1 Project B: Static Quality Assurance

Principal Investigator: Peter Müller, ETHZ
Co-Investigator: Viktor Kuncak, EPFL

1.1 Summary of the Research Plan

Modern society is increasingly relying on software infrastructure, with software crashes potentially having tremendous consequences, from substantial economic damage to human casualties. The absence of a crash due to a run-time error is an example of an easy-to-state formal specification that every piece of software should satisfy. We have recently witnessed significant advances in verification techniques and theorem proving, which made it possible to formally prove that software satisfies such specifications. These techniques have been successful for sequential first-order programs such as device drivers.

However, despite these improvements, existing verification techniques suffer from severe shortcomings: they cannot handle modern object-oriented programs, are limited to simple properties, require interaction of experts in formal methods, or require knowledge of the whole program. The goal of this project is to address some of these issues by developing an automatic program verifier for Scala programs. Scala distinguishes itself from other object-oriented (OO) programming languages by providing powerful language features typically found in functional programming languages (such as higher-order functions and abstract data types) and by offering a strong type and effects system supporting, for instance, path-dependent types. These features make Scala significantly more expressive than mainstream OO-languages such as Java and C#. The objective of this project is to exploit these features for program verification. More specifically, we plan to work on the three main components of an automatic program verifier:

1. Specification: We will develop a contract language for Scala that supports static and run-time assertion checking. We will exploit the functional features of Scala to develop a contract language that is expressive and has a precise formal semantics.
2. Verification: We will develop a modular verification methodology for Scala's mix of object-oriented and functional features, including in particular concurrency. To deal with heap data structures and side effects as well as with concurrency, we will develop new techniques to specify and check the non-interference of heap accesses.
3. Theorem proving: We will develop automated reasoning techniques that will support the analysis and verification of sequential and concurrent Scala programs. Our techniques will handle proof obligations arising both from (a) ensuring a programming discipline such as non-interference, and (b) proving correctness of programs that use such a discipline.

We expect these activities to lead to the following key results: (1) A type system and static analysis to specify and check non-interference of heap operations; (2) A modular specification and verification methodology for Scala; (3) A theorem prover with decision procedures for the developed verification methodology.

The proposed research addresses the above-mentioned shortcomings of program verification by supporting a modern OO-language, by enabling the verification of complex functional behavior, by a high degree of automation, and by being fully modular. It will lead to a precise understanding of the benefits that functional programming concepts in OO-languages have for formal reasoning, and will, therefore, provide the foundations for an important trend in programming language design.

1.2 State of the Art

This subsection summarizes important results in the three main research areas involved in this proposal, program verification, non-interference, and theorem proving.

Verification of functional programs with higher-order logic or type theory can handle features such as higher-order functions. However, these approaches do not handle mutable heap data structures. Therefore, despite Scala's strong functional influence, verification of Scala programs requires a technique capable of handling OO-programs with mutable heap structures, aliasing, and subtyping. There are three major techniques for the modular verification of this class of programs.

Techniques based on ownership [18] structure the heap hierarchically into contexts and control accesses to these contexts. The context structure is used to maintain invariants, to prove disjointness of object structures, to specify and check read and write effects, to impose locking disciplines, to specify termination measures, and for object immutability. Ownership-based techniques are used in two of the most prominent verification projects, the Java Modeling Language JML [47] and Spec# [9]. Ownership-based verification has been very successful, but is too restrictive for non-hierarchical structures [61] and complex interactions between data structures, which are common in Scala (for instance, because of higher-order functions).

Separation logic [69] is an extension to Hoare logic that makes disjointness properties of heap structures explicit in formulas using the so-called separating conjunction. One of the main virtues of separation logic is its support for framing, that is, for reasoning about properties that remain invariant under heap updates. Separation logic has been used successfully to verify concurrent programs [77] and has also been extended to OO-programs [63]. A drawback of separation logic is that it is difficult to automate; essentially, it requires a special theorem prover to handle separating conjunction [62, 59].

Dynamic frames [32, 74] use explicit specifications of read and write effects for framing. Verification based on dynamic frames, especially on implicit dynamic frames [73], is very similar to separation logic; it is equally powerful, but more amenable to automatic first-order theorem provers. Therefore, we will use dynamic frames in this project. Regional logic [7] is very similar to dynamic frames; it uses ghost state rather than functions to denote the effects of operations.

Reasoning about OO-programs benefits greatly from techniques that structure the heap, especially ownership type systems [18]. There is a variety of ownership systems that enforce different encapsulation properties [2, 18, 26], check effects [16, 15], and prescribe locking disciplines [13]. Recent systems go beyond the classical static hierarchies by supporting multiple owners [15] and by permitted dynamic changes of the ownership hierarchy, so-called ownership transfer [3, 17, 57]. We will build on this work to specify and check non-interference of heap operations.

State of the art automated theorem provers such as Z3 [24] and CVC3 [10] are a crucial component of verifiers such as Spec# [9] and ESC/Java2 [19]. These provers typically use a combination of successful SAT solving algorithms with a number of specialized decision procedures [29] and quantifier-instantiation techniques [23] and belong to an emerging field of satisfiability modulo theories [68]. Theorem provers also enable software verification within predicate abstraction [5, 11] and shape analysis algorithms [78, 79]. An alternative approach is to develop the software system within a theorem prover and then generate executable code. This approach has a long tradition in the ACL2 system [33] and is also popular within the interactive theorem provers Coq [54] and Isabelle [36]. Other systems propose a combination of verification of imperative programs and theorem proving through a unified interactive interface [1, 6].

1.3 Previous Work by the Investigators

This subsection outlines the applicants' work related to the proposed project.

1.3.1 Peter Müller

Peter Müller is working on various aspects of programming methodology including program verification, type systems, and language design. He has been one of the main contributors to ownership-based verification techniques. In particular, he developed modular verification techniques for object invariants [56, 49, 53, 28], data abstraction through model fields, pure methods, and model classes [21, 22, 51, 52], and for C# delegates [58]. He has also worked on various aspects of program specification, for instance, information hiding in specifications [70] and well-definedness of contracts [46]. His techniques are used by JML and Spec#, two of the leading frameworks for the specification and verification of OO-programs. Müller also published an overview of specification and verification challenges [45], some of which will be addressed in this project.

Much of Müller's work on program verification is based on heap structuring techniques. To that end, he has developed the Universe Type system [26], an ownership type system that distinguishes itself from other systems by controlling the modification of objects while permitting arbitrary aliases. Therefore, Universe Types provide powerful support for verification, but are also very flexible. They allow dynamic changes of ownership (ownership transfer) [57], and support important language features such as generics [25]. Universe Types are available as part of the JML tool suite; a similar ownership system—based on verification instead of types—has been implemented in the Spec# system [49]. It has been demonstrated that an adaptation of Universe Types to Scala is possible [72].

1.3.2 Viktor Kuncak

Viktor Kuncak is working in the area of automated reasoning, verification, programming languages, and software engineering. He has architected and, with collaborators, developed the Jahob verification system for Java programs [35]. Jahob deploys new theorem proving algorithms and integrates a number of existing techniques into a single verification system [37, 80]. The techniques employed in Jahob include shape analysis with field constraint analysis [79], encodings into first-order logic [12], and new decision procedures for sets with cardinality constraints [41].

Previously, he was involved in the Hob system that combines data structure verification with set-based typestate analysis to verify both high-level and detailed program properties [40, 44]. He also worked on static role analysis [39], a form of type state analysis that captures precise aliasing information.

In the area of foundations of static analysis, he addressed an open question of decidability of subtyping constraints [43], proved undecidability of a class of heap constraints [42], and worked on applications of interactive theorem proving [4]. Has has also contributed to counterexample-finding approaches in potentially infinite structures [38], enabling the application of constraint solvers such as Alloy [31] to integers and inductive data types.

His most recent work includes new algorithms for proving validity of expressive constraints on sets and multisets with cardinality operators [65, 64] and related extensions of linear arithmetic [66]. These results illustrate the need for specialized decision procedures in the context of more general automated reasoning systems.

1.4 Detailed Research Plan

Objectives

The goal of this project is to develop a deductive verification platform for functional, object-oriented, and concurrent programs for modern programming languages such as Scala. We expect these activities to lead to the following key results:

1. A type system and static analysis to specify and check non-interference of heap operations and disjointness of heap regions. Our techniques will exploit Scala's type system and the decision procedures developed in this project to support flexible and dynamically-evolving heap structures.

2. A specification and verification methodology for Scala. This methodology will extend existing work on OO-programs by a treatment of Scala's functional features. It will extend existing work on functional programs by supporting heap data structures with side effects.

3. A theorem prover with decision procedures tailored towards the developed verification methodology. The theorem prover will accept an expressive formula language, including formulas in Scala syntax, and will use a range of decision procedures and proof search strategies to establish validity or find counterexamples of formulas.

Methodology

We will start by identifying a class of target Scala programs that we will use to develop the verification methodology. In parallel, we will develop a basic verification and theorem proving platform. We will perform manual proofs for critical cases and then develop verification conditions that automate the tasks. To prove the generated verification conditions, we will first consider encodings into existing efficient theorem provers, such as Z3. Our experience suggests that such encodings will not be sufficient to automatically prove complex examples. These examples will provide us with verification conditions that will drive the development of our prover technology to achieve the desired level of automation.
The overall architecture of our verifier is shown in Fig. 1. The Sequential Verifier (developed in Workpackage B1) computes verification conditions for Scala programs, correctly handling higher-order features of Scala. The Sequential Verifier passes its verification conditions to the Scala Prover. The Scala Prover (developed in B3)

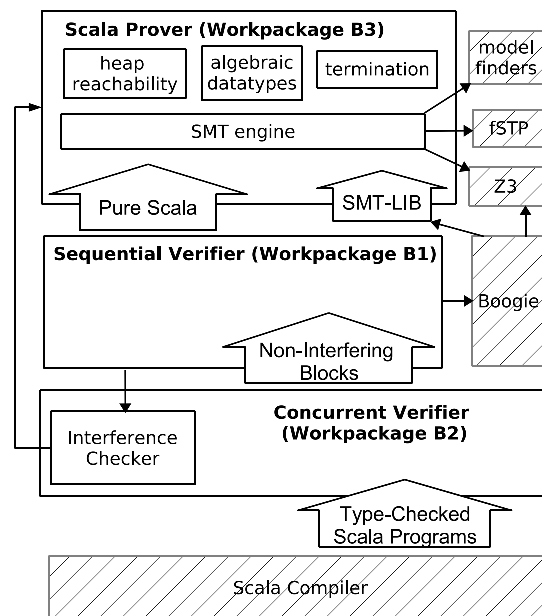

Figure 1: Architecture of the developed verifier. Black boxes depict components developed in this project; grey hatched boxes depict external components that we will build on.

uses its specialized decision procedures and model finders to prove verification conditions or report counterexamples back to the Sequential Verifier. To handle concurrent, actor-based programs, we will develop a Concurrent Verifier (B2), which builds on the Sequential Verifier and uses a type-based non-interference checker to guarantee the isolation of actors. The non-interference checker will also be used for sequential verification to reason about heap operations.

In the following subsections, we present the research plan for the Workpackages B1–B3. Each workpackage corresponds to one PhD project.

Workpackage B1: Scala Verifier

The first workpackage focuses on the automatic verification of sequential Scala programs. Its main scientific objective is to develop a verification methodology that can handle functional language features such as higher-order functions and abstract data types, in the presence of heap data structures and side effects. The extension to concurrent programs will be done in Workpackage B2, described below.

Task B1.1: Basic Verification Methodology. Our verification methodology for Scala will be based on dynamic frames [32, 74]. This approach allows one to reason effectively about the effects of heap updates on the result of function applications by making the read and write effects of operations explicit in specifications. It is, thus, well-suited for the verification of Scala programs, which contain heap updates, w.r.t. specifications written in the annotation language developed in Task B3.1, which make have use of function applications. In particular, dynamic frames are less restrictive than ownership regimes [49, 18] and seem more amenable to automatic verification than separation logic [63, 69].

We will design our verification methodology to be sound (that is, all successfully verified programs are actually correct) and modular (that is, each class can be verified without knowing its clients or subclasses). Modularity is crucial to handle library code and for scalability.

Verification requires a formal semantics of the programming language. We will develop a heap model for Scala, which will in particular support function objects and encode Scala's advanced types (for instance, path-dependent types and family polymorphism). The semantics of Scala will be defined by a translation to a simple imperative intermediate language such as BoogiePL [8]. This translation will especially cover specifics of Scala such as traits and mixins, and pattern matching. The representation in the intermediate language also includes all assertions required to prove the absence of runtime errors and the satisfaction of the annotations provided by the programmer. Verification conditions can then be computed easily on the intermediate representation by a tool such as Boogie [8]. We will implement the translation from Scala to the intermediate representation by extending the existing Scala compiler. For this task, we can build on our earlier work on the Boogie verification methodology [49, 50, 51, 53]. as well as on reasoning about pure methods [21, 52].

Task B1.2: Verification of Functional Features. In this task, we extend the basic verification methodology to the functional aspects of Scala, in particular, to higher-order functions and to properties expressed in the style of algebraic laws.

In Scala, every operation is an object, which leads to a natural encoding of higher-order functions and curried functions. A key issue in verifying programs containing function objects is that the client of a function object can be verified only relatively to the specification of the function object it uses. For instance, the result of filtering a collection depends on the behavior of the used filter

function. To handle such situations, we will equip every function object with functions for their pre- and postcondition as well as their read and write effect, respectively. A client of a function object can use those functions in its specification; for instance, it might require in its precondition that the precondition of a passed function object holds and that the function object has an empty write effect. Dynamic frames as used in Task B1.1 provide the necessary machinery to reason about the effects of heap updates on these properties of function objects. For this task, we can build on our earlier work on reasoning about pure methods [21, 52].

With a programming language with strong functional aspects, it seems natural to write certain specifications in the style of algebraic laws (such as `rev(rev(x))=x`). However, reasoning about such specifications in an OO-language is non-trivial because of side effects (if `rev` has side effects, its use in specifications is questionable) and object identities (if `rev` is side-effect free, its result will be a fresh object and, thus, the validity of the above specification depends on the meaning of '`=`'; in particular, it does not hold if '`=`' denotes reference equality). We will address this problem by allowing programmers to provide functional reference implementations and by proving a simulation relation between this reference implementation and a more efficient implementation using side effects. For this task, we can build on our earlier work on reasoning about JML model classes, which contain algebraic specifications for side-effect free methods [22].

We will implement the results of this task in the verifier developed in Task B1.1.

Task B1.3: Evaluation. In this task, we will evaluate the verifier for sequential Scala programs by specifying and verifying parts of the verifier and of the Scala Prover developed in Workpackge B3. We will evaluate the expressiveness of the annotation language developed in Task B3.1 (can we express the functional behavior of the code), the strength of the program verifier (can correct code actually be verified), and the speed of the program verifier (do we produce verification conditions that can be handled well by automatic theorem provers). Our results will also be compared to similar results from a state-of-the-art verification system such as Spec# [9].

Workpackage B1 Integration and Collaboration. The PhD student working on Workpackage B1 will be based at ETHZ with Peter Müller as the primary advisor. Viktor Kuncak will co-supervise the student, especially to ensure a smooth integration of the verifier with the annotation language and the Scala Prover developed in Workpackge B3. We will collaborate with the staff of Workpackage B2 to integrate the non-interference checker and the verification methodology for concurrent programs into the verifier and with Martin Odersky and the staff from Project A on the formalization of the semantics of Scala.

Workpackage B2: Process Non-Interference

The second workpackage focuses on the automatic verification of concurrent Scala programs. Its main scientific objective is to develop techniques to specify and check the non-interference of heap operations, and to use the non-interference results during verification, especially of concurrent programs.

Task B2.1: Specifying and Checking Non-interference. Verification of OO-programs relies heavily on the non-interference of heap operations, for instance, to prove that a heap update does not affect the result of a function or to prove that two threads or actors operate on disjoint memory regions and, thus, do not lead to data races.

Based on a semantic characterization of the required non-interference properties, we will develop a type system that can specify and check non-interference. The type system will allow programmers to structure the heap into regions and to enforce inclusion and disjointness properties of regions. Our type system will improve on existing heap structuring techniques in four ways. First, our regions are sets of locations (object-field pairs) rather than sets of objects; this fine granularity is for instance important for the precise specification of read and write effects. Second, each location can belong to several regions simultaneously; this form of "multiple ownership" is important to handle sharing in data structures. Third, the type system can express and check effects such as read and write effects of methods; statically-checked effects will greatly simplify verification. Fourth, region membership can change dynamically; this form of "ownership transfer" is crucial for dynamically-evolving data structures. Our type system will combine all four qualities, whereas existing work handles these requirements only partially (for instance, datagroups [48] work on the level of locations, MOJO [15] supports multiple ownership and effects, and Universe Types [57] handle ownership transfer).

We will achieve the required expressiveness of the type system by building on Scala's type system (in particular, path-dependent types) and by combining static type checking with run-time checks. We will reduce the necessary run-time checks by combining our type system with static analyses, for instance, to track the uniqueness or immutability of references. Remaining run-time checks will be turned into proof obligations for the subsequent verification step. For this task, we can build on our earlier work on Universe Types [26, 25, 57] and their use in verification [28, 56].

The type system will be implemented in the verifier developed in Task B1.1. One of the key challenges here is to make the guarantees of the type system available to the verifier without drowning the prover in useless facts.

Task B2.2: Verification of Concurrent Programs. Scala does not have a fixed concurrency model, but supports various styles of concurrent programming through libraries. One of them is Erlang-style actors, isolated processes that communicate via message passing. Since actors do not communicate via shared memory, they avoid many of the difficulties of traditional thread-based concurrency. In this task, we will extend our verification methodology to safety properties of Scala programs using actors.

Unlike in Erlang, Scala's actors are not purely functional; they typically mutate data structures and exchange data via references to mutable objects, which may cause supposedly-isolated actors to interfere with each other and to cause data races. We will use our non-interference system from Task B2.1 to permit controlled sharing between actors and use effect specifications to prove the non-interference of actors. This technique will for instance allow one to implement a concurrent quicksort using actors; a mutable array may be shared between actors, and the effects guarantee that the individual actors operate on disjoint regions.

Once we have achieved non-interference between actors, properties of individual actors such as the absence of runtime exceptions can be verified using sequential reasoning techniques. For interactions between actors, we will build on reasoning techniques for distributed systems, where local proof obligations are used to maintain global invariants. We will extend the verifier developed in Task B1.1 to Scala programs with actors.

Task B2.3: Evaluation. This task will evaluate the results of Workpackage B2 for the verification of sequential and concurrent programs. First, we will re-visit the case study from Task B1.3 and apply the non-interference system from Task B2.1. We will evaluate the expressiveness of the system and compare the performance of the verifier to the results from Task B1.3 in order to

quantify the benefits of a non-interference type system over proving non-interference in a theorem prover. Second, we will apply the extended verifier from Task B2.2 to several examples using actors. We will verify the isolation of actors and simple functional properties.

Workpackage B2 Integration and Collaboration. The PhD student working on Workpackage B2 will be based at ETHZ with Peter Müller as the primary advisor. Martin Odersky will co-supervise the student to ensure a close integration with Project A. In particular, Project A will develop a lightweight type system that guarantees non-interference of actors by preventing mutable state from being shared between actors. In Task B2.2, we will use verification techniques to develop a solution that is more flexible, but might also impose more annotation and checking overhead. In other words, Project A and B explore different ends of a spectrum with a focus on programming and verification, respectively. We will collaborate with the staff of Workpackage B1 to integrate the non-interference checker and the verification methodology for concurrent programs into the verifier, and with the staff of Workpackage B3 on the integration of type systems and decision procedures for non-interference.

Workpackage B3: Prover Technology

The goal of the third workpackage is to develop and implement theorem proving techniques that support the verification of imperative and concurrent programs. The resulting theorem prover, which we call *Scala Prover*, will accept as input logical formulas as well as constraints in a subset of Scala, and will itself be implemented in Scala.

Task B3.1: Specification Using Pure Scala. The first step in developing the Scala Prover is to identify its formula input language, which we call *Pure Scala*. We plan to use classical higher-order logic as the semantic basis of Pure Scala and provide Isabelle/HOL as one of the concrete syntax formats. In addition to quantifiers, sets, and relations, which are a standard part of higher-order logic, the language will directly support data types such as integers, rationals, bit-vectors, lists, arrays, user-defined algebraic data types, and multisets. The explicit account of these constructs in the language will enable the Scala Prover to reason about them by invoking efficient specialized decision procedures.

Pure Scala will also be the basis for the annotation language of the Scala Verifier in Workpackages B1 and B2. To make the annotation language accessible to Scala programmers, we will develop an embedding of higher-order logic formulas into Scala. Syntactically, it will therefore be possible to write formulas using a subset of Scala as another concrete syntax format. In designing the annotation language we will ensure a correspondence between the higher-order logic semantics and the run-time execution semantics of Pure Scala expressions, identifying an executable subset of Pure Scala. Challenges in ensuring the correspondence include recursive definitions, side effects, exceptions, and the evaluation order. We will use Scala's type and effect discipline to show the absence of side effects and exceptions in expressions. To ensure termination in a range of important cases, we will use terminating combinators, and rule out potentially looping constructs using a notion of language *layer discipline*, which restricts Scala expressions to a fragment of the full language.

This development will benefit from Scala's support for functional programming, including anonymous functions, lazy evaluation, and a standard library of higher-order logic functions and immutable collections. We will build on the experience in the development of JML [47] and the

code generation facilities of Isabelle/HOL. We have started exploring some of these problems in a run-time checker for the Jahob verification system [81], a run-time checker for separation logic [60], and recent student projects at EPFL exploring run-time checking of Scala contracts. We will collaborate with the staff of Project C to develop techniques for efficient execution of a class of run-time checks expressed in Pure Scala.

Task B3.2: Proof and Counterexample Construction for Pure Scala. Having designed formula semantics and its connection to Scala, we will develop the automated reasoning capabilities of the Scala Prover, working on two complementary fronts: finite model finding and proof construction.

Finite model finding techniques will enable the Scala Prover to construct concrete counterexamples to verification conditions generated from the Scala Verifier. To ensure a correct interpretation of results of finite model finders, we will build on our notion of existential bounded-universal formulas [38] and the connection to run-time execution semantics from Task B3.1. Our initial model finding techniques will build on the Kodkod relational model finder [76] based on eager encoding into SAT. In later stages we expect to move model finding capabilities directly into the core Scala Prover engine. As part of this effort, we will explore the potential of code execution in finite model finding, inspired by the encouraging results of the Korat family of algorithms [14, 55].

Dually to counterexample generation, we will develop proof construction capabilities within the Scala Prover. Our starting point will be the Isabelle/HOL proof system, whose broad applicability has been demonstrated through proofs of programs, cryptographic protocols, and deep mathematical results. Having adopted a proof system that is complete for practical purposes, we will develop connections with external automated provers to increase proof automation, following our previously developed approach of formula approximation [37]. We will also explore further a novel idea of representing proofs using assertions within programs [80]. We expect these techniques to work even better in the context of Pure Scala and to combine well with model finding. The expected result will be an approach to programmer-assisted theorem proving that is easily accessible to Scala programmers.

Task B3.3: Automating Inductive and Termination Reasoning. Having gained experience with Pure Scala's foundations and semi-interactive reasoning, we plan to increase the automation of the Scala Prover by developing new decision procedures and proof search algorithms. Our starting point for automation will be a satisfiability modulo theory architecture [29] applied to Pure Scala programs and the previously introduced notion of formula approximation [37, 81]. We have already obtained a generalization of these algorithms to richer representations, avoiding the need for encodings using clauses [75].

The emphasis of our automation will be decision procedures and heuristics for reasoning about inductive properties, such as reachability properties in graphs and trees. In this context we will explore both automata-based methods and proof-theoretic methods, leveraging insights from tools such as MONA [34], description logic reasoners, and μ-calculus-based decision procedures [30]. Unlike previous approaches that focused on the use of reachability in mutable recursive data structures, we will also consider reachability properties that arise at a higher level of abstraction where sets and relations are treated abstractly but the application data itself has recursive structure. These techniques will therefore directly apply to Workpackages B1 and B2. In addition to applications to mutable structures expressed using uninterpreted functions in the logic, we will consider inductive properties in algebraic data types that are ubiquitous in functional programming; our initial experience in this direction was positive [27].

The use of Scala layers will ensure termination of formulas written in the spirit of higher-order logic, but will not suffice for recursively defined functions. To soundly admit definitions of such functions we will develop techniques for automated termination proofs and incorporate them into the Scala Prover. In addition to generating an induction principle, our termination algorithms will estimate the computational complexity of the function. We will explore approaches that build on term rewriting techniques [71] as well as on the recent concept of transition invariants [67, 20].

Workpackage B3 Integration and Collaboration. The PhD student working on Workpackage B3 will be based at EPFL with Viktor Kuncak as the primary advisor and Peter Müller as the co-advisor. The staff will collaborate with the staff of Workpackages B1, B2, and Project C, using the benchmarks generated from these applications to drive the development of the theorem prover. This coordination will ensure that the prover can effectively support automated reasoning tasks needed to prove properties of sequential and concurrent Scala programs. We also expect collaboration with Project A staff on verification of invariants of distributed and actor-based systems.

The development of certain standard parts of the Scala Prover (such as the SAT solver engine) will benefit from related efforts in the context of the ProgLab.NET project funded by the Microsoft Innovation Cluster for Embedded Software. However, the present Workpackage B3 focuses on properties such as graph reachability, needed to prove process non-interference in Workpackage B2 as well as the programmer-friendly theorem proving framework, whereas ProgLab.NET focuses on numerical properties arising from verification of resource usage (timing, power, memory) for .NET platform.

1.5 Timeplan and Milestones

The table below summarizes the milestones for each workpackage. Each milestone marks the availability of a major result.

	Workpackage B1 (Scala Verifier)	**Workpackage B2** (Process Non-Interference)	**Workpackage B3** (Prover Technology)
Month 06	Scala semantics (B1.1)	Semantic characterization of non-interference (B2.1)	Pure Scala (B3.1)
Month 12	Basic verifier (B1.1)	Non-interference type system (B2.1)	Run-time assertion checker (B3.1)
Month 18	Methodology for higher-order features (B1.2)	Integration of type system into basic verifier (B2.1)	Core prover (B3.2)
Month 24	Methodology for algebraic specifications (B1.2)	Non-interference checker for actors (B2.2)	Model finder (B3.2)
Month 30	Sequential verifier (B1.2)	Concurrent verifier (B2.2)	Reachability prover (B3.3)
Month 36	Case study (B1.3)	Case study (B2.3)	Termination prover (B3.3)

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 1–25. Springer, 2004.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 311–330. ACM Press, 2002.

[4] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, Seattle, Nov 8-12, 2004 2004.

[5] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *ACM Conf. Programming Language Design and Implementation*, 2001.

[6] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering (FASE)*, number 1783 in LNCS, 2000.

[7] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *LNCS*, pages 387–411. Springer, 2008.

[8] M. Barnett, B.-Y. Evan Chang, R. DeLine, B. Jacobs 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[9] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*, volume 3362 of *LNCS*, pages 49–60. Springer, 2004.

[10] Clark Barrett and Cesare Tinelli. CVC3. In *Computer Aided Verification*, volume 4590 of *LNCS*, 2007.

[11] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[12] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in a data structure verification system. In *Verification, Model-Checking, and Abstract Interpretation*, November 2007.

[13] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2002.

[14] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, July 2002.

[15] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 42(10) of *ACM SIGPLAN Notices*, pages 441–460. ACM, 2007.

[16] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.

[17] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In Luca Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 176–200. Springer, 2003.

[18] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 33(10), pages 48–64. ACM, 1998.

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

[20] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *ACM Symp. Principles of Programming Languages*, 2007.

[21] Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[22] Á. Darvas and P. Müller. Faithful mapping of model classes to mathematical structures. *IET Software*, 2008. To appear.

[23] Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In *Int. Conf. Automated Deduction (CADE)*, 2007.

[24] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[25] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 28–53. Springer, 2007.

[26] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[27] Mirco Dotta, Philippe Suter, and Viktor Kuncak. On static analysis for expressive pattern matching. Technical Report LARA-REPORT-2008-004, EPFL, 2008.

[28] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer, 2008.

[29] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *16th Conf. Computer Aided Verification*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[30] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *ACM PLDI*, 2007.

[31] Daniel Jackson. *Software Abstractions: Logic, Language, & Analysis*. MIT Press, 2006.

[32] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.

[33] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[34] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.

[35] Viktor Kuncak. The Jahob project web page. `http://javaverification.org`. last accessed: September 2009.

[36] Viktor Kuncak. Binary search trees. The Archive of Formal Proofs, `http://afp.sourceforge.net/`, April 2004.

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

[38] Viktor Kuncak and Daniel Jackson. Relational analysis of algebraic datatypes. In *Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, 2005.

[39] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Annual ACM Symp. on Principles of Programming Languages (POPL)*, 2002.

[40] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), December 2006.

[41] Viktor Kuncak, Hai Huu Nguyen, and Martin Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. `http://dx.doi.org/10.1007/s10817-006-9042-1`.

[42] Viktor Kuncak and Martin Rinard. Existential heap abstraction entailment is undecidable. In *10th Annual International Static Analysis Symposium (SAS 2003)*, San Diego, California, June 11-13 2003.

[43] Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science*, 2003.

[44] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *6th Int. Conf. Verification, Model Checking and Abstract Interpretation*, 2005.

[45] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[46] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE, 2007.

[47] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, December 2006.

[48] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153, 1998.

[49] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.

[50] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *LNCS*, pages 26–42. Springer, 2005.

[51] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 115–130. Springer, 2006.

[52] K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In S. Drossopoulou, editor, *European Symposium on Programming (ESOP)*, volume 4960 of *LNCS*, pages 307–321. Springer, 2008.

[53] K. R. M. Leino, P. Müller, and A. Wallenburg. Flexible immutability with frozen objects. In J. Woodcock and N. Shankar, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, LNCS. Springer, 2008.

[54] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *ACM Symp. Principles of Programming Languages*, 2006.

[55] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *International Conference on Software Engineering*, 2007.

[56] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[57] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 42(10) of *ACM SIGPLAN Notices*, pages 461–478. ACM, 2007.

[58] P. Müller and J. N. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, 2007.

[59] Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *Computer-Aided Verification*, pages 355–369, 2008.

[60] Huu Hai Nguyen, Viktor Kuncak, and Wei Ngan Chin. Runtime checking for separation logic. In *9th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS, 2008.

[61] M. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

[62] M. Parkinson and D. Distefano. jStar: Towards practical verification for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, 2008. to apeear.

[63] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages (POPL)*, pages 247–258. ACM, 2005.

[64] Ruzica Piskac and Viktor Kuncak. Decision procedures for multisets with cardinality constraints. In *VMCAI*, number 4905 in LNCS, 2008.

[65] Ruzica Piskac and Viktor Kuncak. Fractional collections with cardinality bounds. In *Int. Conf. Computer Science Logic (CSL)*, 2008.

[66] Ruzica Piskac and Viktor Kuncak. Linear arithmetic with stars. In *Computer-Aided Verification*, 2008.

[67] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Logic in Computer Science*, 2004.

[68] Silvio Ranise and Cesare Tinelli. The SMT-LIB format: An initial proposal. In *Proceedings of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, 2003.

[69] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[70] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods (FM)*, volume 5014 of *LNCS*, pages 68–83. Springer, 2008.

[71] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and Rene Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Trans. Computational Logic*, (accepted for publication), 2008.

[72] D. Schregenberger. Universe type system for Scala. Master's thesis, ETH Zurich, 2007.

[73] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *Formal Techniques for Java-like Programs*, 2008.

[74] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering FASE*, volume 4961 of *LNCS*, pages 261–275. Springer, 2008.

[75] Philipe Suter. Non-clausal satisfiability modulo theories. Master's thesis, EPFL, September 2008.

[76] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.

[77] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *Concurrency Theory (CONCUR)*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[78] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *Verification, Model Checking, and Abstract Interpratation*, 2006.

[79] Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. Verifying complex properties using symbolic shape analysis. In *Workshop on Heap Abstraction and Verification (collocated with ETAPS)*, 2007.

[80] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2008.

[81] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin Rinard. Runtime checking for program verification. In *Workshop on Runtime Verification*, volume 4839 of *LNCS*, 2007.