ERC Starting Grant 2013 Research proposal [Part B2]

Part B2: *The Scientific Proposal*

Section a. State-of-the-Art and Objectives

Individuals, organizations, industries, and nations are increasingly depending on software and systems using software. This software is large and complex and integrated in a continuously changing complex environment. New languages, libraries and tools increase productivity of programmers, creating even more software, but the reliability, safety and security of the software that they produce is still low. We are getting used to the fact that computer systems are error-prone and insecure. Software errors cost world economies billions of euros. They may even result in loss of human lives, for example by causing airplane or car crashes, or malfunctioning medical equipment. To improve software and methods of software development one can use a variety of approaches, including automated software verification and static analysis of programs.

The successful development and application of powerful verification tools such as model checkers [11, 58], static program analyzers [13], symbolic computation algorithms [8], decision procedures for common data structures [46], as well as theorem provers for first- and higher-order logic [50] opened new perspectives for the automated verification of software systems. In particular, increasingly common use of concurrency in the new generation of computer systems has motivated the integration of established reasoning-based methods, such as satisfiability modulo theory (SMT) solvers and first-order (FO) theorem provers, with complimentary techniques such as software testing [28]. This kind of integration has however imposed new requirements on verification tools, such as inductive reasoning [43, 40], interpolation [34], proof generation [19], and non-linear arithmetic symbolic computations [18]. Verification methods combining symbolic computation and automated reasoning are therefore of critical importance for improving software reliability.

In this project we address this challenge by automatic program analysis. Program analysis aims to discover program properties preventing programmers from introducing errors while making software changes and can drastically cut the time needed for program development, making thus a crucial step to automated verification.

Purpose and Objectives. *The research proposed in this project develops new, unconventional methods of reasoning and reasoning-based program analysis, by exploiting new synergies between symbolic computation and FO theorem proving in the automated analysis of software systems. We will implement world-leading tools based on these methods, and apply them in concrete problems of high industrial relevance.* In particular, the project targets the combination of algorithmic combinatorics, computer algebra, FO theorem proving and static analysis of programs. Our proposal includes the following two project parts (PPs):

(PP1) Automatic generation of program properties by symbol elimination;

(PP2) Efficient reasoning with both theories and quantifiers for proving program properties.

Part (PP1) is intended to design algorithms for an *automatic generation of program properties*. Such properties express conditions to hold at intermediate program locations and are used to prove the absence of program errors, hence they are very important for improving automation of program analysis. (PP1) will rely on our recent symbol elimination method [40], but use symbol elimination in the combination of FO theorem provers and symbolic computation methods. Such a combination is highly non-trivial and requires the development of new reasoning methods based on superposition FO theorem proving, Gröbner basis computation, and quantifier elimination. (PP1) will provide fundamentally new ways of generating program properties, such as loop invariants and Craig interpolants, resulting in crucial improvements in the theory and practice of program analysis.

In (PP2) we will design efficient techniques for automatically proving properties of complex software systems. Such properties usually express arithmetic and logical operations over the computer memory, which can be represented by unbounded data types such as an array, list, pointer, or heap. Proving such properties automatically requires efficient *reasoning with both quantifiers and theories*. We expect (PP2) to have a deep and long-lasting impact in reasoning-based program analysis, bringing us closer to the goal of automatically analyzing programs containing millions of lines of code.

The algorithms developed in our project will be supported by the *development of the world-leading theorem prover Vampire [42], and its extension to support program analysis*. The new features of Vampire will be evaluated on academic and industrial programs, for the latter we have collaboration agreements with verification teams at Microsoft Research, Intel, Saab and Ericsson. We believe that the world-leading position of Vampire in FO theorem proving will be a basis for a major breakthrough in the use of FO provers in program analysis.

Thanks to the full automation and tool support of our project, researchers and software engineers/developers will be able to use our results in their work, without the need to become experts in FO theorem proving and symbolic computation.

Timeliness and Importance. The high-gain aspect of our project comes from its creative use of automated reasoning methods, including new developments related to symbol elimination, program analysis, and combining quantifiers with theories. Our project will turn symbol elimination into a powerful tool for program analysis. Various techniques used in program analysis, such as Gröbner basis computation, quantifier elimination, and Craig interpolation, will be considered as applications, or special cases, of symbol elimination.

Analyzing and verifying large programs requires non-trivial automation, and automatic generation and proving of program properties is a key step to such automation. Our project brings new non-standard approaches to program property generation, yielding properties that cannot be generated by others. By doing so, we will considerably reduce the required manual work. We believe reasoning with both theories and quantifiers will be the main topic in automatic reasoning for the next two decades: advances in program analysis and verification crucially depend on it, it is very complex, and requires developing new mathematics and non-trivial algorithms.

The timeliness of our research is witnessed by the growing academic and industrial interest in program analysis. Microsoft is now routinely using program analysis techniques, for example on the Windows operating system having over 50 millions lines of code, which allowed to reduce the main source of the Windows system crashes down to almost zero. Just recently, in Summer 2013 Facebook bought the academic startup verification company Monoidics and uses now program analysis and verification on its software. According to the Open Web Application Security Project - OWASP, the highest ranked security vulnerability comes from the "execution of unintended commands or accessing data without proper authorization", an error that can be discovered by program analysis. Moreover, more than half of the top ten vulnerabilities listed by OWASP can be prevented by program analysis. Time is thus now ripe for reasoning-based program analysis.

Our project will enable the analysis, and hence partial verification of programs that are beyond the power of existing methods since advanced symbolic computation techniques and their combination with FO theorem proving are not yet used in state-of-the-art program analysis tools. Our results will bring breakthrough approaches to software analysis, which, together with other advances in the area, will reduce the cost of developing safe and reliable computer programs used in our daily life.

State-of-the-Art and Innovative Aspects of the Project. Our project addresses problems that other approaches could not solve so far and is fundamentally different from existing techniques, as detailed below.

(1) Developing polynomial invariant generation algorithms for loops with arbitrary polynomial arithmetic. Many classical data flow analysis problems, such as constant propagation and finding definite equalities among program variables, can be seen as problems about polynomial identities. In [45, 56] a method built upon linear and polynomial algebra is developed for computing polynomial equalities of a bounded degree. A related approach was also proposed by [51] using abstract interpretation. Without an a priori fixed polynomial degree, in [52] the polynomial invariant ideal is approximated by a fixed point procedure based on polynomial algebra and abstract interpretation. In [38] we described an automatic approach computing all polynomial invariants of a certain class of loops which strictly generalizes the programming model of [52]. Our project extends [38] in new ways, uses recent advances in computer algebra and FO theorem proving, and requires no a priori fixed set of predicates or polynomial degree bounds. Our results will significantly simplify the analysis of programs with complex arithmetic.

(2) Developing quantified invariant generation methods for loops over unbounded data structures, such as arrays, list, pointers, and heaps. Developing such methods is very important for complex programs, since such programs extensively use both complex data types and memory management. In [59] loop invariants are inferred by predicate abstraction over a set of a priori defined predicates, while [25] employs constraint solving over invariant templates. In [43] invariants are computed using input predicates and interpolation. Unlike these approaches, our project will not use a priori defined predicates or templates. User guidance is also not required in [27, 14], but invariants are derived using abstract interpretation over array indexes [27] and array segments [14]. However, these approaches can only infer universally quantified invariants. Our project, based on our previous works [30, 40], will provide fully automatic methods for generating quantified invariants, including those with quantifier alternations, which cannot be handled by other techniques.

(3) Developing efficient techniques for an automatic generation and optimization of interpolants, including those with quantifiers. Craig interpolation turned out to provide a powerful technique for verifying safety properties of software. The works of [34, 10] derive interpolants from resolution proofs in the ground (that is, quantifier-free) theory of linear arithmetic and uninterpreted functions. The approach described in [55] generates ground interpolants in the combined theory of arithmetic and uninterpreted functions using constraint

solving techniques over an a priori defined interpolant template. The method presented in [43] computes quantified interpolants from restricted FO proofs over scalars, arrays and uninterpreted functions. Unlike these approaches, our results will not be limited to reasoning in ground decidable theories, but use symbol elimination in FO theorem proving in conjunction with quantified theories. Our project builds upon our previous results [41, 33], and computes interpolants of different strength from proofs in FO logic with theories.
(4) Designing an efficient method for the timing analysis of programs. Several existing works make use of abstract interpretation to estimate the number of loop iterations, i.e. loop bounds; however, often loop bounds are assumed to be a priori given, in part, by the user – see e.g. [26]. In [22], an abstract interpretation based linear invariant generation method is used to derive linear bounds over each counter variable. Abstract interpretation is also used in [1] in conjunction with analyzing the non-deterministic behavior of multi-path loops. Contrary to these techniques, our project will derive complex algebraic upper bounds on the number of execution steps of loops by using symbol elimination in symbolic computation. Our project will significantly extend our previous results [7, 36] and advance the timing analysis of programs.
(5) Developing new algorithms for reasoning in FO logic and theories. Combining FO proving and theory reasoning is very hard. Most of the modern systems are based on two approaches to such reasoning: trigger-based (heuristic) ground instantiation of quantified axioms in SMT solvers [19, 54] and (incomplete) axiomatization of theories in FO provers [42]. A tighter integration is described in [17, 2]. Motivated by our experimental results [30, 32], we will integrate theory reasoning with superposition FO theorem proving, solving problems beyond the reach of other methods.

Section b. Methodology

Creating software that is correct by design or finding errors in software is a very hard problem. There are theoretical results showing undecidability of almost every problem related to program correctness. Despite these results, considerable progress has been achieved in the last years in the area of program analysis and verification. Industries running very large programs have started to routinely use methods and tools developed for this area. Such tools are increasingly using methods based on various kinds of reasoning, such as abstraction, model checking, theorem proving, SMT solving, or symbolic computation. Our project will use recent advances in symbolic computation and theorem proving, and design novel methods for reasoning-based program analysis.

Let us first motivate the work in this project on a small example. Consider the program given in Figure 1, written in a C-like syntax. The program fills an integer-valued array B by the positive values of a source array A added to the values of a function call f, and an integer-valued array C with the non-positive values of A. In addition, it computes the sum s of squares of the visited positions in A. A safety assertion, in FO logic (FOL), is specified at the end of the loop, using the `assert` construct. The program of Figure 1 is clearly safe as the assertion is satisfied when the loop is exited. However, to prove program safety we need additional loop properties, i.e. invariants, that hold at any loop iteration. It is not hard to derive that after any iteration k of the loop (assuming $0 \leq k \leq n$), the linear invariant relation $a = b + c$ holds. It is also not hard to argue that, upon exiting the loop, the value of a is n. However, such properties do not give us much information about the arrays A, B, C and the integer s. For proving program safety, we need to derive that each $B[0], \dots, B[b-1]$ is the sum of a strictly positive element in A and the value of f at the corresponding position of B. We also need to infer that s stores the sum of squares of the first n non-negative integers, corresponding to the visited positions in A. Formulating these properties in FOL yields the loop invariant:

```
$a := 0; b := 0; c := 0; s := 0;$
while $(a < n)$ do
  if $A[a] > 0$
    then $B[b] := A[a] + h(b); b := b + 1;$
    else $C[c] := A[a]; c := c + 1;$
  $a := a + 1; s := s + a * a;$
end do
assert$((\forall p)(0 \leq p < b \implies B[p] - h(p) > 0) \wedge$
            $6 * s = n * (n + 1) * (2 * n + 1))$
```

Figure 1: Motivating example.

$$(\forall p)(0 \leq p < b \implies (\exists q)(0 \leq q < a \wedge A[q] > 0 \wedge B[p] = A[q]+h(p))) \wedge 6*s = a*(a+1)*(2*a+1) \quad (1)$$

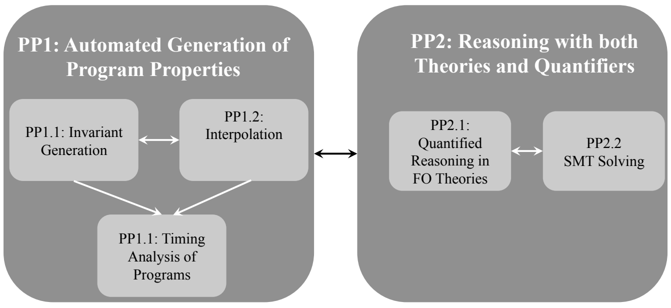
The above property requires quantifier alternations and polynomial arithmetic and can be used to prove the safety assertion of the program. This loop property in fact describes much of the intended behavior of the loop and can be used to analyze properties of programs in which this loop is embedded. Generating such loop invariants requires however reasoning in full FOL with theories, in our example in the FO theory of arrays, polynomial arithmetic and uninterpreted functions. Our project addresses this problem and is divided in two connected project parts (PPs):

(PP1) Automated generation of program properties;
(PP2) Reasoning with both theories and quantifiers.

The relation between the PPs is illustrated in Figure 2. We next describe each project part in detail.

(PP1) Automated Generation of Program Properties

Program analysis aims at generating and annotating program code with program properties, called program annotations, that can be used to prove safety and liveness properties of software. Analyzing such properties becomes an especially challenging task for programs with complex flow and, in particular, with loops or recursion. For such programs one needs additional program annotations, in the form of loop invariants, conditions on ranking functions, or Craig interpolants, that express properties to hold at intermediate program points. Providing such annotations manually requires a considerable amount of work by highly qualified personnel and often makes program analysis prohibitively expensive. Therefore, automation of the generation of program properties is invaluable in making industrial program analysis economically feasible.


PP1: Automated Generation of Program Properties
PP1.1: Invariant Generation
PP1.2: Interpolation
PP1.1: Timing Analysis of Programs
PP2: Reasoning with both Theories and Quantifiers
PP2.1: Quantified Reasoning in FO Theories
PP2.2 SMT Solving
Figure 2: Project Parts and Their Relations.

In [40] we introduced a new method, called *symbol elimination*, for automatic annotation of loops with their invariants. It is the first ever method to generate complex properties with quantifier alternations over arrays. Symbol elimination, unlike other methods, is completely automatic and requires no annotations or templates.

In a nutshell, symbol elimination is based on the following ideas. Suppose we have a program P with a set of variables V. The set V defines the language of P. We extend the language of P to a richer language V^+ by adding functions and predicates, such as loop counters. After that, we automatically generate a set $\mathcal{L}$ of FO properties of the program in the extended language V^+, by using techniques from symbolic computation and theorem proving. These properties are valid properties of the program, however they use the extended language V^+. Then we derive from $\mathcal{L}$ program properties in the original language P, thus "eliminating" the symbols in $V^+ \setminus V$. In [41] we also showed that symbol elimination and interpolation are two related methods of program analysis and verification, and gave a new algorithm for building interpolants from FO proof. We made the Vampire theorem prover into an interpolating FO prover [31] and designed methods for finding small interpolants [33]. In [32] we have shown that symbol elimination actually works well in practice: it produces high quality loop properties which in over 80% of test cases (involving industrial applications) could replace human-produced annotations. With the help of symbol elimination, generating arbitrarily quantified properties of programs using combinations of data structures can thus soon become a reality!

While effective, our initial results of symbol elimination [40, 31, 32] can only be used in the analysis of program loops with limited control flow, arithmetic, and data structures. For example, nested loops, unbounded data structures other than arrays, or arbitrary multiplications among program variables are not yet supported. *The work in (PP1) aims at developing new symbol elimination algorithms for the automated generation of auxiliary program properties for programs with complex arithmetic and unbounded data structures, such as arrays, lists, pointers, and heaps.* The use of unbounded data structures is especially important in the analysis of safety-critical applications, since these programs implement complex operations over memory allocations. The major benefit of (PP1) will be the fully automatic generation of FO invariants and interpolants with polynomial arithmetic, including those with alternations of quantifiers, which is not yet possible by other methods.

(PP1) will turn symbol elimination into a powerful tool for program analysis, as follows. We will design new symbol elimination methods for inferring polynomial and quantified invariants (PP1.1). Since recent results on Craig interpolation provide an alternative approach to invariant generation, we are also interested in developing FO interpolation algorithms such that the resulting interpolants will give better scalability for inferring system requirements (PP1.2). Finally, we aim at developing efficient techniques to derive bounds on program resources, such as time and memory, that can be used in the timing analysis of programs (PP1.3). Our workhorse for the experimental part of the research will be the Vampire theorem prover [42], to which the PI is actively contributing. The outcomes of (PP1) will turn Vampire into a tool that can be used not only for proving, but also for deriving program properties. In more detail, (PP1) brings the following contributions.

(PP1.1) Invariant Generation. Loop invariants form the "backbone" of correctness proofs of programs with loops or recursion, and are of utmost importance in the automatic analysis of software systems. In [40] we

$$\left\{\begin{array}{rl} a^{(k+1)} = & a^{(k)}+1 \\ s^{(k+1)} = & s^{(k)}+a^{(k)}*a^{(k)} \end{array}\right. \qquad \text{(i)}$$

$$\left\{\begin{array}{rl} a^{(k)} = & a^{(0)}+k \\ s^{(k)} = & s^{(0)}+\frac{k*(k+1)*(2*k+1)}{6} \end{array}\right. \qquad \text{(ii)}$$

$$\begin{array}{l} 6*s^{(k)} = \\ a^{(k)}*(a^{(k)}+1)*(2*a^{(k)}+1) \end{array} \qquad \text{(iii)}$$

Figure 3: Polynomial invariant generation using symbolic computation on Figure 1.

described a framework for automatically inferring array invariants without any user guidance, by introducing the symbol elimination method. The approach was ground-breaking in several respects. First, it was the first ever method able to generate quantified loop properties with quantifier alternations and even properties that are not inductive invariants. Moreover, no user guidance or loop annotations were used to generate such properties. While general and effective, symbol elimination is currently limited to programs with arrays and integers, restricting the loop structure to loops with nested conditionals. (PP1.1) will address the limitations of symbol elimination to invariant generation and advance the state-of-the-art in program analysis, as follows.

• We will design new approaches generating *polynomial invariants for loops with arbitrary polynomial assignments.* For doing so, we will use algebraic recurrence solving methods and apply symbol elimination in the variable elimination procedures of Gröbner basis computation [8] and quantifier elimination [12]. For deriving polynomial invariants, we will proceed as follows. We will describe the behavior of programs loops with polynomial arithmetic in terms of recurrence equations over program scalars and loop counters. The obtained recurrences might however express more general recurrences than the C-finite class of linear recurrences handled in our previous work [38]. For example, arbitrary multiplications of loop variables or the use of nested loops can respectively lead to the P-finite class of linear recurrences with polynomial coefficients and $\Pi\Sigma$-recurrences. Algorithmic methods are available for solving such recurrences in terms of hypergeometric sequences [57, 47], and the closed form solutions of these recurrence equations as functions of the loop counters can be derived, though sometimes might not exist. The computed closed forms of such recurrences might however not be polynomials in loop counters, but, in some cases, still could be handled in a "polynomial manner" by introducing extra variables as non-polynomial functions over loop counters. By computing polynomial relations among these extra variables, polynomial invariants over program variables will be inferred.

To illustrate the workflow proposed above, consider Figure 3. Figure 3(i) describes the system of recurrence equations corresponding to the updates over a and s in Figure 1, where $s^{(k)}$ and $a^{(k)}$ denote the values of s and a at the kth loop iteration of Figure 1. That is, program variables become functions of loop iterations k. The closed form solutions of Figure 3(i) is given in Figure 3(ii). After substituting the initial values of a and s, Figure 3(iii) shows a valid polynomial identity among the values of a and s at any loop iteration k. While Figure 3 shows the main steps of polynomial invariant generation in (PP1), note that Figure 3(i) describes a system of C-finite recurrences which can always be solved. As mentioned, (PP1) will focus on more general recurrences describing loops with arbitrary polynomial updates, for which more sophisticated methods than C-finite recurrence solving will be used. For example, the loop assignments $x:=x+1; y:=x*y$ would yield a C-finite recurrence in x and a P-finite recurrence in y, as the closed form solution of x is a polynomial expression in the loop counter and the recurrence equation of y is linear in y. (PP1) will address such, and more general loops and infer polynomial invariants in a fully automated way.

• We will develop new methods computing *quantified invariants for loops with complex arithmetic and unbounded data structures*, by using symbol elimination in the combination of symbolic computation and FO theorem proving. We will first design FO theories of various unbounded data types, including the theory of lists, pointers, and heaps and introduce extra predicates defining updates over these data types similarly to the array update predicates of [40]. For example, we may write $upd(B,p,x)$ to express that an array B was updated at position p by the value x. For our running example from Figure 1, $upd(B,p,x)$ is defined as:

$$upd(B,p,x) \iff 0 \leq k \leq n \wedge A^{(k)}[a^{(k)}] > 0 \wedge p = b^{(k)} \wedge x = A^{(k)}[a^{(k)}] + h(b^{(k)}),$$

expressing that k is a loop iteration value at which the array B was updated (the true-branch of the conditional of Figure 1 was visited). As before, $A^{(k)}$ denotes the value of the array A at the kth loop iteration (A is actually unchanged throughout the loop of Figure 1).

Using such update predicates, the changes made over the elements of an unbounded data structure can be captured precisely. We will then apply symbol elimination in superposition FO proving and eliminate symbols different than program variables, deriving quantified loop invariants, possibly with quantifier alternations (similarly to the invariant (1)). In [41] we introduced special simplification orderings to convert proof search into symbol eliminating proof search. We will exploit our results from [39] and design new kind of simplification orderings and literal selections used in FO theorem provers, with the purpose of guiding superposition proving towards efficient symbol elimination. In addition, reasoning in the FO theories of lists, pointers and heaps will be necessary for extending symbol elimination to derive quantified invariants. To this end, (PP1.1) will use results of (PP2) and help (PP2) by providing hard benchmarks for reasoning with both quantifiers and theories.

• As program loops may have many invariants, finding those invariants that can be used for proving software correctness is a challenging task. An open challenge in (PP1.1) is therefore to design *automated invariant generation methods that are complete.* By completeness we mean that if a program has an invariant of a certain form, our approach will find invariants implying them. We will use methods from the theory of polynomial ideals, characterize the class of polynomial invariants by polynomial ideals, and compute a minimal and unique representation of the polynomial invariant ideal. We will also extend consequence finding in FO provers by efficient generation of various classes of clause sets with eliminated symbols: for example, a minimized set of invariants to avoid redundant clauses that imply each other. However, checking wether a FO formula is implied by another FO formula is undecidable, and thus deriving a minimal set of FO program properties is in general undecidable too. In (PP1.1) we will therefore also develop techniques for fast removal of redundant program properties. We will also generate simple induction axioms for inferring minimized sets of inductive invariants. We also intend to generalize (PP1.2) to generate interpolants from which inductive invariants are inferred. Ultimately, we are interested to develop a theory for symbol elimination and consequence finding.

• Extending our previous work [40] to *handle programs with nested loops and complex arithmetic* is necessary for making symbol elimination applicable to a larger class of practical problems. We aim at using symbol elimination in the combination of symbolic computation and FO theorem proving. As shown by our initial results in [30], such an integration can yield loop invariants over vectors and matrices from numerical libraries, such as the Netlib repository (`www.netlib.org`). Further development of symbol elimination requires more complex kinds of loop analysis followed by or interleaved with theorem proving. For such analysis, in (PP1.1) we need to apply recurrence solving together with the FO interpolation results of (PP1.2), as well as with the best existing static analysis methods using, for example, abstract interpretation [27, 14].

• To make the results of (PP1.1) practically useful, we have to address various implementation issues in making FO theorem provers better suited for program analysis. We will *extend Vampire* to analyze programs with complex arithmetic and unbounded data types, and reason about program properties with both quantifiers and polynomial arithmetic. We will also use Vampire in conjunction with the Aligator software package developed by the PI in [37]. We will evaluate Vampire on examples taken from open-source libraries, for example from the Netlib repository, and from industrial applications. For the latter, we have informal collaboration agreements with verification teams at Microsoft Research, Intel, Saab and Ericsson. We will also test the quality of the generated invariants on the examples of the software verification competition - SV-COMP [4].

Milestones and Deliverables of (PP1.1):

- polynomial invariant generation methods for nested loops with polynomial assignments;
- quantified invariant generation methods for loops with unbounded data structures;
- FO theorem prover with algebraic reasoning and polynomial invariant generation;
- FO theorem prover with quantified invariant generation;
- case studies of polynomial and quantified invariant generation on large programs.

(PP1.2) Interpolation. Craig interpolation [15] provides an alternative approach to invariant generation, and hence program analysis – see e.g. [34, 43]. For example, a Craig interpolant after one loop unrolling of Figure 1 is $B[0] - h(0) > 0 \wedge s = 1$, describing a transition predicate of Figure 1 under the condition that the loop of Figure 1 is exited after one iteration. Taken multiple loop unrollings, a sequence of Craig interpolants is computed from which a quantified loop invariant might be inferred [43].

The key ingredient in theorem proving based interpolation is a so-called local proof, since local proofs admit efficient interpolation algorithms [34, 41]. Informally, in local proofs some symbols are colored in the red or blue colors and others are uncolored. Uncolored symbols are said to be grey. A local proof cannot contain an inference that uses both red and blue symbols. In other words, colors cannot be mixed within the same inference. In [41] we defined an algorithm for building interpolants as boolean combinations of conclusions of so-called symbol eliminating inferences of local proofs in any sound calculus. A symbol eliminating inference is an inference whose premises contain at least one colored non-grey symbol, while its conclusion only contains grey symbols. Unlike other interpolation techniques [34, 10], we are not limited to decidable theories for which interpolation algorithms are known. However, a consequence of this generality is that we do not guarantee finding interpolants even for decidable theories or for theories where interpolants are known to exist. If we provide a (possibly incomplete) axiomatization of a theory and produce proofs in this axiomatization, we are effectively dealing with standard FO proofs, which may be non-local. The objective of (PP1.2) is to develop new efficient algorithms for extracting interpolants from proofs in full FOL with theories, as follows.

• We will develop new algorithms *extracting interpolants from arbitrary, and not only local, FO superposition proofs*. (PP1.2) will therefore extend results of [33, 44], since [33] can handle non-local SMT proofs with only uninterpreted constants, and [44] rewrites non-local SMT proofs by using proof transformations specific to the set of SMT proof rules. Using the theorem proving engines of (PP2), we will design new changes in superposition algorithms for generating interpolants in the combination of various FO theories, such as arithmetic, unbounded data structures and uninterpreted function symbols.

• Since interpolants are used for proving and refining program properties, the efficiency of program analysis crucially depends to which extent nice (e.g., small or useful for some purpose) interpolants can be automatically generated. Various methods of *minimizing interpolants* will also be studied, possibly based on proof transformations and theory-specific proof localisations in FO theories. We will rely on our initial results from [33] and optimize interpolants with respect to several measures. A challenging task of (PP1.2) is to derive inductive interpolants from which inductive invariants can be inferred for (PP1).

• Craig interpolation has been generalized to so-called tree interpolants for their use in concurrent [23] and recursive [29] programs. In this context, dependencies between program paths are encoded using a tree data structure, where a tree node represents a formula valid at an intermediate program location. Proving that a sequence of executions is infeasible in this case reduces to iteratively computing interpolants for each tree node v, by considering interpolants of the children of v. As for a tree with n nodes one would need to compute n interpolants from n local proofs, a challenging task is to compute all n interpolants from exactly one FO local proof. Results of [24] address this problem but can only be applied to generate quantifier-free tree interpolants, using constraint solving in the quantifier-free theory of uninterpreted functions and linear arithmetic. Other theories, especially those with quantifiers are not treated in [24]. Another objective of (PP1.2) is therefore to *generate and optimize tree interpolants* in full FO logic with theories, from only one superposition proof. Our initial experiments in [6] show that FO interpolation over integer arithemtic and arrays can compute tree interpolants no other solver could so far derive.

• On the implementation side of (P1.2), we aim at *extending Vampire* with efficient interpolation algorithms for complex programs, and make Vampire work together with model checkers, such as CPAChecker [5]. Results of (PP1.2) will be evaluated on various benchmarks, including examples from the software verification competition - SV-COMP [4] and examples coming from our industrial collaborations. We will study how the quality of minimized interpolants effects the efficiency of interpolation-based model checking and invariant generation.

Milestones and Deliverables of (PP1.2):

- new methods of interpolation in FO theories using symbol elimination;
- an interpolating theorem prover for FO theories;
- case studies of interpolation on concrete programs of industrial relevance.

(PP1.3) Timing Analysis of Programs. When reasoning about correctness of systems, one needs to also ensure that system operations are executed in a timely fashion. The worst case execution time (WCET) analysis of programs is concerned with providing tight bounds for the execution time of system components. One of the main difficulties in WCET analysis comes from the presence of loops or recursive procedures, as WCET depends on the number of loop iterations (that is, loop bounds) or recursion depth. For example, when analyzing the execution time behavior of Figure 1, one needs to infer that the number of loop iterations is bounded by n. In [35, 36] we described how symbolic computation and automated reasoning can be used for a restricted, yet in practice quite general class of programs, and derived precise WCET values from the inferred loop bounds. However, our method is currently limited to multi-path loops that can be translated into single-path loops by SMT queries over arithmetic. As loops can be arbitrarily nested, in (PP1.3) we aim at developing new symbol elimination techniques for generating assertions over loop bounds, in order to derive tight, and possibly precise, WCET timing constraints.

• We will develop new methods for inferring tight *iteration bounds for program loops with arbitrary nestedness and polynomial assignments*. We will use abstraction techniques, such as [9], to refine and simplify the control flow of the program and derive recurrence equations characterizing the loop behavior. These recurrences can involve sum and product quantifiers over loop counters, and hence could be studied by means of $\Pi\Sigma$ recurrences [57]. Our project will explore such recurrences and compute tight loop iteration bounds by minimizing the closed form expressions over loop counters.

• We will use the SMT and FO theorem proving engines of (PP2) and reason about *loop iterations that depend on the content of unbounded data structures*. This way, loop bounds can be expressed as functions of scalar variables and elements of unbounded data structures.

• On the practical side, we intend to *extend Vampire* to derive and refine non-trivial loop bounds in FO theories. We also aim at using the results of Vampire in combination with existing WCET tools, for example with the r-TuBound toolchain to which the PI has already contributed [35].

Milestones and Deliverables of (PP1.3):

- automated methods for computing loop bounds for nested loops with polynomial assignments;
- new methods for deriving loop bounds over non-numeric data structures;
- case studies of loop bound computation for WCET analysis.

(PP2) Reasoning with both Theories and Quantifiers

The methods described in (PP1) work with symbolic representations of the set of program states, where sets of states are modeled using logical formulas. This allows one to analyze programs working with unbounded data types and so having an infinite set of states. Reasoning about such programs requires proving the validity of sentences in various FO theories, in which the programs and their properties can be formalized. For example, for proving safety of Figure 1 using the invariant (1), one needs to reason in the FO theory of arrays, polynomial arithmetic and uninterpreted functions. Without efficient decision procedures and theorem provers, it is hence impossible to build effective tools for program analysis, which is the main topic of our project.

Decision procedures over quantifier-free theories, implemented in SMT solvers – see e.g. [19], can process huge formulas, but SMT solvers are yet inefficient in handling quantifiers. On the other hand, FO theorem provers, see e.g. [42], are very efficient in working with quantifiers but weak in theory reasoning. Note that a large number of problems in program analysis (for example, reasoning about dynamic memory allocation) can best be expressed using both theories and quantifiers. Reasoning with quantifiers and theories is therefore regarded as probably the main obstacle in applications of reasoning to software analysis.

In (PP2) we will develop automated theorem proving methods and tools supporting reasoning with both quantifiers and theories, proving problems beyond the reach of current methods. We will design new theories and algorithms for combining SMT solving and FO theorem proving, investigate how SMT solvers and theorem provers can cooperate, and carry out industrial-strength case studies. We would like to go beyond just interfacing SMT solvers as black-boxes with FO theorem provers. Our work plan in (PP2) is therefore divided into (PP2.1) quantified reasoning in FO theories and (PP2.2) SMT solving. We present each of these subtasks in detail below. Our results will be implemented and tested using Vampire.

(PP2.1) Quantified Reasoning in FO Theories. Program components involve both bounded and unbounded data structures. Therefore, analyzing the intended behavior of a computer program comes with the challenging task of reasoning in the combination of FOL and various theories, such as arithmetic, arrays, and lists. Quantified reasoning in FO theories can be regarded as an important, challenging, and probably the hardest topic for the next two decades of research in automated reasoning. Developers of SMT solvers try to address this topic by extending SMT solving with quantifiers by adding ground instances of axioms.

The FO theorem proving community starts from the other direction. They use FO theorem provers to generate ground instances of a quantified theory problem, and derive satisfiability/unsatisfiability of such ground instances by using the superposition calculus. This approach of FO theorem proving is known as instantiation-based theorem proving [21], and it is refutationally complete. That is, if a quantified theory problem is unsatisfiable, then the method will eventually derive its unsatisfiability. Implementing the approach of [21], an extension to superposition FO theorem proving is presented in [49], where an SMT procedure for arithmetic and uninterpreted functions is used as a black-box to handle ground instances of a FO problem. Another approach to quantified reasoning in FO theories is described in [53], where problems in the decidable theory of Presburger arithmetic are treated. However, [53] cannot be applied to different (undecidable) FO theories. Unlike the cited works, in [40, 42] we proposed sound but incomplete reasoning in arbitrary FO theories, as follows. We add some subset of theory axioms to a FO theorem prover and use the superposition calculus to prove problems with both quantifiers and theories. While the approach of [40, 42] is incomplete, our experimental results showed good performances in proving interesting program properties.

Just recently, we studied superposition reasoning in the theory of data collections, such as arrays or sets, with extensionality axioms. We showed that the addition of new inference rules to superposition enables us solving problems no other theorem prover could solve far. Let us illustrate this result by considering the following property over set union and intersection:

$$(\forall X, Y, Z)(X \cap Y \subseteq Z \wedge Z \subseteq X \cup Y \implies (X \cup Y) \cap (\overline{X} \cup Z) = Y \cup Z) \qquad (2)$$

where X, Y, and Z denote set variables and $\overline{X}$ is the set complement of X. The above property is a valid property in the theory of sets and a logical consequence of the set theory axiomatization. What is interesting is that while proving such properties poses no problems to humans, especially not to mathematicians, it is very hard for FO theorem provers. None of the FO provers from the CASC theorem proving competition of last year [61] can solve it with one hour time limit. The reason for this lies in the treatment of the extensionality axiom $(\forall X, Y)((\forall e)(e \in X \iff e \in Y) \implies X = Y)$ of set theory, where e denotes a set element and $\in$ is the set membership predicate. This axiom is needed for proving (2), by translating set equality into a set membership property. However, independently of the ordering used by a FO prover, $X = Y$ will always be the smallest (positive) literal, and hence no superposition prover will select it. This is why FO provers are very inefficient on proving properties involving equality reasoning between data collections, as in (2). To overcome such inefficiency, we added a new inference rule to the superposition calculus which, under additional conditions, forces the selection of positive equality literals among data collection variables. Our initial experiments on hard set theory problems (see Table 1), as well as on the TPTP and SMTLib theorem proving benchmark suites [60, 3], show that such a change boosts the performance of FO provers, proving problems that no prover could prove so far. Our experiments therefore suggest that theory-specific reasoning in combination with FO superposition is the right way to go for proving in full FO theories.

In (PP2.1) we rely on this observation and follow the research started in our previous works [40, 42]. The work in (PP2.1) will target the development of new theories, methods and tools supporting efficient quantified reasoning in FO theories, as follows.

• One line of research in (PP2.1) will extend FO theorem provers with *sound but incomplete theory axiomatizations*. In particular, using testcases from (PP1), we will study necessary and sufficient sets of theory axioms from which interesting program properties can be derived in extensions of the superposition calculus. We will need to understand what are the best simplification rules and orderings used in superposition calculi and maybe also add some theory rules.

• We will design *new inference rules for superposition reasoning in FO theories*, with a special focus on the theory of arrays, lists, pointers and heaps. Examples from (PP1) will be used to understand the required logical formalism, the best ways of axiomatizing FO theories, and possible extensions of the superposition calculus, without radical changes in the underlining inference mechanism and implementations of superposition. Our preliminary experiments on proving properties about data collections show that new superposition rules can efficiently be added to FO provers, solving problems no theorem prover can solve.

• We will address the *instantiation-based theorem proving* framework of [21], and use (PP2.2) for proving ground instances of a FO problem in an incremental way. That is, proofs produced by (PP2.2) will be analyzed with the purpose of generating proof certificates. The proof certificates will be propagated back to the FO theorem prover, and new ground instances will be generated. However, understanding what parts of the proof are relevant is a non-trivial task, and requires a deeper understanding of how the SMT engine and the FO prover can work together. (PP2.1) will therefore be based on an empirically driven development of methods, resulting in improvements in algorithms and their use and combination. These improvements will be evaluated on examples coming from (PP1), as well on industrial benchmarks coming from our industrial collaborations.

• The most ambitious objective of (PP2.1) is to *combine FO superposition proving and SMT solving*. In [20] we already described how linear arithmetic reasoning can be used in FO theorem proving; nevertheless, our results addressed only quantifier-free linear real/rational arithmetic problems. We will extend our results in [20] to the full integration of FO reasoning and SMT solving in linear arithmetic, and address also other theories than linear arithmetic. Powerful heuristics on literal selection and variable orderings will be used to block redundant inferences. Results of (PP2.1) will be implemented in Vampire.

Milestones and Deliverables of (PP2.1):

- new methods for building-in theory reasoning in the superposition calculus;

#	BEAGLE	CVC4	E-KR-HYPER	E	IPROVER	MUSCADET	PRINCESS	VAMPIRE	**VAMPIRE^{EX}**	ZIPPER-POSITION
1					13.70	0.10	7.61		0.08	
2		41.54					8.22		0.02	
3									0.29	
4		30.24		1.38	1.47	0.65	9.45	0.24	0.07	
5		56.05		33.98	0.89	0.10	14.64		0.25	
6		54.40			0.29		10.97		0.25	
7									0.03	
8									0.08	
9									0.09	
10									0.09	
11									0.27	
12		50.52			0.58		14.66	0.40	0.25	
13		30.34		0.35	1.10	0.09	15.13	0.17	0.02	
14	7.88			0.07	2.44	0.09	8.09	0.03	0.07	10.59
15		32.15		1.55	13.80		8.04	0.15	0.03	
16									4.14	
17		30.94		24.31				0.02	0.09	0.44
18									1.08	
19									0.04	
20									0.25	
21									0.25	
22									1.76	
23									0.50	
24								0.26	0.42	
25									0.05	
26									0.10	
27				52.47	11.80		20.97		0.08	
28		34.32			11.80		37.05	0.72	0.31	
29		31.33		1.64	38.63			0.26	0.04	
30				27.54	3.32	0.11	11.53	0.07	0.08	23.30
31									0.27	
32									0.09	
33					23.28		21.00		0.01	
34	2.21	30.29		0.03	0.50	0.08	6.71	0.02	0.01	0.59
35		30.34		30.23	8.23		7.24	0.25	0.02	
36		44.77			1.50			21.01	0.03	
	2	13	0	11	16	7	15	13	**36**	4

Table 1: Runtimes in seconds of FO provers from the CASC theorem proving competition 2013 on some hard set theory problems with extensionality axiom reasoning. VAMPIRE refers to the Vampire version without the improved extensionaliy reasoning, whereas **VAMPIRE^{EX} is the new Vampire version that incorporates extensionality axiom reasoning as discussed on page 9**. Experiments were obtained using a GridEngine managed cluster system. Each run of a prover on a problem was assigned a dedicated 2.3 GHz core and 10 GB RAM with the time limit of 60 seconds. Empty cells corresponds to timeouts. The last row counts the number of solved problems.

- new methods for combining theory reasoning with instantiation-based and superposition theorem proving;
- new reasoning methods combining FO provers and SMT solvers;
- FO prover with reasoning with both theories and quantifiers;
- case studies on academic and industrial examples.

(PP2.2) SMT Solving. Modern theorem provers deciding the satisfiability of propositional formulas, called SAT solvers, are based on the DPLL procedure [16]. Motivated by success of SAT-solving, the DPLL procedure has been extended to the DPPL(T) method [48] to treat quantifier-free theories. Deciding such problems is known as an SMT problem and gave rise to the development of very efficient SMT solvers, see e.g. [19]. The goal of (PP2.2) is to develop an efficient SMT solver within a FO theorem prover, by combining decision procedures of various theories in the style of DPLL(T) and understanding how such an SMT solver can assist a FO theorem prover.

• We will first add *existing SMT algorithms* for the theory of integers, reals and arrays to Vampire, study theory behavior and understand the best ways of using them within a FO theorem prover. Our starting point will be our

Current Status - as of March 2014	Proposed Development through the Project
FO theorem prover	FO theorem prover
with analyzing a restricted class of programs	with **analyzing complex programs**
	with **complete polynomial** invariant generation and
with quantified invariant generation of non-nested loops	with **minimized quantified invariant generation** of **arbitrary loops**
with interpolation	with **optimized** interpolation and **resource bound analysis**
with some built-in data types and	with **rich collection of built-in data types and theories**
with real arithmetic	**with incremental SMT solving**
using superposition	**using both superposition and theory reasoning**

Table 2: Proposed development on Vampire.

initial SMT procedure for deciding linear real/rational arithmetic [20], which is already integrated in Vampire. We will also be interested in reasoning about new, less understood, theories, such as those of pointers and heaps, and use results of (PP1) and (PP2.1) to reason about ground properties in such theories.

• To address the problem of reasoning about problems with complex arithmetic, such as non-linear polynomial equalities and inequalities, in (PP2.2) we target the use of advanced *symbolic computation algorithms,* such as Gröbner basis computation [8] and cylindrical algebraic decomposition [12]. While these methods are powerful in theory, their efficiency in practice is limited to problems with a small number of variables and/or polynomials with low degrees. To make these techniques scale to real world problems, we will use additional heuristics for variable orderings and reduce the number of variables by extending the coefficient field to a ring of polynomials.

• We are also interested to *produce SMT proofs*. Analyzing these proofs, one may generate auxiliary formulas as interpolants for (PP1.2), or additional proof lemmas for (PP2.1). The results of (PP2.2) will yield a new SMT solver in Vampire, making an efficient FO prover with theories available to the research community.

Milestones and Deliverables of (PP2.2):

- new SMT methods for fragments of FO theories;
- proof-producing SMT solver in a FO prover.

Tool Development. Our project will provide automatic tool support for (PP1) and (PP2) by extending Vampire as presented in Figure 2. Our project will make Vampire a powerful automated reasoning tool able to perform program analysis, polynomial and quantified invariant generation, interpolation, resource bound analysis, and prove properties involving both quantifiers and theories. The Vampire executable will be made online from the early stages of the project, including its use for industrial and commercial purposes. This will allow us to involve others in testing our methods and tools on their benchmarks.

The PI of this project started her work on Vampire in 2009. Before the PI joined the development of Vampire, Vampire was a very efficient FO theorem prover but could not be easily used for program analysis. As a result of the PI's involvement, starting from 2009 Vampire was extended with new features, such as program analysis, symbol elimination, interpolation, and theory reasoning. These new features have significantly increased the number of Vampire users. For example, in 2010-2011 Vampire was downloaded more than 1,000 times, that is, over 10 times a week. Currently, the research on extending Vampire and its development is led by the PI in Gothenburg and by Prof. Andrei Voronkov in Manchester.

Further Considerations.

Interdisciplinarity. Although the project belongs to computer science and eventually aims at the implementation of systems, it can be regarded as truly interdisciplinary in the following respect. The project will use and design rigorous mathematical techniques in polynomial algebra, quantifier elimination, algorithmic combinatorics, and mathematical logic, to support systems engineering methods and tools. The project will thus encourage interactions with mathematicians and logicians, bridging together computer science, mathematics and logic. Our project will also help to derive program properties that can be used by computer security tools.

Strategies for Dissemination. The scientific results of the project will be disseminated through peer-reviewed publications in international journals; publications and presentations at the top international conferences and workshops in the area with peer-reviewed proceedings; development of world-leading tools used by researchers worldwide; and in participation in tool competitions in the area. Further, we intend to organize workshops and tutorials at top international conferences, and scientific seminars and summer schools at Chalmers. We will also offer undergraduate semester projects on topics of the project.

Risks. The goals of our project are highly ambitious, and so also risky. One significant source of risk arises from the fact that proving minimality of a set of quantified program properties is undecidable. We will use our best theoretical knowledge to come up with a large classification of classes of quantified properties and impose minimality criteria on these classes. We will then adjust theorem proving and symbol elimination to generate a minimal and possibly complete set of valid program properties. Combining the computationally expensive symbolic computation methods with first-order reasoning could also be harder than expected, and hence some parts of the program analysis task cannot be fully completed. However, even in this case it is likely that our project will open a new challenging research direction that others will follow, both in the symbolic computation and automated reasoning communites. Another risky aspect of our project comes from reasoning with both theories and quantifiers. Since the whole problem is highly intractable, we will target sound but incomplete theory reasoning, moving then towards more efficient ways of first-order theory reasoning. We are, however, confident that the project will give a major breakthrough in this area since in automated reasoning new methods can often solve many problems previously unsolvable by other methods.

International collaborations. We will benefit from the existing scientific collaborations of the PI with leading experts in program analysis and automated reasoning. We alphabetically list the researchers whom we will collaborate with: Dr. Nikolaj Bjørner, Microsoft Research; Prof. Armin Biere, JKU Linz; Prof. Thomas A. Henzinger, IST Austria; Dr. Zurab Khasidashvili, Intel; Prof. Jens Knoop, TU Vienna; Prof. Helmut Veith, TU Vienna; Prof. Andrei Voronkov, U. of Manchester.

Integration of the Project in the Host Institution. The PI of this project was appointed to Chalmers in April 2013, with a purpose of building an international competitive research group in formal methods. Topics of our project are in the center of interest of several researchers at Chalmers. We plan to collaborate with Prof. Wolfgang Ahrendt (Formal Methods group), Prof. Koen Claessen (Functional Programming group), and Prof. Andrei Sabelfeld (Computer Security group). Our project will thus encourage cross-institutional collaboration at Chalmers and benefit from the expertise of different groups. The methods of our project will complement the existing research at the university, and the tools designed in the project are likely to be used by other groups. We will also exploit the research facilities of the Chalmers Software Center and conduct industrial collaborations with Ericsson and Saab.

Career Development of the PI. The PI joined Chalmers as an associate professor in formal methods only recently, in April 2013. The ERC Starting Grant would support the PI in building up her own research group at Chalmers by providing funding for graduate students and postdoctoral researchers working under her supervision.

Project Parts / Personnel	**Year1**	**Year2**	**Year3**	**Year4**	**Year5**
PP1 – PhD Student 1					
PP1.1: Invariant generation	✓		✓	✓	
PP1.2: Interpolation	✓	✓		✓	✓
PP1.3: Timing analysis		✓	✓		✓
PP2 – Postdoc					
PP2.1: Reasoning in FO Theories	✓		✓	✓	✓
PP2.2: SMT solving	✓	✓	✓		

Figure 4: SYMCAR Work Plan.

Section c. Resources (including Project Costs)

(c.1) The Team. The SYMCAR project will be led by the PI and she will be involved in both project parts. The two project parts require however somewhat different background skills and expertise. The research in (PP1) concerns with both theoretical and practical issues, and provides sufficient material for one doctoral dissertations. We thus suggest the participation of one doctoral student in the project, working primarily in (PP1) under the supervision of the PI. Significant parts of (PP2) involve tool building and experimentation, with a large amount devoted also to software engineering, and requires already solid knowledge in program analysis, automated reasoning, and practical tool building. We therefore foresee a post-doctoral researcher for (PP2), who will take the lead on the work on Vampire under the guidance of the PI.

(c.1) Time and Work Plan. The project will last 5 years with the activities as indicated in Figure 4. The work plan is designed according to the dependencies among project parts, and as discussed below.

Year 1. The generation of quantified invariants and interpolants requires the development of theory and algorithms in (PP1.1) and (PP1.2). In addition, these tasks depend on FO reasoning with both quantifiers and reasoning. For this reason, (PP2.1) and (PP2.2) will start in the first year of the project, giving reasoning support to (PP1.1) and (PP1.2) from the very beginning of the project.

Year 2. Examples of (PP1.1) and (PP1.2) will help (PP2) to identify principal difficulties in combining SMT solving and FO theorem proving, and heuristics for SMT solving will be derived. The second year of (PP2) will therefore focus on (PP2.2). Results of (PP2.2) will help (PP1.2) to get better interpolants in various theories,

Cost Category			**Total in Euro**
		PI	459,757
		Senior Staff	0
	Personnel	Post docs	339,476
		Students	271,581
		Other	18,186
Direct Costs	*i. Total Direct costs for Personnel (in Euro)*		1,089,000
	Travel		74,000
	Equipment		0
	Other goods and services	Consumables (PCs)	18,000
		Publications (including Open Access fees), etc.	10,000
		Other (Audit fees)	9,000
	ii. Total Other Direct Costs (in Euro)		111,000
A – Total Direct Costs (i + ii) (in Euro)			1,200,000
B – Indirect Costs (overheads) 25% of Direct Costs (in Euro)			300,000
C1 – Subcontracting Costs (no overheads) (in Euro)			0
C2 – Other Direct Costs with no overheads (in Euro)			
Total Estimated Eligible Costs (A + B + C) (in Euro)			1,500,000
Total Requested EU Contribution (in Euro)			1,500,000
For the above cost table, please indicate the % of working time the PI dedicates to the project over the period of the grant:			**70%**

Figure 5: SYMCAR budget.

and guide (PP1.3) to derive tight loop bounds.
Year 3. (PP1.3) will be used to refine the WCET of programs. For such programs, invariant generation methods in (PP1.1) will be developed and further used in (PP1.3). (PP1.1) will also yield the combination of symbolic computation algorithms and FO theorem proving in (PP2.1) and (PP2.2).
Years 4-5. The work within years 4-5 crucially depends on years 1-3. We expect that dependencies among the subtasks of (PP1) and their relations to (PP2) will be the same as in years 1-2. This will however be not the case with (PP2). To support (PP1), new methods will be developed in (PP2.1) during years 4-5, whereas the work in (PP2.2) will be integrated with (PP2.1).

(c.2) Requested Funding. The project will be carried out at the Department of Computer Science and Engineering of Chalmers. We request funding as listed in Figure 5 and explained below. The personnel costs are calculated according to the Chalmers regulations. All costs are calculated with a yearly increase of 3%.

PI. The PI will dedicate at least 70% of her time to the project, for the following reasons. 20% of the PI's working time will be dedicated to teaching, which is important for this project and the PI's future. 10% of the PI's working time will go to other, already committed, projects. We therefore request funding covering 70% of the PI's salary.

One Postdoc. We request funding for one postdoc position for 5 years in total, working on (PP2).

One PhD Student. We request funding for 1 PhD student position for 5 years, working on (PP1).

Other Personnel. We request partial funding for a project administrator to help with reporting.

Travel Expenses. We require all together EUR 74,000 for travels. The travel costs are calculated with EUR 14,800 per year and split as follows. The estimation of these travel costs is based on our previous experience and includes travels, subsistence and conference fees as charged by main conferences in the area of the proposal. For each project year, we expect two conference attendances outside Europe (EUR 2,600 per attendee and conference, EUR 5,200 all together) and three conference attendances in Europe (EUR 1,600 per attendee and conference, EUR 4,800 all together). In addition, we request EUR 4,800 for inviting experts for short stays at Chalmers to collaborate with us on the project, as well as for visiting our collaboration partners.

Consumables (Laptops/PCs). We require three high-end laptops for team members in year 1, plus three replacement laptops in year 4. We need high-end laptops for programming and running our computationally expensive systems. All together, we ask for EUR 18,000 covering laptop costs.

Publication Costs. We request yearly EUR 2,000 for publications and books, all together EUR 10,000.

Other (Audit Fees). We finally require EUR 9,000 for audit certificate fees, according to the Chalmers policy.

Literature

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. Automated Reasoning*, 46(2):161–203, 2011.

[2] L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational Theorem Proving for Hierarchic First-Order Theories. *Appl. Algebra Eng. Commun. Comput.*, 5:193–212, 1994.

[3] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[4] D. Beyer. Competition on Software Verification - SV-COMP. In *TACAS*, pages 504–524, 2012.

[5] D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *CAV*, pages 184–190, 2011.

[6] R. Blanc, A. Gupta, L. Kovács, and B. Kragl. Tree Interpolation in Vampire. In *LPAR-19*, pages 173–181, 2013.

[7] R. Blanc, T. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *LPAR-16*, pages 103–118, 2010.

[8] B. Buchberger. An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. *J. Symbolic Computation*, 41(3-4):475–511, 2006.

[9] P. Cerny, T. Henzinger, and A. Radhakrishna. Quantitative Abstraction Refinement. In *POPL*, pages 115–128, 2013.

[10] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *TACAS*, pages 397–412, 2008.

[11] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, pages 52–71, 1981.

[12] G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. *ATFL*, pages 134–183, 1975.

[13] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[14] P. Cousot, R. Cousot, and F. Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *POPL*, pages 105–118, 2011.

[15] W. Craig. Three uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. Symbolic Logic*, 22(3):269–285, 1957.

[16] M. Davis, G. Logemann, and D.W. Loveland. A machine program for theorem-proving. *Communications of ACM*, 5(7):394–397, 1962.

[17] L. de Moura and N. Bjørner. Engineering DPLL(T) + Saturation. In *IJCAR*, pages 475–490, 2008.

[18] L. de Moura and G. Passmore. Computation in Real Closed Infinitesimal and Transcendental Extensions of the Rationals. In *CADE*, pages 178–192, 2013.

[19] L. M. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In *CEUR Workshop Proceedings*, 2008.

[20] I. Dragan, K. Korovin, L. Kovács, and A. Voronkov. Bound Propagation for Arithmetic Reasoning in Vampire. In *SYNASC*, 2013. To appear.

[21] H. Ganzinger and K. Korovin. Theory Instantiation. In *LPAR-13*, pages 497–511, 2006.

[22] S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *PLDI*, pages 292–304, 2010.

[23] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs. In *POPL*, pages 331–344, 2011.

[24] A. Gupta, C. Popeea, and A. Rybalchenko. Solving Recursion-Free Horn Clauses over LI+UIF. In *APLAS*, pages 188–203, 2011.

[25] A. Gupta and A. Rybalchenko. InvGen: An Efficient Invariant Generator. In *CAV*, pages 634–640, 2009.

[26] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *RTSS*, pages 57–66, 2006.

[27] N. Halbwachs and M. Peron. Discovering Properties about Arrays in Simple Programs. In *PLDI*, pages 339–348, 2008.

[28] G. Hamon, L. de Moura, and J. M. Rushby. Generating Efficient Test Sets with a Model Checker. In *SEFM*, pages 261–270, 2004.

[29] M. Heizmann, J. Hoenicke, and A. Podelski. Nested Interpolants. In *POPL*, pages 471–482, 2010.

[30] T. Henzinger, T. Hottelier, L. Kovács, and A. Voronkov. Invariant and Type Inference for Matrices. In *VMCAI*, pages 163–179, 2010.

[31] K. Hoder, L. Kovács, and A. Voronkov. Interpolation and Symbol Elimination in Vampire. In *IJCAR*, pages 188–195, 2010.

[32] K. Hoder, L. Kovács, and A. Voronkov. Case Studies on Invariant Generation Using a Saturation Theorem Prover. In *MICAI*, pages 1–15, 2011.

[33] K. Hoder, L. Kovács, and A. Voronkov. Playing in the Grey Area of Proofs. In *POPL*, pages 259–272, 2012.

[34] R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *TACAS*, pages 459–473, 2006.

[35] J. Knoop, L. Kovács, and J. Zwirchmayr. r-TuBound: Loop Bounds for WCET Analysis (Tool Paper). In *LPAR-18*, pages 435–444, 2012.

[36] J. Knoop, L. Kovács, and J. Zwirchmayr. WCET Squeezing: On-demand Feasibility Refinement for Proven Precise WCET-bounds. In *RTNS*, pages 161–170, 2013.

[37] L. Kovács. Aligator: A Mathematica Package for Invariant Generation. In *IJCAR*, pages 275–282, 2008.

[38] L. Kovács. Reasoning Algebraically About P-Solvable Loops. In *TACAS*, pages 249–264, 2008.

[39] L. Kovács, G. Moser, and A. Voronkov. On Transfinite Knuth-Bendix Orders. In *CADE*, pages 384–399, 2011.

[40] L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *FASE*, pages 470–485, 2009.

[41] L. Kovács and A. Voronkov. Interpolation and Symbol Elimination. In *CADE*, pages 199–213, 2009.

[42] L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.

[43] K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *TACAS*, pages 413–427, 2008.

[44] K. L. McMillan. Interpolants from Z3 Proofs. In *FMCAD*, pages 19–27, 2011.

[45] M. Mueller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *POPL*, pages 330–341, 2004.

[46] G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, 1980.

[47] I. Nemes and M. Petkovsek. RComp: A Mathematica Package for Computing with Recursive Sequences. *J. Symbolic Computation*, 20(5-6):745–753, 1995.

[48] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*). *J. ACM*, 53(6):937–977, 2006.

[49] V. Prevosto and U. Waldmann. SPASS+T. In *ESCoR*, pages 18–33, 2006.

[50] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[51] E. Rodriguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Invariants of Bounded Degree using Abstract Interpretation. *J. Science of Computer Programming*, 64(1):54–75, 2007.

[52] E. Rodriguez-Carbonell and D. Kapur. Generating All Polynomial Invariants in Simple Loops. *J. Symbolic Computation*, 42(4):443–476, 2007.

[53] P. Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *LPAR-15*, pages 274–289, 2008.

[54] P. Rümmer. E-Matching with Free Variables. In *LPAR-18*, pages 359–374, 2012.

[55] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *VMCAI*, pages 346–362, 2007.

[56] S. Sankaranaryanan, H. B. Sipma, and Z. Manna. Non-Linear Loop Invariant Generation using Groebner Bases. In *POPL*, pages 318–329, 2004.

[57] C. Schneider. Symbolic Summation with Single-Nested Sum Extensions. In *ISSAC*, pages 282–289, 2004.

[58] J. Sifakis. A Unified Approach for Studying the Properties of Transition Systems. *Theor. Comput. Sci.*, 18:227–258, 1982.

[59] S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *PLDI*, pages 223–234, 2009.

[60] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[61] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.