and LED outputs supported by the board. We are now working on obtaining analytical time costs for this architecture.

Application to Traditional Languages

Our final objective is the determination of how our formal models and analyses could be applied to present-generation languages and application frameworks that are in widespread use for the development of embedded systems. A number of common language features, such as assignment, unrestricted exception handling or dynamic method dispatch, are known to both complicate static analyses and to reduce the quality of analytical results. This has motivated our use of Hume as a "virtual laboratory" in the first instance: by eliminating such features it is possible to make more rapid progress on the key issues related to the analysis. In order to extend our work, we will therefore first identify generic language features that are amenable to analysis using our techniques. We will subsequently explore how the analyses can be extended to the other language constructs of interest. Despite the lack of good formal semantics for many traditional languages, we anticipate being able to demonstrate that the use of a suitably restricted, but still powerful, subset of the language will permit the construction of good-quality static analyses for determining bounds on time- and space-resource usage.

1.4 THE STATE OF THE ART IN PROGRAM ANALYSES FOR REAL-TIME EMBEDDED SYSTEMS

Static analysis of *worst-case execution time* (WCET) in real-time systems is an essential part of the analyses of over-all response time and of quality of service [27]. 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 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 considerable and unnecessary hardware costs for the large production runs that are often required.

Memory management is another important issue in real-time and/or embedded systems with their focus on restricted memory settings. Some languages provide automatic dynamic memory management without strong guarantees on time performance (e.g. Java [25]), whilst others rely on more predictable but error-prone explicit memory management (e.g. C, C^{++}, RTSj or Ada). One recent approach [8] is to exploit memory *regions* for some or all allocation and to combine annotations with automatic inference. Such approaches do not, however, provide real-time guarantees, and typically require manual intervention in the allocation process. Moreover, static region analysis can be overly pessimistic [8] for long-lived allocations. Regardless of the memory management method, there is a strong need for static guarantees of memory utilisation bounds.

Three competing technologies can be used for worst-case execution time analysis: experimental or testing-based approaches, probabilistic measures 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, deliver more accurate costs, and can be engineered to deliver high levels of trust in their results. Finally, existing static analyses based on low-level machine models can provide guaranteed upper bounds on execution time, but are time-consuming to construct, and may be unduly pessimistic, especially for recent architectures with complex cache behaviour.

Experimental Approaches to WCET Analysis

Cache memories and pipelines usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in the execution time. For (hard) real-time systems such as a flight-control computer, this is undesirable and possibly even hazardous. The widely used classical methods of predicting execution times are not generally applicable. Software monitoring changes the code, which in turn impacts the cache behaviour. Hardware simulation, emulation, or direct measurement with logic analysers can only determine the execution times for some inputs and cannot be used to infer the execution times for all possible inputs in general.

Some producers of time-critical software have thus developed their own method, which is based on strict design and coding rules, the most deterministic usage of the internal speed-up mechanisms of the microprocessor, and measurements of code fragments whose limited size makes it possible to obtain a WCET for all their possible inputs. This method allows the computation of a safe WCET for the whole program by combining the WCETs of the individual fragments. An appropriate combination formula exists thanks to the design and coding rules. However, this method poses the following drawbacks: it limits the effective power of the CPU, requires manual effort for the measurements and related intellectual analysis, and cannot be performed too early during software development, since the target hardware has to be available for measurement purposes. Moreover, in order to ensure that an upper bound of the WCET is really being observed, complex extensive verification and justification of the measurement process is required. It is also possible that this measurement-based method might not scale up to future projects. Therefore major industries depending on time-critical software are actively studying and evaluating new approaches to WCET determination based on static program analysis, as they are pursued by AbsInt.

Probabilistic WCET Analysis

Probabilistic WCET analysis provides distribution functions, rather than absolute upper bounds, for the execution time. This approach is valid even in the hard-real-time environment, if it can provide a guarantee that the probability of deadline over-run by any mission-critical task is within the accepted safety levels (e.g., less than 10^{-9} per flight hour for avionics applications).

Existing implementations of probabilistic WCET analysis tend to be rather low-level: for example, in [6], the program units used are basic blocks (instruction sequences with one entry and one exit) of either Java byte-code, or machine code compiled from C. The difficulty with this approach is that the information about high-level program structure, which is essential for combining the distribution functions of individual basic blocks into “larger” functions, is then lost, and needs to be re-constructed from specifically-designed program annotations. The analysis is performed in the "bottom-up" direction.

Static Analyses for Execution Cost

There has been a significant amount of work on analyzing general execution costs, typically focusing on time usage, since the pioneering work on *automatic complexity analysis* for first-order Lisp programs undertaken by Wegbreit [34]. There has been progress on automatically costing higher-order functions, and recent work has begun to tackle the many problems surrounding costing recursion (e.g. Amadio et al. [1, 7] consider synthesis of polynomial time bounds for first-order recursive programs). The static analyses for real-time systems of which we are aware (e.g. Verilog's SCADE or stack analysers such as that of Regehr et al. [28] or AbsInt's `StackAnalyzer` tool) are, however, highly conservative in limiting their attention to first-order non-recursive systems with statically allocated data structures. Typically, languages used for real-time systems do not support features such as recursion or higher-order functions because of costing difficulties, and cost analyses that might deal with such features are not applied to real-time systems because the mostly widely-employed languages do not possess the requisite features.

Le Métayer [23] uses *program transformation* via a set of rewrite rules to derive complexity functions for FP programs. A database of known recurrences is used to produce closed forms for some recursive functions. However, the language is restricted to a particular set of higher-order combinators for expressing functions and the analysis is not *modular* as the transformation can only be applied to complete programs.

Rosendahl [30] also uses *program transformation* to obtain a step counting version of first-order Lisp programs; this is followed by abstract interpretation to obtain a program giving an upper bound on the cost. Again this abstract interpretation requires a complete program, limiting both its scalability and its applicability to systems with e.g. compiled libraries. Finally, Benzinger [5] obtains worst-case complexity analysis for NuPrl-synthesized programs by "*symbolic execution*" followed by recurrence solving. The system supports first-order functions and lazy lists but higher-order functions must be annotated with complexity information. Moreover, only a restricted and awkward primitive recursion syntax is supported.

1.5 EXISTING WORK BY THE CONSORTIUM

High-Level Static Analyses for Real-Time, Hard Space Systems

St Andrews and LMU have developed complementary formal models for determining upper bounds on space usage [18, 14] and time usage [26]. LMU has focused on determining formally verified space models for first-order languages [16], whilst St Andrews has focused on models that allow inference of time usage for higher-order, polymorphic and (primitive) recursive programs [33]. The combination of this work will lead to a powerful formal model capable of allowing inference of both time and space bounds for a language supporting modern language technologies, including higher-order definitions, polymorphism, recursion and automatic memory management. Our work is influenced by that of Reistad and Gifford [29] for the cost analysis of higher-order Lisp expressions, by the "time system" of Dornic et al. [9], and by Hughes, Pareto and Sabry's *sized types* [19], for checking (but *not* inferring) *termination* for recursion and *productivity* for reactive streams in a higher-order, recursive, and non-strict functional language. Both St Andrews and LMU have produced automatic analyses [18, 14, 26] based on these resource prediction models using standard *type-and-effect* system technology to automatically *infer* costs from source programs.

Low-Level Static Analyses

Motivated by the problems of measurement-based methods for WCET estimation, AbsInt has investigated a new approach based on static program analysis [22, 15]. This has been evaluated by Airbus France [31] within the Framework V RTD project "DAEDALUS" (IST-1999-20527). 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 estimates valid for all executions of the application.

Resource Certification

In the Framework V MRG project we aimed to develop certificates for bounded resource consumption for higher-level JVM programs, and to use these certificates in a proof-carrying-code infrastructure for mobile systems. In this infrastructure a certifying compiler automatically generates certificates for (linear) bounds on heap space consumption for a strict, first-order language with object-oriented extensions. These certificates can be independently checked when composing software modules. Novel features in the reasoning infrastructure are the use of a hiearchy of programming logics, using high-level type systems to capture information on heap consumption, and the use of tactic-based certificates in the software infrastructure. The latter drastically reduces the size of the certificates that are generated. In the context of embedded systems, the cost model (and thus the certificates built on them) must reflect lower-level architecture features. The bounds for the resource consumption that are expressed in these certificates will be provided by our static analyses, and may also incorporate information gained by measurement on the concrete hardware.

Linear Types for Memory Allocation

LFPL [16, 18] uses linear types to determine resource usage patterns. A special resource type called "*diamond*" is used to count constructors. First-order LFPL definitions can be computed in linearly bounded space, even in the presence of general recursion. More recently, Hofmann and Jost have introduced [18] automatic inference of these resource types, and thus of heap-space consumption, using linear programming. At the same time, the linear typing discipline is relaxed to allow analysis of programs typable in a usage type system such as [21, 4]. Extensions of LFPL to higher-order functions have been studied in [17] where it was shown that such programs can be evaluated using dynamic programming in time $O(2^{p(n)})$ where n is the size of the input and p is a fixed polynomial. It has been shown that this is equivalent to polynomial space plus an unbounded stack.

1.6 CONCLUSIONS

In the **EmBounded** project, we are trying to push back the boundaries of applicability for functional programming by considering hard real-time, hard space systems. We believe that functional programming notations have a great deal to offer to modern software engineering practices, through the twin advantages of abstraction and compositionality. By tackling the long-standing behavioural bugbears of time and space usage through careful language design in conjunction with state-of-the-art static analysis techniques, we hope to show that functional languages can also be highly practical and deliver real benefits in terms of automated support for the development of complex programs in the real-time embedded systems domain.

Having constructed cost models for Hume and the HAM, our immediate challenge in the project is to construct sound resource analyses to determine good upper bounds for recursive higher-order functions. In particular, we need to extend our work on space to also deal with time information and we must also study the integration between time information at the source and binary levels. We must also develop convincing real-time applications that exploit recursion and higher-order functions in an essential way. In order to do this, we are studying applications from the computer vision domain that may be used for real-time object tracking, or direction of autonomous vehicles. Finally, we must demonstrate that functional languages are suitable for use in time- and space-constrained settings. We have constructed one realistic demonstrator based on the Simple Simon game, and will now consider additional.

REFERENCES

[1] R. Amadio. Max-plus Quasi-Interpretations. In *Proc Typed Lambda-Calculi and Applications (TLAC '03), Springer-Verlag LNCS 2701*, 2003.

[2] T. Amtoft, F. Nielson, and H. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[3] D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A Resource-aware Program Logic for Grail. In *Proc. ESOP'04 — European Symposium on Programming*, 2004.

[4] D. Aspinall and M. Hofmann. Another Type System for In-Place Update. In D. Le Metayer, editor, *Programming Languages and Systems (Proc. ESOP'02)*, volume Springer LNCS 2305, 2002.

[5] R. Benzinger. Automated Complexity Analysis of Nuprl Extracted Programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

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

[7] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On Termination Methods with Space Bound Certifications. In *Proc Perspectives of System Informatics, Springer-Verlag LNCS 2244*, 2001.

[8] M. Deters and R.K. Cytron. Automated Discovery of Scoped Memory Regions for Real-Time Java. In *Proc. ACM Intl. Symp. on Memory Management, Berlin, Germany*, pages 132–141, June 2002.

[9] V. Dornic, P. Jouvelot, and D.K. Gifford. Polymorphic Time Systems for Estimating Program Complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, March 1992.

[10] 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*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.

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

[12] K. Hammond and G.J. Michaelson. The Hume Language Definition and Report, Version 0.3. Technical report, Heriot-Watt University and University of St Andrews.

[13] K. Hammond and G.J. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, number 2670 in 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*, number 2670 in 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. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[17] M. Hofmann. The strength of non size-increasing computation. In *Proc. ACM Symp. on Principles of Prog. Langs. (POPL), Portland, Oregon*. ACM Press, 2002.

[18] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symposium on Principles of Programming Languages*, New Orleans, LA, USA, January 2003. ACM Press.

[19] R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems using Sized Types. In *Proc 1996 ACM Symposium on Principles of Programming Languages – POPL '96*, St Petersburg, FL, January 1996.

[20] D. Kästner. TDL: a Hardware Description Language for Retargetable Postpass Optimisations and Analyses. In *Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, – GPCE 2003, Erfurt, Germany*, pages 18–36. Springer-Verlag LNCS 2830, September 2003.

[21] N. Kobayashi and A. Igarashi. Resource Usage Analysis. In *POPL '02 — Principles of Programming Languages*, Portland, Oregon, January 2002.

[22] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proc. 9th International Static Analysis Symposium SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.

[23] D. Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[24] K MacKenzie and N. Wolverson. Camelot and Grail: Compiling a Resource-Aware Functional Language for the Java Virtual Machine. In *TFP'03 — Symposium on Trends in Functional Programming*, Edinburgh, Scotland, Sep 11–12, 2003, 2003.

[25] K. Nilsen. Issues in the Design and Implementation of Real-Time Java. *Java Developers' Journal*, 1(1):44, 1996.

[26] A.J. Rebón Portillo, K. Hammond, H.-W. Loidl, and P. Vasconcelos. A Sized Time System for a Parallel Functional Language (Revised). In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, number 2670 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

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

[28] J. Regehr, A.J. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. In *Proc 3rd Intl. Conf. on Embedded Software (EMSOFT 2003*.

[29] B. Reistad and D.K. Gifford. Static Dependent Costs for Estimating Execution Time. In *LFP'94 — Conference on Lisp and Functional Programming*, pages 65–78, Orlando, Florida, June 27–29, June 1994. ACM Press.

[30] M. Rosendahl. Automatic Complexity Analysis. In *Proc. FPCA'89 — Intl. Conf. on Functional Prog. Langs. and Comp. Arch.*, pages 144–156, 1989.

[31] 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 Intl. Conf. on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, 2003.

[32] Mads Tofte and Jean-Pierre Talpin. Region-based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[33] P.B. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. Implementation of Functional Languages (IFL 2003)*, 2004.

[34] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.