ProgLab.NET

Research Proposal (Round 2)
Microsoft Innovation Cluster for Embedded Software

Proposal Title:
ProgLab.NET: Workbench for Ensuring Embedded Software Quality and Reliability

Abstract:

We propose to build an extensible workbench for automated program analyses of embedded software. The workbench will combine a number of static analyses and coordinate them using an evidence manager. Different program analyses will feed into each other. Their results will be presented to the software developer in a simple, uniform way, via a plugin for an IDE (Visual Studio).

Project Duration: 4 years

Expected Amount of Funding: 290'000 CHF/year

The funding will support:

- Three Ph.D. students to work on particular program analyses in the ProgLab framework (for each student 55'000 CHF/year)
- One Ph.D. level programmer for developing and maintaining the ProgLab.NET infrastructure (110'000 CHF/year)
- Travel to enable collaborations with Microsoft researchers: 15'000 CHF/year

Principal Investigators:

Thomas A. Henzinger

Viktor Kuncak

Martin Odersky

16 May 2008

1 Biographies of Principal Investigators

The principal investigators are leading three research groups, which collaborate as part of the EPFL Thrust in Reliable Software Research (TRESOR).

TRESOR Page: `http://tresor.epfl.ch`

Thomas Henzinger is Professor at EPFL and Adjunct Professor at the University of California, Berkeley. He holds a PhD in Computer Science from Stanford University (1991) and was an Assistant Professor at Cornell University (1992-95), then Assistant Professor (1996-97), Associate Professor (1997-98), and Professor (1998-2004) at UC Berkeley. In Berkeley, he was a Director of the Center for Hybrid and Embedded Software Systems. He was also Director at the Max-Planck Institute for Computer Science in Saarbruecken (1999-2000).

Henzinger is a member of Academia Europaea, a member of the German Academy of Sciences (Leopoldina), a Fellow of the ACM, a Fellow of the IEEE, and an ISI Highly Cited Researcher. He has published more than 200 articles in refereed conferences and journals. He founded the main international conferences on hybrid systems (HSCC) and embedded software (EMSOFT) and chaired the program committees of several international conferences such as Computer-Aided Verification and Computer Science Logic.

Henzinger's research focuses on models, algorithms, and tools for the design and verification of software and embedded systems. His hybrid automaton model has become the standard formalism for analyzing mixed discrete-continuous systems and his HyTech tool was the first model checker for such hybrid systems. Other software from Henzinger's team include Mocha, a toolkit for the verification of reactive modules; Blast, a model checker for C programs; and Giotto, a coordination language for distributed real-time tasks.

Homepage: `http://mtc.epfl.ch/~tah`

Viktor Kuncak is Assistant Professor at EPFL and the head of the Automated Reasoning and Analysis group. His work combines ideas from decision procedures, theorem proving, constraint solving, run-time checking, static analysis, and programming languages. His recent contributions include verification of precise properties of imperative data structures, and decision procedures for reasoning about collections of objects. Before joining EPFL in 2007, Viktor Kuncak was a doctoral student at the Massachusetts Institute of Technology. In his PhD thesis he developed and implemented new algorithms for proving program properties that had been beyond the reach of previous automatic verification techniques. He has also worked on the foundations of expressive subtyping constraints and applications of interactive theorem proving. He was a summer intern at Software Productivity Tools group in Microsoft Research, Redmond, USA, in 2002 and a visitor at the Max-Planck-Institute for Computer Science, Saarbrücken, Germany, in 2003 and 2005.

Homepage: `http://lara.epfl.ch/~kuncak`

Martin Odersky is Professor at EPFL where he heads the programming research group. His research interests cover fundamental as well as applied aspects of programming languages. They include semantics, type systems, programming language design, and compiler construction. The main focus if his work lies in the integration of object-oriented and functional programming. His research thesis is that the two paradigms are just two sides of the same coin and should be unified as much as possible. To prove this he has experimented with a number of language designs, from Pizza to GJ to Functional Nets. He has also influenced the development of Java as a co-designer of Java generics and as the original author of the current javac reference compiler. His current work concentrates on the Scala programming language, which unifies FP and OOP while staying completely interoperable with Java and .NET.

Martin Odersky got his doctorate from ETH Zürich in 1989. He held research positions at the IBM T.J. Watson Research Center from 1989 and at Yale University from 1991. He was a professor at the Univerisity of Karlsruhe from 1993 and at the University of South Australia from 1997. He joined EPFL as full professor in 1999. He is associate editor of the Journal of Functional Programming and member of IFIP WG 2.8. He was conference chair for ICFP 2000, and program chair for ECOOP 2004 as well as ETAPS/CC 2007. He is a fellow of the ACM.

Homepage: `http://lamp.epfl.ch/~odersky`

2 Project Description

2.1 Background: State of the Art

The use of embedded control systems, from medical implants to automobiles (drive-by-wire) and aircraft (fly-by-wire), is increasing rapidly. As they are deployed in safety-critical situations, the verification of such systems has become a central concern [10, 36, 24].

In the 1990s, formal verification techniques such as model checking made a successful transition from theory to practice in hardware design. This transition was enabled by new technologies for exploring very large state spaces, such as symbolic and compositional methods for model checking [8] and efficient methods for SAT solving [35]. Today it would be difficult to find a hardware design corporation that does not use formal verification tools in one way or another [14]. The 2008 Turing award was given for this achievement.

In the current decade, formal verification has begun to make some inroads in software design. The routine use of verification tools that detect common simple errors (e.g., FindBugs [20]) is enforced by a number of companies (such as Google); model-checking techniques are deployed in the verification of protocol-intensive code such as device drivers (e.g., at Microsoft [1]); abstract-interpretation techniques have been used to find numerical and timing errors in safety-critical software [7, 32]. The enablers have been technologies for coping with infinite state spaces, such as counterexample-guided refinement of finite abstractions [3], improved theorem-proving technology [11], shape analysis for dynamic heap structures [6, 5], sophisticated abstract interpretation engines [7], and approaches based on verification condition generation [9]. Recently, the Spec# system [4] has made remarkable steps towards adoption of verification tools, by integrating verification condition generation, theorem proving, abstract interpretation, run-time checking, and interactive deployment in Visual Studio. However, to our knowledge, none of these systems enables the analysis of running time, memory usage, and failure probability, which are essential for embedded software development. More generally, significant work on automation and the scope of applicability remains before we are able to put a formal-verification toolbox on every programmer's desk.

2.2 Objective: Advances Expected from the Proposed Research

We believe that the time is ripe to offer formal-verification technology as an integral part of a workbench that can be shared by both programmers developing embedded software and by verification tool developers, for their mutual benefits. This project will build such a workbench, dubbed ProgLab.NET. For the project to succeed, existing verification technology will need to be harnessed and refocused so that

1. program analyses provide unobtrusive and easily understandable information not only about complete programs but especially about partial programs, i.e., program fragments whose missing pieces are constrained by environment assumptions, component interfaces, procedure summaries, function stubs, and requirement specifications;

2. guided by an evidence manager, program analyses perform efficient and incremental checks as the (partial) programs and their requirements change during software development;

3. different program analyses feed into each other in order to leverage their strengths and present their results to the software developer in a simple, uniform way.

The ProgLab workbench will consist of a set of program analyses (including model checkers, type checkers, and theorem provers) that support demmand-driven and modular verification of software fragments expressed in a common representation format. The verified properties will include both expressive versions of the traditional safety properties and non-functional properties such as execution time, power consumption, and memory usage.

ProgLab will integrate both fully formal and semiformal analyses (such as testing) but will concentrate on analyses that are largely automatic, in particular, type-based analyses and execution-based analyses (such as model checking and abstract interpretation). In contrast to heavy-weight and time-consuming formal verification approaches, the primary purpose of ProgLab is not to provide full formal correctness proofs of already completed programs. Instead, the goal is to aid the programmer throughout the software development process with rapid feedback obtained from tools that work with incomplete programs as

they are being developed. We believe that in this way we can take a significant step toward the wide adoption of formal verification techniques in software design.

ProgLab design will unify a range of program analyses through common APIs and intermediate representation formats targetted towards demand-driven analysis of partial programs. We will evaluate this design through an integration within the Visual Studio IDE.

2.3 Opportunity: Scala as a Language Facilitating Verification

Current research in software verification can be classified into efforts that focus on widely used languages such as C, and efforts that focus on purely academic languages such as ML. The main argument for the former line of work is that in order to have practical impact, one must handle real-world programs in all their complexity and ugliness, even at the price that successes are far more difficult to come by, more modest, and often idiosyncratic. However, the focus on a language like C also has a significant drawback, namely, that insights gained from the verification research have little chance of flowing back into the design of the language and its usage patterns. In other words, there is no feedback loop from the tool designers to the language designers.

This is why we have started to look for a language that has a significant user community and yet is under our complete control. We need both: a broad, progressive user community in order to foster early adoption and collect real data about the use of our verification tools, and complete control over the language design in order to facilitate the verification tasks. For this purpose we have brought together in a single team—in what we believe to be a unique combination in the world—the groups of Martin Odersky, who has designed and implemented the Scala language, Thomas Henzinger, whose expertise lies in formal verification, and Viktor Kuncak, whose expertise lies in automated theorem proving for static analysis. We will extend the state-of-the-art in software verification and at the same time evaluate the new methods on a concrete workbench based on Scala [29, 28], whose support for modularity and functional programming makes it an ideal starting point for verification. The benefits will be mutual: insights into how analyses can be facilitated will flow directly into the design of the language, and insights from usage patterns of the language and verification tools will flow back to guide the verification research.

Scala is remarkable in delivering a modern language design while at the same time seemlessly interoperating with widely used platforms. In particular, Scala already compiles to .NET platform and supports the use of certain .NET libraries. ProbLab workbench aims to leverage this opportunity by extending the developed analyses not only to the code develoepd in Scala but also to .NET libraries and other software developed on .NET platform. Key techniques that will make this possible are proof carrying bytecodes that preserve high-level properties of heterogeneous source languages, and rich interfaces that express these properties at code fragment boundaries.

2.4 Methodology

We will organize the work in four thrusts. The main task of the first thrust is to build the infrastructure: a workbench we plan to call ProgLab in analogy to MatLab. While in Matlab the objects of study are mathematical formulas, in ProgLab the objects of study are programs. ProgLab will be designed to permit the addition of new program analyses that communicate through a common set of data structures representing the following four categories of software artifacts: 1) *Programs* and partial programs, including component interfaces, procedure summaries, environment assumptions, and code; 2) *Requirements*, including assertions, monitors, timing and reliability constraints; 3) *Platforms* for software execution, including semantic definitions, schedulers, memory and resource models; 4) *Evidence*, including abstractions, proofs, and tests. The workbench will manage and integrate evidence that is established by the analyses, and present it to the user in an understandable way. We will put a premium on compositional and incremental analyses. Compositionality means that an analysis does not need to be redone when a (partial) program is reused in a different context; incrementality means that an analysis does not need to be redone in its entirety when the (partial) program changes. Both compositionality and incrementality cannot be achieved in absolute terms, but require a sophisticated evidence manager that tracks assumptions and generates new proof obligations. Such evidence management will be a key task of ProgLab.

The remaining three thrusts will focus on particular program analyses in the ProgLab framework, with special focus on embedded software.

2.5 Proposed Design of ProgLab Infrastructure

We are aware of challenges in engineering an incremental analysis integrated into an IDE through the experience of developing two generations of Eclipse plugins for the Scala programming language, which included a type-checker for Scala's advanced type system. Based on this experience, we propose a layered architecture of ProgLab that will make integration of analysis into an IDE easier.

Part of the ProgLab design will be formats for the four programming artifacts (programs, requirements, platforms, evidence), including their abstract syntax, human-readable and binary concrete syntax, precise semantics, and a set of operations that ProgLab components are expected to perform efficiently. To facilitate compositional and incremental analyses, we envision a representation format where local changes in the source code correspond to changes to a small number of files in the representation. The semantic basis of the representation format will be control-flow graphs whose edges are specified by transition formulas expressed in a standardized and expressive logic. We will also explore connections and translations between our format and future versions of the intermediate language for Spec#.

Given a common serializable representation format, the basic communication between the analyses and the IDE will be through the file system. The granularity for incremental and modular analysis will differ for different program analyses. For many analyses we expect that updating computed results on each file save and tracking information at the level of procedures and basic blocks will be sufficient.

To meet the challenges of deploying the analyses within Visual Studio, we have dedicated part of our funds to a full-time PhD level programmer with a familiarity of Visual Studio. The sole responsibility of the programmer will be the integration of developed analyses into Visual Studio. The Visual Studio Languages Team has committed to act in an advisory role for this effort. That team overseas design, compiler and IDE support for Microsoft's major managed languages, including C#, VB, F#, IronPython and IronRuby. Sean McDirmid at Microsoft Research Bejing will help us with general .NET integration issues. We also expect collaboration with Microsoft Research groups that have experience in integrating advanced verification tools into Visual Studio.

Proposed Analyses. The core expected contribution of the proposal are new program analyses operating on partial programs under development. The analyses will use a range of techniques from advanced type systems, model checking, and theorem proving, but will all communicate through the common representation format (described in the previous section) to provide evidence for software quality and reliability that will be made available to the developer through an IDE. We group the activities of developing the proposed analyses into three thrusts: rich types for effect analyses, analyses capturing physical program properties, and developing theorem provers that enable the analyses.

2.6 Effect Analyses Through Rich Types

There are many properties that go beyond standard types. For instance, given a function, one might be interested in whether it terminates, what exceptions it can throw, what side effects it has, areas of memory it will read, write, or allocate, or what other resources it consumes. Such properties can be regarded as computational effects. Type systems for formalizing such effects have been known for 20 years [26], and there are some strong parallels with monads [33]. In essence, an effect system can be characterized as a monad which admits a least-upper bound operator. In the absence of effect masking operators, the total effect of a computation is the least upper bound of all computations it invokes; so effect analysis has also strong parallels to closure analysis.

Despite these results, effect analysis has not yet found its way into programming languages that are in actual use. It appears the main challenges to overcome before it can be adopted are how to keep the notation lightweight and how to make the framework customizable so that precisely the topics of interest can be handled.

In this project, we will develop a general modular framework for the notation and static analysis of computational effects. The novelty of the framework will be that particular effect domains should be "pluggable" (i.e. user-definable and freely combinable) instead of being fixed in the programming language. User-defined effects will be written down using Scala's type annotation mechanism; they will be analyzed in the ProgLab workbench. The framework will be instantiated for several concrete analyses with particular focus on analyses for real-time and embedded software: A region analysis, a space/time complexity analysis, and an exception escape analysis.

The effect checking type systems will also feed back into other analyses and into general language programming. For instance, it's reasonable to require that functions defined in specifications, such as pre/postconditions or invariants be total and side-effect free. If these guarantees can be established a-priori, the design and implementation of contract checkers becomes simpler. Another issue in the programming languages is the handling of guards in pattern matching code. Again, one should demand that a guard be side-effect free.

The work planned in this thrust will include the following topics:

(1) Augment Scala's existing framework for annotated types with *annotation type parameters*. Such type parameters should be compile-time only; unlike the existing type parameters in .NET generics they would be erased at run-time. These type parameters are needed in polymorphic type annotations, such as the ones that express polymorphic effects. Rules for syntax, type checking and type inference of polymorphic annotations will have to be designed and investigated. This work item should also develop recommendations for adding polymorphic annotations to .NET's standard annotation framework.

(2) Develop a modular framework for effect-checking type systems. Principal questions are: How can one define and combine several effect analyses? Should it be possible to abstract in an effect-polymorphic function over several analyses at once? Can one find a lightweight notation by making intelligent use of defaults?

(3) Investigate how effect information can be retrofitted to existing .NET libraries. This part is crucial because virtually every component will make heavy use of the .NET infrastructure—either directly or indirectly. Assuming a worst-case scenario every time a common CLR library is accessed would weaken analyses to the point of making them impractical.

(4) Based on the infrastructure, develop two simple effect systems for totality and side-effect freedom. Have the results of these effect systems flow back into other analyses and the base language.

(5) Again based on the infrastructure, develop three analyses that are move involved: a region analysis to avoid the need of a garbage collector, a space/time complexity analysis to estimate resource consumption, and an escape analysis to give approximations to which exceptions an application might throw. These analyses have special utility in an embedded systems context, where resources are limited, real-time garbage collection is expensive, and reliability is often of crucial importance.

2.7 Analyses Capturing Physical Program Properties

While much research has been carried out on topics such as scheduling, fault-tolerant architectures, and fault analysis, we are still at an early stage of incorporating nonfunctional properties into programming language design. Indeed, current practice in embedded programming has been likened to the assembly age, with programmers tweaking low-level constructs such as task priorities in order achieve satisfying timing behavior in test suites. We have advocated the view that the programmer should be relieved from managing real time, failure rates, and other physical properties in the same way in which the programmer is relieved, in high-level languages, from managing memory. We designed an experimental language, called Giotto, in which the programmer specifies end-to-end timing behavior and long-term failure probabilities, and the compiler ensures these properties (if possible) by generating appropriate task schedules and by replicating tasks on multiple processors [18, 15]. In order to do this, the compiler must know (or make assumptions about) the worst-case execution times of tasks and hardware failure rates.

In this project, we will develop within ProgLab language-based mechanisms —type systems and program analyses— to give the programmer information about physical program properties such as timing, resource, and reliability properties relative to a specified execution platform. For this purpose, execution platforms will become objects of study within ProgLab, and execution time analysis as well as fault analysis will become objectives of program analyses within ProgLab. Moreover, based on insights from the Giotto project, we will provide type-based extensions to languages for specifying both timing and failure behavior.

More specifically, we advocate a coroutine-style programming model, where individual software tasks operate on logically private memory space in logically private execution time slots under a logically fixed failure rate. All interaction between tasks is restricted to program-specified public variables at program-specified public time instants. For example, a task may announce that it will read an input variable and

write to an output variable every 10 ms, meaning that there will be no additional, intermediate synchronization points with other tasks. The task is admitted only if, for each invocation, sufficient memory space and execution time can be allocated to guarantee that the outputs are computed in time. The admissibility analysis needs to use performance assumptions about the execution platform to compute space and time utilization requirements for the task. The analysis will generate both memory layouts (the assignment of space to tasks) and schedules (the assignment of time to tasks) partly statically, when task bytecode is generated, and partly just-in-time, when native code is generated. We will experiment with various trade-offs between static and JIT resource management and scheduling, using an extended bytecode such as embedded machine code [16, 17], which includes resource management instructions and annotations. Since the performance assumptions about the execution platform may be incorrect, and to allow some amount of dynamic resource management and scheduling, the resource consumption of each task will, in addition, be monitored at run-time, invoking program-specified exception handlers if either space or time overruns occur. The goal is that, once admitted, each task will exhibit deterministic (i.e., predictable) timing behavior, independent of the overall load of the system.

Faults will be handled in a similar way. A program may specify, for example, that in the long run, 99.99% of all computed output values of a periodic task are valid. Invalid output values may be due to hardware failures such as sensor faults, or due to software failures such as insufficient available execution time. As part of the admissibility analysis, the compiler generates a replication mapping of tasks to processors which guarantees the specified task failure rate under given assumptions (e.g., hardware failure rates) about the execution platform. For instance, if the assumed failure rate of a processor is 1%, then in order to achieve the specified task failure rate of 0.01%, each task invocation needs to be replicated on two independent processors.

ProgLab will provide a user-friendly facility for experimenting with novel embedded programming models such as the proposed one. It will support both the capability to define performance assumptions of resource-constrained execution platforms such as cell phones, and the capability to extend high-level languages with types, and intermediate languages with instructions and annotations, for resource management.

2.8 Enabling Analyses using Theorem Provers

Theorem proving and constraint-solving are among key enabling technologies for analysis and verification of software systems. They are an essential component of software model checking tools such as SLAM [2], Static Driver Verifier [1], and BLAST [19]. Software for embedded systems imposes new kinds of constraints on timing, power, and memory use, and requires new advances in constraint solving techniques. The size and complexity of embedded software is increasing, which requires tools for checking rich classes of properties. Furthermore, the integration into development environments such as Visual Studio demands algorithms that produce concrete timely feedback.

Provers for Expressive Specifications. Our view is that theorem provers for future software analysis tasks need native support not only for traditional theories such as uninterpreted function symbols and linear arithmetic, but also for much richer theories. We will develop and integrate decision procedures for expressive constraints appearing in software specifications, such as recently proposed decision procedures for collections such as sets and bags with cardinality operators [23, 30, 31]. For mutable data structures we will develop more efficient decision procedures that support transitive closure on tree-like graphs, building on our experience [34] and taking into account recently proposed techniques [13]. We will also develop and implement decision procedures for inductive data types; the impact of such decision procedures in ProbLab will be reinforced by Scala's strengths in developing and checking properties of functional programs [12].

We will systematically study the complexity, proof systems, and interpolation for these theories, drawing the the boundary between tractability and intractability of proof obligations. We will also explore feasibility of combining such rich constraints over non-disjoint theories [21, Chapter 4]. Just like solvable clases of differential equations are essential for computer algebra and modelling packages, solvable classes of logical constraints will form the foundation of automated reasoning and analysis in ProgLab.

Proofs, Interpolants, and Models. Our algorithms will provide not only yes/no answer to formula validity but provide concrete evidence useful for other ProbLab components and incoporated in evidence

management system. Such evidence will be essential for providing feedback to ProgLab users. For valid constraints, our procedures will generate proof objects that justify reasoning to users and enable computation of interpolants for invariant inference. For satisfiable constraints, our procedures will construct models (solutions), building on model finding using SAT solvers [22]. In addition to satisfiability problems, we will explore more general optimization problems and their use in embedded software analysis [25], especially in integer arithmetic domains.

Deployment in ProgLab. We will implement our provers in the Scala programming language to enable smooth fine-grained integration with ProgLab. As part of an API for reasoning services we will provide a definition of an expressive formula language based on the Isabelle/HOL logic [27], building on our previous experience [21, Chapter 4]. The developed decision procedures will be particularly useful for rich types and effects. They will also provide ProgLab with unique strengths of reasoning about mutable data structures. In addition to its uses within rich type systems and verification condition generation, we expect the prover to directly support deep reasoning about functional Scala programs by embedding a Scala subset into the language of prover's formulas. Such integrations will provide similar benefits to Scala and .NET as the unification of programming and specification language in ACL2 or code generation in Isabelle.

2.9 Expected Collaborations

Interaction with Microsoft groups. We will collaborate with several groups at Microsoft in order to achieve our research objectives. We plan to collaborate with Peter Müller at Microsoft Research Redmond on a variety of ProgLab issues, including annotated types, ownership types, and static analyses. We plan to collaborate with Rustan Leino from Microsoft Research Redmond on aspects of system architecture, as well as exchange formats that will facilitate interoperability with Spec#. We also maintain interactions with Nikolaj Bjørner, Byron Cook, Shaz Qadeer, and Sriram Rajamani on automated software verification. The Visual Studio Languages Team has agreed to act in an advisory role for our efforts to write a visual studio plugin. That team overseas design, compiler and IDE support for Microsoft's major managed languages, including C#, VB, F#, IronPython and IronRuby. Sean McDirmid at Microsoft Research Bejing intends to help is with general .NET integration issues.

Interaction with Researchers in Switzerland. Two related ICES research proposals complement our work: 1) the proposal on *Failure Immunity for Embedded Software in Consumer Devices* by George Candea (EPFL), and 2) the proposal on *Soft Integration of Hard Real-Time Capabilities in C#* by Rachid Guerraoui (EPFL) and Jan Vitek (Purdue). We are already interacting with groups led by George Candea and Rachid Guerraoui (which are also part of TRESOR). We expect ICES program to further facilitate our interaction. As a collaboration activity related to the first proposal, we will investigate specification formats that enable runtime mechanisms to enforce properties that were not established using static techniques. In relation to the second proposal, we will explore a common .NET format for pluggable types that can be shared by the C# extension and by the .NET bytecodes generated by Scala's existing pluggable type system. To complement these activities on Scala and C#, in the course of this project we also expect interactions with researchers from ETH Zürich related to common representations for components written in additional programming languages compiled to .NET platform.

2.10 Relevance of the Proposed Research

ProgLab.NET will bring the benefits of advanced analysis and programming language tools to the realm of embedded software on the .NET platform. Proposed techniques have the potential to increase the productivity of developing reliable software in a range of embedded devices, from safety critical control software to software running in resource-constrained portable computing environments. The research will also have consequences in software development in general, because it will present a unique case of integration of a popular language and platform with advanced analysis techniques. Common representation formats developed in the context of ProgLab in collaboration with other research groups have the potential of unifying previously independent techniques and bringing together the corresponding research communities. Such effort will enable researchers and industry to meet the challenge of cost-effective reliable software development for future platforms.

3 Project Plan

3.1 Activities, Milestones, Deliverables

Figure 1 shows a tentative Gannt chart of the proposed activities described in previous sections. The four thrusts approximately correspond to responsibilities of four research team members for which we are asking support:

ProgLab infrastructure will be the responsibility of the programmer;

Rich type analyses will the responsibility of first PhD student;

Analyses capturing physical properties will the responsibility of the second PhD student;

Theorem provers for program analyses will be the responsibility of the third PhD student.

We expect significant interaction between the four thrusts and the four team members. In particular, all members will work closely with the programmer on the design of ProgLab infrastructure. All three principal investigators will interact with all the team members through regular joint research meetings.

Thrust	Year 1	Year 2	Year 3	Year 4
ProgLab infrastructure	interaction interface, core plugin	manager for evidence, requirements, platforms	interactive and visual feedback	.NET extension recommendations
rich types	polymorphic annotations, simple effects	modular effect composition, retrofitting	escape and region analysis	space/time complexity analysis
physical properties	intermediate real-time language	admissibility analysis	fault-tolerant compilation	case studies and evaluation
theorem provers	prover language, interface, SMT engine, VCGEN	decision procedures (collections, integers)	reachability, interpolation, infinite models	integer optimization, evaluation

Figure 1: Gannt chart of activities in 4 research thrusts over the period of 4 years

Our final deliverables will be take two main forms: 1) the ProgLab infrastructure along with associated analyses and the theorem prover, all available under the BSD licence; 2) publications and presentations describing ProgLab design as well as the foundation and the implementation of the analyses.

Our first milestone, to be delivered after two years, will be a first version of ProgLab with a core Visual Studio plugin, a simple effect system integrated into Scala's type system, a compiler for a domain-specific sublanguage of Scala tailored to embedded systems, and a path-sensitive analysis of this sublanguage based on our satisfiability-modulo-theories theorem prover.

Subsequent two years will lead to a more robust and interactive deployment within Visual Studio, with additional effect systems that track memory usage, the corresponding theorem proving technology for reasoning about heap properties, and the notion of fault-tolerant computation that accounts for errors (such as memory errors) occurring under certain probability.

3.2 Dissemination, Standardization, and Patent Activities

Dissemination. Our research groups pursue active dissemination activities in the form of research publications, public presentations, and released software artifacts. Taken together, the number of publications for the three principal investigators is over 300, and the group has publicly released software HyTech, Mocha, Blast, Giotto, Hob, Jahob, GJ, Pizza, and Scala. We will continue these activities in the context of ProgLab. Following the ICES research program guidelines, we will release our software under the BSD licence.

Standardization. Jointly with our collaborators from Microsoft, EPFL, and ETHZ, we will work on establishing joint formats for exchanging information about software artifacts, working towards community standards for reliable software.

Patents. We do not expect to pursue any patent activities specific to the research in this proposal.

References

[1] T. Ball, B. Cook, and V. L. S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft Research, 2004.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[5] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

[6] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In T. Ball and R. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006, Seattle, WA, August 16-20)*, LNCS 4144, pages 532–546. Springer-Verlag, Berlin, 2006.

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[9] D. R. Cok and J. R. Kiniry. Esc/java2: Uniting ESC/Java and JML. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[10] J. S. C. (Counterpunch). US: The fatal flaws in the Patriot missile system. `http://www.corpwatch.org/article.php?id=11110`, 2003.

[11] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, 2007.

[12] M. Dotta, P. Suter, and V. Kuncak. On static analysis for expressive pattern matching. Technical Report LARA-REPORT-2008-004, EPFL, 2008.

[13] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 342–351, New York, NY, USA, 2007. ACM Press.

[14] R. Gerth. Model checking if your life depends on it: A view from Intel's trenches. In *SPIN Workshop*, volume 2057 of *LNCS*, page 15, 2001.

[15] A. Ghosal, T. Henzinger, C. Pinello, A. Sangiovanni-Vincentelli, K. Chatterjee, D. Iercan, and C. Kirsch. Logical reliability of interacting real-time tasks. In *DATE*, April 2008.

[16] T. Henzinger and C. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 315–326. ACM Press, 2002.

[17] T. Henzinger, C. Kirsch, and S. Matic. Schedule-carrying code. In *EMSOFT: Embedded Software*, Lecture Notes in Computer Science 2855, pages 241–256. Springer, 2003.

[18] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.

[20] D. Hovemeyer and W. Pugh. Finding bugs is easy. Invited talk, MASPLAS, April 2003.

[21] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[22] V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In *Joint 10th European Software Engineering Conference (ESEC) and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[23] V. Kuncak and M. Rinard. Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In *CADE-21*, 2007.

[24] N. G. Leveson. Appendix A, Medical devices: The Therac-25 story, `http://sunnyday.mit.edu/papers/therac.ps`. In *Safeware: System Safety And Computers*. Addison Wesley, 1995.

[25] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, 1996.

[26] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, 1988.

[27] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[28] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2004-006, EPFL IC, 2004.

[29] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57, 2005.

[30] R. Piskac and V. Kuncak. Decision procedures for multisets with cardinality constraints. In *9th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS, 2008.

[31] R. Piskac and V. Kuncak. On linear arithmetic with stars. Technical Report LARA-REPORT-2008-005, EPFL, 2008.

[32] 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 *DSN*, pages 625–, 2003.

[33] P. Wadler. The marriage of effects and monads. In *ICFP*, pages 63–74, 1998.

[34] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.

[35] H. Zhang. SATO: An efficient propositional prover. In *CADE*, pages 272–275, 1997.

[36] Software bug causes train wreck. `http://www.availabilitydigest.com/public_articles/0101/software_bug_causes_train_wreck.pdf`, October 2006.