ENTRA

318337

Whole-Systems ENergy TRAnsparency

Analysis Based Verification and Debugging of Energy, Performance and Precision Properties

Deliverable number:	D3.3
Work package:	Analysis and Verification (WP3)
Delivery date:	1 October 2015 (36 months)
Actual date:	1 March 2016
Nature:	Prototype
Dissemination level:	PU
Lead beneficiary:	IMDEA Software Institute
Partners contributed:	Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited

Project funded by the European Union under the Seventh Framework Programme, FP7-ICT-2011-8 FET Proactive call.

Short description:

This deliverable presents techniques for the verification and debugging of energy, performance and precision properties, based on the comparison of analysis information against specifications. It also describes the final instantiations of the general resource analysis framework for inferring energy consumption, performance, precision, and other parameters such as data sizes. In addition, it reports on a prototype implementation of the analysis and verification techniques as well as on a demonstration.

The deliverable includes the following attachments:

- D3.3.1 [LBLGH16]: *Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks*. Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2016), 2016. arXiv:1601.02800.

- D3.3.2 [LGHK+15]: *Towards Energy Consumption Verification via Static Analysis*. Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2015), arXiv:1512.09369, 2015.

- D3.3.3 [GK14]: *Analysis and Transformation Tools for Constrained Horn Clause Verification*. Theory and Practice of Logic Programming, Vol. 14, Num. 4-5 (supplementary materials), pages 90–101, Cambridge University Press, 2014.

- D3.3.4 [KG15b]: *Tree automata-based refinement with application to Horn clause verification*. Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings, Lecture Notes in Computer Science, Vol. 8931, pages 209–226, Springer, 2015.

- D3.3.5: *Probabilistic Resource Analysis by Program Transformation*. Proc. of the Foundational and Practical Aspects of Resource Analysis, LNCS, Springer, 2015. To appear.

- D3.3.6 [GGP+15]: *Static analysis of energy consumption for LLVM IR programs*. Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2015, pages 12–21, ACM, 2015.

- D3.3.7 [KG15a]: *Constraint Specialisation in Horn Clause Verification*. Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015, pages 85–90, Association for Computing Machinery, 2015.

- D3.3.8 [RK15]: *Probabilistic Output Analysis by Program Manipulation*. Quantitative Aspects of Programming Languages, EPTCS, 2015.
- D3.3.9 [KGG15]: *Decomposition by tree dimension in Horn clause verification*. Proceedings of the Third International Workshop on Verification and Program Transformation (VPT'2015), EPTCS, 2015. To appear.
- D3.3.10 [KG14]: *Convex polyhedral abstractions, specialisation and property-based predicate splitting in Horn clause verification*. Proceedings of First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, July 2014.

Contents

1 Introduction

Static analysis, together with modeling, is the other key component of energy transparency, and, hence, for energy-aware software development. It infers information about energy consumed by programs without actually running them. As with energy models, analysis can be performed on program representations at different levels in the software stack, ranging from source code (in different programming languages) to intermediate compiler representations (such as LLVM IR [LA04]) or Instruction Set Architecture (ISA).

Static analysis at a given level consists of reasoning about the execution traces of a program at that level, in order to infer information (among other things) about how many times certain basic elements of the program will be executed. The role of the energy model is to provide information about the energy consumption of such basic elements; it is used by the analysis to infer information about energy consumption of higher-level entities such as procedures, functions, loops, blocks and the whole program.

Analysis can also be performed at a given software level using energy models for a lower level. Such a model needs to be mapped upwards to the higher level, as described in Deliverable D2.3 (attachment D2.3.3.). The information inferred by static analysis at one level can also be reflected upwards to a higher level using suitable mapping information.

Deliverable D3.1 described the general resource analysis framework we developed, which can be instantiated in many ways for energy consumption estimation, depending on the implementation of the main components of the framework (e.g., analysers and transformations to the internal representation), and the levels at which the analysis is performed and the energy models defined. D3.1 also reported on a preliminary instantiation that used energy models defined at the ISA level and performed the analysis at the same level (in attachment D3.1.1, whose final version was later published in [LKS+14]). Deliverable D3.2 described preliminary instantiations of the framework that performed the analysis at the LLVM IR level, using energy models defined at the ISA level.

In this deliverable, we report on the final versions of the different instantiations of the general analysis framework that we have performed and how the information inferred by the resulting analyzers is compared against energy-related specifications for verification and debugging (Section 10). Both the prototype analyzers described in Sections 2 and 3 are based on a well-developed approach in which recursive equations (cost relations) are extracted from a program, representing the cost of running the program in terms of its input [Weg75, Ros89, DLH90, DLGHL97, NMLH09, AAGP11, AAGP09].

These cost relations are converted to *closed-form*, i.e., without recurrences, by means of a solver. The analysis automatically infers an approximate upper (and lower) bound of the energy

consumed by programs compiled to ISA or to LLVM IR. An energy model defined at the ISA level is used; for LLVM IR analyses the model is propagated to the LLVM IR level via the mapping techniques described in Deliverable D2.3 (attachment D2.3.3.).

An alternative approach, not based on cost relations but on the application of well known techniques for Worst Case Execution Time (WCET), such as Implicit Path Enumeration Techniques (IPET), has been explored (Section 4). This approach is able to deal with multi-threaded embedded programs that use two common patterns: task farms and pipelines. Section 9 presents a more general analysis of multi-threaded programs aimed at discovering properties relevant to energy consumption and optimisation.

We have also explored a combination of static and dynamic (profiling-based) techniques for the inference of energy consumption (Section 6). The dynamic technique uses an evolutionary algorithm to determine bounds on the energy consumption of basic blocks. The static analysis framework is then used to combine the energy values obtained for the basic blocks according to the program control flow, and produce energy consumption bounds of the whole program.

Many applications allow certain levels of variability in the accuracy/precision of their computations (e.g., video and audio encoders, machine learning algorithms, Monte Carlo simulations, etc.) caused by the application of some energy saving techniques. For this reason, we have studied the relationship among energy, precision and performance properties in two different application areas using different techniques (Section 2.2).

It is important to note that the techniques and tools we have developed are language- and architecture-neutral. They have mainly been applied to the analysis of XC programs running on xCORE architectures (the project's proof of concept), by performing suitable instantiations of the general resource analysis framework. However, they have also been applied to other architectures and languages. For example, Deliverable D3.2 (Section 3.2) reported on a prototype implementation for the analysis of C programs running on an ARM Cortex-M3, using an energy model for LLVM IR for that platform.

Analysis/modelling trade-off. At the early stages of the project, our hypothesis was that the choice of level affects the accuracy of the energy models and the precision of the analysis in opposite ways: energy models at lower levels will be more precise than at higher levels while at lower levels more program structure and data structure information is lost, which often implies a corresponding loss of accuracy in the analysis. This hypothesis about the analysis/modelling level trade-off (and potential choices) is illustrated in Figure 1.

In ENTRA we have explored different points in this space of combinations of analysis and modelling. Our experimental results [LGK$^+$16] (Attachment D1.2.1 of deliverable D1.1), that compare the analysis at the LLVM IR and ISA levels, confirm that the expected trade-off exists,

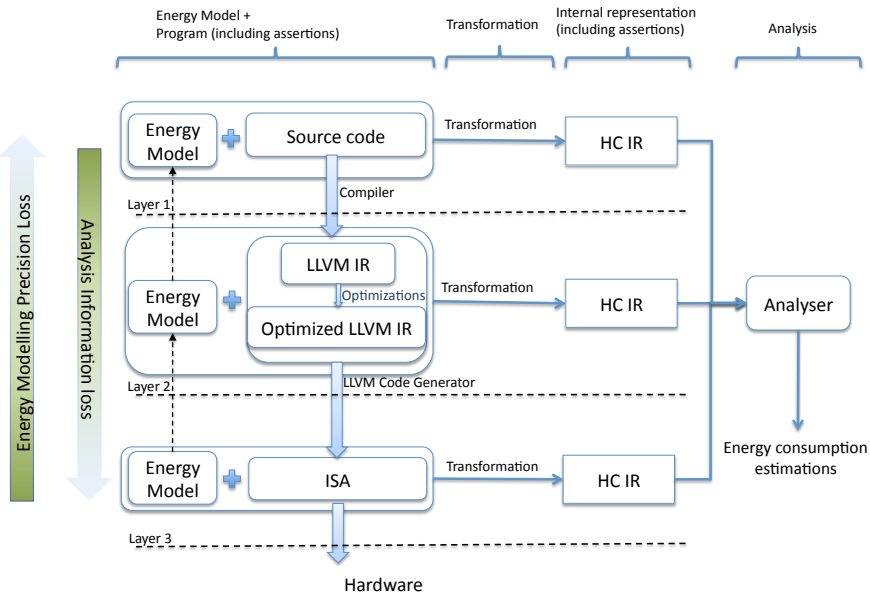


Figure 1: Analysis/modelling level trade-off and potential choices.

but also suggest that performing the static analysis at the LLVM IR level is a good compromise. LLVM IR is close enough to the source code level to preserve most of the program information needed by the static analysis, whilst close enough to the ISA level to allow the propagation of the ISA energy model up to the LLVM IR level without significant loss of accuracy.

In this deliverable we also report on the exploration of another combination of this space: an instantiation of the general resource analysis framework for the analysis at the (XC) source code level using models at the same level (Section 5).

Information inferred by analysis. The information inferred by the analyzers is guided by its final use: program optimisation, verification, helping energy-aware software developers to make design decisions, and so on. For example, they can infer probabilistic information (Section 8), which is useful for optimisation, or safe approximations, namely upper and lower bounds, on the energy consumed by the program or parts of it, which are needed for verification (Section 10). These approximations can be functions parametrised by the sizes of the input data and other hardware features such as clock frequency and voltage. The analyzers can then infer concrete values of the parameters that yield the worst-case energy consumption of the program or its parts.

Static energy profiling [HLGL+16] (Section 7) shows the distribution of energy usage over the parts of the code. This can be very useful to the developer, showing which parts of the pro-

gram are the most energy-critical, and helping design decision making. Some functions or blocks in the program are perhaps not particularly expensive in energy in themselves but are called many times. Such parts are natural targets for optimisation, since there a small improvement can yield important savings.

Note that the safety of bounds depends on energy models giving safe bounds for each instruction or basic block. This is a challenging problem, since the energy cost of executing an instruction depends on the operands. To obtain the worst-case consumption for an instruction we must therefore measure its execution with the operands that induce it. Addressing such a challenge, we have provided solutions in [PKME16, LBLGH16]. Safe bounds are vital for applying energy analysis to verifying or certifying energy consumption.

Prototype tools. The prototype tools developed as part of work package 3, contributing to this deliverable are available at *http://entraproject.eu/software-and-tools*. In addition, this document provides demonstrations of the use of the following prototype tools:

- Prototype implementation based on HC IR transformation and the CiaoPP system in two typical scenarios: analyzing the energy consumed by an XC program and verifying energy related specifications (Section 11).
- Prototype tool for the analysis of (XC) source code (Section 5).

2 Analysis based on HC IR Transformation and the CiaoPP Analyzer

In this Section we report on the final instantiation of the general resource usage framework of deliverable D1.1 that allows the analysis of XC programs both at the ISA and LLVM IR levels, using an energy model at the ISA level. A preliminary prototype implementation was reported in deliverable D3.2.

The analysis of an XC program at the ISA (resp. LLVM IR) level consists of: 1) generating the ISA (resp. LLVM IR) code for such program using the XMOS xcc compiler, 2) transforming the ISA (resp. LLVM IR) into an intermediate block representation based on Horn Clauses (HC IR), 3) using an existing, parametric resource usage analyzer (CiaoPP) to infer energy consumption functions (which depend on input data sizes) for each block in the Horn Clause representation, and 4) mapping the analysis results back to the XC source code.

The instantiation that performs the analysis at the ISA level, using an energy model at the same level, was reported in deliverable D3.1 (Attachment D3.1.1). A preliminary instantiation

that performs the analysis at the LLVM IR level, using the same energy model at the ISA level, together with a mapping tool to propagate the energy model from the ISA level up to the LLVM IR level, was described in Deliverable D3.2 (Attachment D3.2.4).

The final versions of such instantiations are described and published in [LGK+16, LKS+14]. In [LGK+16] (included in D1.2 as Attachment D1.2.1) we also report on an experimental study of the accuracy and efficiency of the instantiation that performs the analysis at the LLVM IR level, comparing it with the one performing the analysis at the ISA level. As mentioned before, from the experimental evaluation we can conclude that performing the static analysis at the LLVM IR level is a good compromise, since 1) LLVM IR is close enough to the source code level to preserve most of the program information needed by the static analysis, and 2) LLVM IR is close enough to the ISA level to allow the propagation of the ISA energy model up to the LLVM IR level without significant loss of accuracy.

2.1 Improvements to the resource usage analysis of CiaoPP

In the process of instantiating the general resource usage analysis framework, we have also identified some limitations or lack of functionality in the state-of-the-art analysis tools. Addressing this challenge, we have developed techniques to fill the identified gaps and implemented them within the CiaoPP system. These include:

1. Development of a novel general resource analysis for HC IR programs based on sized types [SLGH14] (also included as Attachment D3.2.3 in deliverable D3.2). Sized types are representations that incorporate structural (shape) information and allow expressing both lower and upper bounds on the size of a set of data structures and their subterms at any position and depth [SLGBH13]. The new analysis is fully based on the abstract interpretation technique, unlike the previous one present in CiaoPP, improving it in several ways and comparing well in power to state-of-the-art systems.

2. A better design and integration of the first prototype implementation of the resource analysis above into the CiaoPP system, in order to enhance its effectiveness, practicality, maintainability, extensibility and allow an easier combination with other supporting analysis present in the CiaoPP system. This will also eventually allow an evolution of the tool to the industrial application level.

3. Design and development of a component for solving recurrence relations, overcoming important limitations of existing resource analyses, and offering good quality features such as robustness and extensibility. This is crucial for obtaining a practical resource analysis tool

that can be used in real industrial applications. More concretely, the component offers a modular solver framework offering a well-defined interface to the analyzer, and providing all the algebraic-related services, the most important one being the inference of closed-form functions for recurrences. In turn, our solver communicates with a set of *external* solvers using a common interface that we have also defined. Our architecture has two main advantages. Firstly, it establishes a good and clear separation between the analysis and the mathematical machinery. Secondly, results from different external solvers can be combined in order to obtain better solutions. Finally, it makes it easier to add new external solvers in order to handle more classes of recurrences and solving other mathematical problems that can arise during analysis. The use of our new solver component has resulted in a significant improvement of the whole resource analysis.

4. Design and implementation of a specialized solver for a common class of recurrence relations that arise in the analysis of programs with accumulating parameters. This solver has been integrated into the modular framework mentioned above (see Section 2.1.2 for details).

2.1.1 A modular solver architecture

We have developed a new modular architecture for the resource analysis, defining a new component in charge of the algebraic operations of the analysis. In particular, this component is in charge of finding closed-form functions that over-approximate the recurrences set up during the resource analysis. This architecture is shown in Figure 2. The main modules are:

- *Solver_Strategies*: This module defines the common interface of the different strategies to solve recurrence relations.
- $Strat_i$: These are the different strategies to solve or over-approximate recurrences, using the services of the back-end solvers and the *classifier* in order to identify different characteristics of the recurrences.
- *Rec_Classifier*: It associates a *label* to each input recurrence relation that identifies the class of recurrence.
- *Solver_Utils*: Defines the common interface of the different *back-end solvers*.
- BS_i: These are the modules that implement the interface defined by **Solver_Utils**, connecting directly with the particular back-end solvers, such as Mathematica®, CiaoPP's built-in Solver, etc.

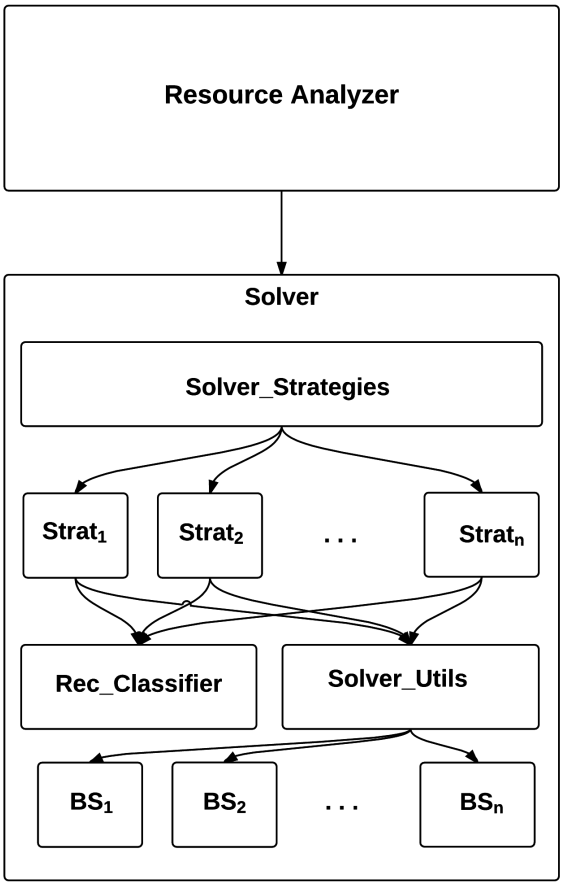


Figure 2: Architecture of the Modular Solver Framework.

The main advantage of this new solver component are the following:

- Being able to integrate easily new back-end solvers, such as existing Computer Algebra Systems (CAS, such as, e.g., Mathematica), existing specialized solvers (e.g., PURRS or PUBS), or even new specialized solvers for a limited set of special recurrences.

- Being able to combine easily the results from different back-end solvers.

Integrating new off-the-shelf systems to solve a larger set of recurrences is crucial for the extensibility and applicability of the whole resource analysis. The ability of combining results from different back-end solvers allows increasing their power, and analyzing (increasingly) more complex programs.

Algebraic Expression Syntax. In order to use different back-end solvers from our architecture, and, more importantly, combine the results coming from different back-end solvers, it is necessary to define a *common expression syntax* for the inputs and outputs of our solver component, delegating the responsibility of translating this syntax back and forth from the internal syntax of the back-end solvers, to each particular back-end solver interface. In Figure 3 we show this common syntax, both for arbitrary expressions and recurrence relations, which we call the *Algebraic Expression Syntax*.

⟨*Exp*⟩ ::= −⟨*Exp*⟩ | ⟨*BuiltinCall*⟩ | ⟨*UserDefCall*⟩ | ⟨*Exp*⟩ ⟨*BinOp*⟩ ⟨*Exp*⟩ | ⟨*Num*⟩ | ⟨*Var*⟩

⟨*BuiltinCall*⟩ ::= `fact (` ⟨*Exp*⟩ `)` | `lucasl (` ⟨*Exp*⟩ `)` | `fibo (` ⟨*Exp*⟩ `)` | `exp (` ⟨*Exp*⟩ `)`
| `max (` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `)` | `min (` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `)`
| `log (` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `)`
| `sum (` ⟨*Var*⟩ `,` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `)`
| `prod (` ⟨*Var*⟩ `,` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `)`

⟨*UserDefCall*⟩ ::= `fun (` ⟨*Name*⟩ `, [` ⟨*Exprs*⟩ `] )`

⟨*BinOp*⟩ ::= `+` | `-` | `*` | `**`

⟨*Exprs*⟩ ::= ϵ | ⟨*Exprs'*⟩

⟨*Exprs'*⟩ ::= ⟨*Expr*⟩ | ⟨*Expr*⟩ `,` ⟨*Exprs'*⟩

⟨*RecRels*⟩ ::= ⟨*RecRel*⟩ | ⟨*RecRel*⟩ `,` ⟨*RecRels*⟩

⟨*RecRel*⟩ ::= `equation (` ⟨*Name*⟩ `, [` ⟨*Vars*⟩ `] ,` ⟨*Exp*⟩ `, [` ⟨*Tests*⟩ `] )`

⟨*Vars*⟩ ::= ϵ | ⟨*Vars'*⟩

⟨*Vars'*⟩ ::= ⟨*Var*⟩ | ⟨*Var*⟩ `,` ⟨*Vars'*⟩

⟨*Tests*⟩ ::= ϵ | ⟨*Tests'*⟩

⟨*Tests'*⟩ ::= ⟨*Test*⟩ | ⟨*Test*⟩ `,` ⟨*Tests'*⟩

⟨*Test*⟩ ::= ⟨*Exp*⟩ ⟨*BinComp*⟩ ⟨*Exp*⟩

⟨*BinOp*⟩ ::= `>` | `<` | `=` | `>=` | `=<`

Figure 3: Algebraic Expression Syntax

Interface to Back-End Solvers. The common interface that the different *back-end_solvers* have to implement is the following (defined by *Solver_Utils*):

- `translate_from_common_syntax(+Exp, -TExp)`: Translates any expression or recurrence relation received as input, in the *common algebraic expression* syntax, to the specific syntax of the corresponding back-end solver.

- `translate_to_common_syntax(+TExp, -Exp)`: Translates expressions or recurrences from the syntax of the back-end solver to the common algebraic expression syntax.
- `simplify(+Exp, -Simp)`: Simplifies the expression or recurrence relation `Exp`. If the input is a recurrence relation, it simplifies the right hand side of the equations.
- `expand(+Exp, -Simp)`: Expands out products and positive integer powers in `Exp`.
- `greater_than(+A,+B),equal_to(+A,+B), ···` : Implement the comparison operators between algebraic expressions.
- `solve_rec_rel(+RecRel,+Bound,-ClosedForm)`: Tries to find an exact closed-form solution for `RecRel`, or a correct approximation with respect to `Bound`, whose value could be `upper`, if we want an upper bound for the recurrence, or `lower` if we want a lower bound.

The translations back and forth to the common expression syntax are performed from - `Solver_Utils`, by calling the corresponding predicates of the back-end solvers. Thus, from the point of view of a user of the solver, the solver receives and returns elements in the common syntax, abstracting away all the translation details.

2.1.2 Using ranking functions for upper-bounding special recurrences

In order to show the advantages of the new architecture proposed for dealing with recurrences, we present in this section how we can implement and integrate a specialized back-end solver for dealing with a special class of logic programs, that commonly appear when the logic program is a translation, more or less direct, of an imperative program with simple loops.

To illustrate the problem we are going to address, we will use a very naive running example. Let us consider an imperative function that calculates the addition of the first N natural numbers.

```
int sum(int N){
  int add = 0;
  for(int i=1;i=<N;i++)
     add += i;
  return add;
}
```

In order to analyze this program with our resource analysis, an automatic translation into a Horn clause representation is performed. Assume that after this step is completed, the loop for this function is translated into the following logic program (Horn clauses):

```
sum(I, N, A, A):-
    I > N.
sum(I, N, AI, AO):-
    I =< N,
    I1 is I + 1,
    AT is AI + I,
    sum(I1, N, AT, AO).
```

Assume also that *execution steps* is the resource to measure. Our resource analysis based on abstract interpretation will set up the following recurrence relation representing an upper bound on the resource usage of a call to that program (predicate):

```
equation(g, [I,N], fun(g,[I + 1,N])+1,[I =< N]),
equation(g, [I,N], 1                  ,[I > N]),
```

This recurrence relation is not in the proper form typically handled by a CAS. Mathematica, for example, cannot deal with it properly because it does not support the constraints associated with each equation, and, moreover, the second argument only appears in constraints. However, we can transform this recurrence into a common, one variable recurrence equation by performing some variable substitution and using the information present in the constraints. Let us define $Y = N - I + 1$. If we rewrite the previous recurrence in terms of Y, we obtain:

```
equation(g, [Y], fun(g,[Y - 1])+1,[Y >= 1]),
equation(g, [Y], 1              ,[Y < 1]),
```

which can be easily solved now by almost any recurrence solver. Its solution, Y, needs to be replaced by its definition, $N - I + 1$. This is a valid upper bound for the recurrence if $I \leq N$. If $I > N$ the result should be C, because the base case condition applies. For that reason, as a last step, we need to use the max operator in order to obtain a valid upper bound for all the possible values of I and N. Therefore, we obtain the following closed-form upper-bound expression for g:

$$max(C, N - I + 1) \tag{1}$$

We have just shown how to solve, in a very intuitive way, a small but tricky kind of recurrence relation. As programs with these characteristics are very common, it is important to automate

this technique. If we pay attention to the reasoning we just followed, we can notice that we looked for a variable replacement that transforms the recurrence into a form amenable for traditional solvers. That means that we need the recurrence in a decremental way, expressing the n-th term as a function of some i-th term, obtaining $i < n$. Therefore, what we need for the variable replacement is an expression that we know *decreases* in each step of the recurrence. The automatic termination community have studied in depth this particular kind of expressions for *loops* (we can see a recurrence as a loop), because they are instrumental for proving termination. They usually prove that there exists a function that maps the arguments of the loop to a *well-founded* partial order, such that this function decreases in each step. These kinds of functions are called *ranking functions* [Flo67]. In [PR04] a complete method for obtaining linear ranking functions is presented.

The use of ranking functions for finding closed-form upper bounds for recursive resource usage expressions (called *cost relations*) was presented in [AAGP11]. In such work, the authors first define a set of *evaluation trees* for a system of cost equations and a given initial call. Then, they try to find an upper bound on the number of nodes, both internal nodes and leaves. Each node of this kind has a *local cost* that needs to be multiplied by the corresponding bound on the number of nodes in order to obtain a closed-form expression. These local costs are also over-approximated by using linear programming techniques, and the result is finally obtained. This idea of over-approximating the number of nodes of any possible evaluation tree was previously given in [DLGHL97], but for inferring both lower and upper bounds for *divide-and-conquer* predicates.

In our case, we are going to limit this technique to simple cases, as the one that we showed above. By simple, we mean recurrences with one single recursive call, possibly multiplied by a constant coefficient, and no other equation call. Even more, we are going to support recurrences (under this technique) with mutually exclusive equations, and only one base case. As we are going to use the algorithm proposed in [PR04], we also need to limit ourselves to linear arithmetic expressions as input.

We follow an approach similar to the ones in [AAGP11, DLGHL97].

Definition 1 (Simple recurrence relation) *We call simple recurrence relation a recurrence of the form:*

$$\begin{aligned} f(\overline{u}) &= C, && \textit{if } \varphi(\overline{u}) \\ f(\overline{u}) &= K * f(\overline{u}') + g(\overline{u}), && \textit{if } \varphi'(\overline{u}) \end{aligned}$$

where

- $C, K \in \mathbb{Z}^+$ *are constants,*

- *g is an expression that does not contain any call to f or any other functions except built-ins,*
- *$\varphi(\overline{u}) \wedge \varphi'(\overline{u})$ is unsatisfiable (mutual exclusion),*
- *$\overline{u}$ is a sequence of arguments, and $\overline{u}'$ is a sequence of linear arithmetic expressions over $\overline{u}$, such that $|\overline{u}| = |\overline{u}'|$.*

Definition 2 (Ranking function for a simple recurrence relation) *A ranking function for the following simple recurrence relation:*

$$\begin{aligned} f(\overline{u}) &= C, && \textit{if } \varphi(\overline{u}) \\ f(\overline{u}) &= K * f(\overline{u}') + g(\overline{u}), && \textit{if } \varphi'(\overline{u}) \end{aligned}$$

is a function $h : \mathbb{Z}^{|\overline{u}|} \to \mathbb{Z}^+$ such that $\varphi'(\overline{u}) \models h(\overline{u}) > h(\overline{u}')$.

Now that we have already defined the class of recurrences we are going to handle, and what a ranking function is for those recurrences, we can show how to obtain a safe closed-form over-approximation for them. Given an initial input $\overline{u}_0$, let us first observe what is the result of applying the recursive case several times, before reaching the base case:

$$\begin{aligned} f(\overline{u}_0) = \\ Kf(\overline{u}_1) + g(\overline{u}_0) = \\ K(Kf(\overline{u}_2) + g(\overline{u}_1)) + g(\overline{u}_0) = K^2 f(\overline{u}_2) + Kg(\overline{u}_1) + g(\overline{u}_0) = \\ \cdots \\ K^{i-1} f(\overline{u}_{i-1}) + K^{i-2} g(\overline{u}_{i-2}) + \cdots + g(\overline{u}_0) = \\ K^i C + K^{i-1} g(\overline{u}_{i-1}) + \cdots + g(\overline{u}_0) \end{aligned}$$

As we can see, the last expression, where the base case is finally reached, is in closed form, with i being the number of applications of the recursive case. We can replace i by a ranking function $h(\overline{u})$ for f, because it is also an upper bound of the number of times a recursive case is applied (see for example [AAGP11]). We also need to find a general form for all the expressions $g(\overline{u}_j)$. In order to do that, and to ensure we are going to obtain a correct upper bound, we can replace each subexpression by the result of maximizing $g(\overline{u})$ under the constraints $\varphi'(\overline{u})$. Let M be the result of this operation. Finally, as we explained before, it is necessary to *wrap* the resulting expression with the max operator and the base case constant C as argument. In conclusion, we obtain the following closed-form expression:

$$\hat{f}(\overline{u}) = max(C, (\sum_{i=0}^{h(\overline{u})-1} K^i M) + K^{h(\overline{u})} C) \geq f(\overline{u}) \quad (2)$$

Coming back to our running example, we can see that the recurrence is a simple recurrence relation with $K = 1, C = 1$ and $\forall \overline{u} : g(\overline{u}) = 1$. If we apply the algorithm in [PR04] (interpreting the recurrence as a single loop), we obtain the ranking function $h(I, N) = N - I + 1$. Finally, by applying the formula we have just derived, we obtain the following upper-bound closed-form expression:

$$max(1, (N - I + 1) * 1 + 1 = max(1, N - I + 2) \quad (3)$$

which is a correct upper bound and is consistent with our first intuition (equation 1).

2.1.3 Implementation

We have implemented a prototype of the proposed architecture taking advantage of the module system of Ciao. We have also performed some *refactorizations* on the abstract interpretation-based resource analysis in order to integrate our prototype of the solver into it. In this prototype, we have developed a *strategy* called *chain*, as a proof of concept. This strategy simply tries to solve a recurrence relation by calling in sequence each available back-end solver. The first solution found is the one that is returned, obtained from one of the back-end solvers. A simple but significant improvement of this strategy would be to compare all of these results and get the maximum or minimum of them, depending on which kind of approximation we are looking for.

The implementation of the method showed in this section mainly requires a procedure for finding (linear) ranking functions. The Parma Polyhedra Library (PPL) provides numerical abstractions especially targeted at applications in the field of analysis and verification of complex systems, including implementations of methods for the synthesis of linear ranking functions [BRZH02, BHZ08]. Thus, we have integrated PPL as a back-end solver in our proposed architecture, implementing the operation `solve_rec_rel` in such a way that it: (a) identifies the different parts of the recurrence relation; (b) obtains a ranking function for the given recurrence; and (c) sets up the closed-form expression of Equation (2), using the ranking function obtained in (b) and the components of the recurrence obtained in (a). We also require an operation that performs the maximization of an expression under a set of constraints. Fortunately, we can use any CAS (in our case Mathematica), or even PPL to perform such operation.

2.2 Relating accuracy and energy consumption properties

Many applications allow certain levels of variability in the accuracy of their computations (e.g., video and audio encoders, machine learning algorithms, Monte Carlo simulations, etc.). This variability in accuracy depends on the nature of such applications in the sense that certain applications may allow, e.g., certain computations to be skipped, data representation to be varied, or different program paths (strategies) to be chosen based on given flags. This makes the relationship between accuracy and energy quite application-specific. As long as the distortion in the output due to variability in the accuracy is within user-defined acceptable levels, the variability can be traded off with performance/energy.

For this reason, we have studied the relationship among energy, precision, and performance in two different application areas using different techniques:

Firstly, in [BLLG15] we have studied the variability in accuracy stemming from the skipping of certain computations using loop perforation [SMR11]. The loop perforation technique transforms loops to execute only a subset of their iterations. This work addresses the problem of energy-efficient scheduling and allocation of tasks in multicore environments, where the tasks can permit certain loss in accuracy of either final or intermediate results, while still providing the proper functionality. We used the CiaoPP analyzer to estimate the energy of the different tasks. The proposed scheduler, enhanced with loop perforation, was shown to improve on the previous one by achieving significant energy savings (31 % on average) for acceptable levels of accuracy loss.

Secondly, we have applied a technique consisting of the variation of the number of bits used for the representation of numeric (integer) operands on a signal filter application (a case study of WP6). The results of our study are shown in Table 1, where we compare the 32-bit version of the *Biquad* benchmark against another version using 16-bit data representation. These two versions of the *Biquad* benchmark have been run for a number of different inputs and the actual energy consumption (measured on the hardware) of the two versions has been compared.

N	**HW (*nJ*)**		**32-bit/16-bit**
	32-bit	**16-bit**	
5	872	741	1.17
7	1190	997	1.19
10	1670	1426	1.16
14	2293	2004	1.14

Table 1: Energy consumption (*nJ*) measurements of *Biquad* for 32- and 16-bit versions.

Column **N** represents the number of *BANKS* for which the *Biquad* filter is applied to a signal. Columns **32-bit** and **16-bit** represent the two versions of the *Biquad* filter where the data (signal and coefficients) are represented using 32 and 16 bits respectively. The last column shows the energy gains (although paying the price in accuracy loss) by using only 16-bit data representation, as the ratio of the 32-bit and 16-bit versions.

The CiaoPP analyzer can infer different energy functions for the two versions of the *Biquad* filter, relating accuracy with energy.

3 Direct Analysis of LLVM IR

As already mentioned, LLVM IR offers a good trade-off between analyzability and accuracy. By contrast to the instantiation of the general analysis framework based on HC IR transformation used in the CiaoPP analyzer, we have also experimented with an approach operating directly on the LLVM IR representation. This means that no transformation to a different representation is necessary; instead, the analysis can be applied directly to the LLVM IR blocks. Such approach is described in [GGP+15] (included this document as attachment D3.3.6). Though relying on similar analysis techniques, the approach can be integrated more directly in the LLVM toolchain and is in principle applicable to any language targeting this toolchain. The approach can use the same LLVM IR energy model as the one applied in [LGK+16].

4 WCET-inspired Energy Consumption Static Analysis

Since the underlying challenges of analysing the timing and energy consumption behaviour of a program are quite similar, in [GKE15], we have applied well known WCET techniques to retrieve energy consumption estimations. One of the most popular WCET techniques is IPET [LM97], which retrieves the worst case control flow path of programs based on a cost model that assigns a timing cost to each CFG basic block. We have replaced the timing cost model by an ISA energy model given in [KE15b]. In the absence of architectural performance enhancing features, such as caches, this technique can provide safe upper bounds for timing. Through our experimental evaluation we have demonstrated that this is not the case for energy, as energy consumption, in contrast to time, is data sensitive, i.e., the energy cost of instructions varies depending on the operands used.

In order to explore the value and limits of applying IPET for energy consumption estimations, we have also extended the analysis to the LLVM IR level, using the LLVM IR energy characterization given by the mapping technique mentioned above (and described in Deliverable

D2.3). Furthermore, we have extended the energy consumption analysis to multi-threaded embedded programs from two commonly used concurrency patterns: task farms and pipelines. The experimental evaluation, on a set of mainly industrial programs, demonstrates that, although the energy bounds retrieved cannot be considered safe, they can still provide valuable information for energy aware development, especially in the absence of widely accessible energy monitoring of software.

5 Energy Analysis of Source Code

In Deliverable D3.1 [LG13] we presented a general framework in which energy analysis could be performed at various levels: source code (XC program); intermediate code (such as LLVM IR); machine instructions (ISA). In previous articles and deliverables we have described energy analyses at LLVM IR and ISA levels along with the associated energy model [LKS+14, LGK+16, GGP+15, GKE15]. In this section we complete the picture by describing how we can perform energy analysis at the source level and describe a prototype source code analysis tool.

To achieve analyses at different levels, there are two associated mappings in opposite directions: firstly, the meaning of the program derives from the source code and is mapped *downwards* (by compiler translations) into LLVM IR and ISA code respectively. Secondly, the energy model is derived from the lowest level and is mapped *upwards* to the intermediate and source code.

For source code analysis there are two main requirements.

- There is an energy model for source code constructs. This is obtained by mapping a lower-level energy model upwards to source code.
- The analysis is based directly on the semantics of the source code rather than on a translation to intermediate code.

The main advantage of source code analysis is that the analysis results are easier to interpret by the developer. The potential disadvantage is that the energy model may be less precise, since the actual energy consumed depends in many cases on exactly how the source code is translated into machine instructions and how the processor handles them.

We now describe how we obtain the source code energy model, and then how we perform analysis at the source level.

5.1 Derivation of a source code energy model

The energy model for source code is derived from the energy model for its LLVM IR representation, which is turn is derived from the ISA energy model. Previously we have presented a

procedure for mapping the ISA model to LLVM IR [GKE15] and demonstrated its usefulness in analysis of LLVM IR [GGP+15, LGK+16].

Obtaining the control flow graph for the source program. Given a source XC program, we compute its control flow graph (CFG) (or rather, a set of CFGs, one CFG for each procedure in the program). The CFG is obtained directly from the abstract syntax tree (AST) of the XC program, which is output by a specially modified version of the XC compiler toolchain [xTI]. We use an attribute grammar to "thread" the AST; each statement node in the AST has an inherited *next* attribute whose value is some other statement node, while each boolean expression has two attributes *true* and *false* whose values are the statement nodes for the true and false control flow. Compound statements have an *edge* attribute whose value is the sub-statement or expression to which control goes; for instance the *edge* attribute for an "if" statement is its boolean expression test. For non-compound statements such as assignment statements, the *edge* attribute evaluates to the *next* attribute's value.

Figure 4 shows the attribute evaluation rules for the main statement types, for attributes *next*, *true*, *false* and *edge*. This approach can also handle the *break* statement (which is not included in Figure 4), which requires only an extra attribute whose value is the *next* attribute of the innermost enclosing loop for each statement; this is the statement to which the *break* statement diverts.

The attributes *edge*, *true* and *false* in the threaded AST induce a directed graph (a set of connected graphs), where the nodes are statement nodes and boolean expressions, and the edges are pairs of the form $(S, S.edge)$, $(B, B.true)$ and $(B, B.false)$, where S and B are statement and boolean expression nodes respectively in the CFG. Each procedure entry point is a start node. We do not at this stage represent function call-return edges in the graph.

Figure 5 shows a screenshot from the prototype tool, with the input XC program on the left, and the CFG (one for each of the two procedures) on the right.

The basic blocks are then obtained straightforwardly from the CFG. A basic block is a maximal sequence of connected nodes where every node in the sequence apart from possibly the first and the last has exactly one in-edge and one out-edge. (Note: a slight modification of this procedure was needed to ensure correspondence with LLVM IR basic blocks; namely, the boolean guarding a loop is included both in the block leading to the loop and in the final block of the loop itself.)

Energy mapping between basic blocks at different levels. The translation of an LLVM instruction to ISA is highly dependent on context. Therefore there is no general energy mapping between ISA and LLVM IR *instructions*. The energy mapping is rather based on *basic blocks*; thus the mapping is program-specific. Given an LLVM program and its corresponding ISA pro-

AST Grammar Rules	Attribute Rules
If → if E then S_1 else S_2	E.true := S_1
	E.false := S_2
	S_1.next := If.next
	S_2.next := If.next
	If.edge := E
While → while E do S_1	E.true := S_1
	E.false := While.next
	S_1.next := While
	While.edge := E
StatementList → S_1;...;S_n	S_j.next = S_{j+1} (j = 1 to n-1)
	S_n.next := StatementList.next
	StatementList .edge := S_1
S → StatementList | If | While | Call | Assign	
	StatementList.next := S.next
	If.next := S.next
	While.next := S.next
	Call.next := S.next
	Assign.next := S.next
Assign → Id = E	Assign.edge := Assign.next
Call → Id (E_1,...,E_n)	Call.edge := Call.next

Figure 4: Attributes for AST threading

gram, there is a fairly stable mapping between the basic blocks in the two programs. That is, the flow of control is the same in the two programs and this implies that the basic blocks are in correspondence. Since the energy of a basic block at the ISA level can be computed from the energy model of the instructions in that block, the same energy can be assigned to the corresponding LLVM IR basic block.

Mapping the LLVM IR energy model. In [GKE15] a tool is described to propagate ISA-level energy information up to the LLVM IR level and obtain *energy values for LLVM IR blocks*. Blocks at both LLVM IR and ISA levels are identified mainly by their start and end source line numbers, which are also preserved in the source-level CFG as shown in Figure 5. We therefore can assign the energy values from an LLVM IR block to the corresponding block in the source code, by the following simple procedure. Figure 6 shows an example of the output of the tool [GKE15] for one block, highlighting the start and end numbers and the energy for the block derived from the corresponding ISA statements.

- Identify the start and end line number of an LLVM IR basic block, say b.
- Find the source block having the same start and end line numbers as b, and assign to that

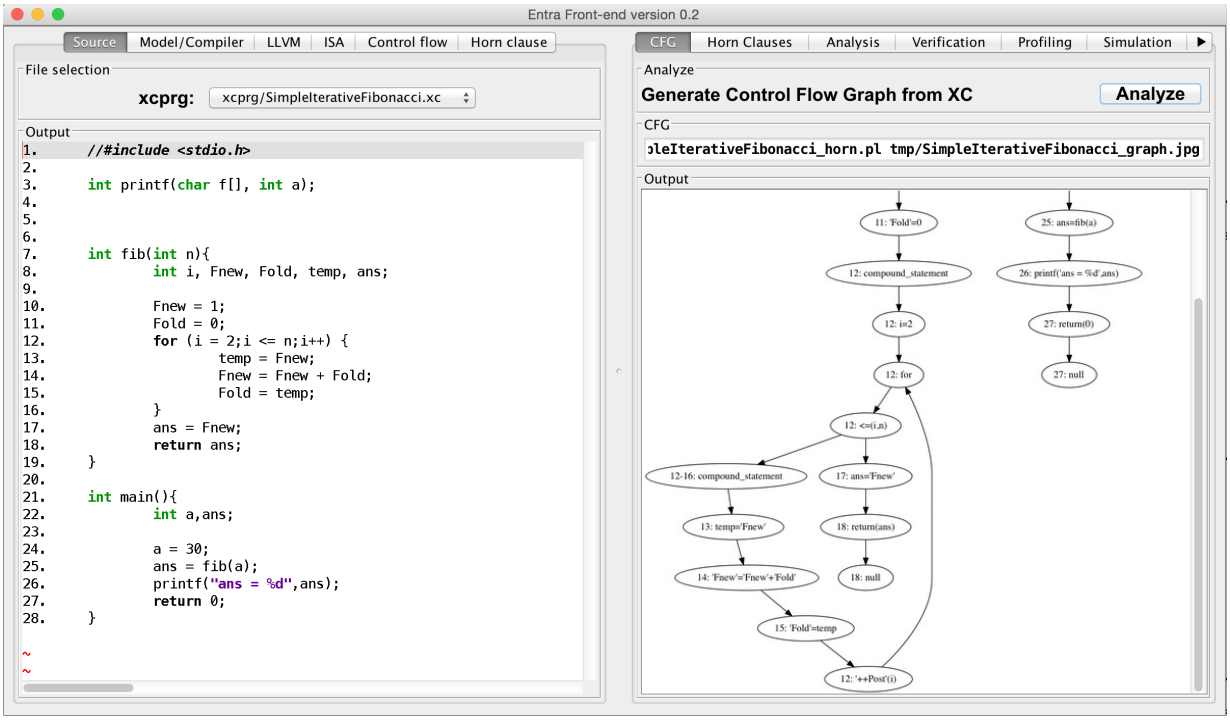


Figure 5: CFG derived from an XC program

block the energy of b.

This simple scheme does depend on the source code being split over lines in a reasonable way. It is possible after all to write a program on one line! Apart from this, there can be tricky points. For example line 12 in the program shown in Figure 5 is `for (i=2; i<=n; i++)` and this contains statements from different blocks. However the combination of start and end line number is usually sufficient to identify each block uniquely.

5.2 Analysis of source code

The analysis of source code follows the same approach adopted in the ENTRA project as for analysis of LLVM IR and ISA. This involves the following steps:

- Translation of the flow graph to Horn clauses.
- Application of the general resource analysis framework.
- Translation and reporting of analysis results back to the source code level.

```
{
 "lineEnds": 12,
 "name": "fib_LoopBody",
 "mapping": [
  {
   "LLVMIRIns": "%i.0 = phi i32 [ %postinc",
   "ISAMap": []
  },
  ... rest of block instructions omitted
  {
   "LLVMIRIns": "%relopcmp9 = icmp sgt i32 %postinc",
   "ISAMap": ["lss_3r"]
  },
  {
   "LLVMIRIns": "br i1 %relopcmp9",
   "ISAMap": ["brbf_ru6"]
  }
 ],
 "lineStarts": 13,
 "energy": 2.5907919155876703E-9
}
```


Figure 6: Example of an LLVM IR block energy information output by mapping tool [GKE15]

Translating the flow graph to Horn clauses. For each statement node in the AST we identify the set of variables in scope at that point. We assume, in order to simplify the explanation, that procedures do not have free variables (i.e. all variables in procedures are either locally declared or parameters), though this restriction is not essential. An attribute grammar defining inherited attribute *vars* and synthesised attribute *newvars* for each statement and boolean expression is given in Figure 8.

For statement node n, let us denote the value of the *vars* attribute for n as X_n, a tuple of variables (we assume that the variables are listed in some fixed order). We denote by X'_n the renamed tuple where each variable x in X_n is renamed as x' in X'_n.

For each edge in the CFG, say (n, m), we define a Horn clause of the form

$$p_m(X_m) \leftarrow \phi_{n,m}(X_m, X'_n), p_n(X'_n)$$

where the constraint $\phi_{n,m}(X_m, X'_n)$ relates the state of the variables at n and m and expresses the effect of executing n. The clause can be read as follows: if the point just before statement n can be reached with variable values given by the tuple X'_n, and $\phi_{n,m}(X_m, X'_n)$ holds, then statement m can be reached with variable values X_m. There are three cases for the constraint $\phi_{n,m}(X_m, X'_n)$.

1. If n is a boolean expression E then $\phi_{n,m}(X_m, X'_n)$ is $E', X_m = X'_n$.

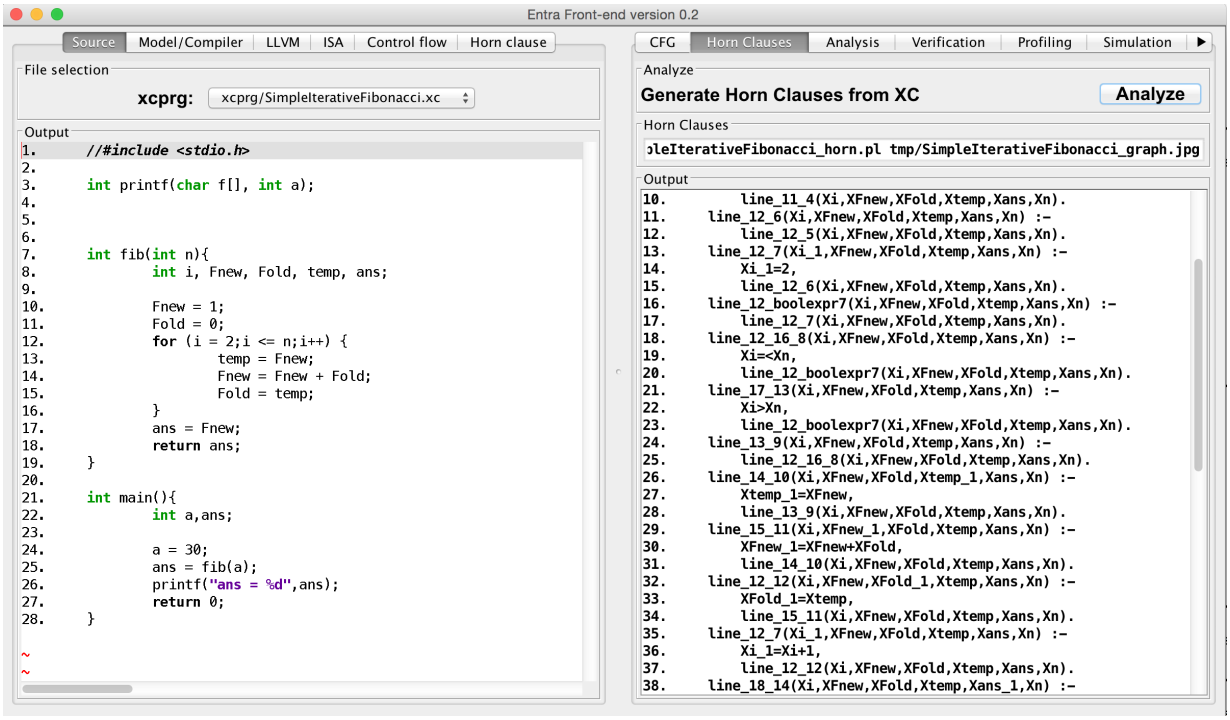


Figure 7: Horn clauses derived from an XC program

2. If n is an assignment $x = E$ then $\phi_{n,m}(X_m, X'_n)$ is $x = E', X_n \setminus \{x\} = X'_n \setminus \{x'\}$.

3. Otherwise, $\phi_{n,m}(X_m, X'_n)$ is $X_m = X'_n$.

The renaming of variables has an equivalent effect to SSA form, but "phi-functions" are not needed, since they are implicitly realised by the Horn clause semantics. Figure 7 shows a section of the Horn clauses derived from the running example program. Note that the line numbers are encoded in the predicate name; a predicate whose name is `line_X_Y` is on line `X`.

Analysis of the Horn clause representation. The resource analysis tools based on Horn clauses were previously applied to LLVM IR and ISA [LGK+16]. The tools derive an invariant on the energy consumption at specified program points. These invariants relate the energy consumed at a program point with the values of the variables in the state. For the *fib* function in the running example, and the energy assignments derived from the LLVM IR blocks as explained above, the energy consumed from the procedure entry to the return statement is derived to be the following.

$$energy = \mathtt{i} * 2.5907919155876703 * 10^{-9} - 8.235050400802908 * 10^{-10}\ Joule.$$

AST Grammar Rules	Semantic Rules
If → if E then S_1 else S_2	E.vars := If.vars S_1.vars := If.vars S_2.vars := If.vars If.newvars = { }
While → while E do S_1	E.vars := While.vars S_1.vars := While.vars While.newvars = { }
StatementList → {S_1,...,S_n}	S_j.vars = S_{j-1}.vars ∪ S_j.newvars (j = 2 to n) S_1.vars := StatementList.vars StatementList.newvars = { }
S → Decl | StatementList | If | While | Call | Assign	StatementList.vars := S.vars If.vars := S.vars While.vars := S.vars Call.vars := S.vars Assign.vars := S.vars S.newvars := Decl.newvars *// { } for other cases*
Decl → declare Type var	Decl.newvars = {var}

Figure 8: Attributes for statement variables

The invariant relates energy to the variable `i`, which has the same value as `n+1` at the return point for all values $\texttt{n} \geq 2$. For all $\texttt{n} < 2$, $\texttt{i} = 2$ and the energy consumption is constant, since the loop is not entered. Note also that if $\texttt{n} \geq 2$, then the loop is traversed only $\texttt{n} - 1$ times.

6 Combining Static and Dynamic Analysis Techniques

We have proposed an approach for inferring parametric upper and lower bounds on the energy consumption of a program using a combination of static and dynamic (i.e., profiling-based) techniques [LBLGH16]. The dynamic technique, based on an evolutionary algorithm, is used to determine the maximal / minimal energy consumption of each basic block. Such blocks contain multiple instructions, which allows this phase to take into account inter-instruction dependencies. Since such basic blocks are branchless, the evolutionary algorithm approach is more practical and efficient and the technique infers energy values that are accurate since no control flow-related variations occur. An instantiation of the static analysis framework based on the CiaoPP system (similar to the one described in Section 2) is then used to combine the energy values obtained for the blocks according to the program control flow, and produce energy consumption bounds of the whole program. We also carried out an experimental evaluation to validate the upper and lower bounds on a set of benchmarks. The results support our hypothesis that the bounds inferred in this way are indeed safe and quite accurate, and the technique practical.

7 Parametric Static Profiling of Energy Consumption

The goal of automatic program resource analysis is to discover the resources a program uses (in our case energy) as a function of the size of the input data or other environmental parameters of the program, without actually executing the program. Previous work on this topic, mainly for inferring asymptotic time complexity bounds, goes back to the 1970s. Recent research has adapted these techniques for inferring energy bounds and other resources and these form the basis for ENTRA prototype tools for energy analysis of programs.

However, as part of the energy-aware software development process investigated in the ENTRA project, it is clear that analysis of the overall energy consumption of a program only partially fulfills the requirements of energy analysis. A more typical requirement is to analyse a program to discover the *distribution* of energy consumption over the parts of the program. In short, we want to perform *static profiling* of its energy usage (by analogy with *dynamic profiling*, which records resource usage during execution at a range of execution points such as function calls). With static profiling the profile is not just a set of counts but rather a set of functions associated with given program points, parameterised by the size of input to the entry point of the program.

For example, suppose that procedure `p(x)` calls procedure `q(y)` (possibly more than once). Rather than analysing the total energy consumed by a call to `p(x)`, including the resulting call(s) to `q(y)`, we would like to discover the resources used by procedure `q(y)` when activated from `p(x)`.

Motivations for static profiling. There are several motivations for static profiling:

- Firstly, a profile of the resource usage of the program can show the developer which parts of the program are the most resource-critical. For example, it can expose the cost of functions that are perhaps not so resource-hungry in themselves but which are called many times. Such parts can be natural targets for optimisation since a small optimisation might yield important savings.

- Secondly, there are cases where the overall resource complexity of a program might not be obtainable. For instance, some program parts may be too complex for analysis or perhaps the code for some parts is not available and the cost cannot even be reasonably estimated. In this case useful information can still be obtained by excluding such parts from the analysis, obtaining information about the resource usage for the rest of the program.

- Thirdly, resource usage models (for example Tiwari's energy consumption model [TMW94])

are sometimes based on summing the individual resource usage of basic components of the program. Static profiling fits naturally with such models.

- Fourthly, static profiling does not introduce the run-time overhead that is inevitable with dynamic profiling due to code instrumentation. The static profiling approach obtains safe upper and lower bounds on resource consumption, because it is based on the semantics of the program rather than on particular executions of it.

Outline of the static profiling technique. Our starting point is the well-developed technique of extracting cost equations or relations from the program, expressing resource usage functions defined by recurrence relations [Weg75, Ros89, DLH90, DL93, DLGHL97, AAG+08]. These are then solved to get a closed-form function expressing the (bounds on) parameterised resource usage. In [HLGL+16] (included in Deliverable 1.2 as attachment D1.2.3) we define a program transformation which allows a standard cost analyzer to infer statically parameterised functions for each of a specified set of program components (called "cost centres" in the paper).

The method has been implemented for the HC IR but not yet integrated with the tools handling XC source, LLVM IR or ISA.

8 Probabilistic Resource Analysis

Bounds on energy consumption are useful, but information about the distribution of consumption within those bounds is even more so. For example, it may be that most execution cases of a program result in consumption close to the lower bound, while the upper bound is reached only in a few outlying cases, or vice versa. From the distribution, estimates of average energy consumption can be derived. One approach to obtaining this kind of information is to perform probabilistic static analysis of a program with respect to its energy consumption. This is a special case of probabilistic output analysis, whose aim is to derive a probability distribution of possible output values for a program from a probability distribution of its input. The output in this case is energy consumption.

The first results published for this approach used fairly simple domains and distributions in the examples [RK15]. The technique, however, is not inherently restricted in this way. In a later paper [KR15] it is shown how to apply the analysis to more complex scenarios such as dependent non-uniform distributions and parametrized domains. The paper also describes how the technique has been integrated with a standard algebraic system, Mathematica. This gives it strength to analyse complex scenarios of dependencies as exemplified with the classic Montyhall

case. A detailed description of our results on this topic can be found in [RK15] and [KR15] (also included in this document as attachments D3.2.8 and D3.2.5, respectively).

9 Analysis of Multi-threaded Programs

In this section we present an analysis of multi-threaded programs aimed at discovering properties relevant to energy consumption and optimisation. We consider the XC language, which permits parallel threads and synchronous channel communication.

The XC Language. XC is "an imperative programming language with a computational framework based on C" [Wat09]. The main relevant language features of XC in this section are those for creating concurrent threads that can communicate and synchronise with each other using channels.

The most distinctive semantic features of XC relate to concurrency. One property is that parallel threads cannot make any updates to shared variables; another is that communication channels have two ends owned by exactly one thread each, so there is no race on being the first to send or get on a given channel. For the purpose of the analysis described here, these features ensure that the only way that a thread can influence another thread is by sending channel communications. Communication is *synchronous*; both sender and receiver wait until both have performed their send or read actions.

Energy-relevant aspects of multi-threaded code. The energy-relevant properties of code that we aim to analyse are mainly concerned with timing and synchronisation. This is because energy can be wasted by threads that wait frequently, or that complete their tasks too early for some external deadline. More specifically, we want to get answers to the following questions.

- How much work is done between thread communications?
- How active/inactive is each thread?
- What is the distribution over time of the number of running threads? What code blocks can (possibly) run in parallel with each other?

Potential power optimisations. If we can answer the questions above, a number of potential energy-wasting behaviours, and hence opportunities for optimisation, can be identified.

- Sometimes, threads can be slowed down (for example by reducing the frequency and voltage of a core) while still meeting deadlines from other threads.

- Threads that communicate frequently should be placed close to each other (taking into account communication infrastructure).
- Bottlenecks can be removed by shifting tasks or introducing more threads.
- Very inactive threads can be merged with other threads.

In addition, information about how many threads can run in parallel is needed for making accurate energy consumptions, according to the multi-threaded energy model for the xCORE [KE15a].

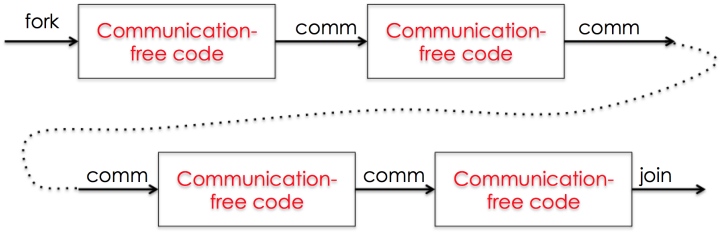


Figure 9: Life cycle of a thread

Thread behaviour: communication actions and communication-free code. Each thread is created by a `par` statement. After creation, it may communicate with other threads. Many of the interesting questions about thread behaviour and synchronisation concern the question "what does a thread do between communications?". Figure 9 illustrates the life-cycle of a thread. After creation, it alternates between executing code that does not communicate and communication actions (either sending or receiving) indefinitely or until the thread is terminated.

The problem of identifying the communication-free code sections is related to the task of generating the control flow graph in Section 5. We can identify all communication points in the code. Then any (feasible) path in the control flow graph from one communication point to another (without passing through any other communication point) can be enumerated. In Figure 10 we illustrate the process. In the thread code on the left, P, Q, $\cdots$ U are pieces of code containing no communications. The graph then shows the behaviour of the thread as a state machine that alternates communication on channel b, c or d with code sections C1, $\cdots$, C5, whose makeup is given on the right. We call the communication-free sections C1, $\cdots$, C5 *tasks*.

Timing of tasks. The analysis requires that the time of each section of communication-free code is given. Such a section is possibly non-deterministic. We assume that a duration can be

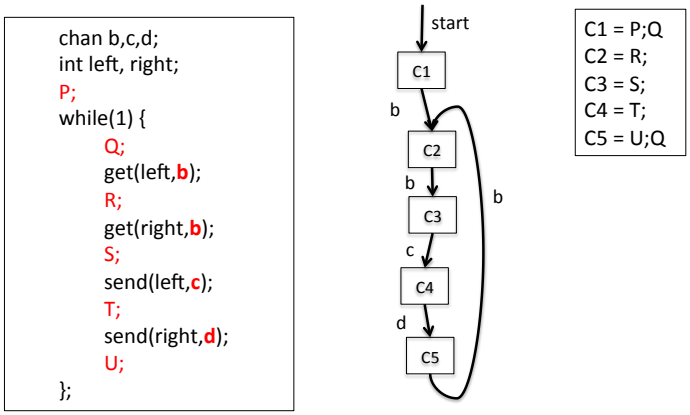


Figure 10: Identifying communication-free code sections

derived for each section, which could be an interval $[x, y]$ where x is the best (lowest) execution time and y is the worst execution time. More generally the time could be a constraint depending on data values, for example derived by an automatic complexity analysis tool or a worst-case execution time analysis. We number the tasks $1, 2, \ldots$ and denote the duration of the k^{th} task as d_k.

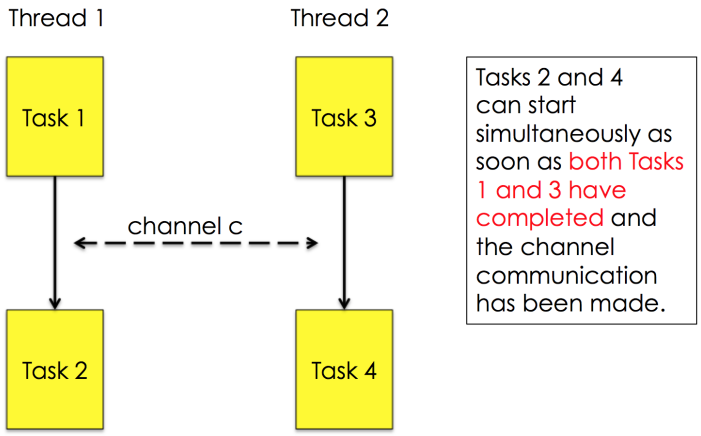


Figure 11: Basic synchronous synchronisation constraint

Constraints on task firing times. We now consider when a task can start or "fire". Consider two threads that communicate on some channel c. Figure 11 shows that the tasks immediately following the communication in each thread, namely tasks 2 and 4, can start simultaneously as soon as the communication on c has occurred, which in turn can happen as soon as both tasks 1

and 3 have completed.

The same task may fire many times, if the thread contains a loop. Let t_k^m denote the time of the m^{th} firing of the k^{th} task. We now formalise the constraints on the firing time, based on the general synchronisation rules described above. Consider again the threads in Figure 11, which we number 1 and 2 respectively, with tasks 1, 2, 3 and 4. Suppose the first thread is on its n^{th} iteration and the second thread is on its m^{th} iteration. We let $\delta(n) = n + 1$ if Task 2 is a loop header and thus the loop counter is incremented, otherwise $\delta(n) = n$; similarly $\delta(m)$ is defined relative to Task 4.

$$n \geq 0 \wedge m \geq 0 \tag{4}$$

$$t_2^{\delta(n)} = \mathsf{max}(t_1^n + d_1, t_3^m + d_3) \tag{5}$$

$$t_4^{\delta(m)} = \mathsf{max}(t_1^n + d_1, t_3^m + d_3) \tag{6}$$

The final group of constraints arise from the fact that the same number of communication events happens at each of the two ends of a channel. In order to formalise this, we define the following constants for each channel occurring in a thread. For simplicity we assume that the thread has one loop and a prefix which is executed once before entering the loop.

- c_{pre}: the number of occurrences of c in the prefix.
- $c_1, \ldots, c_k$: the k occurrences of the channel c within the loop.

Using these constants, we can assert that $c_{pre} + n * k + j$ is the number of communications on c when c_j in the loop is encountered, after n iterations of the loop have been completed. Therefore, if channel c occurs in two threads, the constraint that there is the same number of operations at each channel end, is captured by the following equation. Constants c_{pre_i}, j_i, k_i and n_i are respectively the number of occurrences of c in the prefix of thread i, the index of the occurrence of c in the loop thread i, the number of occurrences of channel c in loop i and the number of completed iterations of the loop i, for $i = 1, 2$.

$$c_{pre_1} + n_1 * k_1 + j_1 = c_{pre_2} + n_2 * k_2 + j_2 \tag{7}$$

Solving the task firing constraints. The constraints 4, 5, 6 and 7 are collected for all cases where a channel c connected two threads at points j_1 and j_2. The four tasks 1, ..., 4 can be identified syntactically for each such case. The solution of the constraints, or more generally an approximate solution, is a set of constraints defining t_k^n, the time of the n^{th} firing of task k, in terms of n only.

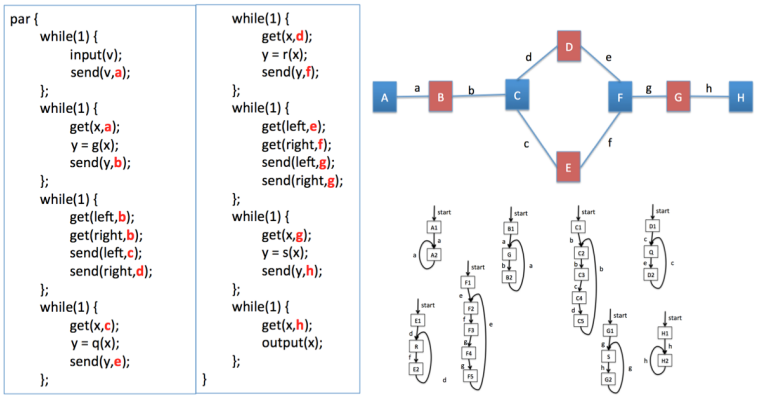


Figure 12: Example of a pipeline process

Illustrative example and prototype implementation. Figure 12 shows the pseudocode for eight threads forming a pipeline process; the pipeline splits in the middle as shown in the figure. The threads are also represented as task flow-graphs as discussed above, with the channel communications labelling the edges. We note that some of the inter-communication tasks are assumed to be very light (those marked in red) while the blue ones are "work" tasks. Let us assign timings to the tasks as follows.

- G = 300, Q = 334, R = 500, S = 250, all other tasks = 5.

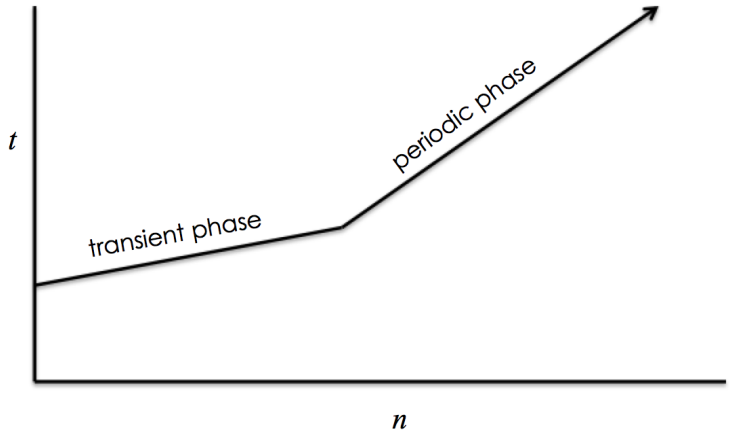


Figure 13: Typical transient and periodic phases in thread timing

We encoded the constraints as Horn clauses, and computed an approximate solution using a convex polyhedral solver, as described in [GK14, KG15a]. We obtained a solution that shows that the program timings have an initial transient phase followed by a periodic phase [GGS+06],

as shown in Figure 13. This is because the threads start off as quickly as possible but delays from the later stages of the pipeline take a few iterations to propagate back to the start of the pipeline. When the threads reach a steady state, we find in the example that all thread loops have a period of either 610 or 305 (some threads loop twice as fast as others because of the split in the pipeline). These periods define the *throughput* of the pipeline, namely the rate at which elements enter the pipeline (which is 305 time units). This is useful information that by no means immediately clear from the code or even from task timings.

Furthermore the analysis allows us to compute the percentage activity for each thread, since we know its period and the sum of the times of the tasks in the thread. Given the timings above, this ranges from 1.6% (thread A) to 83% (thread E). Other information that can be derived easily from the solution includes:

- when one task definitely waits for another;
- which tasks can run simultaneously;
- which tasks on different threads do not run at the same time;
- frequency of each channel communication.

Status of the prototype tool. The constraints are currently generated automatically from a schematic representation of the threads in which tasks have already been identified. The solution of the constraints is semi-automatic in that the transient and periodic phases have to be identified before the precise periodic solution can be found. Current and future work consists of (a) generating the timing constraints directly from XC code, (b) trying successively longer transient phases until a periodic phase is found and (c) integration with the timing analyser tools for XC to determine task durations.

10 Energy Consumption Verification and Debugging

Verification compares actual system properties with required properties in order to prove them or to detect inconsistencies. We have developed techniques that enable software engineers to express and verify properties of the system throughout its life-cycle relevant to energy. For example, our techniques address the important challenge of being able to certify that a given energy budget is met, while maintaining a given level of quality of service.

10.1 Verification based on cost function comparison and the CiaoPP system

We first have defined an energy usage semantics that represents energy as a function of different system and data parameters. Then, we have defined advanced function comparison operations in order to compare the approximated semantics inferred by static analysis with the intended semantics expressed by specifications written in the common assertion language. In general, the result of such comparison gives preconditions under which a given specification is met or not.

We have developed a prototype tool based on these ideas, using the CiaoPP system, for the verification of XC programs running on the XS1-L architecture. The tool is described in [LGHK$^+$15] (included in this document as attachment D3.3.2). The input to the tool is the XC source program together with specifications written in the XC source code as (front-end) assertions. Such specifications can include both lower and upper bounds on energy usage, and they can express intervals within which energy usage is to be certified to be within such bounds. The bounds of the intervals can be given in general as functions on input data sizes. Our verification tool can prove whether such energy usage specifications are met or not. It can also infer the particular conditions under which the specifications hold. These conditions are also expressed as intervals of functions of input data sizes, such that a given specification can be proved for some intervals but disproved for others. The specifications themselves can also include preconditions expressing intervals for input data sizes. Attachment D3.3.2 also illustrates with an example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

If an energy consumption assertion is violated, the software developer is informed and a debugging process is started in order to find the causes of the assertion violation, and help the developer redesign the program.

10.2 Supporting techniques for program verification

Research on Horn clauses as a common semantic representation language. The use of Horn clauses is established as a representational formalism for software verification. Horn clauses have simple logical semantics, but are expressive enough to capture important aspects of the meaning of most programming languages. Therefore it functions as a common semantic representation; source programs in a variety of languages are translated into Horn clauses, in such a way that properties of the Horn clauses can be mapped into properties of the source program from which they are derived. Analysis and verification can then be performed on the Horn

clauses and the results interpreted in the source program. The advantage is that a set of tools for handling Horn clauses can be applied to many different programming languages. It suffices to write a suitable translator to Horn clauses. (This is not a trivial task, but beyond the scope of the discussion here). Horn clauses were adopted as an internal semantic representation in the ENTRA project, and applied to analysis and verification of XC programs, LLVM IR code and ISA.

The analysis of Horn clauses was itself a research goal in order to ensure wider applicability and scalability of the ENTRA approach; results and tools were transferred where possible into the ENTRA tools. The goal was to establish that tools based on Horn clause representations were capable of reaching the state-of-the-art in software verification and analysis.

Convex Polyhedral Analysis Tools. The domain of convex polyhedra is a fundamental abstract domain for relations over numerical values, which are typical for Horn clauses derived from imperative programs. We implemented and tested several tools that combined and extended state-of-the-art techniques for convex polyhedral analysis [GK14, KG14, KG15a] (attachments D3.3.10, D3.3.3 and D3.3.7 in this document respectively). In particular, a generic "constraint strengthener" was developed [KG15a] which can be used to enhance any other Horn clause analyser. This was successfully combined with existing ENTRA analysis prototype tools.

Tools for refinement and decomposition of Horn clause verification problems. Effective scaling up of verification tools requires techniques to decompose complex problems into smaller ones, and to refine analyses that are insufficiently precise to verify a given property. We investigated a novel decomposition technique based on "tree dimension" [KGG15] (see Attachment D3.3.9). Regarding refinement, we developed a novel technique based on transforming Horn clause programs to eliminate false alarms arising from imprecise analyses. An iterative procedure based on this approach was developed [KG15b] (see Attachment D3.3.4), which compared favourably with existing "abstraction-refinement" tools.

11 Demonstration of the Verification Tool based on HC IR Transformation and the CiaoPP System

This section provides a demonstration of the use of the implemented prototype tool based on HC IR transformation and the CiaoPP system in two typical scenarios: analyzing the energy consumed by an XC program and verifying energy related specifications.

We use the ENTRA tools front end for this purpose. Although the ENTRA tools front end is described in detail in deliverable D1.2, we include a short description here, to make the document more self-contained.

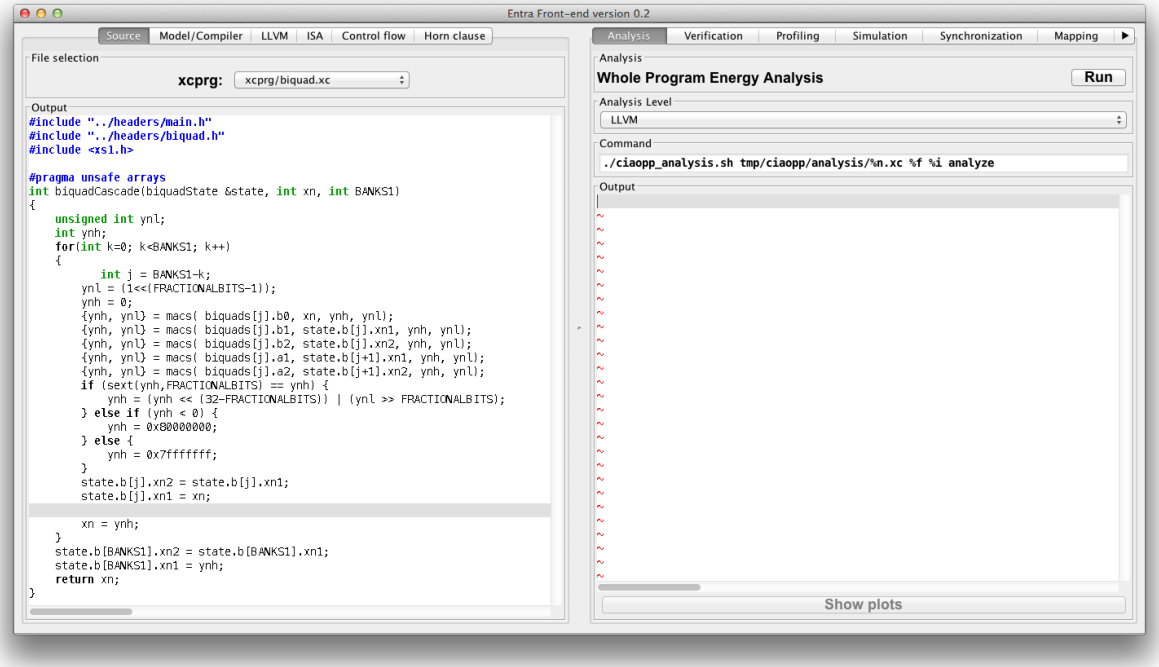


Figure 14: Graphical User Interface.

Figure 14 shows a screenshot of the ENTRA tools front end, which has two main sections, one on the left and the other on the right. The source file is loaded into the buffer on the left section under the *Source* tab. The user can load one of the benchmarks from the *xcprog* drop down. This drop down is populated with all the benchmarks under the directory xcprog which is located inside the corresponding tool's directory.

The *Analysis* and *Verification* tabs (within the section on the right) allow the user to perform analysis and verification of the program loaded into the buffer (within the section on the left). Under each of the two tabs, the user can select the *Analysis Level* to be either *LLVM* or *ISA*, which specify the level (LLVM IR or ISA) to which the XC source is compiled and at which it is then analyzed by the underlying tool (e.g., CiaoPP). Once the user presses the *Run* button, the analysis/verification is run and its output is loaded into the buffer on the right. The underlying command used to invoke the CiaoPP analysis/verification is also shown in the *Command* box. The *Show plots* button allows the user to see the output plotted graphically. It is shown using the gnuplot utility.

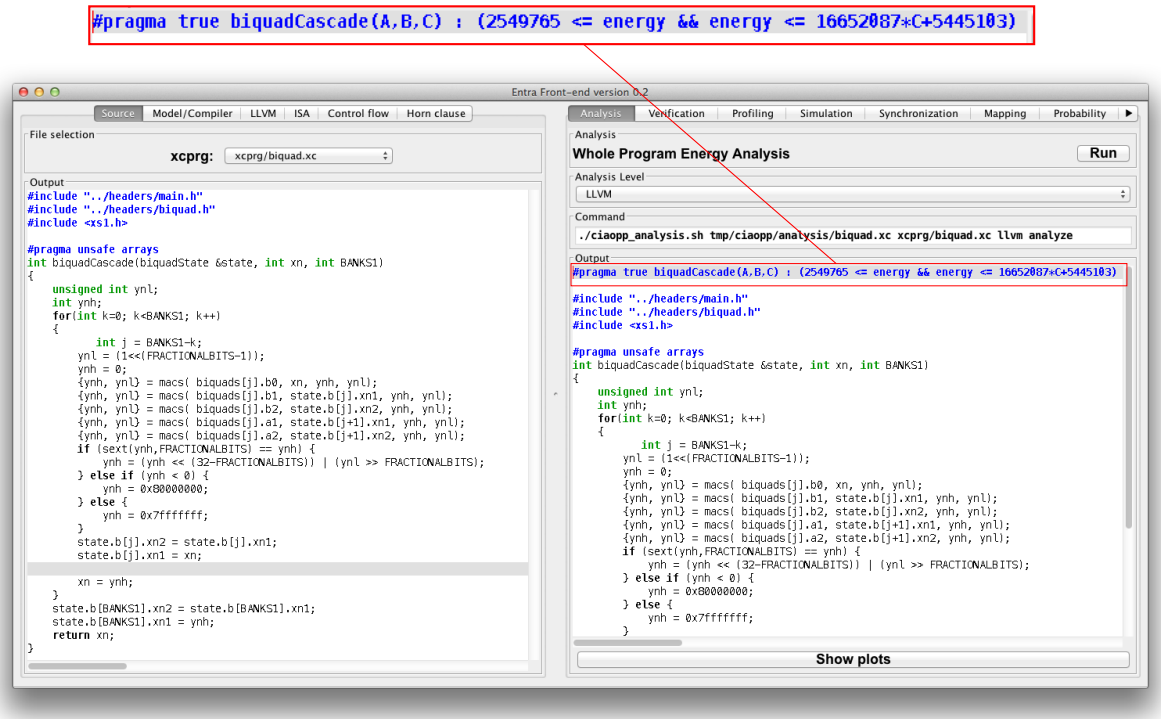


Figure 15: Analysis results (expressed as front end assertions).

Assuming that the CiaoPP system has been properly installed as well as the XMOS compiler, the tool can perform the analysis and verification of XC programs that are loaded into the buffer to the left hand side of the ENTRA tools front end.

11.1 Analyzing the energy consumed by an XC program

We first show how analysis of an XC program is performed using the ENTRA tools front end. For example, we select the *biquad.xc* benchmark from the *xcprog* drop down menu, as shown in Figure 14. Then we select *LLVM* from the *Analysis Level* drop down (which will tell the analysis to take the LLVM IR option by compiling the source code into LLVM IR and transform this into HC IR for analysis). After clicking on the `Run` button, the analysis is performed, producing the results as depicted in Figure 15. Such results are expressed in the front end syntax of the common assertion language, as explained in Deliverables D2.1 [EG13] and D3.1 [LG13]. We can see for example that the upper bound on the energy consumption of the *biquadCascade* program is given as a linear function on the size of the input argument to the program, C (number of BANKS), namely $16652087 * C + 5445103\ nJ$.

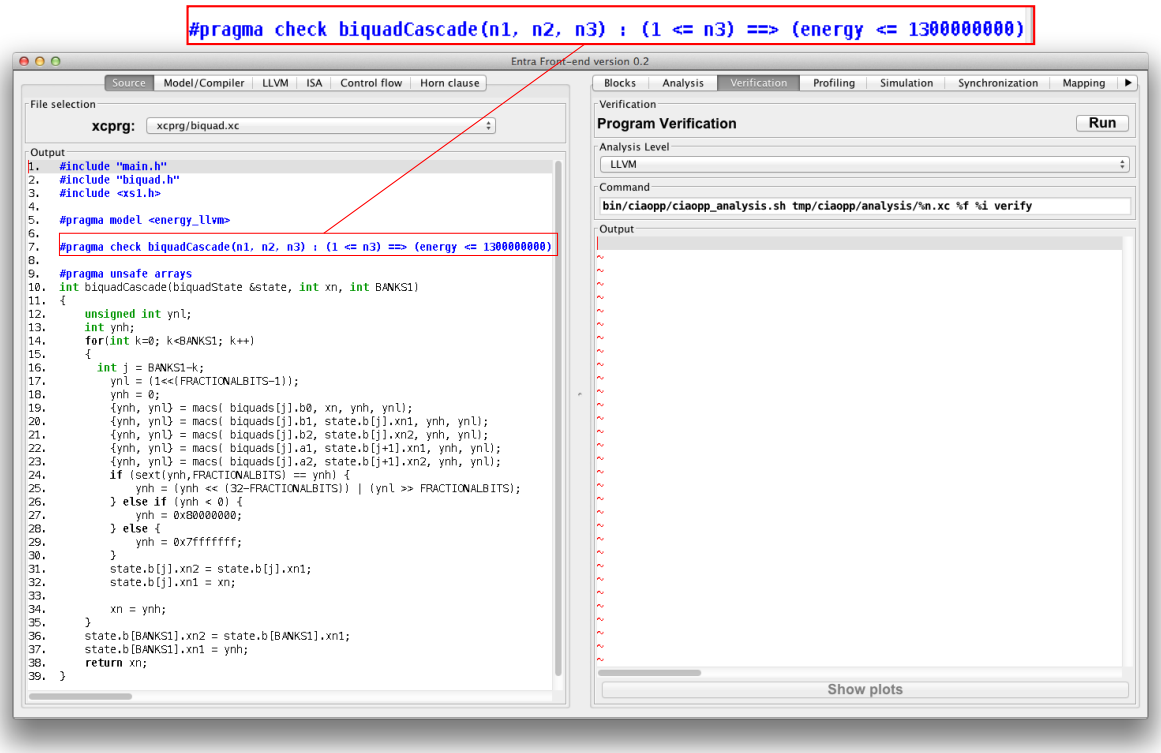


Figure 16: Program verification.

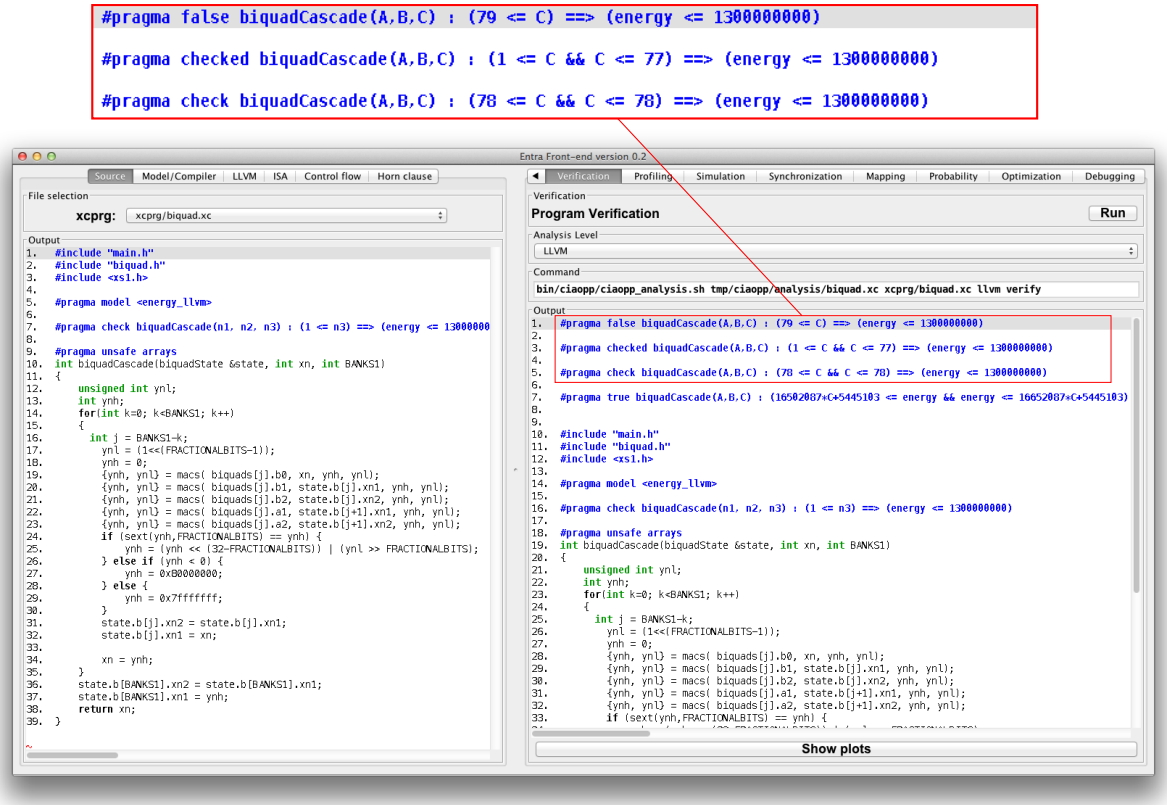


Figure 17: Verification results (expressed as front end assertions).

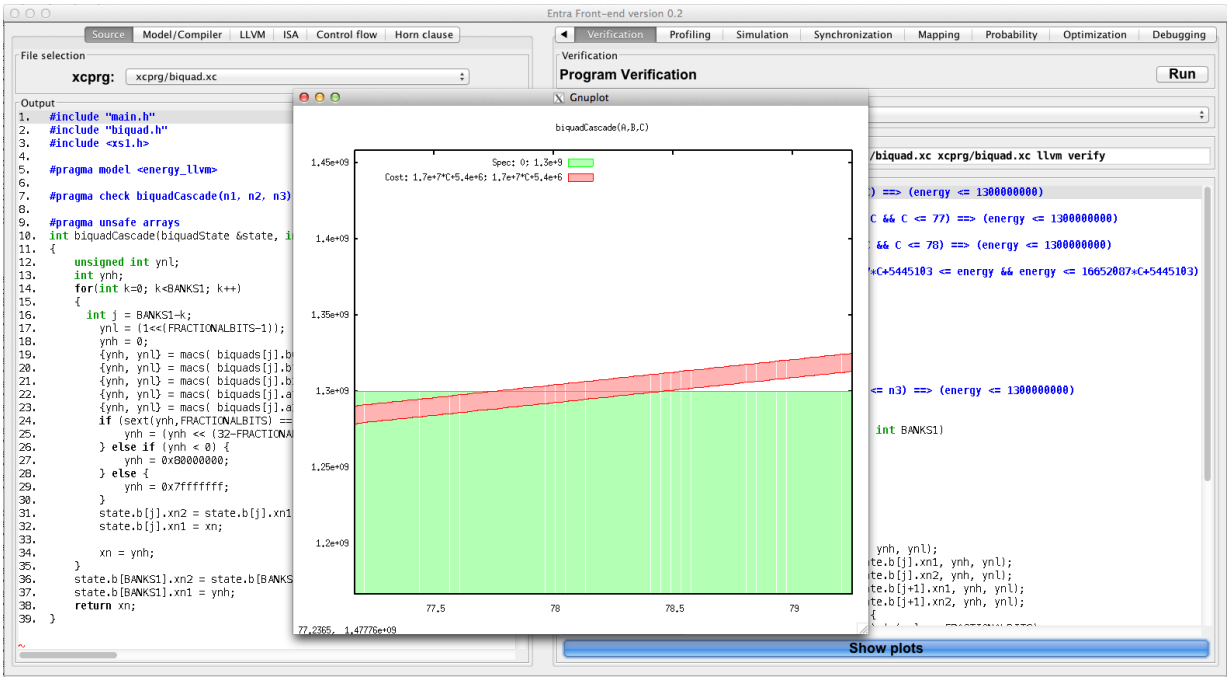


Figure 18: Verification results (graphically plotted).

11.2 Verification of the energy budget for an XC program

In order to verify that an energy budget can be met by a given XC program and to find the optimal values for the inputs for which such budget is respected, we first open it in a buffer using the *xcprog* drop down menu, as shown in Figure 16 (we select the same *biquad.xc* benchmark). Line 7 in the program specifies the energy budget (to be less than or equal to 13000000 *nJ*) with a *check* assertion. We then select *LLVM* from the *Analysis Level* drop down, as before. After clicking on the `Run` button, the analysis and verification is performed, producing the results depicted in Figure 17 (on lines 1, 3, and 5). Such results are expressed again in the front end syntax of the common assertion language, as explained in Deliverables D2.1 [EG13] and D3.1 [LG13].

We can see that the upper bound on the energy consumption of the *biquadCascade* program is given as before as a linear function on the size of the input argument to the program, C (number of BANKS), namely $16652087 * C + 5445103\ nJ$ on line 7. On line 3 the assertion with status *checked* indicates that the energy budget (specified with the *check* assertion on line 16) is met for the interval $1 =< C =< 77$ of values for argument C. On the other hand, the assertion with status *false*, on line 1, indicates that for all values of the argument C such that $C >= 79$ the energy budget *is not met*. Finally, on line 5 the assertion with status *check* indicates that for the value of the argument $C = 78$, the verification cannot conclude if the budget will be met or not.

The tool also allows the user to plot the results. By clicking on the *Show plots* button in

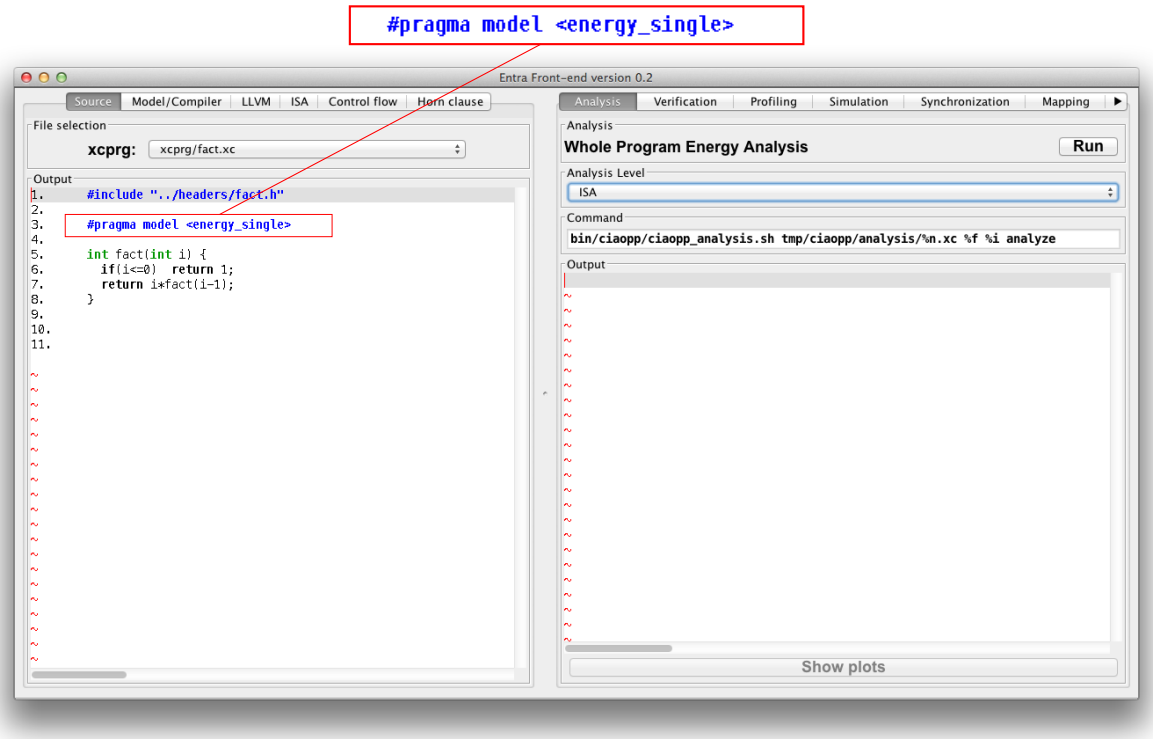


Figure 19: Using different energy models (given by a front end assertion).

Figure 17, the tool will plot the results as shown in Figure 18. The x axis represents the input data size (e.g., C) on which the cost function depends, and the y axis represents the energy consumption. The green region is the specification whereas the pink region represents the cost function. The user can visually notice that for interval $1 =< C =< 77$ of the argument C (x axis), the energy budget is met, while for the interval $C >= 79$ the energy budget will not be met. From the plot it is also clear that for the value $C = 78$ the energy budget cannot be guaranteed.

11.3 Showing that the analysis/verification is parametric w.r.t. the energy models

This section illustrates that the analysis and verification components of the tool are parametric with respect to the energy models used. The analysis and verification techniques produce safe results provided that the energy models express safe information. We show how different energy models can be used for the analysis of a given XC program and what are the results in each case. Once again, we use the ENTRA tools front end for this purpose.

Figure 19 shows a screenshot of the ENTRA tools front end, with an XC program implement-

ing the *factorial* function, which is already loaded into the buffer. As we can see in the figure, there is an assertion indicating which energy model should be used to analyze the program. In this example, we choose a model named `energy_single`, which is an ISA-level energy model that assigns a *single* average energy consumption value to each ISA operation.

As we have shown before, in order to analyze this program using the graphical ENTRA tools front end, we select *ISA* as the level of the analysis from the *Analysis Level* drop down. After clicking on the `Run` button, the analysis is performed using the model indicated in the assertion, in this case `energy-single`. Then, we can see the results of the analysis, and by clicking the *Show plots* button we can see graphically the energy bounds obtained. Figure 20 shows this result. We can observe that both lower and upper bounds are equal, due to the fact that the used model assigns a single energy value to each ISA operation.

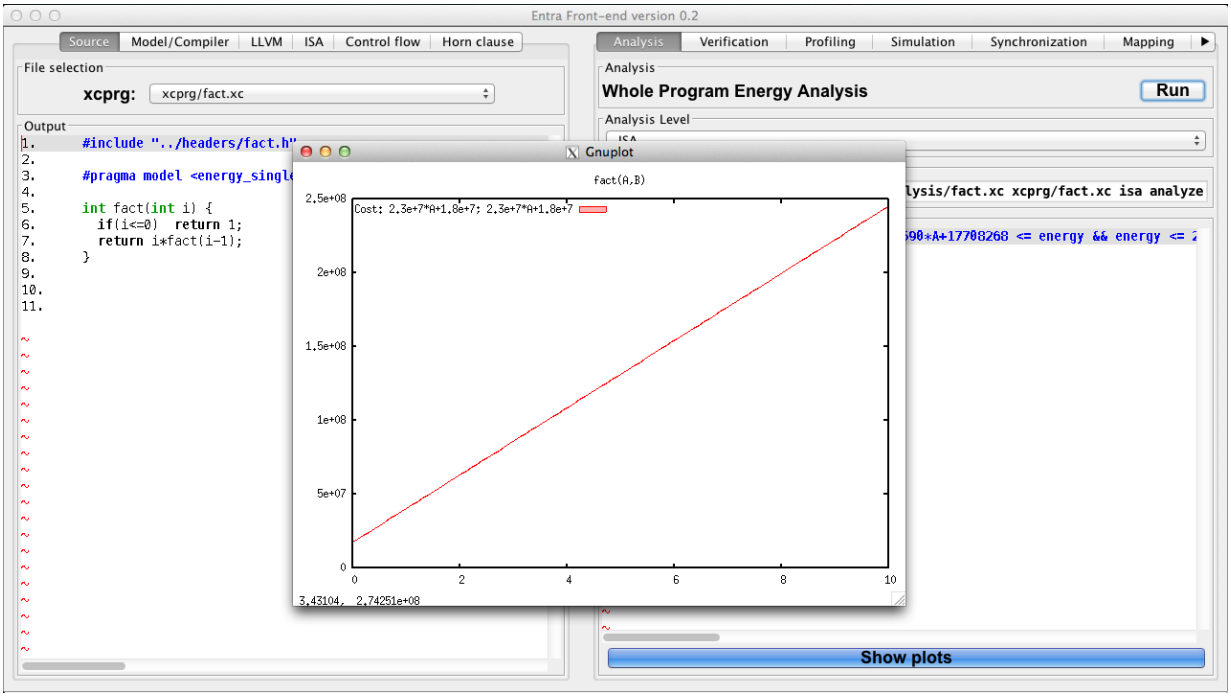


Figure 20: Analysis result using a single-valued, average Energy Model.

To illustrate the different results that can be obtained by using other energy model, we modify the assertion in `fact.xc`, by choosing this time a model named `energy_interval` (see Figure 21). The difference between this model and the previous one is that it has been generated by assigning an energy *interval* to each ISA operation. The end points of such interval have been obtained by both subtracting and adding a *percentage of error* to each average value of the `energy_single` model (in this case the error we have chosen is 10%).

By following exactly the same procedure, we analyze the program again, and this time we ob-

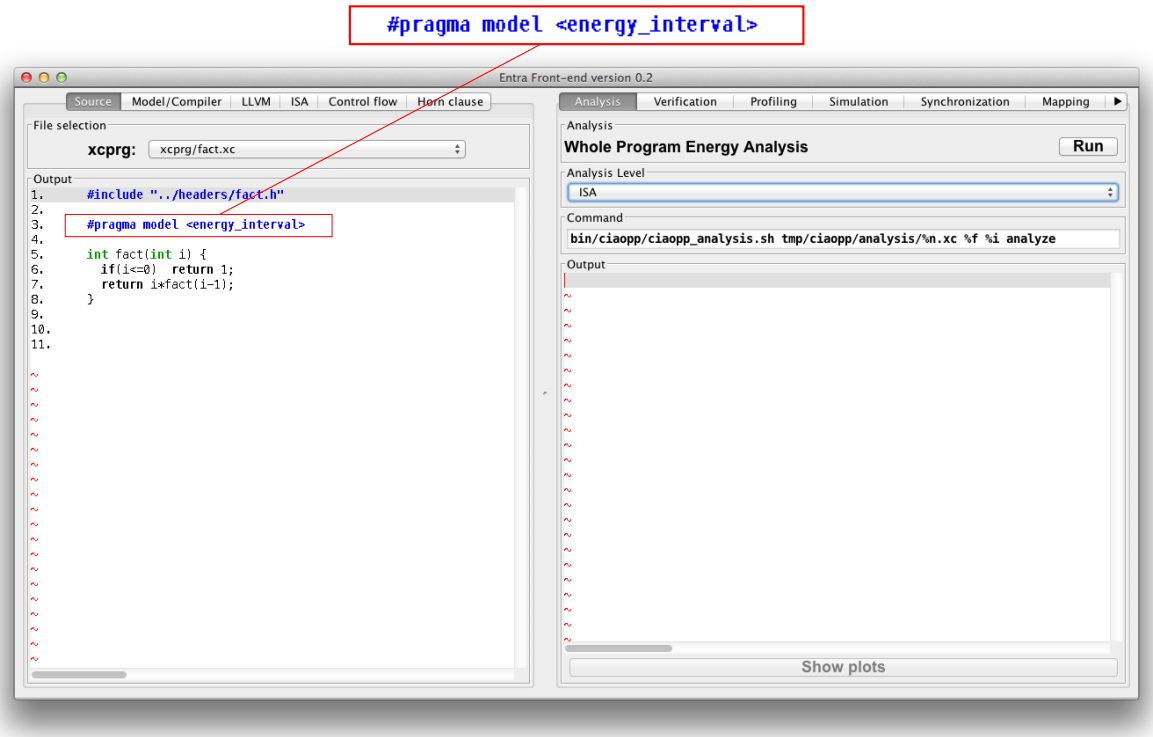


Figure 21: Using different energy models (given by a front end assertion).

tain two different linear functions, corresponding to a lower and upper bound on the energy consumed by the program `fact.xc`. We can see graphically in Figure 22 that these two functions now determine an area within which the actual energy consumed by the program is estimated to lie.

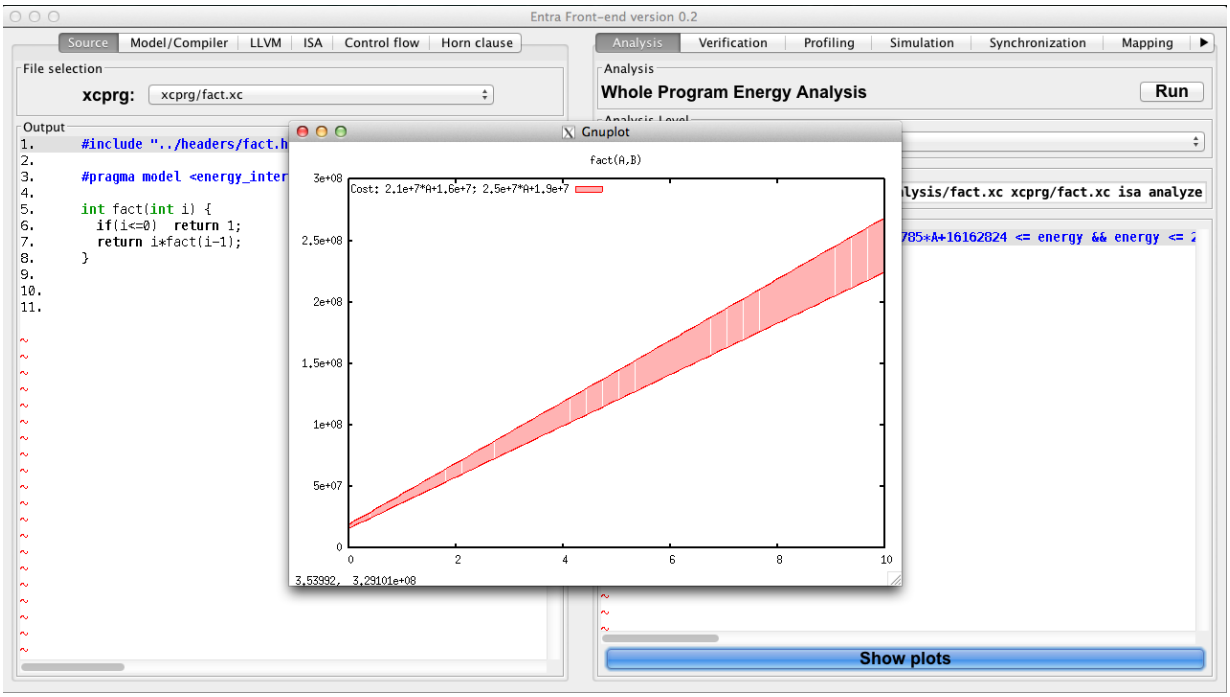


Figure 22: Analysis results using an interval-based energy model.

References

[AAG+08] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: A Cost and Termination Analyzer for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'08)*, Electronic Notes in Theoretical Computer Science, Budapest, Hungary, April 2008. Elsevier.

[AAGP09] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Cost relation systems: A language-independent target language for cost analysis. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 248:31–46, August 2009.

[AAGP11] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.

[BHZ08] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[BLLG15] Z. Banković, U. Liqat, and P. López-García. Trading-off Accuracy vs. Energy in Multicore Processors via Evolutionary Algorithms Combining Loop Perforation and Static Analysis-based Scheduling. In Enrique Onieva, Igor Santos, Eneko Osaba, Héctor Quintián, and Emilio Corchado, editors, *Hybrid Artificial Intelligent Systems (HAIS 2015)*, volume 9121 of *Lecture Notes in Computer Science*, pages 690–701. Springer International Publishing, 2015.

[BRZH02] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis: Proceedings of the 9th International Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229, Madrid, Spain, 2002. Springer-Verlag, Berlin.

[DL93] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[DLGHL97] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[DLH90] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[EG13] K. Eder and N. Grech, editors. *Common Assertion Language*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 2.1, http://entraproject.eu.

[Flo67] R. W. Floyd. Assigning Meanings to Programs. In J.T Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, volume 19, Mathematical Aspects of Computer Science, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[GGP+15] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES 2015, pages 12–21, New York, NY, USA, 2015. ACM.

[GGS+06] Amir Hossein Ghamarian, Marc Geilen, Sander Stuijk, Twan Basten, Bart D. Theelen, Mohammad Reza Mousavi, A. J. M. Moonen, and Marco Bekooij. Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design (ACSD 2006), 28-30 June 2006, Turku, Finland*, pages 25–36. IEEE Computer Society, 2006.

[GK14] John P. Gallagher and Bishoksan Kafle. Analysis and transformation tools for constrained horn clause verification. *Theory and Practice of Logic Programming*, 14(4-5 (supplementary materials)):90–101, Jun. 2014. ICLP, Vienna.

[GKE15] K. Georgiou, S. Kerrison, and K. Eder. On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs. Technical report, oct 2015.

[HLGL+16] R. Haemmerlé, P. Lopez-Garcia, U. Liqat, M. Klemen, J. P. Gallagher, and M. V. Hermenegildo. A Transformational Approach to Parametric Accumulated-cost Static Profiling. In *Thirteenth International Symposium on Functional and Logic Programming (FLOPS 2016)*, LNCS. Springer, 2016. To appear.

[KE15a] S. Kerrison and K. Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):1–25, April 2015.

[KE15b] Steve Kerrison and Kerstin Eder. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Trans. Embedded Comput. Syst.*, 14(3):56, 2015.

[KG14] Bishoksan Kafle and John P. Gallagher. Convex polyhedral abstractions, specialisation and property-based predicate splitting in horn clause verification. In Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko, and Valerio Senni, editors, *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014*, volume 169, pages 53–67. EPTCS, Jan. 2014. HCVS, Vienna.

[KG15a] Bishoksan Kafle and John P. Gallagher. Constraint specialisation in horn clause verification. In Kenichi Asai and Kostis Sagonas, editors, *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015*, pages 85–90. Association for Computing Machinery, Jan. 2015. PEPM, Mumbai.

[KG15b] Bishoksan Kafle and John P. Gallagher. Tree automata-based refinement with application to horn clause verification. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *Lecture Notes in Computer Science*, pages 209–226. Springer, Jan. 2015. VMCAI, Mumbai.

[KGG15] Bishoksan Kafle, John P. Gallagher, and Pierre Ganty. Decomposition by tree dimension in horn clause verification. In Alexei Lisitsa, Andrei P. Nemytykh, and Alberto Pettorossi, editors, *Proc. of the 3rd International Workshop on Verification and Program Transformation (VPT'2015)*, volume 199 of *EPTCS*, pages 1–14, 2015. arXiv:1512.03862.

[KR15] M. Kirkeby and M. Rosendahl. Probabilistic Resource Analysis by Program Transformation. In *Proc. of the Foundational and Practical Aspects of Resource Analysis*, LNCS. Springer, 2015. To appear.

[LA04] C. Lattner and V.S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, March 2004.

[LBLGH16] U. Liqat, Z. Banković, P. Lopez-Garcia, and M. V. Hermenegildo. Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2016)*, 2016. arXiv:1601.02800.

[LG13] P. López-García, editor. *A General Framework for Resource Consumption Analysis and Verification*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 3.1, http://entraproject.eu.

[LGHK+15] P. Lopez-Garcia, R. Haemmerlé, M. Klemen, U. Liqat, and M. V. Hermenegildo. Towards Energy Consumption Verification via Static Analysis. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES), arXiv:1501.03064*, 2015. arXiv:1512.09369.

[LGK+16] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In M. Van Eekelen and U. Dal Lago, editors, *Foundational and Practical Aspects of Resource Analysis. Fourth International Workshop FOPARA 2015, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2016. To appear.

[LKS+14] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In Gopal Gupta and Ricardo Pea, editors, *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.

[LM97] Y.T.-S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(12):1477–1487, Dec 1997.

[NMLH09] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of BYTECODE*, volume 253 of

Electronic Notes in Theoretical Computer Science, pages 65–82. Elsevier - North Holland, March 2009.

[PKME16] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Data Dependent Energy Modeling for Worst Case Energy Consumption Analysis. In *53rd Design Automation Conference (DAC) [under submission]*. ACM, June 2016.

[PR04] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, Lecture Notes in Computer Science, pages 239–251. Springer, 2004.

[RK15] Mads Rosendahl and Maja H. Kirkeby. Probabilistic output analysis by program manipulation. In Nathalie Bertrand and Mirco Tribastone, editors, *Proceedings Thirteenth Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2015, London, UK, 11th-12th April 2015.*, volume 194 of *EPTCS*, pages 110–124, 2015. QAPL, London.

[Ros89] M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156. ACM Press, 1989.

[SLGBH13] A. Serrano, P. Lopez-Garcia, F. Bueno, and M. Hermenegildo. Sized Type Analysis for Logic Programs (technical communication). *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, 13(4-5):1–14, August 2013.

[SLGH14] A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754, 2014.

[SMR11] Henry Hoffmann Sasa Misailovic, Stelios Sidiroglou and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proc. of FSE'11*. ACM Press, 2011.

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[Wat09] Douglas Watt. *Programming XC on XMOS Devices*. XMOS Ltd., 2009.

[Weg75] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[xTI] xtimecomposer. Accessed: 2014.

Attachments

Attachment D3.3.1

Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks

Published at the Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2016)

Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks

U. Liqat [†]
umer.liqat@imdea.org

Z. Banković [†]
zorana.bankovic@imdea.org

P. Lopez-Garcia [*] [†]
pedro.lopez@imdea.org

M.V. Hermenegildo [†] [‡]
manuel.hermenegildo@imdea.org

ABSTRACT

We are currently witnessing an increasing number of energy-bound devices, including in some cases mission critical systems, for which there is a need to optimize their energy consumption and verify that they will perform their function within the available energy budget. In this work we propose a novel parametric approach to estimating tight energy bounds (both upper and lower) that are practical for energy verification and optimization applications in embedded systems. Our approach consists in dividing a program into basic ("branchless") blocks, establishing the maximal (resp. minimal) energy consumption for each block using an evolutionary algorithm, and combining the obtained values according to the program control flow, using static analysis, to produce energy bound functions. Such functions depend on input data sizes, and return upper or lower bounds on the energy consumption of the program for any given set of input values of those sizes, without running the program. The approach has been tested on XMOS chips, but is general enough to be applied to any microprocessor and programming language. Our experimental results show that the bounds obtained by our prototype tool can be tight while remaining on the safe side of budgets in practice.

Keywords

Energy Consumption Analysis, Energy Modeling, Embedded Systems, Static Analysis, Evolutionary Algorithms.

1. INTRODUCTION

We are witnessing an ever-increasing performance and ubiquity of battery- and/or harvested energy-powered devices. An important trend in this context is the so called *Internet of Things* paradigm. It is estimated that by the year 2020, about 50 billion small autonomous devices, embedded in all kind of objects, in our clothes, or stuck to our bodies will operate and intercommunicate continuously for long periods of time, such as years. Such devices rely on small batteries or energy harvested from the environment, which implies that their energy consumption should be very low.

Although there have been improvements in battery and energy harvesting technology, they alone are often not enough to achieve the required level of energy consumption to fully support *Internet of Things* and other energy-bound applications. Thus, better techniques for optimizing the energy consumption of embedded systems are needed. While many energy-saving features have been developed for hardware, far more energy savings remain to be tapped by improving the software that runs on these devices. In addition, there are many critical embedded applications (e.g., sensor-based) for which, beyond optimizing energy consumption, it is actually crucial to guarantee that execution will complete within a specified energy budget, e.g., before the available system energy runs out.

In this work we focus on the *static* estimation of the energy consumed by program executions (i.e., at compile time, before actually running them), as a basis for energy optimization and verification. Such estimations are given as functions on input data sizes, since data sizes typically influence the energy consumed by a program, but are not known at compile time. This approach allows abstracting away such sizes and inferring energy consumption in a way that is parametric on them.

Different types of resource usage estimations are possible, such as, e.g., probabilistic, average, or safe bounds. However, not all types of estimations are valid or useful for a given application. For example, in order to verify/certify energy budgets, *safe upper- and lower-bounds* on energy consumption are required [16, 15]. Unfortunately, current approaches that guarantee that the bounds are always safe tend to compromise their tightness seriously, inferring overly conservative bounds, which are not useful in practice. With this safety/tightness trade-off in mind, our goal is the development of an analysis that infers tight bounds that are on the safe side in most cases, in order to be practical for verification applications, as well as for energy optimization.

Of the small number of static energy analyses proposed to date, only a few [20, 13, 12] use resource analysis frameworks that are aimed at inferring safe upper and lower bounds on the resources used by program executions. A crucial component in order for such frameworks to infer hardware-dependent resources, and, in particular, energy, is a low-level resource usage model, such as, e.g., a model of the energy consumption of individual instructions. Examples of such models are [11], at the Java bytecode level, or [10], at the assembly level.

Clearly, the accuracy of the bounds inferred by analysis depends on the nature and accuracy of the low-level models. Unfortunately, models such as [11, 10] provide *average* energy consumption values or functions, which are not really suitable for upper- or lower-bounds analysis. Furthermore, trying to obtain instruction-level models that provide strict safe energy bounds would result in very conservative bounds. Although when fed with such models the static analysis would infer high-level energy consumption functions providing strictly safe bounds, these bounds would not be useful in general because of their large inaccuracy. For this reason, the analyses in [20, 13, 12] used instead the already mentioned instruction level average energy models [11, 10]. However, this meant

[*]Spanish Council for Scientific Research (CSIC).

[†]IMDEA Software Institute, Madrid, Spain.

[‡]Universidad Politécnica de Madrid (UPM).

that the energy functions inferred for the whole program were not strict bounds, but rather approximations of the actual bounds, and could possibly be below or above. This trade-off between safety and accuracy is a major challenge in energy analysis. In this paper we address this challenge by providing a technique for the generation of lower-level energy models which are useful and effective in practice for verification-type applications.

The main source of inaccuracy in current instruction-level energy models is inter-instruction dependence (including also data dependence), which is not captured in most models. On the other hand, the concrete sequences of instructions that appear in programs exhibit worst cases that are not as pessimistic as considering the worst case for each of the individual intervening instructions. Based on this, we decided to use *branchless blocks* of assembly instructions as the modeling unit instead of individual instructions. We divide the (assembly) program into such *basic blocks*, each a straight-line code sequence with exactly one entry to the block (the first instruction) and one exit from the block (the last instruction). We then measure the energy consumption of these basic blocks, and determine a maximal (resp. minimal) energy consumption for each block. In this way the inter-instruction data dependence discussed above and other factors are accounted for. The energy values obtained for each block are fed to our static resource analysis, which combines them according to the program control flow and produces the energy bound functions.

In order to find the maximum and minimum energy consumption of each basic block we use an evolutionary algorithm (EA). We vary the input values and take energy measurements directly from the hardware for each input combination. This way, we take advantage of the fast search space exploration provided by EAs. EAs have also been used for estimating the worst case energy consumption of *whole* programs [22], due to their fast exploration of the search space. However, if there are data-dependent branches in the programs, which is often the case, applying this approach to whole programs (or program segments that contain branches) quickly loses accuracy, since different input combinations can trigger different sets of instructions [22]. In contrast, our approach combines EAs and static analysis techniques in order to get the best of both worlds. We take out the treatment of data-dependent branches from the EA, so that the same sequence of instructions is always executed in each basic block. The worst (resp. best) case energy of the basic blocks is estimated by the EA with higher accuracy since, not having any branches, the most important deficiency of the EA is avoided. The program control flow dependencies are taken care instead by the static analysis.

In our experiments we focus for concreteness on the energy analysis of programs written in XC [28], running on the XMOS XS1-L architecture. However, our approach is general enough be applied to the analysis of other programming languages (and associated lower level program representations) and architectures as well. XC is a high-level C-based programming language that includes extensions for concurrency, communication, input/output operations, and real-time behavior. Our experimental setup infers energy consumption information by processing the ISA (Instruction Set Architecture) code compiled from XC [28], and reflects it up to the source code level. Such information is provided in the form of *functions on input data sizes*, and is expressed by means of *assertions* [7].

In these experiments, the energy estimations produced by our approach were always safe, in the sense that they over-approximated the actual bounds (i.e., the inferred upper bounds were above the actual upper bounds and the inferred lower bounds below the actual lower bounds). This suggests that, even if we cannot assure formally that such estimations are always safe, they are quite accurate in the sense that the inferred energy bounds are close to the actual bounds, and that in practice they will also be safe/strict in most cases. We argue that our analysis provides a good practical compromise for the verification/certification of energy budgets.

In summary, the main contributions of this paper are:

- A novel approach that combines dynamic and static analysis techniques for inferring the energy consumption of program executions. The dynamic part is based on EAs, and produces low-level energy models.
- The proposal of a new abstraction level at which to perform the energy modeling of program components using dynamic techniques: basic (branchless) blocks of assembly instructions.
- A method based on EAs to dynamically (i.e., by profiling) obtain practical upper and lower bounds on the energy of such basic blocks, with a good safety/accuracy balance.
- The use of a static analysis that takes care of the program control flow, in order to determine how many times blocks are executed, which combined with the information provided by the block models, infers functions that give the energy of a program and its procedures as functions of input data sizes.
- An experimental study that supports our claims.

In the rest of the paper, Section 2 explains how the information inferred by our approach can be used for the energy consumption verification application. Section 3 explains our technique for energy modeling of program basic blocks. Section 4 shows how these models are used by the static analysis to infer upper- and lower-bounds on the energy consumed by programs as functions of their input data sizes. Section 5 reports on an experimental evaluation of our approach. Related work is discussed in Section 6, and finally Section 7 summarises our conclusions.

2. ENERGY CONSUMPTION VERIFICATION/CERTIFICATION

The lower (E_l) and the upper bound (E_u) inferred by our analysis can be used for energy consumption verification and certification. We refer the reader to [14, 15] for a detailed description on how static analysis information can be used for general resource usage verification within the CiaoPP system, and to [16] for how it can be specialized for verifying energy consumption specifications of embedded programs.

Here we only give some intuitive ideas. Assume that a program specification expresses energy budget E_b, e.g., defined by the capacity of the battery, we can conclude the following:

1. $E_u \leq E_b \implies$ the given program can be safely executed within the existing energy budget.
2. $E_l \leq E_b \leq E_u \implies$ it might be possible to execute the program, but we cannot claim it for certain.
3. $E_b < E_l \implies$ it it not possible to execute the program (the system will run out of batteries before program execution is completed.

3. ENERGY MODELING OF BLOCKS

As mentioned before, the first step of our energy bounds analysis is to determine upper and lower bounds on the energy consumption of each basic ("branchless") program block. We perform the modeling at this level rather than at the instruction level in order to cater for inter-instruction dependencies. In order to determine such bounds first all the basic blocks of the program are identified, and then the energy consumption of each of these blocks is profiled for different input data using an EA. These steps are explained in the following sections.

3.1 Generating the Basic Blocks to be Modeled

A *basic block* over an inter-procedural control flow graph (CFG) is a maximal sequence of distinct instructions, S_1 through S_n, such that all instructions $S_k, 1 < k < n$ have exactly one in-edge and one out-edge (excluding call/return edges), S_1 has one out-edge, and S_n has one in-edge. A basic block therefore has exactly one entry point at S_1 and one exit point at S_n.

In order to divide a program into such *basic blocks* (for which an upper bound on the energy consumption of the program will be determined using the EA), the program is first compiled to the lower representation, ISA in our case. A data flow analysis of the ISA representation yields an inter-procedural control flow graph (CFG). A final control flow analysis is carried out to infer *basic blocks* from the CFG. These basic blocks are further modified so that they can be run and measured independently by the EA. Modifications for each basic block include:

1. A basic block with k function call instructions is divided into $k+1$ basic blocks without the function call instructions.

2. A number of special ISA instructions (e.g., *return*, *call*) are omitted from the block. The cost of such instructions is measured separately and added to the cost of the block.

3. Memory read/write instructions are abstracted to a fixed memory region available to each basic block in order to avoid memory violations.

An example of the modification 1 above is shown in Figure 1, Listing 1, which is an ISA representation of a recursive factorial program where the instructions are grouped together into 3 *basic blocks* $B1$, $B2$, and $B3$. Consider *basic block* $B2$. Since it has a (recursive) function call to *fact* at address 12, it is divided further into two blocks in Listing 2, such that the instructions before and after the function call form two blocks $B2_1$ and $B2_2$, respectively. The energy consumption of these two blocks is maximized (minimized) by providing values to the input arguments to the block (see below) using the EA. The energy consumption of $B2$ can then be characterized as:

$$B2_e^A = B2_{1e}^A + B2_{2e}^A + bl_e^A$$

where $B2_{1e}^A$, $B2_{2e}^A$, and bl_e^A denote the energy consumption of the $B2_1$, $B2_2$ blocks and the *bl* ISA instruction, with approximation A (where A=upper or A=lower).

For each modified basic block, a set of input arguments is inferred. This set is used for an individual representation to drive the EA algorithm to maximize the energy consumption of the block. For the entry block, the input arguments are derived from the signature of the function. The set $gen(B)$ characterizes the set of variables read without being previously defined in block B. It is defined as:

$$gen(b) = \bigcup_{k=1}^{n} \{v \mid v \in ref(k) \wedge \forall (j < k).v \notin def(j)\}$$

where $ref(n)$ and $def(n)$ denote the variables referred to and defined/updated at a node n in block b respectively.

For the basic blocks in Figure 1 in Listing 1, the set of input arguments are $gen(B1)$={r0}, $gen(B2_1)$={sp[0x1]}, $gen(B2_2)$={sp[0x1],r0} and $gen(B3) = \emptyset$.

3.2 EA for Estimating the Energy of a Basic Block

In the following we detail the most important aspects of the EA used for estimating the maximal (i.e., worst case) and minimal (i.e., best case) energy consumption of a basic block. The only difference between the two algorithms is the way we interpret the objective function: in the first case we want to maximize it, while in the second we want to minimize it.

Individual. The search space dimensions are the different input variables to the blocks. Our goal is to find the combination of input values which maximizes (minimizes) the energy of each block. The set of input variables to a block is inferred using a dataflow analysis (explained in the next section). Thus, an individual is simply an array of input values given in the order of their appearance in the block. The input values in an individual are coded as integers, since they represent 32-bit values stored in different hardware registers.

The majority of individuals are initialised with random 32-bit numbers. However, we also include corner cases to the initial population that are known to cause high (low) energy consumption for particular instructions. For example all 1s for high energy consumption, or all 0s for low energy consumption as operands to a multiply ISA instruction. This speeds up the EA algorithm in finding inputs to some basic blocks that maximize/minimize their overall energy consumption.

Crossover. The crossover operation is implemented as an even-odd crossover, since it provides more variability than a standard n-point crossover. In this crossover the first child is created by taking the first element and every other one after it from one of the parents, e.g., the mother. The second element and every other one come from the other parent, i.e., the father. The second child is created in the opposite way: the first element and every other one after it are taken from the father, while the second and every other one come from the mother. The process is depicted in Figure 2, where $P1$ and $P2$ are the parents, and $C1$ and $C2$ are their children created by the crossover operation.

Mutation. For the purpose of this work we have created a custom mutation operator. Since the energy consumption in digital circuits is mainly the result of bit flipping, we believe that the most optimal way to explore the search space is by performing some bit flipping in the mutation operation. This is implemented in the following way. For each gene (i.e., input value to the basic block):

1. We create a random 32-bit integer, i.e., a random mask.

2. Then we perform the XOR operation of that integer and the corresponding gene. This way, we perform random flipping of the bits of each gene, since we only flip the bits of the gene at positions where the value of the random mask is 1.

The process is depicted in Figure 3, where the input values are given as binary numbers.

Objective function. The objective function that we want to maximize (minimize) is the energy of a basic block, which is measured

Listing 1: Basic blocks of a factorial function.

```
<fact>:
B1  01: entsp 0x2
    02: stw   r0, sp[0x1]
    03: ldw   r1, sp[0x1]
    04: ldc   r0, 0x0
    05: lss   r0, r0, r1
    06: bf    r0, <08>

B2  07: bu    <010>
    10: ldw   r0, sp[0x1]
    11: sub   r0, r0, 0x1
    12: bl    <fact>
    13: ldw   r1, sp[0x1]
    14: mul   r0, r1, r0
    15: retsp 0x2

B3  08: mkmsk r0, 0x1
    09: retsp 0x2
```

Listing 2: Modified basic blocks.

```
<fact>:
 01: entsp 0x2                    } B1
 02: stw   r0, sp[0x1]
 03: ldw   r1, sp[0x1]
 04: ldc   r0, 0x0
 05: lss   r0, r0, r1
 06: bf    r0, <08_NEW>
08_NEW:

 07: bu    <010>                  } B2_1
 10: ldw   r0, sp[0x1]
 11: sub   r0, r0, 0x1

 12: bl <fact>

 13: ldw   r1, sp[0x1]            } B2_2
 14: mul   r0, r1, r0
 15: retsp 0x2

 08: mkmsk r0, 0x1                } B3
 09: retsp 0x2
```

block before call

block after call

Figure 1: Example: Basic block modifications.

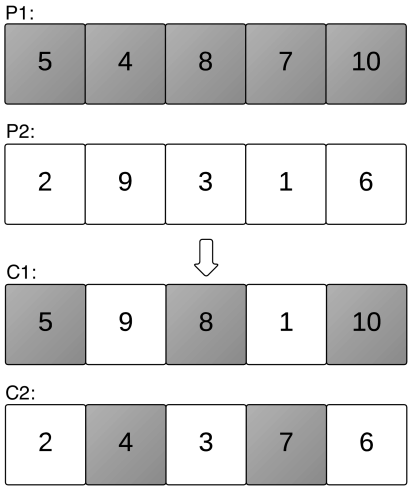


Figure 2: Example of even-odd crossover.

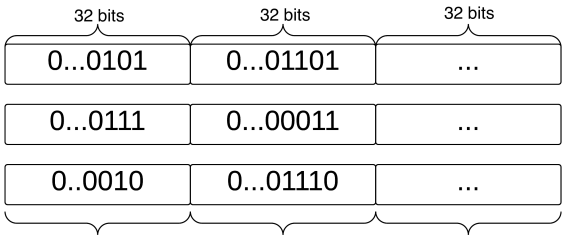


Figure 3: Mutation.

directly from the chip. The concrete setting of the experiment will be explained in the following section.

In general, pipeline effects such as stalls (to resolve pipeline hazards), which depend on the state of the processor at the start of the execution of a basic block, can affect the upper/lower bound estimated on the energy consumption of such block. In our approach intra-block pipeline effects are accounted for, since, the dependences among the instructions within a block are preserved. However, the inter-block pipeline effects need to be accounted for. These can be modeled in a conservative way by assuming a maximum stall penalty for the upper bound estimation of each block (e.g., by adding a stall penalty, say three cycles, to the execution time of the block). Similarly, for the lower bound estimation a zero stall penalty can be used. To approximate this effect, in [3], the authors characterize each block through pairwise executions with all of its possible predecessors. Each basic block pair is characterized by executing it on an Instruction Set Simulation (ISS) to collect cycle counts.

The XMOS XS1 architecture used in our experiments does not have these pipeline effects by design, since exactly one instruction per thread is executed in a 4-stage pipeline (more details in Section 5.1).

4. ENERGY CONSUMPTION OF THE PROGRAM

Once the energy models of each basic block of the program are known, the energy consumption of the whole program is bounded by a static analyser that takes into account the control flow of the program and infers safe upper/lower bounds on its energy consumption. We have implemented such analyser by specialising the generic resource analysis framework provided by CiaoPP [24] for programs written in the XC programming language [28] and running on the XMOS XS1-L architecture. We have also written the necessary code (i.e., assertions [7]) to feed such analyser with the block-level upper/lower bound energy model obtained by using the technique explained in Section 3.

The generic resource analyser ensures that the inferred bounds are strict/safe if it is fed with energy models providing safe bounds. As mentioned in the introduction, in [13] we performed a previous instantiation of such generic analyser by using the instruction-level energy model described in [10], which provided average energy

values. As a result, the analysis inferred an upper-bound energy function for the whole program that was an approximation of the actual upper bound, and could possibly be below it.

The analysis is general enough to be applied to other programming languages and architectures (see [13, 12] for details) provided that energy models for each architecture exist. It enables a programmer to symbolically bound the energy consumption of a program P on input data $\bar{x}$ without actually running $P(\bar{x})$. It is based on setting up a system of recursive cost equations over a program P that capture its cost (energy consumption) as a function of the sizes of its input arguments $\bar{x}$. The transformation-based analysis framework of [13, 12] transforms the assembly (or LLVM IR) representation of the program into an intermediate semantic program representation (HC IR), that the analysis operates on, which is a series of connected code blocks, represented as Horn Clauses. The analyser deals with this HC IR always in the same way, independent of where it originates from, setting up cost equations for all code blocks (predicates).

Consider the example in Listing 1. The recursive cost equations are set up over the function *fact* that characterize the energy consumption of the whole function using the approximation A of each block inferred by the EA:

$$fact_e^A(R0) \quad = \quad B1_e^A + fact\_aux_e^A(0 \leq R0, R0)$$

$$fact\_aux_e^A(B, R0) = \begin{cases} B2_e^A + fact_e^A(R0-1) & \text{if } B \text{ is } \texttt{true} \\ B3_e^A & \text{if } B \text{ is } \texttt{false} \end{cases}$$

The cost of the $fact$ function is captured by the equation $fact_e^A(R0)$ under an approximation A (e.g., upper/lower) which in turn depends on $B1_e^A$ (i.e., the energy consumption of block $B1$) and the equation $fact\_aux_e^A$, which represents the branching originated from the last instruction of block *B1*. It captures the cost of blocks $B2$ and $B3$ based on the condition on the input size $R0$.

If we assume (for simplicity of exposition) that each basic block has unitary cost in terms of energy consumption, i.e., $Bi_e = 1$ for all i, we obtain the energy consumed by `fact` as a function of its input data size ($R0$): $fact_e(R0) = R0 + 1$.

The functions inferred by the static analysis are arithmetic functions (polynomial, exponential, logarithmic, etc.) that depend on input data sizes (natural numbers).

5. EXPERIMENTAL EVALUATION

In this Section we report on an experimental evaluation of our approach to inferring both upper and lower bounds on the energy consumed by program executions, given as functions on input data sizes. The experiments have been performed with programs written in XC running on the XMOS XS1-L architecture. However, as already said, our approach is general enough be applied to the analysis of other programming languages (and associated lower level program representations) and architectures as well.

5.1 Evaluation Platform

We use a hardware and software platform created by XMOS that enables us to measure the energy [19], time, and power used during program executions on real hardware. The developed board is a dual-tile board that contains an XS1-A16-128-FB217 processor. The board is fed with a 3.3 V power supply, and supports voltage scaling, although both tiles have to run at the same voltage supply. It also supports frequency scaling, where the tiles can have different frequencies. The XMOS XS1 [17] is a cache-less, predictable architecture by design and manages threads on the hardware. The threads are executed in a round-robin fashion, using a 4-stage pipeline which only permits a single instruction per thread to be active within the pipeline at the same time. This restriction avoids pipeline hazards.

In order to support the process of measuring power, the following has been implemented:

- An extension to the XMOS toolchain that allows power measurements to be recorded and/or displayed in real time. In essence, a small shunt resistor has been added in series with the voltage supply. By measuring the voltage drop on the shunt, we can calculate the current I, which is also the current of the voltage supply, since the shunt is connected in series. In this way, we estimate the power consumption as $V_{sup} \cdot I$, where V_{sup} is the voltage of the power supply.
- A variant of the XTAG-2 debug adapter (called XTAG3) that enables power to be measured [31]. Basically, it has an extra connector that carries analog signals necessary to estimate the power consumption, as explained above. The measurements regarding these signals are transported to the host computer over USB using the xSCOPE interface [32]. In addition, a protocol that enables power measurements and application probing to be performed simultaneously, and data to be transported simultaneously over the USB connection to the host computer, has been designed.

The tool that collects data from the XTAG is `xgdb`, the debugger that is part of the XMOS toolchain. xgdb connects to the XTAG over a USB interface (using libusb), and reads both ordinary xSCOPE traffic and voltage/current measurements. The collected data is normally stored in an XML file, or instead, `xgdb` can pipe the data directly into an analysis program that can only record data that is relevant (between start and end) and only compute the relevant metrics (maximum current, total energy, etc.).

5.2 Results and Discussion

The aim of the experimental evaluation is to perform a first comparison of actual hardware energy measurements against the upper- and lower-bounds on energy consumption obtained by evaluating the functions inferred by our proposed approach (which depend on input data sizes), for each program considered and for different input data sizes. The actual energy consumption of the programs, for each value of input data sizes, is measured with the evaluation platform, i.e., the same used to build the upper- and lower-bound models of the blocks of each program.

Program	Upper/Lower Bounds (nJ)$\times 10^3$	vs. HW
$fact(N)$	$ub = 5.1\,N + 4.2$	7%
	$lb = 4.1\,N + 3.8$	-11.7%
$fibonacci(N)$	$ub = 5.2\,lucas(N)$[1]$+6\,fib(N) - 6.6$	8.71%
	$lb = 4.5\,lucas(N) + 5\,fib(N) - 4.2$	-4.69%
$reverse(N)$	$ub = 3.7\,N + 13.3$	8%
	$lb = 2.95\,N + 12$	-8.8%
$findMax(N)$	$ub = 5\,N + 6.9$	8.7%
	$lb = 3.3\,N + 5.6$	-9.1%
$fir(N)$	$ub = 6\,N + 26.4$	8.9%
	$lb = 4.8\,N + 22.9$	-9.7%
$biquad(N)$	$ub = 29.6\,N + 10$	9.8%
	$lb = 23.5\,N + 9$	-11.9%

Table 1: Upper and lower bounds accuracy.

[1]The mathematical function $lucas(n)$ satisfies the recurrence relation $lucas(n) = lucas(n-1) + lucas(n-2)$ with $lucas(1) = 1$ and $lucas(2) = 3$.

A number of selected benchmarks are shown in Table 1 that are either iterative or recursive. The **Upper/lower Bounds** column depicts the energy estimation functions (on input data sizes) for upper and lower bounds. The column **vs. HW** shows the average over- and under-approximations of the estimation versus the actual measurements on the hardware.

The first two benchmarks are small arithmetic benchmarks. The third benchmark $reverse(N)$ reverses elements of an input array of size N. The list of benchmarks also includes two filter benchmarks, namely *biquad* and fir (Finite Impulse Response). Both programs attenuate or amplify specific frequency ranges of a given input signal. The $fir(N)$ benchmark computes the inner-product of two vectors: a vector of input samples, and a vector of coefficients. The more coefficients, the higher the fidelity, and the lower the frequencies that can be filtered. The $biquad(N)$ benchmark is an equaliser, i.e., it takes a signal and attenuates/amplifies different frequency bands. It uses a cascade of Biquad filters where each filter attenuates or amplifies one specific frequency range. The energy consumed depends on the number of banks N, typically between 3 and 30 for an audio equaliser. A higher number of banks enables a designer to create more precise frequency response curves. A simple $findMax$ benchmark (finding the maximum number in an array) is also included in the list. This is a program where data-dependent branching can bring significant variations of the worst (best) case energy consumption. Note that unlike the first three benchmarks, fir, *biquad*, and $findMax$ all have data-dependent branches.

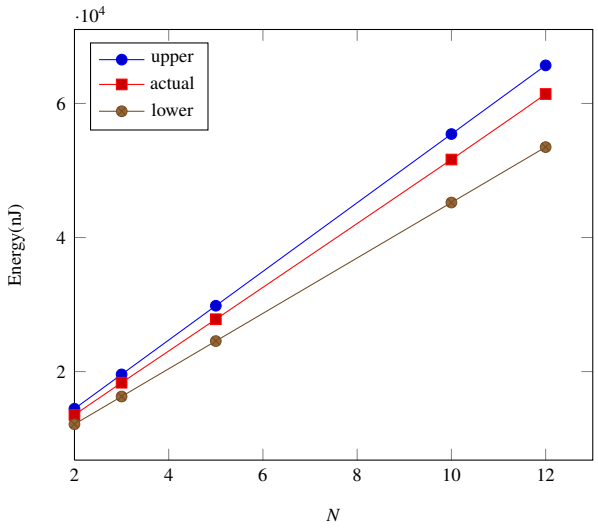


Figure 4: $fact$ **upper/lower bounds vs. actual measurement.**

Figure 4 depicts the upper/lower bound inferred by the analysis vs. the actual measurement on the hardware for the factorial program. The actual program consumption is measured for several values of N, the input value, resulting in the middle curve. The other two curves are the result of plotting the upper- and lower-bound energy functions for different sizes (in the case of an integer, its size is its value).

The upper bound values inferred by the static analysis and the EA over-approximate the actual hardware measurements by 7%, whereas the lower-bound values under-approximate the actual measurements by 11.7% –see Table 1.

The $findMax$ benchmark, which, as mentioned before, has significant data dependent branching, is shown in Figure 5. Unlike $fact$, the upper and lower-bounds in $findMax$ are more distant due

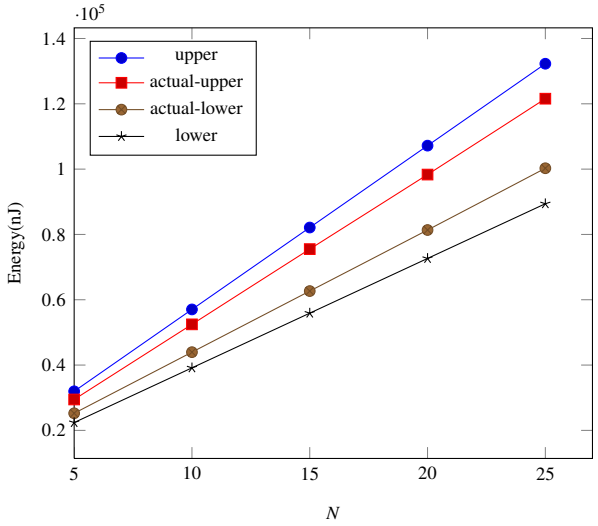


Figure 5: $findMax$ **example upper/lower bounds vs actual-lower and upper energy consumption measurement based on data.**

to the data sensitive branching. A call to $findMax$ with a sorted array in ascending order will discover a new *max* element in each iteration and hence update the current *max* variable resulting in an actual worst case of the algorithm. In contrast, if the array is sorted in descending order then the algorithm will find the *max* element in the first iteration and the rest of the iterations will never update the current *max* variable, resulting in the actual best case.

Figure 5 depicts the upper and lower bounds inferred by the static analysis as well as the actual worst and best case measurements of $findMax$ (first with ascending order and then with descending order array data). The upper and lower bounds inferred are compared against the actual worst and best case measurements. The upper bound over-approximates by 8.7% whereas the lower bound under-approximates by 9.1%. Note that it is not always trivial to find data that exhibit program worst and best case behaviors.

In Table 2, different executions of the $findMax$ benchmark are shown for particular input sizes **N** but using a random data input array in one case and actual worst/best case array input data in the other case. Column **N** shows the size of the input array. Column **Cost App** indicates the type of approximation of the automatically inferred cost functions which estimate energy consumption (depending on input data size **N**): upper bound (**U**) and lower bound (**L**). Such energy functions for the $findMax$ benchmark are shown in Table 1. In order to assess the accuracy of the cost functions we have evaluated them for particular sets of input data corresponding to different input (array) sizes (N), yielding different energy consumption estimations. We have then compared such estimations (column **Est**) with the observed energy consumptions of the hardware measurements (column **Obs**). Column **D** shows the relative harmonic difference between the estimated and the observed energy consumption, given by the formula:

$$rel\_harmonic\_diff(Est, Obs) = \frac{(Est - Obs) \times (\frac{1}{Est} + \frac{1}{Obs})}{2}$$

The inaccuracies in the energy estimations of our technique come mainly from two sources: the energy model, which assigns an energy value to each basic block as described in Section 3, and the Static Resource Analysis (SRA), described in Section 4, which estimates the number of times that the basic blocks are executed de-

pending on the input data sizes.

In order to investigate the source(s) of inaccuracies, we have also introduced Column **Prof**. It shows the result of estimating the energy consumption using the energy model and assuming that the SRA was perfect and estimated the exact number of times that the basic blocks were executed. This obviously represents the case in which all loss of accuracy must be attributed to the energy model. The values in Column **Prof** have been obtained by profiling actual executions of the program with particular input data, where the profiler has been instrumented to record the number of times each basic block is executed. The energy consumption of the program is then obtained by multiplying such numbers by the energy values provided by the energy model for each basic block, and adding all of them. Column **PrD** shows the relative harmonic difference between **Prof** and the observed energy consumption **Obs**, which represents the inaccuracy due to the energy modeling of basic blocks using the EA.

In the case of random data, both the SRA and the energy modeling contribute to the inaccuracy of the energy estimation for the whole program. In contrast, in the second case two sets of array data are used: one that makes $findMax$ exhibit its worst case behavior and another that makes it exhibit the best. These are then compared against the upper- and lower-bound estimations. Since Columns **Est** and **Prof** show the same values in this case, it means that there was no inaccuracy due to the SRA, and that the overall inaccuracy is due to the over- and under-approximation in the EA to model energy consumption of each basic block. In other words, the analysis of the $findMax$ program provides accurate bounds for each data size N.

N	Cost App	Energy(nJ)$\times 10^3$ Est	Prof	Obs	D %	PrD %
Random array data						
5	L	22.3	24.9	27.3	-20.1	-9.2
	U	31.9	30.2		15.6	10
15	L	55.9	61.8	69.1	-17	-11
	U	82.1	75.1		21	8.3
25	L	89.4	99.6	110.9	-17.6	-10.7
	U	132.2	120.8		21.7	8.5
Actual worst- and best-case array data						
5	L	22.3	22.3	25.2	-12.2	-12.2
	U	31.9	31.9	29.4	8.1	8.1
15	L	55.9	55.9	62.6	-11.3	-11.3
	U	82.1	82.1	75.5	8.3	8.3
25	L	89.4	89.4	100.2	-11.4	-11.4
	U	132.2	132.2	121.5	8.4	8.4

Table 2: $findMax$: Source of inaccuracies in prediction: static analysis vs. energy modeling.

Regarding the time taken by the EA, it can vary depending on the parameters it is initialized with, as well as the initial population. This population is different every time the EA is initiated, except for a fixed number of individuals that represent corner cases. In the experiments, the EA is run for up to a maximum of 20 generations, and is stopped when the fitness value does not improve for four consecutive generations. In all the experiments the *biquad* benchmark took the most time (a maximum time of 230 minutes) for maximizing the energy consumption. In contrast, the *fact* benchmark took the least time (a maximum time of 121 minutes). The times remained within the 150-200 minutes range on average. Time speedups were also achieved by reusing the EA results for sequences of instructions that were already processed in a previous benchmark (e.g., return blocks, loop header blocks, etc.). This makes us believe that our approach could be used in practice in an iterative development process, where the developer gets feedback from our tool and modifies the program in order to reduce its energy consumption. The first time the EA is run would take the highest time, since it would have to determine the energy consumption of all the program blocks. After a focused modification of the program that only affects a small number of blocks, most of the results from the previous run could be reused, so that the EA would run much faster during this development process. In other words, the EA processing can easily be made incremental.

The static analysis, on the other hand, is quite efficient, with analysis times of about 4 to 5 seconds on average, despite the naive implementation of the interface with external recurrence equation solvers, which can be improved significantly.

6. RELATED WORK

Static dataflow analysis of the energy consumed by program executions has received relatively little attention until recently. An analysis of Java bytecode programs for inferring upper-bounds on energy consumption as functions on input data sizes was proposed in [20], where the Jimple (a typed three-address code) representation of Java bytecode was transformed into Horn Clauses, and a simple energy model at the Java bytecode level [11] was used. However the energy model used average estimations of the Java opcodes and an opcode cost verification found the estimation to be between -5% and 10%. Furthermore, this work did not compare the results with actual, measured energy consumption. A similar approach was proposed in [13] for the analysis of a C-based programming language. It performed a transformation of the assembly code generated by the compilation of the source program into Horn Clauses, which were then analyzed by using the accurate assembly level energy models presented in [10]. The experiments, performed for a number of small numerical programs, showed for the first time that energy bound functions inferred statically from low-level model could be inferred that provided energy consumption estimations were reasonably accurate with respect to actual executions for any input data size.

Similarly to the work presented here, the approaches mentioned above used instantiations for energy consumption of general resource analyzers, namely [21] in [20] and [13], and [24] in [12] and this paper. Such resource analyzers are based on setting up and solving recurrence equations, an approach proposed by Wegbreit [29] that has been developed significantly in subsequent work [23, 4, 5, 26, 21, 1, 24]. Other approaches to static analysis based on the transformation of the analyzed code into another (intermediate) representation have been proposed for analyzing low-level languages [6] and Java (by means of a transformation into Java bytecode) [2]. In [2], cost relations are inferred directly for these bytecode programs, whereas in [20] the bytecode is first transformed into Horn Clauses. The general resource analyzer in [21] was also instantiated in [18] for the estimation of execution times of logic programs running on a bytecode-based abstract machine. The approach used timing models at the bytecode instruction level, for each particular platform, and program-specific mappings to lift such models up to the Horn Clause level, at which the analysis was performed.

Other work has taken as its starting point techniques referred to generally as "WCET" (worst case execution time analyses), which have been applied, usually for imperative languages, in different application domains (see e.g., [30] and its references). These techniques generally require the programmer to bound the number of iterations of loops, and then apply an Implicit Path Enumeration

technique to identify the path of maximal consumption in the control flow graph of the resulting loop-less program. This approach has inspired some worst case energy analyses, such as the one presented in [9]. It distinguishes instruction-specific (not proportional to time, but to data) from pipeline-specific (roughly proportional to time) energy consumption. The approach also takes into account complex issues such as branch prediction and cache misses. However, they rely on the user to identify the input which will trigger the maximal energy consumption. In [27] the same approach is applied and further refined for estimating *hard* (i.e., over-approximated) energy bounds. The main novelty of this work consists in introducing relative energy models (implemented at the LLVM level in this case), where the energy of each instruction is given *in relation to each other* (e.g., if we assume that all the instructions have relative energy 1, this means that they all have the same absolute energy), which does not depend on the specific hardware, but can be applied for all the platforms where a mapping between LLVM and low-level assembly instructions exists. On the other hand, in the situations when the energy bounds are not *hard* (i.e., the application allows their violation) they use a genetic algorithm to obtain an under-approximation of the energy bounds. However, this approach loses accuracy when there are data dependent branches present in the program, since different inputs can lead to the execution of different set of instructions.

A similar approach is used in [22] to find the worst-case energy consumption of two benchmarks using a genetic algorithm. In contrast to our approach, the evolutionary algorithm is applied to whole programs, and these do not have any data-dependent branching. The authors further introduce probability distributions for the transition costs among pairs of independent instructions, which can be then be convolved to give a probability distribution of the energy for a sequence of instructions.

In contrast to the work presented here and in [18], all these WCET-style methods (either for execution time or energy) do not infer cost functions on input data sizes but rather absolute maximum values, and, as mentioned before, they generally require the manual annotation of all loops to express an upper bound on the number of iterations, which can be tedious (or impossible) and effectively reduces the case to that of programs with no loops.

Another alternative approach to WCET-style methods was presented in [8]. It is based on the idea of amortization, which allows inferring more accurate yet safe upper bounds by averaging the worst execution time of operations over time. It was applied to a functional language, but the approach is in principle generally applicable. A timing analysis based on game-theoretic learning was presented in [25]. The approach combines static analysis to find a set of basic paths which are then tested. Its main advantage is that it can infer distributions on time, not only average values. In principle, both approaches could be adapted to infer energy usage.

7. CONCLUSIONS

We have proposed an approach for inferring parametric upper and lower bounds on the energy consumption of a program using a combination of static and dynamic techniques. The dynamic technique, based on an evolutionary algorithm, is used to determine the maximal / minimal energy consumption of each basic block. Such blocks contain multiple instructions, which allows this phase to take into account inter-instruction dependencies. Since such basic blocks are branchless, the evolutionary algorithm approach is more practical and efficient and the technique infers energy values that are accurate since no control flow-related variations occur. A static analysis is then used to combine the energy values obtained for the blocks according to the program control flow, and produce energy consumption bounds of the whole program. We also carried out an experimental evaluation to validate the upper and lower bounds on a set of benchmarks. The results support our hypothesis that the bounds inferred in this way are indeed safe and quite accurate, and the technique practical.

8. ACKNOWLEDGMENTS

This research has received funding from the European Union 7th Framework Program agreement no 318337, ENTRA, Spanish MINECO TIN'12-39391 *StrongSoft* project, and the Madrid M141047003 *N-GREENS* program. We also thank Henk Muller, Principal Technologist, XMOS, for his help with the measurement boards, evaluation platform, benchmarks, and overall support.

9. REFERENCES

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

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[3] S. Chakravarty, Z. Zhao, and A. Gerstlauer. Automated, Retargetable Back-annotation for Host Compiled Performance and Power Modeling. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 36:1–36:10, USA, 2013. IEEE Press.

[4] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[5] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[6] K. S. Henriksen and J. P. Gallagher. Abstract interpretation of PIC programs through logic programming. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 184–196. IEEE Computer Society, 2006.

[7] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.

[8] C. Herrmann, A. Bonenfant, K. Hammond, S. Jost, H.-W. Loidl, and R. Pointon. Automatic Amortised Worst-Case Execution Time Analysis. In *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OASIcs*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007.

[9] R. Jayaseelan, T. Mitra, and X. Li. Estimating the Worst-Case Energy Consumption of Embedded Software. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, pages 81–90. IEEE Computer Society, 2006.

[10] S. Kerrison and K. Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):1–25, April 2015.

[11] S. Lafond and J. Lilius. Energy consumption analysis for two embedded Java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.

[12] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR. In *Proc. of the Foundational and Practical Aspects of Resource Analysis*, LNCS. Springer, 2015. To appear.

[13] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.

[14] P. López-García, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In M. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 104–113, Dagstuhl, Germany, July 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo. Interval-Based Resource Usage Verification: Formalization and Prototype. In R. P. na, M. Eekelen, and O. Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis. Second Iternational Workshop FOPARA 2011, Revised Selected Papers*, volume 7177 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 2012.

[16] P. Lopez-Garcia, R. Haemmerlé, M. Klemen, U. Liqat, and M. V. Hermenegildo. Towards Energy Consumption Verification via Static Analysis. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2015), arXiv: 1501.03064*, 2015.

[17] D. May. The XMOS XS1 architecture. available online: http://www.xmos.com/published/xmos-xs1-architecture, 2013.

[18] E. Mera, P. López-García, M. Carro, and M. Hermenegildo. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[19] H. Muller, editor. *Metrics and Case Studies*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 6.1, http://entraproject.eu.

[20] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32, April 2008. Extended Abstract.

[21] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP'07)*, Lecture Notes in Computer Science, pages 348–363. Springer, 2007.

[22] J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data dependent energy modeling for worst case energy consumption analysis. *arXiv preprint arXiv:1505.03374*, 2015.

[23] M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156. ACM Press, 1989.

[24] A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754, 2014.

[25] S. A. Seshia and J. Kotker. Gametime: A toolkit for timing analysis of software. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 388–392. Springer, 2011.

[26] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *15th International Workshop on Implementation of Functional Languages (IFL'03), Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Sep 2005.

[27] P. Wagemann, T. Distler, T. Honig, H. Janker, R. Kapitza, and W. Schroder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 105–114, July 2015.

[28] D. Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.

[29] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[30] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - Overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[31] XMOS. The XTAG-2 Hardware Manual, September 2009. Available online at: https://www.xmos.com/download/private/XTAG-2-Hardware-Manual

[32] XMOS. Use xTIMEcomposer and xSCOPE to trace data in real-time, 2013. Available online at: https://www.xmos.com/download/public/Trace-data-with-XScope(X9923H).pdf.

Attachment D3.3.2

Towards Energy Consumption Verification via Static Analysis

Published at the Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2015)

Towards Energy Consumption Verification via Static Analysis

P. Lopez-Garcia [*] [†]
pedro.lopez@imdea.org

R. Haemmerlé [†]
remy.haemmerle@imdea.org

M. Klemen [†]
maximiliano.klemen@imdea.org

U. Liqat [†]
umer.liqat@imdea.org

M. Hermenegildo [† ‡]
manuel.hermenegildo@imdea.org

ABSTRACT

In this paper we leverage an existing general framework for resource usage verification and specialize it for *verifying* energy consumption specifications of embedded programs. Such specifications can include both lower and upper bounds on energy usage, and they can express intervals within which energy usage is to be certified to be within such bounds. The bounds of the intervals can be given in general as functions on input data sizes. Our verification system can prove whether such energy usage specifications are met or not. It can also infer the particular conditions under which the specifications hold. To this end, these conditions are also expressed as intervals of functions of input data sizes, such that a given specification can be proved for some intervals but disproved for others. The specifications themselves can also include preconditions expressing intervals for input data sizes. We report on a prototype implementation of our approach within the CiaoPP system for the XC language and XS1-L architecture, and illustrate with an example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

Keywords: Energy consumption analysis and verification, resource usage analysis and verification, static analysis, verification.

1. INTRODUCTION

In an increasing number of applications, particularly those running on devices with limited resources, it is very important and sometimes essential to ensure conformance with respect to specifications expressing non-functional global properties such as energy consumption, maximum execution time, memory usage, or user-defined resources. For example, in a real-time application, a program completing an action later than required is as erroneous as a program not computing the correct answer. The same applies to an embedded application in a battery-operated device (e.g., a portable or implantable medical device, an autonomous space vehicle, or even a mobile phone) if the application makes the device run out of batteries earlier than required, making the whole system useless in practice.

In general, high performance embedded systems must control, react to, and survive in a given environment, and this in turn establishes constraints about the system's performance parameters including energy consumption and reaction times. Therefore, a mechanism is necessary in these systems in order to prove correctness with respect to specifications about such non-functional global properties.

To address this problem we leverage an existing general framework for resource usage analysis and verification [22, 23], and specialize it for *verifying* energy consumption specifications of embedded programs. As a case study, we focus on the energy verification of embedded programs written in the XC language [37] and running on the XMOS XS1-L architecture (XC is a high-level C-based programming language that includes extensions for communication, input/output operations, real-time behavior, and concurrency). However, the approach presented here can also be applied to the analysis of other programming languages and architectures. We will illustrate with an example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

2. OVERVIEW OF THE ENERGY VERIFICATION TOOL

In this section we give an overview of the prototype tool for *energy consumption verification* of XC programs running on the XMOS XS1-L architecture, which we have implemented within the CiaoPP system [13]. As in previous work [20, 25], we differentiate between the *input language*, which can be XC source, LLVM IR [17], or Instruction Set Architecture (ISA) code, and the *intermediate semantic program representation* that the CiaoPP core components (e.g., the analyzer) take as input. The latter is a series of connected code blocks, represented by Horn Clauses, that we will refer to as "HC IR" from now on. We perform a transformation from each *input language* into the HC IR and pass it to the corresponding CiaoPP component. The main reason for choosing Horn Clauses as the intermediate representation is that it offers a good number of features that make it very convenient for the analysis [25]. For instance, it supports naturally Static Single Assignment (SSA) and recursive forms, as will be explained later. In fact, there is a current trend favoring the use of Horn Clause programs

[*]Spanish Council for Scientific Research (CSIC).

[†]IMDEA Software Institute, Madrid, Spain.

[‡]Universidad Politécnica de Madrid (UPM).

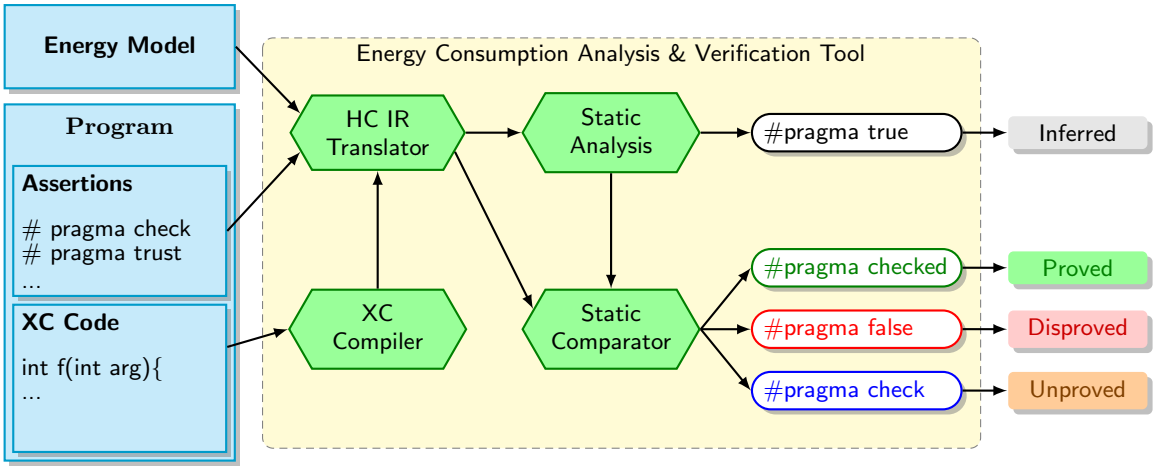


Figure 1: Energy consumption verification tool using CiaoPP.

as intermediate representations in analysis and verification tools [2].

Figure 1 shows an overview diagram of the architecture of the prototype tool we have developed. Hexagons represent different tool components and arrows indicate the communication paths among them.

The tool takes as input an XC source program (left part of Figure 1) that can optionally contain assertions in a C-style syntax. As we will see later, such assertions are translated into Ciao assertions, the internal representation used in the Ciao/CiaoPP system.

The energy specifications that the tool will try to prove or disprove are expressed by means of assertions with `check` *status*. These specifications can include both lower and upper bounds on energy usage, and they can express intervals within which energy usage is to be certified to be within such bounds. The bounds of the intervals can be given in general as functions on input data sizes. Our tool can prove whether such energy usage specifications are met or not. It can also infer the particular conditions under which the specifications hold. To this end, these conditions are also expressed as intervals of functions of input data sizes, such that a given specification can be proved for some intervals but disproved for others.

In addition, assertions can also express trusted information such as the energy usage of procedures that are not developed yet, or useful hints and information to the tool. In general, assertions with status `trust` can be used to provide information about the program and its constituent parts (e.g., individual instructions or whole procedures or functions) to be trusted by the analysis system, i.e., they provide base information assumed to be true by the inference mechanism of the analysis in order to propagate it throughout the program and obtain information for the rest of its constituent parts.

In our tool the user can choose between performing the analysis at the ISA or LLVM IR levels (or both). We refer the reader to [19] for an experimental study that sheds light on the trade-offs implied by performing the analysis at each of these two levels, which can help the user to choose the level that fits the problem best.

The associated ISA and/or LLVM IR representations of the XC program are generated using the xcc compiler. Such representations include useful metadata. The *HC IR translator* component (described in Section 4) produces the internal representation used by the tool, HC IR, which includes the program and possibly specifications and/or trusted information (expressed in the Ciao assertion language [32, 15]).

The tool performs several tasks:

1. Transforming the ISA and/or LLVM IR into HC IR. Such transformation preserves the resource consumption semantics, in the sense that the resource usage information inferred by the tool is applicable to the original XC program.

2. Transforming specifications (and trusted information) written as C-like assertions into the Ciao assertion language.

3. Transforming the energy model at the ISA level [16], expressed in JSON format, into the Ciao assertion language. Such assertions express the energy consumed by individual ISA instruction representations, information which is required by the analyzer in order to propagate it during the static analysis of a program through code segments, conditionals, loops, recursions, etc., in order to infer analysis information (energy consumption functions) for higher-level entities such as procedures, functions, or loops in the program.

4. In the case that the analysis is performed at the LLVM IR level, the *HC IR translator* component produces a set of Ciao assertions expressing the energy consumption corresponding to LLVM IR block representations in HC IR. Such information is produced from a mapping of LLVM IR instructions with sequences of ISA instructions and the ISA-level energy model. The mapping information is produced by the *mapping tool* that was first outlined in [3] (Section 2 and Attachments D3.2.4 and D3.2.5) and is described in detail in [11].

Then, following the approach described in [20], the CiaoPP parametric static resource usage analyzer [30, 28, 34] takes

the HC IR, together with the assertions which express the energy consumed by LLVM IR blocks and/or individual ISA instructions, and possibly some additional (trusted) information, and processes them, producing the analysis results, which are expressed also using Ciao assertions. Such results include energy usage functions (which depend on input data sizes) for each block in the HC IR (i.e., for the whole program and for all the procedures and functions in it.). The analysis can infer different types of energy functions (e.g., polynomial, exponential, or logarithmic). The procedural interpretation of the HC IR programs, coupled with the resource-related information contained in the (Ciao) assertions, together allow the resource analysis to infer static bounds on the energy consumption of the HC IR programs that are applicable to the original LLVM IR and, hence, to their corresponding XC programs. Analysis results are given using the assertion language, to ensure interoperability and make them understandable by the programmer.

The verification of energy specifications is performed by a specialized component which compares the energy specifications with the (safe) approximated information inferred by the static resource analysis. Such component is based on our previous work on general resource usage verification presented in [22, 23], where we extended the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties, such as upper and lower bounds on execution time, memory, energy, or user defined resources, given as functions on input data sizes. We also defined an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program (i.e., the specification) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential, or logarithmic functions) that may come from the specifications or from the analysis results. As a possible result of the comparison in the output of the tool, either:

1. The original (specification) assertion (i.e., with status `check`) is included with status `checked` (resp. `false`), meaning that the assertion is correct (resp. incorrect) for all input data meeting the precondition of the assertion,

2. the assertion is "split" into two or three assertions with different status (`checked`, `false`, or `check`) whose preconditions include a conjunct expressing that the size of the input data belongs to the interval(s) for which the assertion is correct (status `checked`), incorrect (status `false`), or the tool is not able to determine whether the assertion is correct or incorrect (status `check`), or

3. in the worst case, the assertion is included with status `check`, meaning that the tool is not able to prove nor to disprove (any part of) it.

If all assertions are `checked` then the program is *verified*. Otherwise, for assertions (or parts of them) that get `false` status, a *compile-time error* is reported. Even if a program contains no assertions, it can be checked against the assertions contained in the libraries used by the program, potentially catching bugs at compile time. Finally, and most importantly, for assertion (or parts of them) left with status `check`, the tool can optionally produce a *verification warning* (also referred to as an "alarm"). In addition, optional run-time checks can also be generated.

3. THE ASSERTION LANGUAGE

Two aspects of the assertion language are described here: the front-end language in which assertions are written and included in the XC programs to be verified, and the internal language in which such assertions are translated into and passed, together with the HC IR program representation, to the core analysis and verification tools, the Ciao assertion language.

3.1 The Ciao Assertion Language

We describe here the subset of the Ciao assertion language which allows expressing global "computational" properties and, in particular, resource usage. We refer the reader to [32, 13, 15] and their references for a full description of this assertion language.

For brevity, we only introduce here the class of **`pred` assertions**, which describes a particular predicate and, in general, follows the schema:

`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-Props*]`.`

where *Pred* is a predicate symbol applied to distinct free variables while *Precond* and *Postcond* are logic formulae about execution states. An execution state is defined by variable/value bindings in a given execution step. The assertion indicates that in any call to *Pred*, if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* also holds in the success state. Finally, the *Comp-Props* field is used to describe properties of the whole computation for calls to predicate *Pred* that meet *Precond*. In our application *Comp-Props* are precisely the resource usage properties.

For example, the following assertion for a typical `append/3` predicate:

```
:- pred append(A,B,C)
   : (list(A,num),list(B,num),var(C))
  => (list(C,num),
      rsize(A,list(ALb,AUb,num(ANl,ANu))),
      rsize(B,list(BLb,BUb,num(BNl,BNu))),
      rsize(C,
       list(ALb+BLb,AUb+BUb,
         num(min(ANl,BNl),max(ANu,BNu)))))
   + resource(steps,ALb+1, AUb+1).
```

states that for any call to predicate `append/3` with the first and second arguments bound to lists of numbers, and the third one unbound, if the call succeeds, then the third argument will also be bound to a list of numbers. It also states that an upper bound on the number of resolution steps required to execute any of such calls is $AUb + 1$, a function on the length of list A. The `rsize` terms are the *sized types* derived from the regular types, containing variables that represent explicitly lower and upper bounds on the size of terms and subterms appearing in arguments. See Section 5 for an overview of the general resource analysis framework and how sized types are used.

The global non-functional property `resource/3` (appearing in the "`+`" field), is used for expressing resource usages and follows the schema:

`resource(`*Res_Name***`,`** *Low_Arith_Expr, Upp_Arith_Expr)*

where *Res_Name* is a user-provided identifier for the resource the assertion refers to, *Low_Arith_Expr* and *Upp_Arith_Expr* are arithmetic functions that map input data sizes to re-

source usage, representing respectively lower and upper bounds on the resource consumption.

Each assertion can be in a particular *status*, marked with the following prefixes, placed just before the `pred` keyword: `check` (indicating the assertion needs to be checked), `checked` (it has been checked and proved correct by the system), `false` (it has been checked and proved incorrect by the system; a compile-time error is reported in this case), `trust` (it provides information coming from the programmer and needs to be trusted), or `true` (it is the result of static analysis and thus correct, i.e., safely approximated). The default status (i.e., if no status appears before `pred`) is `check`.

3.2 The XC Assertion Language

The assertions within XC files use instead a different syntax that is closer to standard C notation and friendlier for C developers. These assertions are transparently translated into Ciao assertions when XC files are loaded into the tool. The Ciao assertions output by the analysis are also translated back into XC assertions and added inline to a copy of the original XC file.

More concretely, the syntax of the XC assertions accepted by our tool is given by the following grammar, where the non-terminal ⟨*identifier*⟩ stands for a standard C identifier, ⟨*integer*⟩ stands for a standard C integer, and the non-terminal ⟨*ground-expr*⟩ for a ground expression, i.e., an expression of type ⟨*expr*⟩ that does not contain any C identifiers that appear in the assertion scope (the non-terminal ⟨*scope*⟩).

⟨*assertion*⟩ ::= '`#pragma`' ⟨*status*⟩ ⟨*scope*⟩ '`:`' ⟨*body*⟩

⟨*status*⟩ ::= '`check`' | '`trust`' | '`true`' | '`checked`' | '`false`'

⟨*scope*⟩ ::= ⟨*identifier*⟩ '`(`' '`)`'
| ⟨*identifier*⟩ '`(`' ⟨*arguments*⟩ '`)`'

⟨*arguments*⟩ ::= ⟨*identifier*⟩ | ⟨*arguments*⟩ '`,`' ⟨*identifier*⟩

⟨*body*⟩ ::= ⟨*precond*⟩ '`==>`' ⟨*cost-bounds*⟩ | ⟨*cost-bounds*⟩

⟨*precond*⟩ ::= ⟨*upper-cond*⟩ | ⟨*lower-cond*⟩
| ⟨*lower-cond*⟩ '`&&`' ⟨*upper-cond*⟩

⟨*lower-cond*⟩ ::= ⟨*ground-expr*⟩ '`<=`' ⟨*identifier*⟩

⟨*upper-cond*⟩ ::= ⟨*identifier*⟩ '`<=`' ⟨*ground-expr*⟩

⟨*cost-bounds*⟩ ::= ⟨*lower-bound*⟩ | ⟨*upper-bound*⟩
| ⟨*lower-bound*⟩ '`&&`' ⟨*upper-bound*⟩

⟨*lower-bound*⟩ := ⟨*expr*⟩ '`<=`' '`energy`'

⟨*upper-bound*⟩ := '`energy`' '`<=`' ⟨*expr*⟩

⟨*expr*⟩ := ⟨*expr*⟩ '`+`' ⟨*mult-expr*⟩
| ⟨*expr*⟩ '`-`' ⟨*mult-expr*⟩

⟨*mult-expr*⟩ := ⟨*mult-expr*⟩ '`*`' ⟨*unary-expr*⟩
| ⟨*mult-expr*⟩ '`/`' ⟨*unary-expr*⟩

⟨*unary-expr*⟩ := ⟨*identifier*⟩
| ⟨*integer*⟩
| '`sum`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`prod`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩'`,`' ⟨*expr*⟩ '`)`'
| '`power`' '`(`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`log`' '`(`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`(`' ⟨*expr*⟩ '`)`'
| '`+`' ⟨*unary-expr*⟩
| '`-`' ⟨*unary-expr*⟩
| '`min`' '`(`' ⟨*identifier*⟩ '`)`'
| '`max`' '`(`' ⟨*identifier*⟩ '`)`'

XC assertions are directives starting with the token `#pragma` followed by the assertion *status*, the assertion *scope*, and the assertion *body*. The assertion *status* can take several values, including `check`, `checked`, `false`, `trust` or `true`, with the same meaning as in the Ciao assertions. Again, the default status is `check`.

The assertion scope identifies the function the assertion is referring to, and provides the local names for the arguments of the function to be used in the body of the assertion. For instance, the scope `biquadCascade(state, xn, N)` refers to the function `biquadCascade` and binds the arguments within the body of the assertion to the respective identifiers `state`, `xn`, `N`. While the arguments do not need to be named in a consistent way w.r.t. the function definition, it is highly recommended for the sake of clarity. The *body* of the assertion expresses bounds on the energy consumed by the function and optionally contains preconditions (the left hand side of the `==>` arrow) that constrain the argument sizes.

Within the body, expressions of type ⟨*expr*⟩ are built from standard integer arithmetic functions (i.e., `+`, `-`, `*`, `/`) plus the following extra functions:

- `power(base, exp)` is the exponentiation of `base` by `exp`;
- `log(base, expr)` is the logarithm of `expr` in base `base`;
- `sum(id, lower, upper, expr)` is the summation of the sequence of the values of `expr` for `id` ranging from `lower` to `upper`;
- `prod(id, lower, upper, expr)` is the product of the sequence of the values of `expr` for `id` ranging from `lower` to `upper`;
- `min(arr)` is the minimal value of the array `arr`;
- `max(arr)` is the maximal value of the array `arr`.

Note that the argument of `min` and `max` must be an identifier appearing in the assertion scope that corresponds to an array of integers (of arbitrary dimension).

4. ISA/LLVM IR TO HC IR TRANSFORMATION

In this section we describe briefly the HC IR representation and the transformations into it that we developed in order to achieve the verification tool presented in Section 2 and depicted in Figure 1. The transformation of ISA code into HC IR was described in [21]. We provide herein an overview of the LLVM IR to HC IR transformation.

The HC IR representation consists of a sequence of *blocks* where each block is represented as a *Horn clause*:

$$< block\_id > (< params >) \ :- \ S_1, \ \ldots \ , S_n.$$

Each block has an entry point, that we call the *head* of the block (to the left of the $:-$ symbol), with a number

of parameters $< params >$, and a sequence of steps (the *body*, to the right of the $:-$ symbol). Each of these S_i steps (or *literals*) is either (the representation of) an LLVM IR *instruction*, or a *call* to another (or the same) block. The analyzer deals with the HC IR always in the same way, independent of its origin.

LLVM IR programs are expressed using typed assembly-like instructions. Each function is in SSA form, represented as a sequence of basic blocks. Each basic block is a sequence of LLVM IR instructions that are guaranteed to be executed in the same order. Each block ends in either a branching or a return instruction. In order to represent each of the basic blocks of the LLVM IR in the HC IR, we follow a similar approach as in the ISA-level transformation [21]. However, the LLVM IR includes an additional type transformation as well as better memory modelling. It is explained in detail in Appendix 5 of [3]. The main aspects of this process, are the following:

1. Infer input/output parameters to each block.
2. Transform LLVM IR types into HC IR types.
3. Represent each LLVM IR block as an HC IR block and each instruction in the LLVM IR block as a literal (S_i).
4. Resolve branching to multiple blocks by creating clauses with the same signature (i.e., the same name and arguments in the head), where each clause denotes one of the blocks the branch may jump to.

The translator component is also in charge of translating the XC assertions to Ciao assertions and back. Assuming the Ciao type of the input and output of the function is known, the translation of assertions from Ciao to XC (and back) is relatively straightforward. The *Pred* field of the Ciao assertion is obtained from the scope of the XC assertion to which an extra argument is added representing the output of the function. The *Precond* fields are produced directly from the type of the input arguments: to each input variable, its regular type and its regular type size are added to the precondition, while the added output argument is declared as a free variable. Finally the *Comp-Props* field is set to the usage of the resource `energy`, i.e., a literal of the form `resource(energy, Lower, Upper)` where `Lower` and `Upper` are the lower and upper bounds from the energy consumption specification.

5. ENERGY CONSUMPTION ANALYSIS

As already mentioned in Section 2, we use an existing static analysis to infer the energy consumption of XC programs [21]. It is a specialization of the generic resource analysis presented in [35] that uses the instruction-level models described in [16]. Such generic resource analysis is fully based on *abstract interpretation* [9], defining the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework [27, 33] of CiaoPP, obtaining features such as *multivariance*, efficient fixpoints, and assertion-based verification and user interaction.

In the rest of this section we give an overview of the general resource analysis, using the following `append/3` predicate as a running example:

```
append([],    S, S).
append([E|R], S, [E|T]) :- append(R,S,T).
```

The first step consists of obtaining the regular type of the arguments for each predicate. To this end, we use one of the type analyses present in the CiaoPP system [36]. In our example, the system infers that for any call to the predicate `append(X, Y, Z)` with `X` and `Y` bound to lists of numbers and `Z` a free variable, if the call succeeds, then `Z` also gets bound to a list of numbers. The regular type for representing "list of numbers" is defined as follows:

```
listnum := [] | [num | listnum].
```

From this type definition, sized type schemas are derived, which incorporate variables representing explicitly lower and upper bounds on the size of terms and subterms. For example, in the following sized type schema (named *listnum-s*):

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)})$$

α and β represent lower and upper bounds on the length of the list, respectively, while γ and δ represent lower and upper bounds of the numbers in the list, respectively.

In a subsequent phase, these sized type schemas are put into relation, producing a system of recurrence equations where output argument sizes are expressed as functions of input argument sizes.

The resource analysis is in fact an extension of the sized type analysis that adds recurrence equations for each resource. As the HC IR representation is a logic program, it is necessary to consider that a predicate can fail or have more than one solution, so we need an auxiliary *cardinality analysis* to get more precise results.

We develop the `append` example for the simple case of the resource being the number of resolution steps performed by a call to `append/3` and we will only focus on upper bounds, r_U. For the first clause, we know that only one resolution step is needed, so:

$$r_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leq 1$$

The second clause performs one resolution step plus all the resolution steps performed by all possible backtrackings over the call in the body of the clause. This number can be bounded as a function of the number of solutions. After setting up and solving these equations we infer that an upper bound on the number of resolution steps is the (upper bound on) the length of the input list `X` plus one. This is expressed as:

$$r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leq \beta_X + 1$$

We refer the reader to [35] for a full description of this analysis and tool.

6. THE GENERAL RESOURCE USAGE VERIFICATION FRAMEWORK

In this section we describe the general framework for (static) resource usage *verification* [22, 24] that we have specialized in this paper for verifying energy consumption specifications of XC programs.

The framework, that we introduced in [22], extends the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties, such as upper and lower bounds on execution time, memory, energy, or user defined resources, given as functions on input data sizes.

Both program verification and debugging compare the *actual semantics* $[\![P]\!]$ of a program P with an *intended semantics* for the same program, which we will denote by I. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. In the framework, both semantics are given in the form of (*safe*) approximations. The abstract (safe) approximation $[\![P]\!]_\alpha$ of the concrete semantics $[\![P]\!]$ of the program is actually computed by (abstract interpretation-based) *static analyses*, and compared directly to the (also approximate) specification, which is safely assumed to be also given as an abstract value I_α. Such approximated specification is expressed by *assertions* in the program. Program verification is then performed by comparing I_α and $[\![P]\!]_\alpha$.

In this paper, we assume that the program P is in HC IR form (i.e., a logic program), which is the result of the transformation of the ISA or LLVM IR code corresponding to an XC program. As already said, such transformation preserves the resource consumption semantics, in the sense that the resource usage information inferred by the static analysis (and hence the result of the verification process) is applicable to the original XC program.

Resource usage semantics.

Given a program p, let $\mathcal{C}_p$ be the set of all calls to p. The concrete resource usage semantics of a program p, for a particular resource of interest, $[\![P]\!]$, is a set of pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of data (either simple data such as numbers, or compound data structures), $p(\bar{t}) \in \mathcal{C}_p$ is a call to procedure[1] p with actual parameters $\bar{t}$, and r is a number expressing the amount of resource usage of the computation of the call $p(\bar{t})$. The concrete resource usage semantics can be defined as a function $[\![P]\!] : \mathcal{C}_p \mapsto \mathcal{R}$ where $\mathcal{R}$ is the set of real numbers (note that depending on the type of resource we can take other set of numbers, e.g., the set of natural numbers).

The abstract resource usage semantics is a set of 4-tuples:

$$(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$$

where $p(\bar{v}) : c(\bar{v})$ is an abstraction of a set of calls. $\bar{v}$ is a tuple of variables and $c(\bar{v})$ is an abstraction representing a set of tuples of data which are instances of $\bar{v}$. $c(\bar{v})$ is an element of some abstract domain expressing instantiation states. Φ is an abstraction of the resource usage of the calls represented by $p(\bar{v}) : c(\bar{v})$. We refer to it as a *resource usage interval function* for p, defined as follows:

- A *resource usage bound function* for p is a monotonic arithmetic function, $\Psi : S \mapsto \mathcal{R}_\infty$, for a given subset $S \subseteq \mathcal{R}^k$, where $\mathcal{R}$ is the set of real numbers, k is the number of input arguments to procedure p and $\mathcal{R}_\infty$ is the set of real numbers augmented with the special symbols ∞ and $-\infty$. We use such functions to express lower and upper bounds on the resource usage of procedure p depending on input data sizes.

- A *resource usage interval function* for p is an arithmetic function, $\Phi : S \mapsto \mathcal{RI}$, where S is defined as before and $\mathcal{RI}$ is the set of intervals of real numbers, such that $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ for all $\bar{n} \in S$, where $\Phi^l(\bar{n})$ and $\Phi^u(\bar{n})$ are *resource usage bound functions* that denote the lower and upper endpoints of the interval $\Phi(\bar{n})$ respectively for the tuple of input data sizes $\bar{n}$. Although $\bar{n}$ is typically a tuple of natural numbers, we do not want to restrict our framework. We require that Φ be well defined so that $\forall \bar{n}\ (\Phi^l(\bar{n}) \leq \Phi^u(\bar{n}))$.

[1]Also called *predicate* in the HC IR.

$input_p$ is a function that takes a tuple of data $\bar{t}$ and returns a tuple with the input arguments to p. This function can be inferred by using the existing mode analysis or be given by the user by means of assertions. $size_p(\bar{t})$ is a function that takes a tuple of terms $\bar{t}$ and returns a tuple with the sizes of those data under the size metric described in Section 5.

In order to make the presentation simpler, we will omit the $input_p$ and $size_p$ functions in abstract tuples, with the understanding that they are present in all such tuples.

Intended meaning.

The intended approximated meaning I_α of a program is an abstract semantic object with the same kind of tuples: $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$, which is represented by using Ciao assertions (which are part of the HC IR) of the form:

`:- check` *Pred* [`:` *Precond*] `+` *ResUsage*`.`

where $p(\bar{v}) : c(\bar{v})$ is defined by *Pred* and *Precond*, and Φ is defined by *ResUsage*. The information about $input_p$ and $size_p$ is implicit in *Precond* and *ResUsage*. The concretization of I_α, $\gamma(I_\alpha)$, is the set of all pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of terms and $p(\bar{t})$ is an instance of *Pred* that meets precondition *Precond*, and r is a number that meets the condition expressed by *ResUsage* (i.e., r lies in the interval defined by *ResUsage*) for some assertion.

EXAMPLE 6.1. Consider the following HC IR program that computes the factorial of an integer.

```
fact(N,Fact) :- N=<0, Fact=1.
fact(N,Fact) :- N>0, N1 is N-1,
        fact(N1,Fact1), Fact is N*Fact1.
```

One could use the assertion:

```
:- check pred fact(N,F)
   : (num(N), var(F))
  => (num(N), num(F),
      rsize(N, num(Nmin, Nmax)),
      rsize(F, num(Fmin, Fmax)))
   + resource(steps, Nmin+1, Nmax+1).
```

to express that for any call to `fact(N,F)` with the first argument bound to a number and the second one a free variable, the number of resolution (execution) steps performed by the computation is always between `Nmin`$+1$ and `Nmax`$+1$, where `Nmin` and `Nmax` respectively stand for a lower and an upper bound of `N`. In this concrete example, the lower and upper bounds are the same, i.e., the number of resolution steps is exactly `N` $+ 1$, but note that they could be different. □

EXAMPLE 6.2. The assertion in Example 6.1 captures the following concrete semantic tuples:

```
( fact(0, Y), 1 )     ( fact(8, Y), 9 )
```

but it does not capture the following ones:

```
( fact(N, Y), 1 )     ( fact(1, Y), 35 )
```

the left one in the first line above because it is outside the scope of the assertion (i.e., N being a variable, it does

not meet the precondition *Precond*), and the right one because it violates the assertion (i.e., it meets the precondition *Precond*, but does not meet the condition expressed by *ResUsage*). □

Partial correctness: comparing to the abstract semantics.

Given a program p and an intended resource usage semantics I, where $I : \mathcal{C}_p \mapsto \mathcal{R}$, we say that p is partially correct w.r.t. I if for all $p(\bar{t}) \in \mathcal{C}_p$ we have that $(p(\bar{t}), r) \in I$, where r is precisely the amount of resource usage of the computation of the call $p(\bar{t})$. We say that p is partially correct with respect to a tuple of the form $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ if for all $p(\bar{t}) \in \mathcal{C}_p$ such that r is the amount of resource usage of the computation of the call $p(\bar{t})$, it holds that: if $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$ then $r \in \Phi_I(\bar{s})$, where $\bar{s} = size_p(input_p(\bar{t}))$. Finally, we say that p is partially correct with respect to I_α if:

- For all $p(\bar{t}) \in \mathcal{C}_p$, there is a tuple $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in I_α such that $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$, and
- p is partially correct with respect to every tuple in I_α.

Let $(p(\bar{v}) : c(\bar{v}), \Phi)$ and $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ be tuples expressing an abstract semantics $[\![P]\!]_\alpha$ inferred by analysis and an intended abstract semantics I_α, respectively, such that $c_I(\bar{v}) \sqsubseteq c(\bar{v})$,[2] and for all $\bar{n} \in S$ ($S \subseteq \mathcal{R}^k$), $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ and $\Phi_I(\bar{n}) = [\Phi_I^l(\bar{n}), \Phi_I^u(\bar{n})]$. We have that:

(1) If for all $\bar{n} \in S$, $\Phi_I^l(\bar{n}) \leq \Phi^l(\bar{n})$ and $\Phi^u(\bar{n}) \leq \Phi_I^u(\bar{n})$, then p is partially correct with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

(2) If for all $\bar{n} \in S$ $\Phi^u(\bar{n}) < \Phi_I^l(\bar{n})$ or $\Phi_I^u(\bar{n}) < \Phi^l(\bar{n})$, then p is incorrect with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

Checking the two conditions above requires the comparison of resource usage bound functions.

Resource Usage Bound Function Comparison.

Since the resource analysis we use is able to infer different types of functions (e.g., polynomial, exponential, and logarithmic), it is also desirable to be able to compare all of these functions.

For simplicity of exposition, consider first the case where resource usage bound functions depend on one argument. Given two resource usage bound functions (one of them inferred by the static analysis and the other one given in an assertion/specification present in the program), $\Psi_1(n)$ and $\Psi_2(n)$, $n \in \mathcal{R}$ the objective of the comparison operation is to determine intervals for n in which $\Psi_1(n) > \Psi_2(n)$, $\Psi_1(n) = \Psi_2(n)$, or $\Psi_1(n) < \Psi_2(n)$. For this, we define $f(n) = \Psi_1(n) - \Psi_2(n)$ and find the roots of the equation $f(n) = 0$. Assume that the equation has m roots, $n_1, \ldots, n_m$. These roots are intersection points of $\Psi_1(n)$ and $\Psi_2(n)$. We consider the intervals $S_1 = [0, n_1)$, $S_2 = (n_1, n_2)$, $S_m = \ldots (n_{m-1}, n_m)$, $S_{m+1} = (n_m, \infty)$. For each interval S_i, $1 \leq i \leq m$, we select a value v_i in the interval. If $f(v_i) > 0$ (respectively $f(v_i) < 0$), then $\Psi_1(n) > \Psi_2(n)$ (respectively $\Psi_1(n) < \Psi_2(n)$) for all $n \in S_i$.

There exist powerful algorithms for obtaining roots of polynomial functions. In our implementation we have used the GNU Scientific Library [10], which offers a specific polynomial function library that uses analytical methods for finding roots of polynomials up to order four, and uses numerical methods for higher order polynomials.

We approximate exponential and logarithmic resource usage functions using Taylor series. In particular, for exponential functions we use the following formulae:

$$e^x \approx \Sigma_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \qquad for\ all\ x$$

$$a^x = e^{x\ ln\ a} \approx 1 + x\ ln\ a + \frac{(x\ ln\ a)^2}{2!} + \frac{(x\ ln\ a)^3}{3!} + \ldots$$

In our implementation these series are limited up to order 8. This decision has been taken based on experiments we have carried out that show that higher orders do not bring a significant difference in practice. Also, in our implementation, the computation of the factorials is done separately and the results are kept in a table in order to reuse them.

Dealing with logarithmic functions is more complex, as Taylor series for such functions can only be defined for the interval $(-1, 1)$.

For resource usage functions depending on more than one variable, the comparison is performed using constraint solving techniques.

Safety of the Approximations.

When the roots obtained for function comparison are approximations of the actual roots, we must guarantee that their values are safe, i.e., that they can be used for verification purposes, in particular, for safely checking the conditions presented above. In other words, we should guarantee that the error falls on the safe side when comparing the corresponding resource usage bound functions. For this purpose we developed an algorithm for detecting whether the approximated root falls on the safe side or not, and in the case it does not fall on the safe side, performing an iterative process to increment (or decrement) it by a small value until the approximated root falls on the safe side.

7. USING THE TOOL: EXAMPLE

As an illustrative example of a scenario where the embedded software developer has to decide values for program parameters that meet an energy budget, we consider the development of an equaliser (XC) program using a biquad filter. In Figure 2 we can see what the graphical user interface of our prototype looks like, with the code of this biquad example ready to be verified. The purpose of an equaliser is to take a signal, and to attenuate / amplify different frequency bands. For example, in the case of an audio signal, this can be used to correct for a speaker or microphone frequency response. The energy consumed by such a program directly depends on several parameters, such as the sample rate of the signal, and the number of banks (typically between 3 and 30 for an audio equaliser). A higher number of banks enables the designer to create more precise frequency response curves.

Assume that the developer has to decide how many banks to use in order to meet an energy budget while maximizing the precision of frequency response curves at the same time. In this example, the developer writes an XC program where the number of banks is a variable, say `N`. Assume also that the energy constraint to be met is that an application of the

[2]Note that the condition $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ can be checked using the CiaoPP capabilities for comparing program state properties such as types.

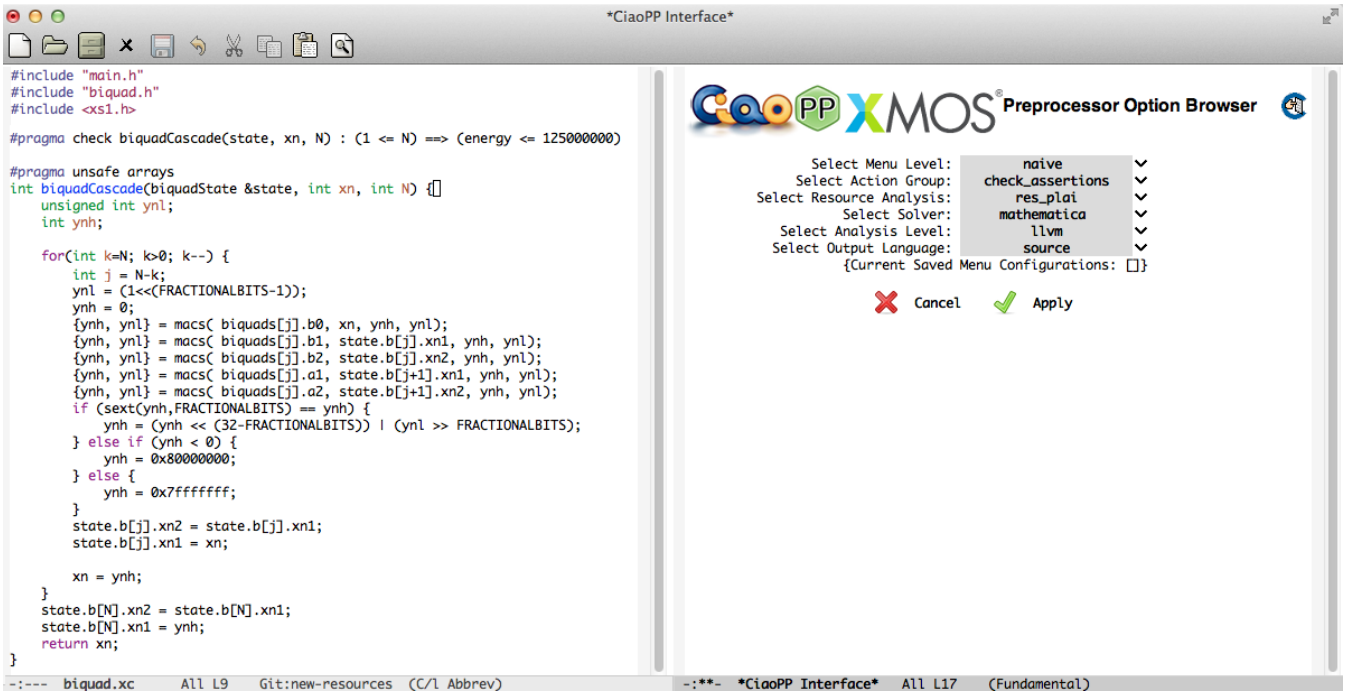


Figure 2: Graphical User Interface of the prototype with the XC biquad program.

biquad program should consume less than 125 millijoules (i.e., 125000000 nanojoules). This constraint is expressed by the following check assertion (specification):

```
#pragma check biquadCascade(state,xn,N) :
   (1 <= N) ==> (energy <= 125000000)
```

where the precondition `1 <= N` in the assertion (left hand side of `==>`) expresses that the number of banks should be at least 1.

Then, the developer makes use of the tool, by selecting the following menu options, as shown in the right hand side of Figure 2: `check_assertions`, for `Action Group`, `res_plai`, for `Resource Analysis`, `mathematica`, for `Solver`, `llvm`, for `Analysis Level` (which will tell the analysis to take the LLVM IR option by compiling the source code into LLVM IR and transform into HC IR for analysis) and finally `source`, for `Output Language` (the language in which the analysis / verification results are shown). After clicking on the `Apply` button below the menu options, the analysis is performed, which infers a lower and an upper bound function for the consumption of the program. Concretely those bounds are represented by the following assertion, which is included in the output of the tool:

```
#pragma true biquadCascade(state,xn,N) :
   (16502087*N + 5445103 <= energy &&
    energy <= 16502087*N + 5445103)
```

In this particular case, both bounds are identical. In other words, the energy consumed by the program is exactly characterized by the following function, depending on `N` only:

$$E_{\text{biquad}}(\texttt{N}) = 16502087 \times \texttt{N} + 5445103 \text{ nJ}$$

Then, the verification of the specification (check assertion) is performed by comparing the energy bound functions above with the upper bound expressed in the specification, i.e., 125000000, a constant value in this case. As a result, the two following assertions are produced (and included in the output file of the tool):

```
#pragma checked biquadCascade(state,xn,N) :
   (1 <= N && N <= 7)
       ==> (energy <= 125000000)
#pragma false biquadCascade(state,xn,N) :
   (8 <= N)
       ==> (energy <= 125000000)
```

The first one expresses that the original assertion holds subject to a precondition on the parameter `N`, i.e., in order to meet the energy budget of 125 millijoules, the number of banks `N` should be a natural number in the interval [1, 7] (precondition `1 <= N && N <= 7`). The second one expresses that the original specification is not met (status `false`) if the number of banks is greater or equal to 8.

Since the goal is to maximize the precision of frequency response curves and to meet the energy budget at the same time, the number of banks should be set to 7. The developer could also be interested in meeting an energy budget but this time ensuring a lower bound on the precision of frequency response curves. For example by ensuring that $\texttt{N} \geq 3$, the acceptable values for `N` would be in the range [3, 7].

In the more general case where the energy function inferred by the tool depends on more than one parameter, the determination of the values for such parameters is reduced to a constraint solving problem. The advantage of this approach is that the parameters can be determined analytically at the program development phase, without the need of determining them experimentally by measuring the energy of expensive program runs with different input parameters.

8. RELATED WORK

As mentioned before, this work adds verification capabilities to our previous work on energy consumption analysis for XC/XS1-L [21], which builds on of our general framework for resource usage analysis [31, 28, 35, 14, 26] and its support for resource verification [22, 24], and the energy models of [16].

Regarding the support for verification of properties expressed as functions, the closest related work we are aware of presents a method for comparison of cost functions inferred by the COSTA system for Java bytecode [1]. The method proves whether a cost function is smaller than another one *for all the values* of a given initial set of input data sizes. The result of this comparison is a Boolean value. However, as mentioned before, in our approach [22, 24] the result is in general a set of subsets (intervals) in which the initial set of input data sizes is partitioned, so that the result of the comparison is different for each subset. Also, [1] differs in that comparison is syntactic, using a method similar to what was already being done in the CiaoPP system: performing a function normalization and then using some syntactic comparison rules. Our technique goes beyond these syntactic comparison rules. Moreover, [1] only covers (generic) cost function comparisons while we have addressed the whole process for the case of energy consumption *verification*. Note also that, although we have presented our work applied to XC programs, the CiaoPP system can also deal with other high- and low-level languages, including, e.g., Java bytecode [29, 26].

In a more general context, using abstract interpretation in debugging and/or verification tasks has now become well established. To cite some early work, abstractions were used in the context of algorithmic debugging in [18]. Abstract interpretation has been applied by Bourdoncle [4] to debugging of imperative programs and by Comini et al. to the algorithmic debugging of logic programs [7] (making use of partial specifications in [6]), and by P. Cousot [8] to verification, among others. The CiaoPP framework [5, 12, 14] was pioneering in many aspects, offering an integrated approach combining abstraction-based verification, debugging, and run-time checking with an assertion language.

9. CONCLUSIONS

We have specialized an existing general framework for resource usage verification for verifying energy consumption specifications of embedded programs. These specifications can include both lower and upper bounds on energy usage, expressed as intervals within which the energy usage is supposed to be included, the bounds (end points of the intervals) being expressed as functions on input data sizes. Our tool can deal with different types of energy functions (e.g., polynomial, exponential or logarithmic functions), in the sense that the analysis can infer them, and the specifications can involve them. We have shown through an example, and using the prototype implementation of our approach within the Ciao/CiaoPP system and for the XC language and XS1-L architecture, how our verification system can prove whether such energy usage specifications are met or not, or infer particular conditions under which the specifications hold. These conditions are expressed as intervals of input data sizes such that a given specification can be proved for some intervals but disproved for others. The specifications themselves can also include preconditions expressing intervals for input data sizes. We have illustrated through this example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

10. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement 318337, ENTRA - Whole-Systems Energy Transparency, Spanish MINECO TIN'12-39391 *StrongSoft* and TIN'08-05624 *DOVES* projects, and Madrid TIC-1465 *PROMETIDOS-CM* project. We also thank all the participants of the ENTRA project team, and in particular John P. Gallagher, Henk Muller, Kyriakos Georgiou, Steve Kerrison, and Kerstin Eder for useful and fruitful discussions. Henk Muller (XMOS Ltd.) also provided benchmarks (e.g., the biquad program) that we used to test our tool.

11. ADDITIONAL AUTHORS

Additional authors: John Smith (The Thørväld Group, email: `jsmith@affiliation.org`) and Julius P. Kumquat (The Kumquat Consortium, email: `jpkumquat@consortium.net`).

12. REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, volume 6234 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010.

[2] N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, editors. *Workshop on Horn Clauses for Verification and Synthesis*, July 2014. To appear in Electronic Proceedings in Theoretical Computer Science.

[3] N. Bohr, K. Eder, J. P. Gallagher, K. Georgiou, R. Haemmerlé, M. V. Hermenegildo, B. Kafle, S. Kerrison, M. Kirkeby, X. Li, U. Liqat, P. Lopez-Garcia, H. Muller, M. Rhiger, and M. Rosendahl. Initial Energy Consumption Analysis. Technical report, FET 318337 ENTRA Project, April 2014. Deliverable 3.2, `http://entraproject.eu`.

[4] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

[5] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[6] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

[7] M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[8] P. Cousot. Automatic Verification by Abstract Interpretation, Invited Tutorial. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 2575 in LNCS, pages 20–24. Springer, January 2003.

[9] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, 1977.

[10] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. Network Theory Ltd, 2009. Available at `http://www.gnu.org/software/gsl/`.

[11] K. Georgiou, S. Kerrison, and K. Eder. A Multi-level Worst Case Energy Consumption Static Analysis for Single and Multi-threaded Embedded Programs. Technical Report CSTR-14-003, University of Bristol, December 2014.

[12] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[13] M. Hermenegildo, G. Puebla, F. Bueno, and P. L. García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.

[14] M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[15] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *TPLP*, 12(1–2):219–252, 2012. http://arxiv.org/abs/1102.5497.

[16] S. Kerrison and K. Eder. Energy modelling of software for a hardware multi-threaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems (TECS)*, 2015. To appear.

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, March 2004.

[18] Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.

[19] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR. Technical report, ENTRA Project, April 2014. Appendix D3.2.4 of Deliverable D3.2. Available at `http://entraproject.eu`.

[20] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Proceedings of LOPSTR'13*, 2014.

[21] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, 2014.

[22] P. López-García, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of ICLP*, volume 7 of *LIPIcs*, pages 104–113. Schloss Dagstuhl, July 2010.

[23] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo. Interval-Based Resource Usage Verification: Formalization and Prototype. In R. P. na, M. Eekelen, and O. Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis*, volume 7177 of *LNCS*, pages 54–71. Springer-Verlag, 2012.

[24] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo. Interval-Based Resource Usage Verification: Formalization and Prototype. In R. P. na, M. Eekelen, and O. Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis. Second Iternational Workshop FOPARA 2011, Revised Selected Papers*, volume 7177 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 2012.

[25] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *LOPSTR 2007*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

[26] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

[27] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[28] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.

[29] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *BYTECODE'09*, volume 253 of *ENTCS*, pages 6–86. Elsevier, March 2009.

[30] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds

Analysis for Logic Programs. In *Proc. of ICLP'07*, volume 4670 of *LNCS*, pages 348–363. Springer, 2007.

[31] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.

[32] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, 2000.

[33] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[34] A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP, ICLP'14 Special Issue*, 14(4-5):739–754, 2014.

[35] A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754, 2014.

[36] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

[37] D. Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.

Attachment D3.3.3

Analysis and Transformation Tools for Constrained Horn Clause Verification

Published in Theory and Practice of Logic Programming 2014

Analysis and Transformation Tools for Constrained Horn Clause Verification∗

John P. Gallagher

Roskilde University, Denmark and IMDEA Software Institute, Madrid, Spain
(*e-mail:* jpg@ruc.dk)

Bishoksan Kafle

Roskilde University, Denmark
(*e-mail:* kafle@ruc.dk)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Several techniques and tools have been developed for verification of properties expressed as Horn clauses with constraints over a background theory (CHC). Current CHC verification tools implement intricate algorithms and are often limited to certain subclasses of CHC problems. Our aim in this work is to investigate the use of a combination of off-the-shelf techniques from the literature in analysis and transformation of Constraint Logic Programs (CLPs) to solve challenging CHC verification problems. We find that many problems can be solved using a combination of tools based on well-known techniques from abstract interpretation, semantics-preserving transformations, program specialisation and query-answer transformations. This gives insights into the design of automatic, more general CHC verification tools based on a library of components.

KEYWORDS: Constraint Logic Program, Constrained Horn Clause, Abstract Interpretation, Software Verification.

1 Introduction

CHCs provide a suitable intermediate form for expressing the semantics of a variety of programming languages (imperative, functional, concurrent, *etc.*) and computational models (state machines, transition systems, big- and small-step operational semantics, Petri nets, *etc.*). As a result it has been used as a target language for software verification. Recently there is a growing interest in CHC verification from both the logic programming and software verification communities, and several verification techniques and tools have been developed for CHC.

Pure CLPs are syntactically and semantically the same as CHC. The main difference is that sets of constrained Horn clauses are not necessarily intended for execution,

∗ The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement no. 318337, ENTRA - Whole-Systems Energy Transparency and the Danish Natural Science Research Council grant NUSA: Numerical and Symbolic Abstractions for Software Model Checking.

but rather as specifications. From the point of view of verification, we do not distinguish between CHC and pure CLP. Much research has been carried out on the analysis and transformation of CLP programs, typically for synthesising efficient programs or for inferring run-time properties of programs for the purpose of debugging, compile-time optimisations or program understanding. In this paper we investigate the application of this research to the CHC verification problem.

In Section 2 we define the CHC verification problem. In Section 3 we define basic transformation and analysis components drawn from or inspired by the CLP literature. Section 4 discusses the role of these components in verification, illustrating them on an example problem. In Section 5 we construct a tool-chain out of these components and test it on a range of CHC verification benchmark problems. The results reported represent one of the main contributions of this work. In Section 6 we propose possible extensions of the basic tool-chain and compare them with related work on CHC verification tool architectures. Finally in Section 7 we summarise the conclusions from this work.

2 Background: The CHC Verification Problem

A CHC is a first order predicate logic formula of the form $\forall(\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k) \rightarrow H(X))$ $(k \geq 0)$, where ϕ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $B_1, \ldots, B_k, H$ are predicate symbols, $H(X)$ is the head of the clause and $\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k)$ is the body. Sometimes the clause is written $H(X) \leftarrow \phi \wedge B_1(X_1), \ldots, B_k(X_k)$ and in concrete examples it is written in the form `H :-` ϕ`,` B_1`(`X_1`),...,`B_k`(`X_k`)`. In examples, predicate symbols start with lowercase letters while we use uppercase letters for variables.

We assume here that the constraint theory is linear arithmetic with relation symbols $\leq$, $\geq$, $>$, $<$ and $=$ and that there is a distinguished predicate symbol false which is interpreted as false. In practice the predicate false only occurs in the head of clauses; we call clauses whose head is false *integrity constraints*, following the terminology of deductive databases. Thus the formula $\phi_1 \leftarrow \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$ is equivalent to the formula $\mathsf{false} \leftarrow \neg\phi_1 \wedge \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$. The latter might not be a CHC but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula `X=Y :- p(X,Y)` is equivalent to the set of CHCs `false :- X>Y, p(X,Y)` and `false :- X<Y, p(X,Y)`. Integrity constraints can be viewed as safety properties. If a set of CHCs encodes the behaviour of some system, the bodies of integrity constraints represent unsafe states. Thus proving safety consists of showing that the bodies of integrity constraints are false in all models of the CHC clauses.

The CHC verification problem. To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. We restate this property in terms of the derivability of the predicate false.

Lemma 2.1
P has a model if and only if $P \not\models \mathsf{false}$.

Proof
Let us write $I(F)$ to mean that interpretation I satisfies F (I is a model of F).

$$\begin{array}{rcl} P \not\models \mathsf{false} & \equiv & \exists I.(I(P) \text{ and } \neg I(\mathsf{false})) \\ & \equiv & \exists I.I(P) \quad \textit{(since } \neg I(\mathsf{false}) \textit{ is true by defn. of } \mathsf{false}\textit{)} \\ & \equiv & P \text{ has a model.} \end{array}$$

□

Obviously any model of P assigns false to the bodies of integrity constraints.

The verification problem can be formulated deductively rather than model-theoretically. Let the relation $P \vdash A$ denote that A is derivable from P using some proof procedure. If the proof procedure is sound and complete then $P \not\models A$ if and only if $P \not\vdash A$. So the verification problem is to show (using CLP terminology) that the computation of the goal $\leftarrow \mathsf{false}$ in program P does not succeed using a complete proof procedure. Although in this work we follow the model-based formulation of the problem, we exploit the equivalence with the deductive formulation, which underlies, for example, the query-answer transformation and specialisation techniques to be presented.

2.1 Representation of Interpretations

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \mathcal{C}$ where A is an atomic formula $p(Z_1, \ldots, Z_n)$ where $Z_1, \ldots, Z_n$ are distinct variables and $\mathcal{C}$ is a constraint over $Z_1, \ldots, Z_n$. If $\mathcal{C}$ is true we write $A \leftarrow$ or just A. The constrained fact $A \leftarrow \mathcal{C}$ is shorthand for the set of variable-free facts $A\theta$ such that $\mathcal{C}\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

Minimal models. A model of a set of CHCs is an interpretation that satisfies each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator, $T_P^{\mathcal{C}}$, which is an extension of the standard T_P operator from logic programming, extended to handle constraints (Jaffar and Maher 1994). Furthermore $\mathsf{lfp}(T_P^{\mathcal{C}})$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, T_P^{\mathcal{C}}(\emptyset), T_P^{\mathcal{C}}(T_P^{\mathcal{C}}(\emptyset)), \ldots$. For more details, see (Jaffar and Maher 1994). This sequence provides a basis for abstract interpretation of CHC clauses.

Proof by over-approximation of the minimal model. It is a standard theorem of CLP that the minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P. That is, $P \models p(v_1, \ldots, v_n)$ if and only if $p(v_1, \ldots, v_n) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that $\mathsf{false} \notin M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where $\mathsf{false} \notin M'$. This technique is called proof by over-approximation of the minimal model.

3 Relevant tools for CHC Verification

In this section, we give a brief description of some relevant tools borrowed from the literature in analysis and transformation of CLP.

Unfolding. Let P be a set of CHCs and $c_0 \in P$ be $H(X) \leftarrow \mathcal{B}_1, p(Y), \mathcal{B}_2$ where $\mathcal{B}_1, \mathcal{B}_2$ are possibly empty conjunctions of atomic formulas and constraints. Let $\{c_1, \ldots, c_m\}$ be the set of clauses of P that have predicate p in the head, that is, $c_i = p(Z_i) \leftarrow \mathcal{D}_i$, where the variables of these clauses are standardised apart from the variables of c_0 and from each other. Then the result of unfolding c_0 on $p(Y)$ is the set of CHCs $P' = P \setminus \{c_0\} \cup \{c'_1, \ldots, c'_m\}$ where $c'_i = H(X) \leftarrow \mathcal{B}_1, Y = Z_i, \mathcal{D}_i, \mathcal{B}_2$. The equality $Y = Z_i$ stands for the conjunction of the equality of the respective elements of the vectors Y and Z_i. It is a standard result that unfolding a clause in P preserves P's minimal model (Pettorossi and Proietti 1999). In particular, $P \models \mathsf{false} \equiv P' \models \mathsf{false}$.

Specialisation. A set of CHCs P can be specialised with respect to a query. Assume A is an atomic formula; then we can derive a set P_A such that $P \models A \equiv P_A \models A$. P_A could be simpler than P, for instance, parts of P that are irrelevant to A could be omitted in P_A. In particular, the CHC verification problem for P_{false} and P are equivalent. There are many techniques in the CLP literature for deriving a specialised program P_A. Partial evaluation is a well-developed method (Gallagher 1993; Leuschel 1999).

We make use a form of specialisation know as forward slicing, more specifically redundant argument filtering (Leuschel and Sørensen 1996), in which predicate arguments can be removed if they do not affect a computation. Given a set of CHCs P and a query A, denote by P_A^{raf} the program obtained by applying the RAF algorithm from (Leuschel and Sørensen 1996) with respect to the goal A. We have the property that $P \models A \equiv P_A^{\mathsf{raf}} \models A$ and in particular that $P \models \mathsf{false} \equiv P_{\mathsf{false}}^{\mathsf{raf}} \models \mathsf{false}$.

Query-answer transformation. Given a set of CHCs P and an atomic query A, the query-answer transformation of P with respect to A is a set of CHCs which simulates the computation of the goal $\leftarrow A$ in P, using a left-to-right computation rule. Query-answer transformation is a generalisation of the magic set transformations for Datalog. For each predicate p, two new predicates p_{ans} and p_{query} are defined. For an atomic formula A, A_{ans} and A_{query} denote the replacement of A's predicate symbol p by p_{ans} and p_{query} respectively. Given a program P and query A, the idea is to derive a program P_A^{qa} with the following property $P \models A$ iff $P_A^{\mathsf{qa}} \models A_{ans}$. The A_{query} predicates represent calls in the computation tree generated during the execution of the goal. For more details see (Debray and Ramakrishnan 1994; Gallagher and de Waal 1993; Codish and Demoen 1993). In particular, $P_{\mathsf{false}}^{\mathsf{qa}} \models \mathsf{false}_{ans} \equiv P \models \mathsf{false}$, so we can transform a CHC verification problem to an equivalent CHC verification problem on the query-answer program generated with respect to the goal $\leftarrow \mathsf{false}$.

Predicate splitting. Let P be a set of CHCs and let $\{c_1, \ldots, c_m\}$ be the set of clauses in P having some given predicate p in the head, where $c_i = p(X) \leftarrow \mathcal{D}_i$. Let $C_1, \ldots, C_k$ be some partition of $\{c_1, \ldots, c_m\}$, where $C_j = \{c_{j_1}, \ldots, c_{j_{n_j}}\}$. Define k new predicates $p_1 \ldots p_k$, where p_j is defined by the bodies of clauses in partition C_j, namely $Q^j = \{p_j(X) \leftarrow \mathcal{D}_{j_1}, \ldots, p_j(X) \leftarrow \mathcal{D}_{j_{n_j}}\}$. Finally, define k clauses $C_p = \{p(X) \leftarrow p_1(X), \ldots, p(X) \leftarrow p_k(X)\}$. Then we define a splitting transformation as follows.

1. Let $P' = P \setminus \{c_1, \ldots, c_m\} \cup C_p \cup Q^1 \cup \ldots \cup Q^k$.
2. Let P^{split} be the result of unfolding every clause in P' whose body contains $p(Y)$ with the clauses C_p.

In our applications, we use splitting to create separate predicates for clauses for a given predicate whose constraints are mutually exclusive. For example, given the clauses `new3(A,B) :- A=<99, new4(A,B)` and `new3(A,B) :- A>=100, new5(A,B)`, we produce two new predicates, since the constraints `A=<99` and `A>=100` are disjoint. The new predicates are defined by clauses $\texttt{new3}_1\texttt{(A,B) :- A=<99, new4(A,B)}$ and $\texttt{new3}_2\texttt{(A,B) :- A>=100, new5(A,B)}$, and all calls to `new3` throughout the program are unfolded using these new clauses. Splitting has been used in the CLP literature to improve the precision of program analyses, for example in (Serebrenik and De Schreye 2001). In our case it improves the precision of the convex polyhedron analysis discussed below, since separate polyhedra will be maintained for each of the disjoint cases. The correctness of splitting can be shown using standard transformations that preserve the minimal model of the program (with respect to the predicates of the original program) (Pettorossi and Proietti 1999). Assuming that the predicate false is not split, we have that $P \models \mathsf{false} \equiv P^{\mathsf{split}} \models \mathsf{false}$.

Convex polyhedron approximation. Convex polyhedron analysis (Cousot and Halbwachs 1978) is a program analysis technique based on abstract interpretation (Cousot and Cousot 1977). When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \mathcal{C}$ for each predicate p. The constraint $\mathcal{C}$ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CLP was by Benoy and King (1996). Since the domain of convex polyhedra contains infinite increasing chains, the use of a widening operator is needed to ensure convergence of the abstract interpretation. Furthermore much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds (Halbwachs et al. 1994).

Recently, a technique for deriving more effective thresholds was developed (Lakhdar-Chaouch et al. 2011), which we have adapted and found to be effective in experimental studies. The thresholds are computed by the following method. Let $T_P^{\mathcal{C}}$ be the standard immediate consequence operator for CHCs, that is, $T_P^{\mathcal{C}}(I)$ is the set of constrained facts that can be derived in one step from a set of constrained facts I. Given a constrained fact $p(\bar{Z}) \leftarrow \mathcal{C}$, define $\mathsf{atomconstraints}(p(\bar{Z}) \leftarrow \mathcal{C})$ to be the set of constrained facts $\{p(\bar{Z}) \leftarrow C_i \mid \mathcal{C} = C_1 \wedge \ldots \wedge C_k, 1 \leq i \leq k)\}$. The function $\mathsf{atomconstraints}$ is extended to interpretations by $\mathsf{atomconstraints}(I) = \bigcup_{p(\bar{Z}) \leftarrow \mathcal{C} \in I} \{\mathsf{atomconstraints}(p(\bar{Z}) \leftarrow \mathcal{C})\}$.

Let $I_\top$ be the interpretation consisting of the set of constrained facts $p(\bar{Z}) \leftarrow \mathsf{true}$ for each predicate p. We perform three iterations of $T_P^{\mathcal{C}}$ starting with $I_\top$ (the first three elements of a "top-down" Kleene sequence) and then extract the atomic constraints. That is, $\mathsf{thresholds}$ is defined as follows.

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(T_P^{\mathcal{C}(3)}(I_\top))$$

A difference from the method in (Lakhdar-Chaouch et al. 2011) is that we use the concrete semantic function $T_P^{\mathcal{C}}$ rather than the abstract semantic function when computing thresholds. The set of threshold constraints represents an attempt to find useful predicate properties and when widening they help to preserve invariants that might otherwise be lost during widening. See (Lakhdar-Chaouch et al. 2011) for further details. Threshold constraints that are not invariants are simply discarded during widening.

```
new6(A,B) :- B=<99.                    new4(A,B) :- C=1+A,D=1+B,A>=50,new3(C,D).
new5(A,B) :- B>=101.                   new3(A,B) :- A=<99, new4(A,B).
new5(A,B) :- B=<100, new6(A,B).        new3(A,B) :- A>=100, new5(A,B).
new4(A,B) :- C=1+A, A=<49, new3(C,B).  false :- A=0, B=50, new3(A,B).
```

Fig. 1. The example program `MAP-disj.c.map.pl`

4 The role of CLP tools in verification

The techniques discussed in the previous section play various roles. The convex polyhedron analysis, together with the helper tool to derive threshold constraints, constructs an approximation of the minimal model of a CHC theory. If false (or false_{ans}) is not in the approximate model, then the verification problem is solved. Otherwise the problem is not solved; in effect a "don't know" answer is returned. We have found that polyhedron analysis alone is seldom precise enough to solve non-trivial CHC verification problems; in combination with the other tools, it is very effective.

Unfolding can improve the structure of a program, removing some cases of mutual recursion, or propagating constraints upwards towards the integrity constraints, and can improve the precision and performance of convex polyhedron analysis.

Specialisation can remove parts of theories not relevant to the verification problem, and can also propagate constraint downwards from the integrity constraints. Both of these have a beneficial effect on performance and precision of polyhedron analysis.

Analysis of a query-answer program (with respect to false) is in effect the search for a derivation tree for false. Its effectiveness in CHC verification problems is variable. It can sometimes worsen performance since the query-answer transformed program is larger and contains more recursive dependencies than the original. On the other hand, one seldom loses precision and it is often more effective in allowing constraints to be propagated upwards (through the *ans* predicates) and downwards (through the *query* predicates).

4.1 Application of the tools

We illustrate the tools on a running example (Figure 1), one of the benchmark suite of the VeriMAP system De Angelis et al. (2014). The result of applying unfolding is shown in Figure 2 (omitting the definitions of the unfolded predicates `new4, new5` and `new6`, which are no longer reachable from `false`). The unfolding strategy we adopt is the following: the predicate dependency graph of a program consists of the set of edges (p, q) such that there is clause where p is the predicate of the head and q is a predicate occurring in the body. We perform a depth-first search of the predicate dependency graph, starting from `false`, and identify the backward edges, namely those edges (p, q) where q is an ancestor of p in the depth-first search. We then unfold every body call whose predicate is not at the end of a backward edge. In Figure 1, we thus unfold calls to `new4, new5` and `new6`.

The query-answer transformation is applied to the program in Figure 2, with respect to the goal `false` resulting in the program shown in Figure 3. The model of the predicate `new3_query` corresponds to those calls to `new3` that are reachable from the call in the integrity constraint. Explicit representation of the query predicates permits more effective propagation of constraints from the integrity clauses during model approximation.

The splitting transformation is now applied to the program in Figure 3. We do not

```
false :- A=0, B=50, new3(A,B).
new3(A,B) :- A=<99, C = 1+A, A=<49, new3(C,B).
new3(A,B) :- A=<99, C = 1+A, D = 1+B, A>=50, new3(C,D).
new3(A,B) :- A>=100, B>=101.
new3(A,B) :- A>=100, B=<100, B=<99.
```

Fig. 2. Result of unfolding `MAP-disj.c.map.pl`

```
false_ans :- false_query, A=0, B=50, new3_ans(A,B).
new3_ans(A,B) :- new3_query(A,B), A=<99, C = 1+A, A=<49, new3_ans(C,B).
new3_ans(A,B) :- new3_query(A,B),A=<99,C is 1+A,D is 1+B, A>=50, new3_ans(C,D).
new3_ans(A,B) :- new3_query(A,B), A>=100, B>=101.
new3_ans(A,B) :- new3_query(A,B), A>=100, B=<100, B=<99.
new3_query(A,B) :- false_query, A=0, B=50.
new3_query(A,B) :- new3_query(C,B), C=<99, A = 1+C, C=<49.
new3_query(A,B) :- new3_query(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
false_query.
```

Fig. 3. The query-answer transformed program for program of Figure 2

show the complete program, which contains 30 clauses. Figure 4 shows the split definition of `new3_query`, which is split since the last two clauses for `new3_query` in Figure 3 have mutually disjoint constraints, when projected onto the head variables.

A convex polyhedron approximation is then computed for the split program, after computing threshold constraints for the predicates. The resulting approximate model is shown in Figure 5 as a set of constrained facts. Since the model does not contain any constrained fact for `false_ans` we conclude that `false_ans` is not a consequence of the split program. Hence, applying the various correctness results for the unfolding, query-answer and splitting transformations, `false` is not a consequence of the original program.

Discussion of the example. Application of the convex polyhedron tool to the original, or the intermediate programs, does not solve the problem; all the transformations are needed in this case, apart from redundant argument filtering, which only affects efficiency. The ordering of the tool-chain can be varied somewhat, for instance switching query-answer transformation with splitting or unfolding. In our experiments we found the ordering in Figure 6 to be the most effective.

The model of the query-answer program is finite for this example. However, the problem is essentially the same if the constants are scaled; for instance we could replace 50 by 5000, 49 by 4999, 100 by 10000 and 101 by 10001, and the problem is essentially unchanged. We noted that some CHC verification tools applied to this example solve the problem, but essentially by enumeration of the finite set of values encountered in the search. Such

```
new3_query___1(A,B) :- false_query___1, A=0, B=50.
new3_query___1(A,B) :- new3_query___1(C,B), C=<99, A = 1+C, C=<49.
new3_query___1(A,B) :- new3_query___2(C,B), C=<99, A = 1+C, C=<49.
new3_query___2(A,B) :- new3_query___1(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
new3_query___2(A,B) :- new3_query___2(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
```

Fig. 4. Part of the split program for the program in Figure 3

```
false_query___1 :- []
new3_query___1(A,B) :- [1*A>=0,-1*A>= -50,1*B=50]
new3_query___2(A,B) :- [1*A>=51,-1*A>= -100,1*A+ -1*B=0]
```

Fig. 5. The convex polyhedral approximate model for the split program

a solution does not scale well. On the other hand the polyhedral abstraction shown above is not an enumeration; an essentially similar polyhedron abstraction is generated for the scaled version of the example, in the same time. The VeriMAP tool (De Angelis et al. 2014) also handles the original and scaled versions of the example in the same time.

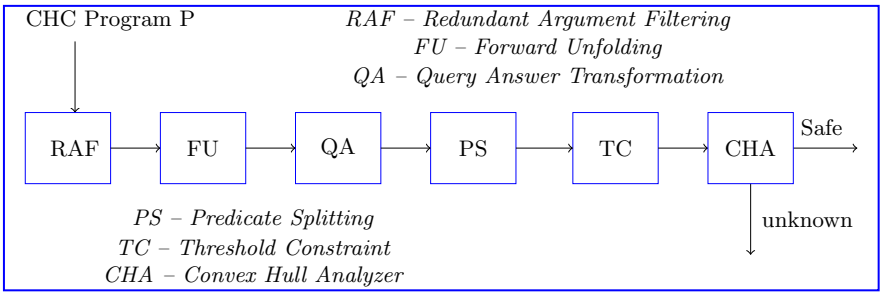


Fig. 6. *The basic tool chain for CHC verification.*

5 Combining off-the-shelf tools: Experiments

The motivation for our tool-chain, summarised in Figure 6, comes from our example program, which is a simple yet challenging program. We applied the tool-chain to a number of benchmarks from the literature, taken mainly from the repository of Horn clause benchmarks in SMT-LIB2 (https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/) and other sources including (Gange et al. 2013) and some of the VeriMap benchmarks (De Angelis et al. 2014). We selected these examples because many of them are considered challenging because they cannot be solved by one or more of the state-of-the-art-verification tools discussed below. Programs taken from the SMT-LIB2 repository are first translated to CHC form. The results are summarised in Table 1.

In Table 1, columns Program and Result respectively represent the benchmark program and the results of verification using our tool combination. Problems marked with (*) could not be handled by our tool-chain since they contain numbers which do not fit in 32 bits, the limit of our Ciao Prolog implementation. whereas problems marked with (**) are solvable by simple ad hoc modification of the tool-chain, which we are currently investigating (see Section 7). Problems such as systemc-token-ring.01-safeil.c contain complicated loop structure with large strongly connected components in the predicate dependency graph and our convex polyhedron analysis tool is unable to derive the required invariant. However overall results show that our simple tool-chain begins to compete with advanced tools like HSF (Grebenshchikov et al. 2012), VeriMAP (De Angelis et al. 2014), TRACER (Jaffar et al. 2012), *etc.* We do not report timings, though all these

results are obtained in a matter of seconds, since our tool-chain is not at all optimised, relying on file input-output and the individual components are often prototypes.

Table 1. *Experiments results on CHC benchmark program*

SN	Program	Result	SN	Program	Result
1	MAP-disj.c.map.pl	verified	17	MAP-forward.c.map.pl	verified
2	MAP-disj.c.map-scaled.pl	verified	18	tridag.smt2	verified
3	t1.pl	verified	19	qrdcmp.smt2	verified
4	t1-a.pl	verified	20	choldc.smt2	verified
5	t2.pl	verified	21	lop.smt2	verified
6	t3.pl	verified	22	pzextr.smt2	verified
7	t4.pl	verified	23	qrsolv.smt2	verified
8	t5.pl	verified	24	INVGEN-apache-escape-absolute	verified
9	pldi12.pl	verified	25	TRACER-testabs15	verified
10	INVGEN-id-build	verified	26**	amebsa.smt2	verified
11	INVGEN-nested5	verified	27**	DAGGER-barbr.map.c	verified
12	INVGEN-nested6	verified	28*	sshsimpl-s3-srvr-1a-safeil.c	NOT
13	INVGEN-nested8	verified	29	sshsimpl-s3-srvr-1b-safeil.c	NOT
14	INVGEN-svd-some-loop	verified	30*	bandec.smt2	NOT
15	INVGEN-svd1	verified	31	systemc-token-ring.01-safeil.c	NOT
16	INVGEN-svd4	verified	32*	crank.smt2	NOT

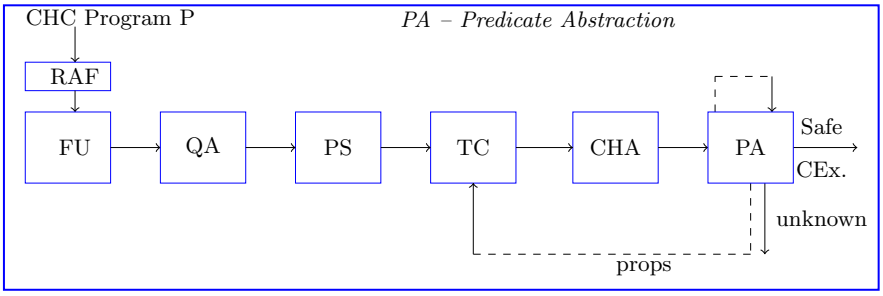


Fig. 7. *Future extension of our tool-chain.*

6 Discussion and Related Work

The most similar work to ours is by De Angelis et al. (2013) which is also based on CLP program transformation and specialisation. They construct a sequence of transformations of P, say, $P, P_1, P_2, \ldots, P_k$; if P_k contains no clause with head false then the verification problem is solved. A proof of unsafety is obtained if P_k contains a clause $\mathsf{false} \leftarrow$. Both our approach and theirs repeatedly apply specialisations preserving the property to be proved. However the difference is that their specialisation techniques are based on unfold-fold transformations, with a sophisticated control procedure controlling unfolding and

generalisation. Our specialisations are restricted to redundant argument filtering and the query-answer transformation, which specialises predicate answers with respect to a goal. Their test for success or failure is a simple syntactic check, whereas ours is based on an abstract interpretation to derive an over-approximation. Informally one can say that the hard work in their approach is performed by the specialisation procedure, whereas the hard work in our approach is done by the abstract interpretation. We believe that our tool-chain-based approach gives more insight into the role of each transformation.

Work by Gange et al. (2013) is a top-town evaluation of CLP programs which records certain derivations and learns only from failed derivations. This helps to prune further derivations and helps to achieve termination in the presence of infinite executions. Duality (http://research.microsoft.com/en-us/projects/duality/) and HSF(C) (Grebenshchikov et al. 2012) are examples of the CEGAR approach (Counter-Example-Guided Abstraction Refinement). This approach can be viewed as property-based abstract interpretation based on a set of properties that is refined on each iteration. The refinement of the properties is the key problem in CEGAR; an abstract proof of unsafety is used to generate properties (often using interpolation) that prevent that proof from arising again. Thus, abstract counter-examples are successively eliminated. The relatively good performance of our tool-chain, without any refinement step at all, suggests that finding the right invariants is aided by a tool such as the convex polyhedron solver and the pre-processing steps we applied. In Figure 7 we sketch possible extensions of our basic tool-chain, incorporating a refinement loop and property-based abstraction.

It should be noted that the query-answer transformation, predicate splitting and unfolding may all cause an blow-up in the program size. The convex polyhedron analysis becomes more effective as a result, but for scalability we need more sophisticated heuristics controlling these transformations, especially unfolding and splitting, as well as lazy or implicit generation of transformed programs, using techniques such as a fixpoint engine that simulates query-answer programs (Codish 1999).

7 Concluding remarks and future work

We have shown that a combination of off-the-shelf tools from CLP transformation and analysis, combined in a sensible way, is surprisingly effective in CHC verification. The component-based approach allowed us to experiment with the tool-chain until we found an effective combination. This experimentation is continuing and we are confident of making improvements by incorporating other standard techniques and by finding better heuristics for applying the tools. Further we would like to investigate the choice of chain suitable for each example since more complicated problems can be handled just by altering the chain. We also suspect from initial experiments that an advanced partial evaluator such as ECCE (Leuschel et al. 2006) will play a useful role. Our results give insights for further development of automatic CHC verification tools. We would like to combine our program transformation techniques with abstraction refinement techniques and experiment with the combination.

References

Benoy, F. and King, A. 1996. Inferring argument size relationships with CLP(R). In *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, J. P. Gallagher, Ed. Lecture Notes in Computer Science, vol. 1207. Springer, 204–223.

Codish, M. 1999. Efficient goal directed bottom-up evaluation of logic programs. *J. Log. Program. 38,* 3, 355–370.

Codish, M. and Demoen, B. 1993. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*, D. Miller, Ed. MIT Press.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 238–252.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 84–96.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2013. Verifying programs via iterated specialization. In *PEPM*, E. Albert and S.-C. Mu, Eds. ACM, 43–52.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2014. Verimap: A tool for verifying programs through transformations. In *TACAS*, E. Ábrahám and K. Havelund, Eds. Lecture Notes in Computer Science, vol. 8413. Springer, 568–574.

Debray, S. and Ramakrishnan, R. 1994. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming 18*, 149–176.

Gallagher, J. P. 1993. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, Copenhagen, 88–98.

Gallagher, J. P. and de Waal, D. 1993. Deletion of redundant unary type predicates from logic programs. In *Logic Program Synthesis and Transformation*, K. Lau and T. Clement, Eds. Workshops in Computing. Springer-Verlag, 151–167.

Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., and Stuckey, P. J. 2013. Failure tabled constraint logic programming by interpolation. *TPLP 13,* 4-5, 593–607.

Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. HSF(C): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, C. Flanagan and B. König, Eds. LNCS, vol. 7214. Springer, 549–551.

Halbwachs, N., Proy, Y. E., and Raymound, P. 1994. Verification of linear hybrid systems by means of convex approximations. In *Proceedings of the First Symposium on Static Analysis*. Lecture Notes in Computer Science, vol. 864. Springer, 223–237.

Jaffar, J. and Maher, M. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19/20*, 503–581.

Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. 2012. TRACER: A symbolic execution tool for verification. In *CAV*, P. Madhusudan and S. A. Seshia, Eds. Lecture Notes in Computer Science, vol. 7358. Springer, 758–766.

Lakhdar-Chaouch, L., Jeannet, B., and Girault, A. 2011. Widening with thresholds for programs with complex control graphs. In *ATVA 2011*, T. Bultan and P.-A. Hsiung, Eds. Lecture Notes in Computer Science, vol. 6996. Springer, 492–502.

Leuschel, M. 1999. Advanced logic program specialisation. In *Partial Evaluation - Practice and Theory*, J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1706. Springer, 271–292.

Leuschel, M., Elphick, D., Varea, M., Craig, S.-J., and Fontaine, M. 2006. The Ecce and Logen partial evaluators and their web interfaces. In *PEPM 2006*, J. Hatcliff and F. Tip, Eds. ACM, 88–94.

Leuschel, M. and Sørensen, M. H. 1996. Redundant argument filtering of logic programs. In *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, J. P. Gallagher, Ed. Lecture Notes in Computer Science, vol. 1207. Springer, 83–103.

Pettorossi, A. and Proietti, M. 1999. Synthesis and transformation of logic programs using unfold/fold proofs. *J. Log. Program. 41,* 2-3, 197–230.

Serebrenik, A. and De Schreye, D. 2001. Inference of termination conditions for numerical loops in Prolog. In *LPAR 2001*, R. Nieuwenhuis and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 2250. Springer, 654–668.

Attachment D3.3.4

Tree automata-based refinement with application to Horn clause verification

Published at the 16th International Conference of Verification, Model Checking, and Abstract Interpretation (VMCAI 2015)

Tree Automata-Based Refinement with Application to Horn Clause Verification

Bishoksan Kafle[1,⋆] and John P. Gallagher[1,2,⋆⋆]

[1] Roskilde University, Denmark
[2] IMDEA Software Institute, Madrid, Spain

Abstract. In this paper we apply tree-automata techniques to refinement of abstract interpretation in Horn clause verification. We go beyond previous work on refining trace abstractions; firstly we handle tree automata rather than string automata and thereby can capture traces in any Horn clause derivations rather than just transition systems; secondly, we show how algorithms manipulating tree automata interact with abstract interpretations, establishing progress in refinement and generating refined clauses that eliminate causes of imprecision. We show how to derive a refined set of Horn clauses in which given infeasible traces have been eliminated, using a recent optimised algorithm for tree automata determinisation. We also show how we can introduce disjunctive abstractions selectively by splitting states in the tree automaton. The approach is independent of the abstract domain and constraint theory underlying the Horn clauses. Experiments using linear constraint problems and the abstract domain of convex polyhedra show that the refinement technique is practical and that iteration of abstract interpretation with tree automata-based refinement solves many challenging Horn clause verification problems. We compare the results with other state of the art Horn clause verification tools.

1 Introduction

In this paper we apply tree-automata techniques to refinement of abstract interpretation in Horn clause verification. We go beyond previous work on refining trace abstractions [23]; firstly, we handle tree automata rather than word automata and thereby can capture traces in any Horn clause derivations rather than just transition systems; secondly, we show how algorithms manipulating tree automata interact with abstract interpretations, establishing progress in refinement and generating refined clauses that eliminate causes of imprecision.

More specifically, we show how to construct tree automata capturing both the traces (derivations) of a given set of Horn clauses and also one or more infeasible traces discovered after abstract interpretation of the clauses. From these we construct a refined automaton in which the infeasible trace(s) have been eliminated and a new set of clauses is constructed from the refined automaton.

⋆ Supported by EU FP7 project ENTRA (Project 318337).
⋆⋆ Supported by Danish Research Council grant FNU 10-084290.

D. D'Souza et al. (Eds.): VMCAI 2015, LNCS 8931, pp. 209–226, 2015.

This guarantees progress in that the same infeasible trace cannot be generated (in *any* abstract interpretation). In addition, the clauses are restructured during the elimination of the trace, leading to more precise abstractions which can lead to better invariant generation in subsequent iterations. The refinement is manifested in the refined clauses, rather than in an accumulated set of properties as in the counterexample-guided abstraction refinement (CEGAR) [8] approach. We rely on the abstract interpretation of the clauses to generate useful properties, rather than hoping to find them during the refinement itself.

We also show how we can introduce disjunctive abstractions selectively by splitting states in the tree automaton. The approach is independent of the abstract domain and constraint theory underlying the Horn clauses. Experiments using linear constraint problems and the abstract domain of convex polyhedra show that the refinement technique is practical and that iteration of abstract interpretation with tree automata-based refinement solves many challenging Horn clause verification problems. We compare the results with other state of the art Horn clause verification tools.

The main contributions of this paper are the following; (1) We construct a correspondence between computations using Horn clauses and finite tree automata (FTA) (Section 3). (2) We construct a refined set of clauses directly from a tree automaton representation of the clauses and an infeasible trace; the trace is eliminated from the refined clauses (Section 3.5) (3) We propose a "splitting" operator on FTAs (Section 2) and describe its role in Horn clause verification (Section 4.1). (4) We demonstrate the feasibility of our approach in practice applying it to Horn clause verification problems (Section 5).

2 Finite Tree Automata

Finite tree automata (FTAs) are mathematical machines that define so-called recognisable tree languages, which are possibly infinite sets of terms that have desirable properties such as closure under Boolean set operations and decidability of membership and emptiness.

Definition 1 (Finite tree automaton). *An FTA $\mathcal{A}$ is a tuple (Q, Q_f, Σ, Δ), where Q is a finite set of states, $Q \subseteq Q_f$ is a set of final states, Σ is a set of function symbols, and Δ is a set of transitions. We assume that Q and Σ are disjoint.*

Each function symbol $f \in \Sigma$ has an arity $n \geq 0$, written as $\mathsf{ar}(f) = n$. The function symbols with arity 0 are called constants. $\mathsf{Term}(\Sigma)$ is the set of ground terms or trees constructed from Σ where $t \in \mathsf{Term}(\Sigma)$ iff $t \in \Sigma$ is a constant or $t = f(t_1, t_2, ..., t_n)$ where $\mathsf{ar}(f) = n$ and $t_1, t_2, ..., t_n \in \mathsf{Term}(\Sigma)$. Similarly $\mathsf{Term}(\Sigma \cup Q)$ is the set of terms/trees constructed from Σ and Q, treating the elements of Q as constants.

Each transition in Δ is of the form $f(q_1, q_2, ..., q_n) \to q$ where $\mathsf{ar}(f) = n$. Given $\delta \in \Delta$ we refer to its left- and right-hand-sides as $\mathsf{lhs}(\delta)$ and $\mathsf{rhs}(\delta)$ respectively. Let $\Rightarrow$ be a one-step rewrite in which $t_1 \Rightarrow t_2$ iff t_2 is the result of replacing one

subterm of t_1 equal to $\mathsf{lhs}(\delta)$ by $\mathsf{rhs}(\delta)$, from some $\delta \in \Delta$. The reflexive, transitive closure of $\Rightarrow$ is $\Rightarrow^*$. We say there is a run (resp. successful run) for $t \in \mathsf{Term}(\Sigma)$ if $t \Rightarrow^* q$ where $q \in Q$ (resp. $q \in Q_f$), and we say that t is *accepted* if t has a successful run. An FTA $\mathcal{A}$ defines a set of terms, that is, a tree language, denoted by $\mathcal{L}(\mathcal{A})$, as the set of all terms accepted by $\mathcal{A}$.

Definition 2 (Deterministic FTA (DFTA)). *An FTA (Q, Q_f, Σ, Δ) is called bottom-up deterministic iff Δ has no two transitions with the same left hand side.*

We omit the adjective "bottom-up" in this paper and just refer to deterministic FTAs. Runs of a DFTA are deterministic in the sense that for every $t \in \mathsf{Term}(\Sigma)$ there is at most one $q \in Q$ such that $t \Rightarrow^* q$.

2.1 Operations on FTAs

FTAs are closed under Boolean set operations, but for our purposes we mention only union and difference of automata, where in addition we assume that the signature Σ is fixed and that the states of FTAs are disjoint from each other when applying operations (the states can be renamed apart).

Definition 3 (Union of FTAs). *Let $\mathcal{A}^1, \mathcal{A}^2$ be FTAs $(Q^1, Q^1_f, \Sigma, \Delta^1)$ and $(Q^2, Q^2_f, \Sigma, \Delta^2)$ respectively. Then $\mathcal{A}^1 \cup \mathcal{A}^2 = (Q^1 \cup Q^2, Q^1_f \cup Q^2_f, \Sigma, \Delta^1 \cup \Delta^2)$, and we have $\mathcal{L}(\mathcal{A}^1 \cup \mathcal{A}^2) = \mathcal{L}(\mathcal{A}^1) \cup \mathcal{L}(\mathcal{A}^2)$.*

Determinisation plays a key role in the theory of FTAs. As far as expressiveness is concerned, we can limit our attention to DFTAs since for every FTA $\mathcal{A}$ there exists a DFTA $\mathcal{A}^d$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^d)$ [9]. The standard construction builds a DFTA $\mathcal{A}^d$ whose states are elements of the powerset of the states of $\mathcal{A}$. The textbook procedure for constructing $\mathcal{A}^d$ from $\mathcal{A}$ [9] is not viewed as a practical procedure for manipulating tree automata, even fairly small ones. In a recent work Gallagher *et al.* [14] developed an optimised algorithm for determinisation, whose worst-case complexity remains unchanged, but which performs dramatically better than existing algorithms in practice. A critical aspect of the algorithm is that the transitions of the determinised automaton are generated in a potentially very compact form called *product form*, which can often be used directly when manipulating the determinised automaton.

Definition 4 (Product Transition). *A* product transition *is of the form $f(Q_1, \ldots, Q_n) \to q$ where Q_i are sets of states and q is a state. The product transition represents a set of transitions $\{f(q_1, \ldots, q_n) \to q \mid q_i \in Q_i, i = 1..n\}$. Thus $\Pi_{i=1}^n |Q_i|$ transitions are represented by a single product transition.*

Alternatively, we can regard a product transition as introducing ϵ-transitions. An ϵ-transition has the form $q_1 \to q_2$ where q_1, q_2 are states. ϵ-transitions can be eliminated, if desired. Given a product transition $f(Q_1, \ldots, Q_n) \to q$, introduce n new non-final states $s_1, \ldots, s_n$ corresponding to $Q_1, \ldots, Q_n$ respectively and replace the product transition by the set of transitions $\{f(s_1, \ldots, s_n) \to q\} \cup$

$\{q' \to s_i \mid q' \in Q_i, 1 = 1..n\}$. It can be shown that this transformation preserves the language of the FTA.

Given FTAs $\mathcal{A}^1$ and $\mathcal{A}^2$ there exists an FTA $\mathcal{A}^1 \setminus \mathcal{A}^2$ such that $\mathcal{L}(\mathcal{A}^1 \setminus \mathcal{A}^2) = \mathcal{L}(\mathcal{A}_1) \setminus \mathcal{L}(\mathcal{A}^2)$. To construct the difference FTA we use union and determinisation and exploit the following property of determinised states [14].

Property 1. Let $\mathcal{A}^d$ be the DFTA constructed from $\mathcal{A}$. Let Q be the states of $\mathcal{A}$. Then there is a run $t \Rightarrow^* q$ in $\mathcal{A}$ if and only if there is a run $t \Rightarrow^* Q'$ in $\mathcal{A}^d$ where $Q' \in 2^Q$, such that $q \in Q'$.

Furthermore recall that a term is accepted by at most one state in a DFTA. This gives rise to the following construction of the difference FTA $\mathcal{A}^1 \setminus \mathcal{A}^2$. We first form the DFTA for the union of the two FTAs and then remove those of its final states containing the final states of $\mathcal{A}^2$. In this way we remove the terms, and only the terms (by Property 1), accepted by $\mathcal{A}^2$. The availability of a practical algorithm for determinisation is what makes this construction of the difference FTA feasible.

Definition 5 (Construction of difference of FTAs). *Let $\mathcal{A}^1, \mathcal{A}^2$ be FTAs $(Q^1, Q^1_f, \Sigma, \Delta^1)$ and $(Q^2, Q^2_f, \Sigma, \Delta^2)$ respectively. Let $(\mathcal{Q}', \mathcal{Q}'_f, \Sigma, \Delta')$ be the determinisation of $\mathcal{A}^1 \cup \mathcal{A}^2$. Let $\mathcal{Q}^2 = \{Q' \in \mathcal{Q}' \mid Q' \cap Q^2_f \neq \emptyset\}$. Then $\mathcal{A}^1 \setminus \mathcal{A}^2 = (\mathcal{Q}', \mathcal{Q}'_f \setminus \mathcal{Q}^2, \Sigma, \Delta')$.*

Next we introduce a new operation over FTA called *state splitting*. which consists of splitting a state q into a number of states, based on a partition of the set of transitions whose rhs is q. We define this splitting as follows:

Definition 6 (Splitting a state in an FTA). *Let $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$ be an FTA. Let $q \in Q$ and $\Delta_q = \{t \in \Delta \mid \mathsf{rhs}(t) = q\}$. Let $\Phi = \{\Delta^1_q, \ldots, \Delta^k_q\}$ $(k > 1)$ be some partition of Δ_q. Introduce k new states $q_1, \ldots, q_k$. Then the FTA $\mathsf{split}_\Phi(\mathcal{A})$ is $(Q^s, Q^s_f, \Sigma, \Delta^s)$ where:*

- $Q^s = Q \setminus \{q\} \cup \{q_1, \ldots, q_k\}$;
- $Q^s_f = Q_f \setminus \{q\} \cup \{q_1, \ldots, q_k\}$ *if* $q \in Q_f$, *otherwise* $Q^s_f = Q_f$;
- $\Delta^s = \mathsf{unfold}_q(\Delta \setminus \Delta_q \cup \{\mathsf{lhs}(t) \to q_i \mid t \in \Delta^i_q, i = 1..k\})$, *where* $\mathsf{unfold}_q(\Delta')$ *is the result of repeatedly replacing a transition $f(\ldots, q, \ldots) \to s \in \Delta'$ by the set of k transitions $\{f(\ldots, q_1, \ldots) \to s, \ldots, f(\ldots, q_k, \ldots) \to s\}$ until no more such replacements can be made.*

We have $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{split}_\Phi(\mathcal{A}))$.

3 Horn Clauses and Their Trace Automata

A constrained Horn clause (CHC) is a first order predicate logic formula of the form $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \to p(X))$ $(k \geq 0)$, where ϕ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty)

vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k)$ is the body.

There is a distinguished predicate symbol false which is interpreted as false. In practice the predicate false only occurs in the head of clauses; we call clauses whose head is false *integrity constraints*, following the terminology of deductive databases. They are also sometimes referred to as negative clauses. We follow the syntactic conventions of constraint logic programs and write a clause as $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$.

3.1 Interpretations and Models

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atomic formula $p(Z_1, \ldots, Z_n)$ where $Z_1, \ldots, Z_n$ are distinct variables and ϕ is a constraint over $Z_1, \ldots, Z_n$. The constrained fact $A \leftarrow \phi$ is shorthand for the set of variable-free facts $A\theta$ such that $\phi\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

Minimal models. A model of a set of CHCs is an interpretation that satisfies each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator (called S_P^D in [25, Section 4]), which is an extension of the standard T_P operator from logic programming, extended to handle the constraint domain D. Furthermore $\mathsf{lfp}(S_P^D)$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, S_P^D(\emptyset), S_P^D(S_P^D(\emptyset)), \ldots$. This sequence provides a basis for abstract interpretation of CHC clauses. The minimal model of P is equivalent to the set of atomic logic consequences of P.

3.2 The Constrained Horn Clause Verification Problem.

Given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. Obviously any model of P assigns false to the bodies of integrity constraints. We restate this property in terms of the derivability of the predicate false. Let $P \models F$ mean that F is a logical consequence of P, that is, that every interpretation satisifying P also satisfies F.

Lemma 1. *P has a model if and only if $P \not\models \mathsf{false}$.*

This lemma holds for arbitrary interpretations (only assuming that the predicate false is interpreted as false), uses only the textbook definitions of "interpretation" and "model" and does not depend on the constraint theory. Due to the equivalence of the minimal model of P with the set of atomic logical consequences of P, we have yet another equivalent formulation of the CHC verification problem.

Lemma 2. *P has a model if and only if $\mathsf{false} \notin M[\![P]\!]$.*

```
c1. mc91(A,B) :- A > 100, B = A-10.
c2. mc91(A,B) :- A =< 100, C = A+11, mc91(C,D), mc91(D,B).
c3. false :- A =< 100,  B > 91, mc91(A,B).
c4. false :- A =< 100, B =< 90, mc91(A,B).
```

Fig. 1. Example CHCs. The McCarthy 91-function.

It is this formulation that is most relevant to our method, since we compute over-approximations of $M[\![P]\!]$ by abstract interpretation. That is, if $\mathsf{false} \notin M'$ where $M[\![P]\!] \subseteq M'$ then we have shown that P has a model.

3.3 Trace Automata for CHCs

Before constructing the trace automaton we introduce identifiers for each clause. An identifier is a function symbol whose arity is the same as the number of atoms in the clause body. For instance a clause $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ is assigned a function symbol with arity k. More than one clause can be assigned the same function symbol, but all the clauses with the same identifier have the same structure, including their constraints; that is, they differ only in one or more predicate names. Given a set of CHCs and a set Σ of ranked function symbols, let $\mathsf{id}_P : P \to \Sigma$ be the assignment of function symbols to clauses.

Definition 7 (Trace FTA for a set of CHCs). *Let P be a set of CHCs. Define the trace FTA for P as $\mathcal{A}_P = (Q, Q_f, \Sigma, \Delta)$ where*

- *Q is the set of predicate symbols of P;*
- *$Q_f \subseteq Q$ is the set of predicate symbols occurring in the heads of clauses of P;*
- *Σ is a set of function symbols;*
- *$\Delta = \{c(p_1, \ldots, p_k) \to p \mid \text{where } c \in \Sigma,\ c = \mathsf{id}_P(cl), \text{where } cl = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)\}$.*

The elements of $\mathcal{L}(\mathcal{A}_P)$ are called trace terms for P. In Section 4 we will see that several clauses differing only in their predicate names are assigned the same function symbol.

To motivate readers, we present an example set of CHCs P in Figure 1 which will be used throughout this paper. This is an interesting problem in which the computations are trees rather than linear sequences.

Example 1. Let P be the set of CHCs in Figure 1. Let id_P map the clauses to $c_1, \ldots, c_4$ respectively. Then $\mathcal{A}_P = (Q, Q_f, \Sigma, \Delta)$ where:

$$\begin{array}{ll} Q = \{\texttt{mc91}, \texttt{false}\} & \Delta = \{c_1 \to \texttt{mc91}, \\ Q_f = \{\texttt{mc91}, \texttt{false}\} & \quad c_2(\texttt{mc91}, \texttt{mc91}) \to \texttt{mc91}, \\ \Sigma = \{c_1, c_2, c_3, c_4\} & \quad c_3(\texttt{mc91}) \to \texttt{false},\ c_4(\texttt{mc91}) \to \texttt{false}\} \end{array}$$

For each trace term there exists a corresponding derivation tree called an AND-tree, which is unique up to variable renaming. The concept of an AND-tree is derived from [33] and [16].

Definition 8 (AND-tree for a trace term). *Let P be a set of CHCs and let $t \in \mathcal{L}(\mathcal{A}_P)$. Denote by $\mathsf{AND}(t)$ the following labelled tree, where each node of $\mathsf{AND}(t)$ is labelled by a clause and an atomic formula.*

1. *For each subterm $c_j(t_1, \ldots, t_k)$ of t there is a corresponding node in $\mathsf{AND}(t)$ labelled by an atom $p(X)$ and (a renamed variant of) some clause $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ such that $c_j = \mathsf{id}_P(p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k))$; the node's children (if $k > 0$) are the nodes corresponding to $t_1, \ldots, t_k$ and are labelled by $p_1(X_1), \ldots, p_k(X_k)$.*
2. *The variables in the labels are chosen such that if a node n is labelled by a clause, the local variables in the clause body do not occur outside the subtree rooted at n.*

Definition 9 (Trace constraints). *Let P be a set of CHCs. The set of constraints of a trace $t \in \mathcal{L}(\mathcal{A}_P)$, represented as $\mathsf{constr}(t)$ is the set of all constraints in the clause labels of $\mathsf{AND}(t)$.*

Definition 10 (Feasible trace). *We say that a trace term t is feasible if $\mathsf{constr}(t)$ is satisfiable.*

Definition 11 (FTA for a trace term). *Let P be a set of CHCs and $t \in \mathcal{L}(\mathcal{A}_P)$. The FTA A_t (whose construction is trivial) such that $\mathcal{L}(\mathcal{A}_t) = \{t\}$ is called* the FTA for t. *The states of $\mathcal{A}_t$ are chosen to be disjoint from those of $\mathcal{A}_P$.*

Example 2 (Trace FTA). Consider the FTA in Example 1. Let $t = c_3(c_2(c_1, c_1))$. Each $\mathtt{node_i}$ represents a label in the trace. Then $\mathcal{A}_t = (Q, Q_f, \Sigma, \Delta)$ is defined as:

$$\begin{aligned} Q &= \{\mathtt{node_1}, \mathtt{node_2}, \mathtt{node_3}, \mathtt{node_4}\} \\ Q_f &= \{\mathtt{node_1}\} \\ \Sigma &= \{c_1, c_2, c_3, c_4\} \\ \Delta &= \{c_1 \to \mathtt{node_3},\ c_1 \to \mathtt{node_4},\ c_2(\mathtt{node_3}, \mathtt{node_4}) \to \mathtt{node_2}, \\ &\quad\ c_3(\mathtt{node_2}) \to \mathtt{node_1}\} \end{aligned}$$

and Σ is the same as in $\mathcal{A}_P$. The trace t is not feasible since $\mathsf{constr}(t) = \{\mathtt{A} \leq 100, \mathtt{B} > 91, \mathtt{A} \leq 100, \mathtt{C} = \mathtt{A} + 11, \mathtt{C} > 100, \mathtt{D} = \mathtt{C} - 10, \mathtt{D} > 100, \mathtt{B} = \mathtt{D} - 10\}$ and this is not satisfiable.

Definition 12 (Constrained trace atom). *Let P be a set of CHCs and $t \in \mathcal{L}(\mathcal{A}_P)$. Let $p(X)$ be the atom labelling the root of $\mathsf{AND}(t)$. Then the constrained trace atom of t is $\forall X.(\exists \bar{Z}.\mathsf{constr}(t) \to p(X))$, where $\bar{Z} = \mathsf{vars}(\mathsf{constr}(t)) \setminus X$.*

We now restate a standard result from constraint logic programming [25] in terms of the concepts defined above.

Proposition 1. *Let P be a set of CHCs.*

1. *Then for all $t \in \mathcal{L}(\mathcal{A}_P)$ the constrained trace atom for t is a logical consequence of P. (Note that if t is not feasible this is trivially true).*

2. *If $p(a)$ is in the minimal model of P, there exists a feasible trace $t \in \mathcal{L}(\mathcal{A}_P)$ whose constrained trace atom is of the form $\forall X.\phi \rightarrow p(X)$ where the constraint $\phi[X/a]$ is true.*

Assuming that the constraint theory has a complete satisfiability procedure, part 1 of Proposition 1 corresponds to the standard soundness result for resolution-based proof systems, and part 2 corresponds to completeness.

3.4 Model-Preserving Transformation of Trace Automata

Proposition 1 implies that the constrained trace atoms for the feasible traces describe exactly the elements of the minimal model, which is equivalent to the set of atomic logical consequences of P. As a consequence the set of feasible traces in $\mathcal{L}(\mathcal{A}_P)$ can be regarded as a representation of the minimal model of P.

If we transform $\mathcal{A}_P$ to another FTA while preserving the set of traces, we also preserve the feasible traces. More generally, we can transform $\mathcal{A}_P$ to another FTA $\mathcal{A}'$ so long as $\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A}_P)$ and the elements of $\mathcal{L}(\mathcal{A}_P) \setminus \mathcal{L}(\mathcal{A}')$ are all infeasible. In this case the feasible traces of $\mathcal{L}(\mathcal{A}')$ are still a representation of the minimal model of P. We will exploit this in our refinement procedure (see Section 4).

3.5 Generation of CHCs from a Trace FTA

Now we describe a procedure (Algorithm 1) for generating a set of clauses P' from an FTA $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$ and a set of clauses P. We assume that Σ is the same as that of $\mathcal{A}_P$; so Σ is the range of the function id_P mapping clauses of P to function symbols. The transitions Δ are not in product form; a modification of the algorithm and its correctness proposition is possible for product form but we omit that here. We first introduce an injective function for renaming the states of $\mathcal{A}$ since we need predicate names for the generated clauses.

$$\rho : Q \rightarrow \mathsf{Predicates}$$

The function ρ maps each FTA state to a distinct predicate name. The algorithm simply generates a clause for each transition, applying the renaming function from states to predicates, and introducing variables arguments according to the pattern obtained from any clause with the corresponding identifier (all clauses with the same identifier having the same variable pattern).

Apart from generating a set of clauses P', Algorithm 1 also generates the clause identification mapping $\mathsf{id}_{P'}$, preserving the function symbols from the FTA. In this way the set of traces is preserved from P to P'. The correctness of Algorithm 1 is expressed by the following proposition.

Proposition 2. *Let P be a set of CHCs and let $\mathcal{A}$ be an FTA whose signature is the same as that of $\mathcal{A}_P$. Let P' be the set of clauses generated from $\mathcal{A}$ and P by Algorithm 1. Then $\mathcal{L}(\mathcal{A}_{P'}) = \mathcal{L}(\mathcal{A})$. Furthermore if $\mathcal{L}(\mathcal{A}_{P'})$ includes all the feasible traces of $\mathcal{L}(\mathcal{A}_P)$ then the minimal model of P' is the same as the minimal model of P, modulo predicate renaming.*

Input: An FTA $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$ and a set of Horn clauses P
Output: A set of Horn clauses P'
$P' \leftarrow \emptyset$;
for *each* $c_i(q_1, \ldots, q_n) \rightarrow q$ *(where* $n \geq 0$*)* $\in \Delta$ **do**
  let $c = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n)$ be any clause in P where $\mathsf{id}_P(c) = c_i$;
  $c_{new} = \rho(q)(X) \leftarrow \phi, \rho(q_1)(X_1), \ldots, \rho(q_n)(X_n)$;
  $\mathsf{id}_{P'}(c_{new}) = c_i$;
  $P' \leftarrow P' \cup \{c_{new}\}$;
end
return P';

Algorithm 1. ALGORITHM for generating a set of clauses from an FTA

Example 3 (Generation of clauses from an FTA). Consider the following transitions, relating to the signature for the program in Figure 1. The set of states is {`[false],[mc91],[e,false],[mc91,e1]`}. (These are elements of the powerset of the set of states {`false,mc91,e,e1`}, which were generated by the determinisation algorithm).

```
c1 -> [mc91, e1].
c2([mc91, e1],[mc91, e1]) -> [mc91].
c2([mc91],[mc91]) -> [mc91].
c2([mc91, e1],[mc91]) -> [mc91].
c2([mc91],[mc91, e1]) -> [mc91].
c3([mc91]) -> [false].
c4([mc91, e1]) -> [false].
c4([mc91]) -> [false].
c3([mc91, e1]) -> [e, false].
```

The clauses generated by Algorithm 1 are the following, with the renaming function $\rho = \{[\texttt{false}] \mapsto \texttt{false}, [\texttt{mc91}] \mapsto \texttt{mc91}, [\texttt{e}, \texttt{false}] \mapsto \texttt{false\_1}, [\texttt{mc91}, \texttt{e1}] \mapsto \texttt{mc91\_1}\}$. Below we also show the clause identifiers (the id function for the generated clauses) showing that several clauses can have the same identifier, thus preserving traces.

```
c1: mc91_1(A,B) :-  A>100, B=A-10.
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91_1(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91_1(D,B).
c3: false :- A =< 100, B > 91, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91_1(A,B).
c3: false_1 :- A =< 100, B > 91, mc91_1(A,B).
```

3.6 Abstract Interpretation of Constrained Horn Clauses

Abstract interpretation [10] is a static program analysis techniques which derives sound over-approximations by computing abstract fixed points. Convex polyhedron analysis (CPA) [11] is a program analysis technique based on abstract interpretation [10]. When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \phi$ for each predicate p. The constraint ϕ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CHCs was by Benoy and King [4].

We summarise briefly the elements of convex polyhedron analysis for CHC; further details (with application to CHC) can be found in [11,4]. The abstract interpretation consists of the computation of an increasing sequence of elements of the abstract domain of tuples of convex polyhedra (one for each predicate) $\mathcal{D}^n$. We construct a monotonic *abstract semantic function* $F_P : \mathcal{D}^n \rightarrow \mathcal{D}^n$ for the set of Horn clauses P, approximating the concrete semantic "immediate consequences" operator. Since $\mathcal{D}^n$ contains infinite increasing chains, a *widening* operator for convex polyhedra [11] is needed to ensure convergence of the sequence. The sequence computed is $Z_0 = \bot^n$, $Z_{n+1} = Z_n \nabla F_P(Z_n)$ where ∇ is a widening operator for convex polyhedra and the empty polyhedron is denoted $\bot$. The conditions on ∇ ensure that the sequence stabilises; thus for some finite j, $Z_i = Z_j$ for all $i > j$ and furthermore the value Z_j represents an over-approximation of the least model of P. Much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds [22]. A threshold is an assertion that is combined with a widening operator to improve its precision.

Our tool for convex polyhedral abstract interpretation, called CPA in the rest of this paper, uses the Parma Polyhedra Library [2] to implement the operations on convex polyhedra, and incorporates a threshold generation phase based on the method described by Lakhdar-Chaouch *et al.* [27], as well as a constraint strengthening pre-processing which propagates constraints both forwards and backwards in the clauses of P. Space does not permit a detailed explanation.

4 Refinement of Horn Clauses Using Trace Automata

If an over-approximation of the clauses derived by polyhedral abstraction does not contain false, the clauses are safe. However if false is contained in the approximation, we do not know whether the clauses are unsafe or whether the approximation was too imprecise. In such cases we can produce a trace term using the clauses in P which justifies the abstract derivation of false. The feasibility of this trace can be checked by a constraint satisfiability check. If the trace is feasible, then it corresponds to a proof of unsafety. Otherwise, refinement is considered based on this trace. In some approaches, a more precise abstract domain is derived from the trace. In our refinement approach, which is described next, we aim to generate a modified set of clauses that could yield a better approximation. This is achieved through the steps shown in Algorithm 2.

Input: A set of Horn clauses P and an infeasible trace t
Output: A set of Horn clauses P'

1. construct the trace FTA $\mathcal{A}_P$ (Definition 7);
2. construct an FTA $\mathcal{A}_t$ such that $\mathcal{L}(\mathcal{A}_t) = \{t\}$ (Definition 11);
3. compute the difference FTA $\mathcal{A}_P \setminus \mathcal{A}_t$ (Definition 5);
4. generate P' from $\mathcal{A}_P \setminus \mathcal{A}_t$ and P (Algorithm 1) ;

return P';

Algorithm 2. ALGORITHM for clause refinement

Both $\mathcal{A}_P$ and $\mathcal{A}_t$ in Algorithm 2 are deterministic by construction, however their union is not. Determinisation is used to generate the difference FTA (step 3) and its result is in product form. The program P' has the same model (modulo predicate renaming) as P, since the steps result in the removal of an infeasible trace but all other traces are preserved.

Removal of one trace from the clauses might not seem much of a refinement. However, the restructuring of the clauses required to remove a trace can split the predicates. This restructuring is the effect of determinisation, which isolates the infeasible trace. This in turn can induce a more precise abstract interpretation, with less precision loss due to convex hull operations and widening.

The correctness of this refinement follows from Proposition 2. In particular $\mathsf{false} \in M[\![P]\!]$ if and only if $\mathsf{false} \in M[\![P']\!]$ (assuming that the predicate renaming at least preserves the predicate name false).

Example 4. Consider again the FTA shown in Example 3. This is in fact the determinisation of $\mathcal{A}_P \cup \mathcal{A}_t$ where P is the set of clauses in Figure 1 and $\mathcal{A}_t$ where t is the infeasible trace `c3(c1)`. The only accepting state of $\mathcal{A}_t$ is `e`; thus to construct the difference $\mathcal{A}_P \setminus \mathcal{A}_t$ we need only to remove from the automaton the states containing `e`, namely `[mc91,e]`. We can also remove any transitions containing this state in the right hand side. This leaves the following FTA and refined program, using the same renaming function as in Example 3. In this program, the infeasible trace corresponding to `c3(c1)` cannot be constructed.

```
c1 -> [mc91, e1].
c2([mc91, e1],[mc91, e1]) -> [mc91].
c2([mc91],[mc91]) -> [mc91].
c2([mc91, e1],[mc91]) -> [mc91].
c2([mc91],[mc91, e1]) -> [mc91].
c3([mc91]) -> [false].
c4([mc91]) -> [false].
c4([mc91, e1]) -> [false].
```

```
c1: mc91_1(A,B) :-  A>100, B=A-10.
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91_1(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91_1(D,B).
```

```
c3: false :- A =< 100, B > 91, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91_1(A,B).
```

It can be seen that although the infeasible trace was very simple, its removal led to a considerably restructured set of clauses. We have not shown the product form here, which is in fact somewhat more compact.

The refinement process guarantees progress; that is, the infeasible computation once eliminated never arises again. Due to the construction of the id mapping for P' the traces in the languages of the FTAs of P and P' are preserved, apart from the eliminated trace.

Proposition 3 (Progress). *Let P be a set of CHCs, and t be a trace in P. Let P' be a refined set of CHCs obtained from P after the removal of t. Then t cannot be generated in any approximation of P'.*

After the removal of the trace t (step 3 of Algorithm 2) the language of $\mathcal{A}_P \setminus \mathcal{A}_t$ does not contain t. Then using Algorithm 1 to generate P', t will not be a possible trace in P'. It is physically impossible to construct t, in any abstract domain.

4.1 Further Refinement: Splitting a State in the Trace FTA

We also apply a tree-automata-based transformation to split states representing predicates where convex hull operations have lost precision. A typical case is where a number of clauses with the same head predicate contain disjoint constraints, such as a predicate representing an if-then-else statement in an imperative program. The clauses defining the statement will have a clause for the *then* branch and a clause for the *else* branch. The respective constraints in these clauses are disjoint since one is the negation of the other. The convex hull will thus contain the whole space for the variables involved in these constraints.

As defined in Definition 6, the FTA state corresponding to such a predicate can be split. We partition the transitions corresponding to the clauses according to the disjoint groups of constraints and apply the procedure in Definition 6, preserving the set of traces. Thus the feasible traces and the model of the resulting clauses is preserved. This enhances precision of polyhedral analysis [15].

Splitting has to be carried out in a controlled manner to prevent blow up in the size of FTA and hence on the size of the clauses generated. With this in mind we split only those states appearing in a counterexample trace.

5 Experiments on CHC Benchmark Problems

Our tool consists of an implementation of a *convex polyhedra analyser* for CLP written in Ciao Prolog[1] interfaced to the Parma Polyhedra Library [2] as well as an implementation of an FTA determiniser written in Java. It takes as input a

[1] http://ciao-lang.org/

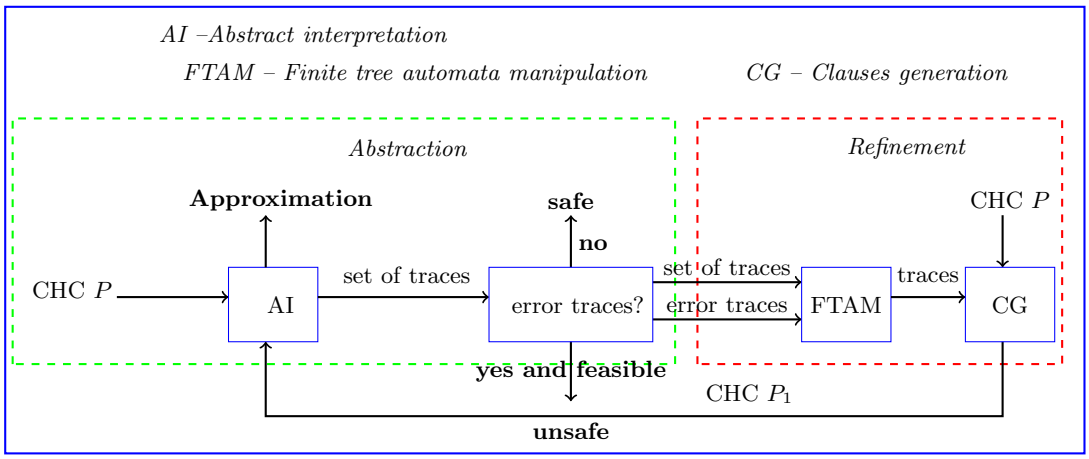


Fig. 2. *Abstraction-refinement scheme in Horn clause verification*

CLP program and returns "safe", "unsafe" or "unknown" (after timeout). The benchmark set contains 216 CHCs verification problems (179 safe and 37 unsafe problems), taken mainly from the repositories of several state-of-the-art software verification tools such as DAGGER [19] (21 problems), TRACER [26] (66 problems), InvGen [21] (68 problems), and also from the TACAS 2013 Software Verification Competition [5] (52 problems). Most of these problems are available in `C` and they were first translated to CLP form[2]. The chosen problems are representatives of different categories of the Software Verification Competition (loops, control flow and integer, SystemC etc.) as well as specific problems used to demonstrate the strength of different verification tools. The benchmarks are available from `http://akira.ruc.dk/~kafle/VMCAI15-Benchmarks.zip`. The experiments were carried out on an Intel(R) quad-core computer with a 2.66GHz processor running Debian 5 in 6 GB memory.

5.1 Summary of Results

The results of our experiments are summarised in Table 3. Column CPA summarises the results using our own *convex polyhedra analyser* (Section 3.6) with no refinement step. Column CPA+R shows the results obtained by iterating the CPA algorithm with the refinement step described in Section 4, Algorithm 2. Column CPA+R+Split incorporates the FTA-based state splitting into the refinement step (Section 4.1). Column QARMC shows the results obtained on the same problems using the QARMC tool [31].

5.2 Discussion of Results

The results show that CPA is reasonably effective on its own, solving 74% (160/216) of the problems, though it times out for seven problems. When combined with a refinement phase we can solve 22 further problems. Although only

[2] Thanks to Emanuele De Angelis for the translation.

	CPA	CPA+R	CPA+R+Split	QARMC
solved (safe/unsafe)	160 (142/18)	182 (160/22)	195 (164/31)	178 (141/37)
unknown/ timeout	49/7	-/34	-/22	-/38
average time (secs.)	5.98	51.66	50.08	59.1
% solved	74	84.25	90.27	82.4

Fig. 3. Experimental results on 216 (179 safe / 37 unsafe) CHC verification problems with a timeout of five minutes

one infeasible trace is eliminated in each refinement step, the refined program splits some of the predicates appearing in the trace, which we noted to be a crucial point of precision for polyhedral analysis [15]. When adding the state splitting refinement we solve an additional 13 problems. Further splitting would solve more problems but we are unwilling to introduce uncontrolled splitting due to the blow up in program size that could result. The maximum number of iterations required to solve a problem was 8. Although the timeout limit was five minutes, only 5% of the solved problems required more than one minute. QARMC tends to perform more (but faster) iterations.

Our implementation uses the product form for DFTAs produced by the determinisation algorithm, although the formalisation of refinement in Section 4 uses only standard FTA transitions. Although the traces for clauses with predicates produced from product states differ from the original clauses, they can be regarded as representing the original traces, by unfolding the clauses resulting from ϵ-transitions. Product form adds to the scalability of the approach, especially for Horn clauses with more than one body atom.

5.3 Comparison with Other Tools

Our results improve on QARMC both in average time and the number of instances solved. Out of 216 problems QARMC solves 178 problems with an average time of 59 seconds whereas we can solve 195 problems with an average time of 50 seconds. However, all unsafe programs in the benchmark set are solved by QARMC in contrast to ours. Convex polyhedral analysis is good at finding the required invariants to prove a program safe and due to this we solved more safe problems than QARMC. QARMC seems to be more effective at finding bugs. Most of the problems challenging to us come from particular categories e.g. SystemC (modelled over fixed size integers) and Control Flow and Integer Variables of [5] which requires some specific techniques to solve. Safe problems challenging to us are also challenging to QARMC though this is not the case for unsafe problems.

6 Related Work

The work by Heizmann et al. [23,24] uses word automata to construct a framework for abstraction refinement. Our work could certainly be regarded as

extending that framework to tree-structured computations, using tree automata instead of (nested) word automata. However our aim is rather different. We use automata techniques to *perform* the refinement whereas in [23] automata notation is only used to re-express the verification problem, shifting the verification problem to the construction of "interpolant automata", without providing any automata-based algorithms to do this. On the other hand we discuss the practicality of the automata-based approach on a set of challenging problems.

While we eliminate only one trace at a time in the described procedure, the FTA difference algorithm extends naturally to eliminating (infinite) sets of traces. However in our setting that does not seem a useful goal – to find an automaton describing an infinite set of infeasible traces often amounts to solving the original problem.

Verification of CLP programs using abstract interpretation and specialisation has been studied for some time. The use of an over-approximation of the semantics of a program can be used to establish safety properties – if a state or property does not appear in an over-approximation, it certainly does not appear in the actual program behaviour. A general framework for logic program verification through abstraction was described by Levi [29]. Peralta *et al.* [30] introduced the idea of using a Horn clause representation of imperative languages and a convex polyhedral analyser to discover invariants of a program. Another approach is taken in the work of De Angelis *et al.* [12,13] on applying program specialisation to achieve verification. Unfolding and folding operations play a vital role in that approach, and hence the program structure is changed much more fundamentally than in our approach.

CEGAR [8] has been successfully used in verification to automatically refine (predicate) abstractions [7,28] to reduce false alarms but not much has been explored in refining abstractions in the convex polyhedral domain. Some work on this (with progress guarantee) has been done in [1] and [19]. [1] uses the powerset domain, while [19] uses a Hint DAG to gain precision lost during the convex hull operation. Both make use of interpolation. The use of interpolation in refinement in verification of Horn clauses is explored in [6,20]. In our approach we guarantee elimination of only one trace and elimination of others depends on properties of the abstract interpretation techniques. By contrast in interpolation-based techniques the refinement introduces new properties which guarantee progress and the elimination of all counterexamples covered by those properties. However the effectiveness of interpolation-based refinement depends on the generation of "good" interpolants, which is a matter of continuing research, for example by Rümmer *et al.* [32]. A number of tools implementing predicate abstraction and refinement are available, such as HSF [18] and BLAST [3]. TRACER [17] is a verification tool based on CLP that uses symbolic execution.

A point of contrast is that in our approach, the refinements are embedded in the clauses whereas in CEGAR they are accumulated in the set of properties used for property-based abstraction. Also we rely on the abstraction using convex polyhedral analysis to discover invariants whereas CEGAR-based approaches rely on interpolation in the refinement stage to discover relevant

properties. Polyhedral analysis is more expensive, yet seems (along with the threshold assertions, see Section 3.6) to be very effective at finding invariants even on the first iteration. A weakness of invariant generation using interpolation is that the interpolants must share variables with the unsatisfiable part of the constraints, typically those in the integrity constraints, which can be insufficient for finding invariants of inner recursive predicates. Informally one can say that approaches differ in where the "hard work" is performed. In the CEGAR approaches and in [23] the refinement step is crucial, and interpolation plays a central role. In our approach, by contrast, most of the hard work is done by the abstract interpretation, which finds useful invariants. Finding the most effective balance between abstraction and refinement techniques is a matter of ongoing research.

7 Conclusion and Future work

In this paper we presented a procedure for abstraction refinement in Horn clause verification based on tree automata. This was achieved through a combination of abstraction (using abstraction interpretation) followed by a trace refinement (using finite tree automata). The refinement is independent of the abstract domain used. The practicality of our approach was demonstrated on a set of Horn clause verification problems.

In the future, we will investigate the elimination of a larger set of infeasible traces in each refinement step, possibly by generalising a trace using interpolation or by discovering a set of infeasible traces. The optimisation of our tool chain is also an important topic for future work as it is clear that our prototype, built by chaining together tools using shell scripts, contains much redundancy.

Acknowledgements. We thank the anonymous referees for useful comments.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 300–316. Springer, Heidelberg (2012)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1–2), 3–21 (2008)
3. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM 54(7), 68–76 (2011)
4. Benoy, F., King, A.: Inferring argument size relationships with CLP($\mathcal{R}$). In: Gallagher, J. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)
5. Beyer, D.: Second competition on software verification - (summary of SV-COMP 2013). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)

6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)
7. Burke, M., Soffa, M.L. (eds.): Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22. ACM (2001)
8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
9. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), `http://www.grappa.univ-lille3.fr/tata` (release October 12, 2007)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) POPL, pp. 238–252. ACM (1977)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages, pp. 84–96 (1978)
12. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying programs via iterated specialization. In: Albert, E., Mu, S.-C. (eds.) PEPM, pp. 43–52. ACM (2013)
13. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: VeriMAP: A tool for verifying programs through transformations. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 568–574. Springer, Heidelberg (2014)
14. Gallagher, J.P., Ajspur, M., Kafle, B.: An Optimised Algorithm for Determinisation and Completion of Finite Tree Automata. Technical Report 145, Roskilde University, Denmark, (September 2014), `http://akira.ruc.dk/~jpg/dfta.pdf`
15. Gallagher, J.P., Kafle, B.: Analysis and transformation tools for constrained Horn clause verification. TPLP, 14(4-5) (additional materials in online edition), 90–101 (2014)
16. Gallagher, J.P., Lafave, L.: Regular approximation of computation paths in logic and functional languages. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 115–136. Springer, Heidelberg (1996)
17. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Failure tabled constraint logic programming by interpolation. TPLP 13(4-5), 593–607 (2013)
18. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A software verifier based on Horn clauses - (competition contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
19. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
20. Gupta, A., Popeea, C., Rybalchenko, A.: Solving recursion-free horn clauses over LI+UIF. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 188–203. Springer, Heidelberg (2011)
21. Gupta, A., Rybalchenko, A.: InvGen: An efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
22. Halbwachs, N., Proy, Y.E., Raymound, P.: Verification of linear hybrid systems by means of convex approximations. In: LeCharlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 223–237. Springer, Heidelberg (1994)

23. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009)
24. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of POPL 2010, pp. 471–482. ACM (2010)
25. Jaffar, J., Maher, M.: Constraint Logic Programming: A Survey. Journal of Logic Programming 19/20, 503–581 (1994)
26. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: TRACER: A symbolic execution tool for verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 758–766. Springer, Heidelberg (2012)
27. Lakhdar-Chaouch, L., Jeannet, B., Girault, A.: Widening with thresholds for programs with complex control graphs. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 492–502. Springer, Heidelberg (2011)
28. Launchbury, J., Mitchell, J.C. (eds.): Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18. ACM (2002)
29. Levi, G.: Abstract interpretation based verification of logic programs. Electr. Notes Theor. Comput. Sci. 40, 243 (2000)
30. Peralta, J.C., Gallagher, J.P., Saglam, H.: Analysis of imperative programs through analysis of constraint logic programs. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 246–261. Springer, Heidelberg (1998)
31. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
32. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for horn-clause verification. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 347–363. Springer, Heidelberg (2013)
33. Stärk, R.F.: A direct proof for the completeness of SLD-resolution. In: Börger, E., Kleine Büning, H., Richter, M.M. (eds.) CSL 1989. LNCS, vol. 440, pp. 382–383. Springer, Heidelberg (1990)

Attachment D3.3.5

Probabilistic Resource Analysis by Program Transformation

Accepted for publication at the Foundational and Practical Aspects of Resource Analysis (FOPARA 2015)

Probabilistic Resource Analysis by Program Transformation

Maja H. Kirkeby and Mads Rosendahl

Computer Science, Roskilde University
Roskilde, Denmark
majaht@ruc.dk, madsr@ruc.dk **

Abstract. The aim of a probabilistic resource analysis is to derive a probability distribution of possible resource usage for a program from a probability distribution of its input. We present an automated multi-phase rewriting based method to analyze programs written in a subset of C. It generates a probability distribution of the resource usage as a possibly uncomputable expression and then transforms it into a closed form expression using over-approximations. We present the technique, outline the implementation and show results from experiments with the system.

1 Introduction

The main contribution in this paper is to present a technique for probabilistic resource analysis where the analysis is seen as a program-to-program translation. This means that the transformation to closed form is a source code program transformation problem and not specific to the analysis. Any necessary approximations in the analysis are performed at the source code level. The technique also makes it possible to balance the precision of the analysis against the brevity of the result.

Many optimizations for increased energy efficiency require probabilistic and average case analysis as part of the transformations. Wierman et al. states that *"global energy consumption is affected by the average case, rather than the worst case"* [37]. Also in scheduling *"an accurate measurement of a task's average-case execution time can assist in the calculation of more appropriate deadlines"* [17]. For a subset of programs a precise average case execution time can be found using static analysis [12, 31, 14]. Applications of such analysis may be in improving scheduling of operations or in temperature management. Because the analysis returns a distribution it can be used to calculate the probability of energy consumptions above a certain limit, and thereby indicating the risk of over-heating.

** The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

The central idea in this paper is to use probabilistic output analysis in combination with a preprocessing phase that instruments programs with resource usage. We translate programs into an intermediate language program that computes the probability distribution of resource usage. This program is then analyzed, transformed, and approximated with the aim of obtaing a closed form expression. It is an alternative to deriving cost relations directly from the program [7] or expressing costs as abstract values in a semantics for the language.

As with automatic complexity analysis the aim of probabilistic resource analysis is to express the result as a parameterized expression. The time complexity of a program should be expressed as a closed form expression in the input size, and for probabilistic resource analysis the aim is to express the probability of resource usage of the program parameterized by input size or range. If input values are not independent we can specify a joint distribution for the values. Values do not have to be restricted to a finite range but for infinite ranges the distribution would converge to zero towards the limit.

The current work extends our previous work on probabilistic analysis [29] in three ways. We show how to use a preprocessing phase to instrument programs with resource usage such that the resource analysis can be expressed as an analysis of possible output of a program. The resource analysis can handle an extended class of programs with structured data as long as the program flow does not depend on the probabilistic data in composite data structures. Finally, we present an implementation of the analysis in the Ciao language [5] which uses algebraic reductions in the Mathematica system [39].

The focus in this paper is on using fairly simple local resource measures where we count core operations on data. Since the instrumentation is done at the source code level we can use flow information so that the local costs can depend on actual data to operations and which operations are executed before and after. This is not normally relevant for time complexity but does play an important role for energy consumption analysis [19, 32].

2 Probability distributions in static analysis

In our approach to probabilistic analysis the result of an analysis is an approximation of a probability distribution. We will here present the concepts and notation we will use in the rest of the paper. A probability distribution is also often referred to as the *probability mass function* in the discrete case, and in the continuous case it is a *probability density functions*. We will use an upper case P letter to denote a probability distribution.

Definition 1 (input probability). *For a countable set X an input probability distribution is a mapping $P_x : X \to \{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$, where*

$$\sum_{x \in X} P_x(x) = 1$$

We define the output probability distribution for a program $\mathtt{p}$ in a forward manner. It is the *weight* or sum of all probabilities of input values where the program returns the desired value z as output.

Definition 2 (output probability). *Given a program,* $\mathtt{p} : X \to Z$ *and a probability distribution for the input,* P_X*, the output probability distribution,* $P_{\mathtt{p}}(z)$*, is defined as:*

$$P_{\mathtt{p}}(z) = \sum_{x \in X \wedge \mathtt{p}(x) = z} P_X(x)$$

Note that Kozen also uses a similar forward definition [20], whereas Monniaux constructs the inverse mapping from output to input for each program statement and express the relationship in a backwards style [23].

Lemma 1. *The output probability distribution,* $P_{\mathtt{p}}(z)$*, satisfies*

$$0 \leq \sum_{z} P_{\mathtt{p}}(z) \leq 1$$

The program may not terminate for all input and this means that the sum may be less than one. If we expand the domain Z with an element to denote non-termination, $Z_\perp$, the total sum of the output distribution $P_{\mathtt{p}}(z)$ would be 1.

In our static analysis we will use approximations to obtain safe and simplified results. Various approaches to approximations of probability distributions have been proposed and can be interpreted as *imprecise probabilities* [1, 10, 9]. Dempster-Shafer structures [16, 2] and P-boxes [11] can be used to capture and propagate uncertainties of probability distributions. There are several results on extending arithmetic operations to probability distributions for both known dependencies between random variables and when the dependency is unknown or only partially known [3, 4, 18, 33, 38]. Algorithms for lifting basic operations on numbers to basic operations on probability distributions can be used as abstractions in static analysis based on abstract interpretation. Our approach uses the P-boxes as bounds of probability distributions. P-boxes are normally expressed in terms of the cumulative probability distribution but we will here use the probability mass function. We do not, however, use the various basic operations on P-boxes, but apply approximations to a probability program such that it forms a P-box.

Definition 3 (over-approximation). *For a distribution* $P_{\mathtt{p}}$ *an over-approximation (*$\overline{P}_{\mathtt{p}}$*) of the distribution satisfies the condition:*

$$\overline{P}_{\mathtt{p}} : \forall z. P_{\mathtt{p}}(z) \leq \overline{P}_{\mathtt{p}}(z) \leq 1 \ .$$

The aim of the probabilistic resource analysis is to derive an approximation $\overline{P}_{\mathtt{p}}$ as tight as possible.

The over-approximation of the probability distribution can be used to derive lower and upper bounds of the expected value and will thus approximate the expected value as an interval [29].

3 Architecture of the transformation system

The system contains five main phases. The input to the system is a program in a small subset of C with annotations of which part we want to analyze. It could be the whole program but can also be a specific subroutine which is called repeatedly with varying arguments according to some input distribution.

The first phase will instrument the program with resource measuring operations. The instrumented program will perform the same operations as the original program in addition to recording and printing resource usage information. This program can still be compiled and run, and it will also produce the same results as the original program.

The second phase translates the program into an intermediate language for further analysis. We use a small first order functional language for the analysis process. The translation has two core elements. We slice [36] the program with respect to the resource measuring operations and transform loops into primitive recursion in the intermediate language. The transformed program can still be executed and will produce the same resource usage information as the instrumented program. Since the instrumentation is done before the translation to intermediate language any interpretation overhead or speed-up due to slicing does not influence the result [28].

In the third phase we construct a probability output program that computes the probability output function. In this case it is a probability distribution of possible resource usages of the original program. This program can also run but will often be extremely inefficient since it will merge information for all possible input to the original program.

The fourth phase transforms the probability program into a large expression without further function calls. Recursive calls are removed using summations and the transformed program computes the same result as the program did before this phase.

In the final phase the probability function is transformed into closed form using symbolic summation and over-approximation. In this phase we exploit the Mathematica system [39]. The final probability program computes the same result or an over-approximation of the function produced in the fourth phase.

4 Instrumenting programs for resource analysis

The input to the analysis is a program in a subset of C. In the next section we define the intermediate language for further analysis and it is the restrictions on the intermediate language that limits the source programs we can analyze with our system. The source program may contain integer variable and arrays, usual loop constructs and non-recursive function calls. The program should be annotated with specification on which part of the program to analyse. The following is an example of such a program.

```
// ToAnalyse: multa(_,_,_,N)
```

```
void multa(int a1[MX],int a2[MX],int a3[MX],int n){
  int i1,i2,i3,d;
  for(i1 = 0; i1 < n; i1++) {
    for(i2 = 0; i2 < n; i2++) {
      d = 0;
      for(i3 = 0; i3 < n; i3++) {
         d = d + a1[i1*n+i3]*a2[i3*n+i2];
      }
      a3[i1*n+i2] = d;
    }
  }
}
```

This example program describes a matrix multiplication for which we would like to analyze the probability distribution for the number of steps when parameterized with the size (`N`) of the matrices.

Instrumentation. The program is then instrumented with resource usage information and translated into a intermediate language for further analysis. The instrumented program is also a valid program in the source language and can be executed with the same results as the original program. It will, however, also collect resource usage information.

In our example we instrument the program with step counting information where we count the number of assignment statement being executed. This is done by inserting a variable into the program and incrementing it once for each assignment statement.

```
int multa(int a1[MX],int a2[MX],int a3[MX],int n){
  int i1,i2,i3,d;
  int step; step=0;
  for(i1 = 0; i1 < n; i1++) {
    for(i2 = 0; i2 < n; i2++) {
      d = 0; step++;
      for(i3 = 0; i3 < n; i3++) {
         d = d + a1[i1*n+i3]*a2[i3*n+i2]; step++;
      }
      a3[i1*n+i2] = d; step++;
    }
  }
  return step;
}
```

The outer loop does not update the step counter, whereas the first inner loop updates it twice per iteration and the innermost loop updates it once per loop iteration.

Slicing. The second phase will slice the program with respect to resource usage and translate the program into the intermediate language of first order functions that we will use in the subsequent stages. Loops in the program are translated into primitive recursion.

```
for3(i3, step, n) =
  if(i3 = n) then step else for3(i3 + 1,step+1,n)

for2(i2, step, n) =
  if(i2 = n) then step else for2(i2 + 1,for3(0,step+2,n),n)

for1(i1, step, n) =
  if(i1 = n) then step else for1(i1 + 1,for2(0,step,n),n)

tmulta(n)= for1(0,step,n)
```

Each function in the recursive program corresponds to a for loop with their related step-updates. The step counter is given as input argument to the next function in a continuation-passing style.

Intermediate language. An intermediate program, `Prg`, consists of integer functions, $\mathtt{f}_i$: $Int^* \to Int$, as given by the abstract syntax given in Figure 1. In the examples we relax the restrictions on function and parameter names.

$$
\begin{aligned}
&\mathtt{f}_i(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \langle \text{exp} \rangle \\
&\qquad \langle \text{aexp} \rangle \models \mathtt{x}_i \mid \mathtt{c} \mid \langle \text{aexp} \rangle \ \mathtt{+_i} \ \langle \text{aexp} \rangle \mid \langle \text{aexp} \rangle \ \mathtt{-_i} \ \langle \text{aexp} \rangle \mid \\
&\qquad\qquad \langle \text{aexp} \rangle \times_{\mathtt{i}} \langle \text{aexp} \rangle \mid \langle \text{aexp} \rangle \ \mathtt{div_i} \ \langle \text{aexp} \rangle \\
&\qquad \langle \text{bexp} \rangle \models \langle \text{aexp} \rangle \ \mathtt{=_i} \ \langle \text{aexp} \rangle \mid \langle \text{aexp} \rangle \ \mathtt{<_i} \ \langle \text{aexp} \rangle \mid \langle \text{aexp} \rangle \leq_{\mathtt{i}} \langle \text{aexp} \rangle \mid \\
&\qquad\qquad \mathtt{true} \mid \mathtt{false} \mid \mathtt{not}(\langle \text{bexp} \rangle) \\
&\qquad\ \langle \text{exp} \rangle \models \langle \text{aexp} \rangle \mid \mathtt{f}_i(\langle \text{exp}_1 \rangle, \ \ldots, \langle \text{exp}_n \rangle) \mid \\
&\qquad\qquad \mathtt{if} \ \langle \text{bexp} \rangle \ \mathtt{then} \ \langle \text{exp} \rangle \ \mathtt{else} \ \langle \text{exp} \rangle
\end{aligned}
$$

Fig. 1. The abstract syntax describing the intermediate programs.

Definition 4. *A program is* well-formed *if it follows the abstract syntax and it contains a finite number of function definitions, that each is of one of the following form and can internally be enumerated with a natural number such that:*

$\mathtt{f_i}(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \mathtt{if}\ b\ \mathtt{then}\ e_0\ \mathtt{else}\ \mathtt{f_i}(e_1, \ldots, e_n)$
where $\mathtt{f_i}$ *is simple,* e_0 *only contains calls to functions* $\mathtt{f_j}$ *where* $j < i$.

$\mathtt{f_i}(x_1, \ldots, x_n) \stackrel{\text{def}}{=} e$
where e *only contain calls to functions* $\mathtt{f_j}$ *where* $j < i$.

The enumeration prevents mutual recursion, and ensures that non-recursive calls cannot create an infinite call-chain.

5 Probabilistic output analysis

The analysis is applied to the intermediate program and an input probability program in the intermediate language. The output is a new program that can be

described by a subset of the intermediate language; this will be clarified later in the definition of pure and closed form programs. The analysis consists of three phases:

Create, where the probability program describing the output distribution is created as a possibly uncomputable expression.
Separate, where we remove all calls from the probability program.
Simplify, where we transform the program into closed form using safe over-approximations when necessary.

The analysis is constructed as three sets of transformation rules, one for each of the three phases. All transformations are syntax directed, and a strategy is to apply them in a depth-first manner. The program output analysis is implemented in Ciao and integrates with Mathematica in the third phase to reduce expressions.

In the following we use $\mathcal{V}ar(e)$ to represent the set of variables occurring in expression e, and $\mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n) \stackrel{\text{def}}{=} e$ to represent the function `f` is defined in the input program. Some side conditions are explained in an informal way, as in "$\mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n) \stackrel{\text{def}}{=} e$, where e is non-recursive".

$$\text{name}\frac{\text{precondition}_1 \quad ... \quad \text{precondition}_n}{\text{original term} \rightarrow \text{rewritten term}}$$

The preconditions are evaluated from left to right, and if all succeeds we can use the transformation.When substituting a variable `x` to an expression e, we denote it $[\mathtt{x}/e]$.

In the following we will begin by extending the intermediate language presented in Figure 1 such that it can express probabilities, and afterwards describe the transformation rules for each phase.

The intermediate language. The intermediate language is, as previously mentioned, a first order functional language. A probability program can be evaluated at any stage through the transformation process.

We extend the abstract syntax given in Figure 1 such that it can easily describe probability distributions. We introduce probability functions, `P`: $Int^* \rightarrow Real$, which follows the expanded syntax given in Figure 2. The dots indicate the syntax described in Figure 1. Again, ⟨aexp⟩ and ⟨exp⟩ are of type integer, ⟨bexp⟩ is boolean, and the new ⟨qexp⟩ is a real. In ⟨qexp⟩ the method `i2r` type casts an integer expression to a real. We introduce `c`, `sum`, `prod` and `argDev` functions. `c` evaluates to either 1, if its boolean expression evaluates to true, or 0 when it evaluates to false. Evaluating `sum` instantiates the variable with all possible values and sum all the results of the evaluation the ⟨qexp⟩. `prod` instantiates its variable with all values for which the first ⟨qexp⟩ evaluates to 1, and then it multiply all the results from evaluating the second ⟨qexp⟩. The last expression introduced is `argDev` which describes the development of the variable $\mathtt{x}_i$ as a function of the number of updates, $\mathtt{x}_j$. The expression ⟨exp⟩ computes the development of $\mathtt{x}_i$ for one incrementation of $\mathtt{x}_j$ (e.g. the argument

$\mathtt{x}_i$ in a function $\mathtt{f}(\mathtt{x}_i)$ with a recursive call $\mathtt{f}(\mathtt{x}_i{-}_\mathtt{i}2)$ has a argument development $\mathtt{argDev}(\mathtt{x}_i, \mathtt{x}_i{-}2, \mathtt{x}_j)$).

$$
\begin{aligned}
\mathtt{f}_i(x_1, \ldots, x_n) &\stackrel{\text{def}}{=} \langle \text{exp} \rangle \\
\langle \text{aexp} \rangle &\models \ldots \mid \mathtt{min}(\langle \text{aexp} \rangle, \langle \text{aexp} \rangle) \mid \mathtt{max}(\langle \text{aexp} \rangle, \langle \text{aexp} \rangle) \\
\langle \text{bexp} \rangle &\models \ldots \mid \langle \text{aexp} \rangle \ {=}_\mathtt{i} \ \langle \text{exp} \rangle \\
\langle \text{exp} \rangle &\models \ldots \mid \mathtt{argDev}(\mathtt{x}_i, \langle \text{exp} \rangle, \mathtt{x}_j) \\
\mathtt{P}_i(x_1, \ldots, x_n) &\stackrel{\text{def}}{=} \langle \text{qexp} \rangle \\
\langle \text{qexp} \rangle &\models \mathtt{i2r}(\langle \text{aexp} \rangle) \mid \mathtt{c}(\langle \text{bexp} \rangle) \mid \langle \text{qexp} \rangle \ op_q \ \langle \text{qexp} \rangle \mid \\
&\qquad \mathtt{sum}(\mathtt{x}_i, \langle \text{qexp} \rangle) \mid \mathtt{prod}(\mathtt{x}_i, \langle \text{qexp} \rangle, \langle \text{qexp} \rangle) \mid \\
&\qquad \mathtt{P}_i(\langle \text{aexp}_1 \rangle, \ \ldots, \langle \text{aexp}_n \rangle) \\
op_q &= +_\mathtt{q} \mid -_\mathtt{q} \mid \times_\mathtt{q} \mid /^\mathtt{q}
\end{aligned}
$$

Fig. 2. The expanded abstract syntax describing probability programs.

A program that computes a probability distribution is referred to as a probability program.

Definition 5. *A probability program that has no if-expressions no function calls is* pure *and a pure probability program without any* ***sum*** *and* ***prod*** *is in* closed form.

A program is *pure* after it is transformed in the separation phase and is pure and in *closed form* after the simplification phase.

The create phase. This phase has only one rule which creates a program that computes a probability distribution from the intermediate program and input distributions.

$$
\mathsf{create}\frac{\mathtt{f}(\mathtt{u}_1, ..., \mathtt{u}_n) \stackrel{\text{def}}{=} e \qquad \mathtt{P}(\mathtt{v}_1, ..., \mathtt{v}_n) \stackrel{\text{def}}{=} e_p}{P_\mathtt{f}(\mathtt{z}) \stackrel{\text{def}}{=} \mathtt{sum}(\mathtt{x}_1 \, ; ... \mathtt{sum}(\mathtt{x}_n \, ; \mathtt{c}(\mathtt{z} =_\mathtt{i} \mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n)) \times_\mathtt{q} \mathtt{P}(\mathtt{x}_1, ..., \mathtt{x}_n)))}
$$

We use the create rule to make a new probability function describing the probability distribution for the integer function we are interested in.

The separate phase. In this phase function calls are removed by repeatedly exposing calls and replacing them. Non-recursive function calls are unfolded using their definitions. Function calls can occur inside if-expressions or as nested

calls; these are extracted and handled one at a time.

$$\text{f-simple}\ \frac{\mathtt{f}(\mathtt{y}_1,...,\mathtt{y}_n) \stackrel{\text{def}}{=} e \quad \text{, where } e \text{ is non-recursive} \quad \mathtt{x}_1,...,\mathtt{x}_n \in \mathcal{V}ar}{\mathtt{c}(\mathtt{z} =_\mathtt{i} \mathtt{f}(\mathtt{x}_1,...,\mathtt{x}_n)) \to \mathtt{c}(\mathtt{z} =_\mathtt{i} e[\mathtt{y}_1/\mathtt{x}_1,...,\mathtt{y}_n/\mathtt{x}_n])}$$

$$\text{rem-P}\ \frac{\mathtt{P}(x_1,...,x_n) \stackrel{\text{def}}{=} e}{\mathtt{P}(e_1,...,e_n) \to e[x_1/e_1,...,x_n/e_n]}$$

$$\text{rem-if}\ \frac{}{\mathtt{c}(\mathtt{z} =_\mathtt{i} \mathtt{if}\ b\ \mathtt{then}\ e_0\ \mathtt{else}\ e_1) \to (\mathtt{c}(b) \times_\mathtt{q} \mathtt{c}(\mathtt{z} =_\mathtt{i} e_0) +_\mathtt{q} \mathtt{c}(\mathtt{not}(b)) \times_\mathtt{q} \mathtt{c}(\mathtt{z} =_\mathtt{i} e_1))}$$

$$\text{no-nest(f)}\ \frac{\{e_1,...,e_n\} \not\subseteq \mathcal{V}ar}{\begin{array}{l}\mathtt{c}(\mathtt{z} =_\mathtt{i} \mathtt{f}(e_1,...,e_n)) \to \\ \mathtt{sum}(u_1\,;...\mathtt{sum}(u_n\,;\mathtt{c}(\mathtt{z} =_\mathtt{i} \mathtt{f}(\mathtt{u}_1,...,\mathtt{u}_n)) \times_\mathtt{q} \mathtt{c}(\mathtt{u}_1 =_\mathtt{i} e_1) \times_\mathtt{q} ... \times_\mathtt{q} \mathtt{c}(\mathtt{u}_n =_\mathtt{i} e_n)))\end{array}}$$

We replace calls to recursive functions by a summation over the number of recursions using argument development constructors to describe the value of each argument as a function of the index of the summation. This way of defining argument development has similarities with size change functions derived using recurrence equations. Argument development functions do not depend on the base-case unlike size-change functions [40]. The summation also contains a product which ensures that the condition evaluates to false for argument values less than the current value of the index of summation. When the expression in a product contains only c-constructors, then the product is evaluated to 1 if either the range is empty or the expression is evaluated to true for the full range. The following rewrite rules are all that is needed for transforming probability programs into pure probability programs.

$$\text{f-rec}\ \frac{\begin{array}{c}\mathtt{f}(\mathtt{y}_1,...,\mathtt{y}_n) \stackrel{\text{def}}{=} \mathtt{if}\ b\ \mathtt{then}\ e_0\ \mathtt{else}\ \mathtt{f}(e_1,...,e_n) \qquad \mathtt{x}_1,...\mathtt{x}_n \in Vars \\ \sigma_{y/i} = [\mathtt{y}_1/\mathtt{i}_1,...,\mathtt{y}_n/\mathtt{i}_n]\sigma_{y/x} = [\mathtt{y}_1/\mathtt{x}_1,...,\mathtt{y}_n/\mathtt{x}_n]\sigma_{y/j} = [\mathtt{y}_1/\mathtt{j}_1,...,\mathtt{y}_n/\mathtt{j}_n]\end{array}}{\begin{array}{l}\mathtt{c}(\mathtt{z} =_\mathtt{i} \mathtt{f}(\mathtt{x}_1,...,\mathtt{x}_n)) \to \\ \mathtt{sum}(\mathtt{i}\,;\mathtt{c}(0 \le_\mathtt{i} \mathtt{i}) \times_\mathtt{q} \\ \quad \mathtt{sum}(\mathtt{i}_1\,;...\mathtt{sum}(\mathtt{i}_n\,;\mathtt{c}(\sigma_{y/i}(b)) \times_\mathtt{q} \mathtt{c}(\mathtt{i}_1 =_\mathtt{i} \mathtt{argDev}(\mathtt{x}_1, \sigma_{y/x}(e_1), \mathtt{i})) \times_\mathtt{q} \\ \qquad \mathtt{c}(\mathtt{z} =_\mathtt{i} \sigma_{y/i}(e_0)) \times_\mathtt{q} ... \times_\mathtt{q} \mathtt{c}(\mathtt{i}_n =_\mathtt{i} \mathtt{argDev}(\mathtt{x}_n, \sigma_{y/x}(e_n), \mathtt{i}))\ )...) \times_\mathtt{q} \\ \quad \mathtt{prod}(\mathtt{j}\,;\mathtt{c}(0 \le_\mathtt{i} \mathtt{j}) \times_\mathtt{q} \mathtt{c}(\mathtt{j} \le_\mathtt{i} \mathtt{i} -_\mathtt{i} 1)\,; \\ \quad \mathtt{sum}(\mathtt{j}_1\,;...\mathtt{sum}(\mathtt{j}_n\,;\mathtt{c}(\mathtt{not}(\sigma_{y/j}(b))) \times_\mathtt{q} \\ \qquad \mathtt{c}(\mathtt{j}_1 =_\mathtt{i} \mathtt{argDev}(\mathtt{x}_1, \sigma_{y/x}(e_1), \mathtt{j})) \times_\mathtt{q} ... \times_\mathtt{q} \mathtt{c}(\mathtt{j}_n =_\mathtt{i} \mathtt{argDev}(\mathtt{x}_n, \sigma_{y/x}(e_n), \mathtt{j})) \\ \quad)...)\ ))\end{array}}$$

The argument development expression may contain function calls as well, and these are extracted equivalently to nested functions.

$$\text{no-nest(argDev)}\ \frac{}{\begin{array}{l}\mathtt{c}(\mathtt{z} =_\mathtt{i} \mathtt{argDev}(\mathtt{x}, \mathtt{f}(e_1,...,e_n)\,, \mathtt{i})) \to \\ \mathtt{sum}(u\,;\mathtt{c}(\mathtt{z} =_\mathtt{i} \mathtt{argDev}(\mathtt{x}, \mathtt{f}(e_1,...,e_n)\,, \mathtt{i})) \times_\mathtt{q} \mathtt{c}(\mathtt{u} =_\mathtt{i} \mathtt{f}(e_1,...,e_n)))\end{array}}$$

After applying these rules until they cannot be applied no more, the probability program has been transformed to pure form.

The simplification phase. We have presented the rules for obtaining a pure probability program, and in this section we outline the rules used to reach closed

form. A pure probability function consists of a series of nested summations multiplied with an expression (e.g. input probability). The rules are applied in no particular order and the phase ends when no more rules can be applied. In this phase we integrate with Mathematica. A call to Mathematica is denoted $\texttt{mm:Function}(Arg) = Answer$, where `Function` denotes the actual function called in Mathematica (e.g. `mm:Expand` calls Mathematica's `Expand` function). The translation between the intermediate language and Mathematica's representation will not be discussed further here. implicitly in the call.

The rules can be grouped by their functionality: preparing expressions, removal of summations and removal of products. The latter are currently the only rules containing over-approximations.

Preparing expressions for removal of either summations or products involve moving expressions that do not depend on the index of summation outside the summation, dividing summations of additions into simpler ones, reducing expressions, dividing summations in ranges, and remove argument development constructors. Please notice that div-sum(x≤) has an equivalent rule for upper bounds.

$$\text{move-c}\frac{x \notin \mathcal{V}ar(e_1)}{\texttt{sum}(x\,;\, e_1 \times_{\texttt{q}} e_2) \rightarrow e_1 \times_{\texttt{q}} \texttt{sum}(x\,;\, e_2)}$$

$$\text{div-sum}(+)\frac{x \in \mathcal{V}ar(e_1) \qquad x \in \mathcal{V}ar(e_2)}{\texttt{sum}(x\,;\, e_1 +_{\texttt{q}} e_2) \rightarrow \texttt{sum}(x\,;\, e_1) +_{\texttt{q}} \texttt{sum}(x\,;\, e_2)}$$

$$\text{div-sum}(\text{x}\leq)\frac{x \notin \mathcal{V}ar(e_1, e_2) \qquad x \in \mathcal{V}ar(e_2)}{\begin{array}{c}\texttt{sum}(x\,;\, \texttt{c}(\texttt{x} \leq_{\texttt{i}} e_1) \times_{\texttt{q}} \texttt{c}(\texttt{x} \leq_{\texttt{i}} e_2) \times_{\texttt{q}} e_3) \rightarrow \\ \texttt{c}(e_1 \leq_{\texttt{i}} e_2) \times_{\texttt{q}} \texttt{sum}(x\,;\, \texttt{c}(\texttt{x} \leq_{\texttt{i}} e_1) \times_{\texttt{q}} e_3) +_{\texttt{q}} \\ \texttt{c}(e_2 \leq_{\texttt{i}} e_1 -_{\texttt{i}} 1) \times_{\texttt{q}} \texttt{sum}(x\,;\, \texttt{c}(\texttt{x} \leq_{\texttt{i}} e_2) \times_{\texttt{q}} e_3)\end{array}}$$

$$\text{rem(argDev)}\frac{\texttt{c} \in \texttt{n}}{\texttt{c}(\texttt{z} =_{\texttt{i}} \texttt{argDev}(\texttt{x}, \texttt{x} +_{\texttt{i}} \texttt{c}, \texttt{i})) \rightarrow \texttt{c}(\texttt{z} =_{\texttt{i}} \texttt{x} +_{\texttt{i}} \texttt{c} \times_{\texttt{i}} \texttt{i})}$$

$$\text{reduceAexp}\frac{\texttt{mm:Reduce}(e_1) = e_2}{\texttt{c}(e_1) \rightarrow \texttt{c}(e_2)}$$

$$\text{reduce}(=)\frac{}{\texttt{c}(true) \rightarrow \texttt{i2r}(1)}$$

Removal of summations can be done in two ways. Either the index of the summation can only be one value or it can be a limited range of values, and depending on which case different transformations are used. In the first case, there exists an equation containing the variable index of the innermost summation. The equation is solved for the variable and the rest of the variable occurrences are replaced by the new value.

$$\text{rem-sum}(=)\frac{\texttt{mm:Solve}(e_1 =_{\texttt{i}} e_2, \texttt{x}) \;=\; \texttt{c}(\texttt{x} =_{\texttt{i}} e_3)}{\texttt{sum}(\texttt{x}\,;\, \texttt{c}(e_1 =_{\texttt{i}} e_2) \times_{\texttt{q}} e) \rightarrow e[\texttt{x}/e_3]}$$

Removing a summation by its range involves using standard mathematical formulas for rewriting series. The last part of the following rule uses $\sum_{k=1}^{n} k^2 = n(n+1)(2n+1)/6$. We only present transformations up to quadratic series and our pragmatic implementation contains rules for transforming series of power of degree up to 10. A more general rewrite rule for series of power of degree up p could be implemented, but is more complicated as it includes Bernoulli numbers and binomial coefficients. The precondition uses Mathematica's `Expand` to transform the expression into the right pattern.

$$\text{rem-sum}(\leq)\frac{x \notin \mathcal{V}ar(e_1,...,e_6)\ \ \texttt{mm:Expand}(e_3) = \texttt{i2r}(e_4 +_\texttt{i} e_5 \times_\texttt{i} \texttt{x} +_\texttt{i} e_6 \times_\texttt{i} \texttt{x} \times_\texttt{i} \texttt{x})}{\begin{array}{l}\texttt{sum}(x\,;\texttt{c}(e_1 \leq_\texttt{i} x) \times_\texttt{q} \texttt{c}(x \leq_\texttt{i} e_2) \times_\texttt{q} \texttt{i2r}(e_3)) \rightarrow \\ \quad \texttt{i2r}(e_4) \times_\texttt{q} \texttt{i2r}(e_2 -_\texttt{i} e_1 +_\texttt{i} 1) +_\texttt{q} \\ \quad \texttt{i2r}(e_5) \times_\texttt{q} \texttt{i2r}(e_2 \times_\texttt{i} (e_2 +_\texttt{i} 1))/^\texttt{q} 2 -_\texttt{q} \\ \quad \texttt{i2r}(e_5) \times_\texttt{q} \texttt{i2r}(e_2 \times_\texttt{i} (e_2 -_\texttt{i} 1))/^\texttt{q} 2 +_\texttt{q} \\ \quad \texttt{i2r}(e_6) \times_\texttt{q} \texttt{i2r}(e_2 \times_\texttt{i} (e_2 +_\texttt{i} 1) \times_\texttt{i} (2 \times_\texttt{i} e_2 +_\texttt{i} 1))/^\texttt{q} 6 -_\texttt{q} \\ \quad \texttt{i2r}(e_6) \times_\texttt{q} \texttt{i2r}(e_2 \times_\texttt{i} (e_2 -_\texttt{i} 1) \times_\texttt{i} (2 \times_\texttt{i} e_2 -_\texttt{i} 1))/^\texttt{q} 6\end{array}}$$

Removal of Product involves a safe over-approximation. The implementation of POA contains two different over-approximations and in many cases the probability program can be transformed into closed form in a precise manner. In the following paragraph we describe when the transformation preserves the accuracy of the transformed term.

The probability function can always be over-approximated to 1. The rule f-rec is an exact rule and introduces a product-expression which may not be possible to rewrite into closed form. We only introduce the product-expression with c-expressions in its body, and therefore it will always either evaluate to 1 or to 0. A safe over-approximation of such a product-expression is 1.

$$\text{rem-prod-one}\frac{x \notin \mathcal{V}ar(e_1,e_2) \qquad x \in \mathcal{V}ar(e_3)}{\texttt{prod}(x\,;\texttt{c}(e_1 \leq_\texttt{i} x) \times_\texttt{q} \texttt{c}(x \leq_\texttt{i} e_2)\,;c(e_3)) \rightarrow 1}$$

For the summation describing recursive calls, this transformation is exact when the condition, b, evaluates to true for exactly one value (eg. it is an equation).

A broader class of recursive programs (than those having an equation in the condition) is those where the c-expression is monotone in x; meaning that there exists a k for which $c(e_3) = 1$ for $x \leq k$ and $c(e_3) = 0$ for $x > k$. This case covers many for-loops. In this case we can accurately replace the `prod`-expression with two c-expressions one checking the lower and one checking the upper range-limit. The empty product (the lower limit is larger than the upper) is 1.

$$\text{rem-prod-mon}\frac{x \notin \mathcal{V}ar(e_1,e_2) \qquad x \in \mathcal{V}ar(e_3) \qquad e_3 \text{ is monotone in } x}{\begin{array}{l}\texttt{prod}(x\,;\texttt{c}(e_1 \leq_\texttt{i} x) \times_\texttt{q} \texttt{c}(x \leq_\texttt{i} e_2)\,;c(e_3)) \rightarrow \\ \big(c(e_3)[x/e_1] \times_\texttt{q} c(e_3)[x/e_2] \times_\texttt{q} \texttt{c}(e_1 \leq_\texttt{i} e_2) +_\texttt{q} \texttt{c}(e_2 \leq_\texttt{i} e_1 -_\texttt{i} 1)\big)\end{array}}$$

This rule does not preserve accuracy when the c-expression is not monotone in x (e.g. $\texttt{c}(2 \leq_\texttt{i} x || 4 \leq_\texttt{i} x)$).

6 Implementation and results

In the following we present three examples which show results of programs with nested loops parameterized input distribution of multiple variables. The probability distribution computed by the output program varies in complexity; the first program calculates a single parameterized output, the second program computes a triangular shaped output distribution and third computes a distribution converging towards a standard normal distribution. The results are presented in a reduced and readable form extracted from our implementation.

Matrix multiplication. The original matrix multiplication program uses composite types and contains nested loops. The intermediate program, defined in Figure 3, contains nested recursive calls but has no dependency on data in composite types.

```
for3(i3,step,n) = if(i3>=n) then step else for3(i3+1,step+1,n)
for2(i2,step,n) = if(i2>=n) then step else for2(i2+1,for3(0,step+2,n),n)
for1(i1,step,n) = if(i1>=n) then step else for1(i1+1,for2(0,step,n),n)
tmulta(step,n) = for1(0,step,n)
P(step,n1) = c(step=0)*c(n1=n)
```

Fig. 3. The intermediate program containing also the parameterized probability distribution. The parameter `n` can obtain only one value.

The nested calls create argument development functions that depend on function calls. These are transformed to a simple form and then removed. The introduced products are over approximated, but due to the form of the condition the result is precise. The output program computes a single value distribution (when specialized with the size of the matrix). It is given in Figure 4 along with an array describing a subset of specializations of the output program with respect to a value of `n`.

```
Ptmulta(out) =
 c(3=<out/(n*n))*
 c(1=<n)*
 c(out/n*n=2+n)*1
```

n	program
1	`Ptmulta(out) = c(out=3)`
2	`Ptmulta(out) = c(out=16)`
3	`Ptmulta(out) = c(out=45)`
4	`Ptmulta(out) = c(out=96)`
...	...

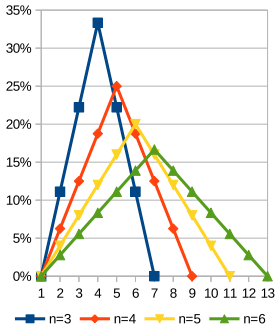
Fig. 4. The general output probability program (left) and the program specialized whith the value of `n` (right).

Adding parameterized distributions. This example is a recursive program computing the addition of two numbers; the input program and the input probability distribution can be seen in Figure 5. The output depends on both increasing and decreasing values. In this example we use a parameter `n` as the upper limit of a range of input values. The input distribution describes two independent variables, each having a uniform distribution from 1 to n.

```
add(x,y) = if x=<0 then y else add(x-1,y+1)
P(x) = c(1=<x)*c(x=<n)*1/n
Pxy(x,y) = P(x)*P(y)
```

Fig. 5. The intermediate program containing both the function `add` and the input probability distribution. Here, the parameter `n` is used to describe a range.

The analysis gives a precise probability distribution and computes a triangular distribution (or pyramid shaped distribution). The output probability program is described in Figure 6 along with a graph depicturing the pyramid shaped output probability distributions for different initializations of `n`. The lower bound on `out` arises from the input probability distribution and not from the condition. The upper bound `2*n` of the analysis result shows that the output depends on both input variables, despite that one is increasing and the other is decreasing.




```
Padd(out) =
    c(2<=out)*c(out<=n)*(1/n*1/n*(out-1))+
    c(1+n<=out)*c(out<=2*n)*(1/n*1/n*(1+2*n-out))
```

Fig. 6. The general output program and the graphs for the output probability distribution with n set to 3, 4, 5, and 6, respectively.

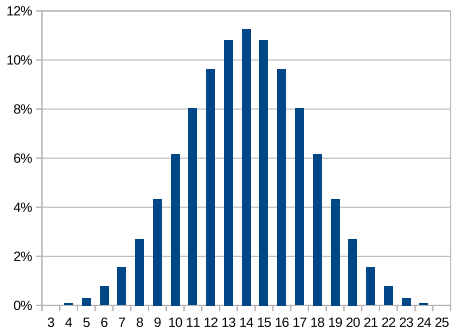
Adding 4 independent variables. The program `sum4` adds four variables and was presented by Monniaux [23]. Certain over-approximations were applied so as to obtain a safe and simplified result.

The program is recursive and in this example we use independent input variables each uniformly distributed input from 1 to 6, as described in Figure 7.

```
add(x,y) = if x=0 then y else add(x-1,y+1)
sum4(x,y,z,w) = add(x,add(y,add(z,w)))
tsum4(x,y,z,w) = sum4(x,y,z,w)
P(x) = c(1=<x)*c(x=<6)*1/6
Pxyzw(x,y,z,w) = P(x)*P(y)*P(z)*P(w)
```

Fig. 7. Intermediate program.

Despite the ranges and their associated value are not symmetric, the resulting program computes a precise and perfectly symmetric probability distribution as shown in Figure 8. The differences in the choice of ranges comes (among other things) from the range dividing rules, as they do not divide the range symmetrically. As expected from the central limit theorem of probability theory,




```
Psum4(out) =
c(4=<out)*c(out=<7)*(-6 + 11*out -
6*out^2 + out^3)/7776+
c(8=<out)*c(out=<12)*
(-1014+169*out+6*out^2-out^3)/7776+
c(9=<out)*c(out=<12)*
(1512-461*out+42*out^2-out^3)/3888+
c(out=13)*(265/648-5*out/216)+
c(14=<out)*c(out=<18)*
(-4790+923*out-54*out^2+out^3)/2592+
c(19=<out)*c(out=<24)*
(17550-2027*out+78*out^2-out^3)/7776
```

Fig. 8. The output program and graph for its computed probability distribution for `out` from 3 to 25.

the resulting probability program describes a distribution that has similarities with a normal distribution.

Monty Hall. The Monty Hall problem is often used to exemplify how gained knowledge influences probabilities (conditional probability). In this problem there are three closed doors; one hiding a price and two that are empty. The doors have an equal chance of hiding the price. There is a contestant, who should choose one of the doors, then the game host will open an empty door and the contestant can either stick with the first choice or can change to the other unopened door. The problem lies in showing whether the best winning-strategy is to stick with the first choice or to switch to the other?

If the strategy is to stick with the first choice and that door has a price then the contestant has won. If the contestant changes door he/she only loses if the first choice was the door hiding the price; if the first choice was an empty door, then the game host would open the other empty door leaving only the price door for a second choice.

The program `monty` models the two strategies; if the strategy variable is 1 then the strategy is to change the door, and otherwise the strategy is to stick with the first choice. The program takes as input the contestant's first guess, the door hiding the price, the empty door which is not opened by the game host and the strategy the contestant uses.

Let us assume the contestant has an equal chance of choosing each of the doors. The input variables `guess`, `price`, and `empty` models the first choice, the price door and the empty door which is left after the game host has opened an empty door. All three doors have a value between 1 and 3, and the empty door cannot be the same as the price door. We have parametrized the strategy with a weight `p` between the two, such that when `p` = 1 then the strategy is to always change door, and when `p`=0 the strategy is to always keep the first choice (e.g. letting `p` = 0.75 we change doors in 3/4 cases and 1/4 we keep the first door). Such a parametrization allows us to execute the analysis once and use the lighter closed form result for that calculation instead. In a problem where the winning-probability of a strategy is dependent on the other input, such input

could be used for optimizing the choice of strategy. The program `monty` and the parametrized input probability distribution can be seen in Figure 9.

```
monty(guess,price,empty,strategy)=
  if strategy = 0
  then finalGuess(guess,price)
  else change(guess,price,empty)

finalGuess(guess,price)=
  if price=guess then 1 else 0

change(guess,price,empty)=
  if price=guess
  then finalGuess(empty,price)
  else finalGuess(price,price)
```

```
Pin(guess,price,empty,strategy) =
  1/18*c(1=<guess)*c(guess=<3)
       *c(1=<price)*c(price=<3)
       *c(1=<empty)*c(empty=<3)
       *c(not(price = empty))
       *Pstrat(strategy)
Pstrat(strategy) =
  p*c(1 = strategy)
    + (1-p)*c(0 = strategy)
```

Fig. 9. The program `monty` models the event flow depending on the chosen strategy; if the strategy is `0` then the contestant keeps the first door and if it is `1` then the contestant changes his mind. There are three doors and the input of `monty` describes the contestants first guess, the door hiding the price, the empty door which is not opened by the game host (and is different from the price door) and the strategy of the contestant. If the final choice hides the price then the program returns 1 and otherwise 0. The probability of the strategy is an expression parametrized with a weight, `p` between the two strategies instead of executing the analysis twice with different strategy.

The analysis was capable of handling the program correctly and the result can be seen in Figure 10.

```
pmonty(out) =
 1/18 *
 (c(out=0)*
   (12*(1-p)+6*p)+
  c(out=1)*
   (6*(1-p)+12*p))
```

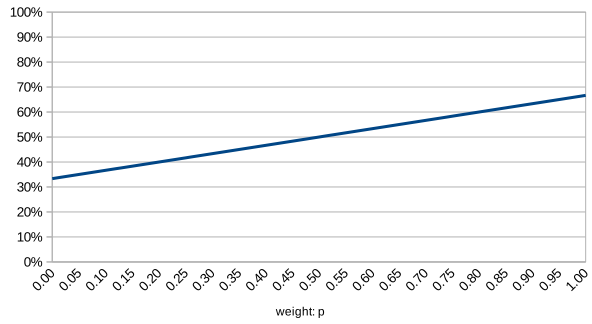


Fig. 10. The probability of winning the Monty Hall as a function of the weight given to change-strategy. The probabilistic output analysis reveals that the best weight between the keep strategy and the change strategy is to always use change strategy.

The probabilities 1/3 and 2/3 does not occur directly in the output probability program, but are found in the constants 6, 12 and 1/18.

Adding dependent non-uniform variables. A function call may have interdependent and non-uniform arguments that can only be represented as a joint probability distribution (ie. polygon), and in this example we demonstrate that the analysis can handle such function calls. We focus on the dependencies, analyze a simple add program and discuss the limits of the interdependencies. The

program also shows that interdependencies quickly lead to the occurrence of integer division in the output.

The input arguments are interdependent; the second argument is always less than or equal to the value of the first argument and the joint distribution depends only on the value of the first argument resulting in a triangular and skewed probability distribution. The probability program is defined in Figure 11.

```
Pxy(x,y) =  c(1=<y)*c(y=<3) *
              c(1=<x)*c(x=<y) * x/10
add(x,y) = x+z

Padd(out) =
  c(2 =< out)*c(out=< 3) *   1/20 * out%2 * (1 + out%2) +
  c(4 =< out)*c(out=< 6) * -(1/20)*(-4+out-out%2)*(-3+out+out%2)
```

Fig. 11. An input program, `add`, its skewed joint distribution, `Pxy`, and the closed form probability program, `Padd`, produced by the analysis. The integer division is noted by a "`%`".

The create rule generates nested summations, and removing such inner summations imply that their values must be expressed using the variables of the outer summations or the input variable (ie. `out`). Comparing the result from this experiment with the output probability distribution for addition of two random variable in Figure 6 indicates that integer division is a special case arising from dependent input. The following interesting expressions are extracted during analysis execution, and they shows how the integer division arises from dependency of input. The first expressions is the result from the create rule and the last expression is the result after removal of the inner `y`-summation.

$$
\begin{aligned}
&P_{\texttt{add}}(\texttt{out}) = \\
&\texttt{sum}(\texttt{x}\,;\texttt{sum}(\texttt{y}\,;\texttt{c}(\texttt{out} =_{\texttt{i}} \texttt{x} +_{\texttt{i}} \texttt{y}) \times_{\texttt{q}} \\
&\qquad \texttt{c}(1 \leq_{\texttt{i}} \texttt{x}) \times_{\texttt{q}} \texttt{c}(\texttt{x} \leq_{\texttt{i}} \texttt{y}) \times_{\texttt{q}} \texttt{c}(1 \leq_{\texttt{i}} \texttt{y}) \times_{\texttt{q}} \texttt{c}(\texttt{y} \leq_{\texttt{i}} 3) \times_{\texttt{q}} (\texttt{i2r(x)} /^{\texttt{q}} \texttt{i2r(10)}))) = \\
&\texttt{sum}(\texttt{x}\,;\texttt{c}(2 \leq_{\texttt{i}} \texttt{out}) \times_{\texttt{q}} \texttt{c}(\texttt{out} \leq_{\texttt{i}} 3) \times_{\texttt{q}} \texttt{c}(1 \leq_{\texttt{i}} \texttt{x}) \times_{\texttt{q}} \\
&\quad \texttt{c}(2 \times_{\texttt{i}} \texttt{x} \leq_{\texttt{i}} \texttt{out}) \times_{\texttt{q}} (\texttt{i2r(x)} /^{\texttt{q}} \texttt{i2r(10)})) +_{\texttt{q}} \\
&\texttt{sum}(\texttt{x}\,;\texttt{c}(4 \leq_{\texttt{i}} \texttt{out}) \times_{\texttt{q}} \texttt{c}(\texttt{out} \leq_{\texttt{i}} 3 +_{\texttt{i}} \texttt{x}) \times_{\texttt{q}} \texttt{c}(2 \times_{\texttt{i}} \texttt{x} \leq_{\texttt{i}} \texttt{out}) \times_{\texttt{q}} (\texttt{i2r(x)} /^{\texttt{q}} \texttt{i2r(10)}))
\end{aligned}
$$

In the last expression there are two summations, each leading to its own part in the resulting program. Looking closely at each summation, we see that they share the upper limit for `x`, $\texttt{c}(2 \times_{\texttt{i}} \texttt{x} \leq_{\texttt{i}} \texttt{out})$, which currently contains an integer multiplication and when solved with respect to `x` contains the integer division. In the final result the second part of the expression has an upper limit for `out`, $\texttt{c}(\texttt{out} \leq_{\texttt{i}} 6)$ which is a constraint that the summation-removal-rule introduces to ensure that the lower limit of the summation (i.e. $\texttt{out} -_{\texttt{i}} 3$) is less than or equal to the upper limit (i.e. $\texttt{out} \,\%_{\texttt{i}}\, 2$).

The original probability ($\texttt{i2r(x)} /^{\texttt{q}} \texttt{i2r(10)}$) occurs directly in the summations, and this indicates a limit of this implementation and approach. To be able to handle a probability, the rewrite rules for summations must transform summations over the probability expression. There are limits to which series

the system currently can transform, Sum of reciprocals (e.g. $\sum_{k=1}^{n} \frac{1}{k}$) known as harmonic series or variations hereof such as generalized harmonic series are currently not implemented. The current analysis is limited to finite summations of at least order of 1, but a closer integration with Mathematica that exploits more of Mathematicas rewriting mechanisms should be able to handle such series.

7 Related works

Probabilistic analysis is related to the analysis of probabilistic programs. Probabilistic analysis is analysis of programs with a normal semantics where the input variables are interpreted over probability distributions. Analysis of probabilistic programs analyse programs with probabilistic semantics where the values of the input variables are unknown (e.g. flow analysis [25]).

In probabilistic analysis it is important to determine how variables depend on each other, but already in 1976 Denning proposed a flow analysis for revealing whether variables depend on each other [8]. This was presented in the field of secure flow analysis. Denning introduced a lattice-based analysis where she, given the name of a variable, that should be kept secret, deducted which other variables those should be kept secret in order to avoid leaking information. In 1996, Denning's method was refined by Volpano et al. into a type system and for the first time, it was proven sound [34].

Reasoning about probabilistic semantics is a closely related area to probabilistic analysis, as they both work with nested probabilistic influence. The probabilistic analysis work on standard semantic and analyze it using input probability distributions, where a probabilistic semantics allow for random assignments and probabilistic choices [20] and is normally analyzed using an expanded classical analysis or verification method [6].

Probabilistic model checking is an automated technique for formally verifying quantitative properties for systems with probabilistic behaviors. It is mainly focused on Markov decision processes, which can model both stochastic and non-deterministic behavior [13, 21]. It differs from probabilistic analysis as it assumes the Markov property.

In 2000, Monniaux applied abstract interpretation to programs with probabilistic semantics and gained safe bounds for worst case analysis [23]. Pierro et al. introduce a linear mapping structure, a Moore-Penrose pseudo-inverse, instead of a Galois connection. They use the linear structures to compare 'closeness' of approximations as an expression using the average approximation error. Pierro et al. further explores using probabilistic abstract interpretation to calculate the average case analysis [24]. In 2012, Cousot and Monerau gave a general probabilistic abstraction framework [6] and stated, in section 5.3, that Pierro et al.'s method and many other abstraction methods can be expressed in this new framework.

When analysing probabilities the main challenge is to maintain the dependencies throughout the program. Schellekens defines this as *Randomness preservation* [31] (or random bag preservation) which in his (and Gao's [14]) case enables

tracking of certain data structures and their distributions. They use special data structures as they find these suitable to derive the average number of basic operations. In another approach [35, 26], tests in programs has been assumed to be independent of previous history, also known as the Markov property (the probability of true is fixed). As Wegbreit remarked, this is true only for some programs (e.g. linear search for repeating lists) and others, this is not the case (linear search for non-repeating lists). The Markov property is the foundation in Markov decision processes which is used in probabilistic model-checking [13]. Cousot et al. presents a probabilistic abstraction framework where they divide the program semantics into probabilistic behavior and (non-)deterministic behavior. They propose handling of tests when it is possible to assume the Markov property, and handle loops by using a probability distribution describing the probability of entering the loop in the ith iteration. Monniaux propose another approach for abstracting probabilistic semantics [23]; he first lifts a normal semantics to a probabilistic semantics where random generators are allowed and then uses an abstraction to reach a closed form. Monniaux's semantic approach uses a backward probabilistic semantics operating on measurable functions. This is closely related to the forward probabilistic semantics proposed earlier by Kozen [20].

An alternative approach to probabilistic analysis is based on symbolic execution of programs with symbolic values [15]. Such techniques can also be used on programs with infinitely many execution paths by limiting the analysis to a finite set of paths at the expense of tightness of probability intervals [30].

8 Conclusion

Probabilistic analysis of program has a renewed interest for analysing programs for energy consumptions. Numerous embedded systems and mobile applications are limited by restricted battery life on the hardware. In this paper we describe a rewrite system that derives a resource probability distribution for programs given distributions of the input. It can analyze programs in subset of C where we have known distribution of input variables. From the original program we create a probability distribution program, where we remove calls to original functions and transform it into closed form. We have presented the transformation rules for each step and outlined the implementation of the system. We discuss over-approximating rules and their influence on the accuracy of the output probability and show that our analysis improves on related analysis in the literature.

References

1. A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault, and S. Putot. Static analysis of programs with imprecise probabilistic inputs. In *In Verified Software: Theories, Tools, Experiments*, pages 22–47. Springer Berlin Heidelberg., 2014.
2. M. Bauer. Approximations for decision making in the Dempster-Shafer theory of evidence. In E. Horvitz and F. V. Jensen, editors, *UAI*, pages 73–80. Morgan Kaufmann, 1996.

3. D. Berleant and H. Cheng. A software tool for automatically verified operations on intervals and probability distributions. *Reliable Computing*, 4(1):71–82, 1998.
4. O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 94(2-4):189–201, 2012.
5. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-Garcıa, and G. Puebla. The ciao prolog system. *Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM)*, 95:96, 1997.
6. P. Cousot and M. Monerau. Probabilistic abstract interpretation. In H. Seidl, editor, *ESOP*, volume 7211 of *LNCS*, pages 169–193. Springer, 2012.
7. Saumya K Debray, P López García, Manuel Hermenegildo, and N-W Lin. Estimating the computational cost of logic programs. In *Static Analysis*, pages 255–265. Springer, 1994.
8. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
9. S. Destercke and D. Dubois. The role of generalised p-boxes in imprecise probability models. In *6th International Symposium on Imprecise Probability: Theories and Applications*, 2009.
10. S. Ferson. Model uncertainty in risk analysis. Tech. report, Centre de Recherches de Royallieu, Universite de Technologie de Compiegne, 2014.
11. S. Ferson, V. Kreinovich, L. Ginzburg, D. S. Myers, and K. Sentz. Constructing probability boxes and Dempster-Shafer structures. Sand2002-4015, Sandia National Laboratories, 2002.
12. P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithm. *Theor. Comput. Sci.*, 79(1):37–109, 1991.
13. V. Forejt, M. Z. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *SFM*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.
14. A. Gao. *Modular average case analysis: Language implementation and extension.* Ph.d. thesis, University College Cork, 2013.
15. J. Geldenhuys, M. B Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176. ACM, 2012.
16. J. Gordon and E. H. Shortliffe. The Dempster-Shafer theory of evidence. In *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, page 21 pp, 1984.
17. X. Guo, M. Boubekeur, J. McEnery, and D. Hickey. ACET based scheduling of soft real-time systems: An approach to optimise resource budgeting. *International Journal of Computers and Communications*, 1(1):82–86, 2007.
18. R. U. Kay. Fundamentals of the Dempster-Shafer theory and its applications to system safety and reliability modelling. In *RTA*, pages 173–185, 2007.
19. S. Kerrison and K. Eder. Energy modelling and optimisation of software for a hardware multi-threaded embedded microprocessor. *University of Bristol, Bristol, Tech. Rep*, 2013.
20. D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.
21. M. Kwiatkowska, G. Norman, and D. Parker. Advances and challenges of probabilistic model checking. In *48th Annual Allerton Conference on Communication, Control, and Computing*, pages 1691–1698. IEEE, September 2010.

22. U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. Energy consumption analysis of programs based on xmos isa level models. In *23rd International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR*, volume 8901 of *LNCS*, pages 72–90, 2013.
23. D. Monniaux. Abstract interpretation of probabilistic semantics. In Jens Palsberg, editor, *SAS*, volume 1824 of *LNCS*, pages 322–339. Springer, 2000.
24. A. Di Pierro, C. Hankin, and H. Wiklicky. Abstract interpretation for worst and average case analysis. In T. W. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation*, volume 4444 of *LNCS*, pages 160–174. Springer, 2006.
25. A. Di Pierro, H. Wiklicky, G. Puppis, and T. Villa. Probabilistic data flow analysis: a linear equational approach. In *Proceedings Fourth International Symposium*, volume 119, pages 150–165. Open Publishing Association, 2013.
26. H. Soza Pollman, M. Carro, and P. Lopez Garcia. Probabilistic cost analysis of logic programs: A first case study. *INGENIARE - Revista Chilena de Ingeniera*, 17(2):195–204, 2009.
27. M. Rosendahl. Automatic program analysis. Master's thesis, University of Copenhagen, 1986.
28. M. Rosendahl. Automatic complexity analysis. In *FPCA*, pages 144–156, 1989.
29. M. Rosendahl and M. H. Kirkeby. Probabilistic output analysis by program manipulation. In *Quantitative Aspects of Programming Languages*, EPTCS, 2015.
30. S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *In Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 447–458. ACM., June 2013.
31. M. P. Schellekens. *A modular calculus for the average cost of data structuring.* Springer, 2008.
32. V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.
33. A. Uwimbabazi. Extended probabilistic symbolic execution. Master's thesis, University of Stellenbosch, 2013.
34. D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
35. B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.
36. M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
37. A. Wierman, L. L. H. Andrew, and A. Tang. Stochastic analysis of power-aware scheduling. In *Proceedings of Allerton Conference on Communication, Control and Computing*. Urbana-Champaign, IL, 2008.
38. N. Wilson. Algorithms for Dempster-Shafer theory. In *Handbook of defeasible reasoning and uncertainty management systems*, pages 421–475. Springer Netherlands, 2000.
39. S. Wolfram. The Mathematica book. *Cambridge University Press and Wolfram Research, Inc., New York, NY, USA and*, 100:61820–7237, 2000.
40. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis*, pages 280–297. Springer, 2011.

Attachment D3.3.6

Static analysis of energy consumption for LLVM IR programs

Published at the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2015)

Static analysis of energy consumption for LLVM IR programs

Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse and Kerstin Eder
University of Bristol, Merchant Venturers Building, Woodland Road
Bristol, BS8 1UB, United Kingdom
nevillegrech@gmail.com, {kyriakos.georgiou, james.pallister, steve.kerrison, jeremy.morse, kerstin.eder}@bristol.ac.uk

Abstract

Energy models can be constructed by characterizing the energy consumed when executing each instruction in a processor's instruction set. This can be used to determine how much energy is required to execute a sequence of assembly instructions, without the need to instrument or measure hardware.

However, statically analyzing low-level program structures is hard, and the gap between the high-level program structure and the low-level energy models needs to be bridged. We have developed techniques for performing a static analysis on the intermediate compiler representations of a program. Specifically, we target LLVM IR, a representation used by modern compilers, including Clang. Using these techniques we can automatically infer an estimate of the energy consumed when running a function under different platforms and compilers.

One of the challenges in doing so is that of determining the energy cost of executing LLVM IR program segments, for which we have developed two different approaches. When this information is used in conjunction with our analysis, we are able to infer energy formulae that characterize the energy consumption for a particular program. This approach can be applied to any languages targeting the LLVM toolchain, including C and XC or architectures such as ARM Cortex-M or XMOS xCORE, with a focus towards embedded platforms. Our techniques are validated on these platforms by comparing the static analysis results to the physical measurements taken from the hardware. Static energy consumption estimation enables energy-aware software development by providing instant feedback to the developer, without requiring simulations or hardware knowledge.

Categories and Subject Descriptors D.2.8 [*Software Metrics*]: Performance measures

General Terms Energy estimation, Software Analysis

1. Introduction

In embedded systems, low energy consumption is a very important requirement. The software running on these systems has a profound effect on the energy consumed. The design of software and algorithms, the programming language and the compiler together with its optimization level all contribute towards the energy consumption of an application. Measuring such consumption, however, requires hardware specific knowledge and instrumentation, making such measurements challenging for software engineers.

Estimations of energy consumption of programs are very useful to software engineers, so that they can understand the effect of their code on the energy consumption of the final system, without the need to instrument or even have the system. Accurate energy consumption and timing analysis of programs involves analyzing low-level machine code representations. However, programs are written in high-level languages with rich abstraction mechanisms, and the relation between the two is often blurred. For instance, optimizations such as dead code elimination, various kinds of code motion, inlining and other clever loop optimization techniques obfuscate the structure of the program and make the resultant code difficult to analyze [27].

In this paper, we develop a static analyzer that works on the intermediate compiler representation of the program (LLVM IR). Our analysis is based on a well-developed approach in which recursive equations (cost relations) are extracted from a program, representing the cost of running the program in terms of its input [2, 3, 25, 32]. These cost relations (CRs) are finally converted to *closed-form*, i.e. without recurrences, by means of a solver. For example, we can analyze the following program.

```
1 void proc(int v[], int l) {
2     for (int i = 0; i < l; i++)
3       if(v[i] & 1)
4         odd();
5       else
6         even();
7 }
```

The following CRs are extracted from the program,

$$
\begin{array}{ll}
(a)\ C_{\mathrm{proc}}(l) = k_1 + C_{\mathrm{for}}(l,0) & \text{if } l \geq 0 \\
(b)\ C_{\mathrm{for}}(l,i) = k_2 & \text{if } i \geq l \wedge l \geq 0 \\
(c)\ C_{\mathrm{for}}(l,i) = k_3 + C_{\mathrm{odd}}() + C_{\mathrm{for}}(l,i+1) & \text{if } i \leq l \wedge l \geq 0 \\
(d)\ C_{\mathrm{for}}(l,i) = k_4 + C_{\mathrm{even}}() + C_{\mathrm{for}}(l,i+1) & \text{if } i \leq l \wedge l \geq 0
\end{array}
$$

where l denotes the length of the array `v`, i stands for the counter of the loop and C_{proc}, C_{odd} and C_{even} approximate, respectively, the costs of executing their corresponding methods. The constraints, denoted on the right hand side of the relations, specify a condition that must be true for the cost relation to be applicable. For instance, relation (a) corresponds to the cost of executing `proc` with an array of length greater than 0 (stated in the condition $l > 0$), where cost k_1 is accumulated to the cost of executing the loop, given by C_{for}. Note that the transition into (c) and (d) is non deterministic. The

SCOPES'15, June 1–3, 2015, Sankt Goar, Germany.

ACM 978-1-4503-3593-5/15/06…$15.00.
http://dx.doi.org/10.1145/2764967.2764974

constants $k_1, \ldots, k_4$ take different values depending on the cost model that one adopts. In this paper, our cost model focuses on energy. These constants are obtained from energy models created at the Instruction Set Architecture (ISA) level [13]. Such models have previously been applied to analysis at the same level [15, 18], and in this paper we propagate this up to the LLVM level.

Many modern compilers such as Clang or XCC are built using the LLVM framework. These internally transform source programs into intermediate compiler representations, which are more amenable to analysis than either source or machine level programs. We show how resource consumption analysis techniques can be adapted and applied to programming languages targeting LLVM IR (such as C or XC [31]) by reusing some of the existing machinery available in the compiler framework (for instance LLVM analysis passes). We show how cost relations can be extracted from programs, such that these can be solved using an existing solver [3]. Specifically, we focus on *optimized LLVM IR*, that has been compiled with optimization levels used in production software (i.e. `O2` or higher).

Time is a significant component of energy consumption, in that a program that computes its result quicker will typically consume less energy by virtue of a shorter run-time. However, the correlation between time and energy varies between architectures, and is related to the complexity of the processor's pipeline [23]. For example, one of the target architectures for this paper exhibits an approximately $2\times$ difference in energy depending on the instructions that are executed, with a similar relationship for the number of threads executed upon it [13]. Analysis of system energy and not just of execution time will therefore garner better information on the energy characteristics of a program.

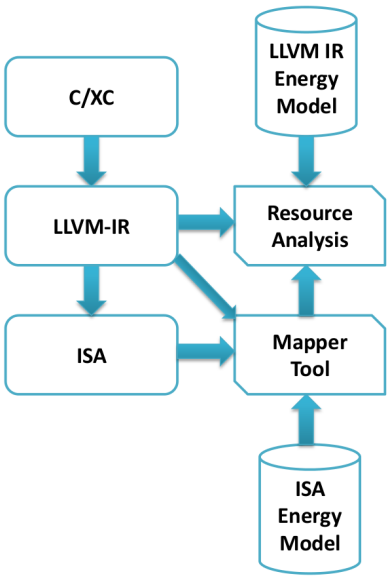


Figure 1. Illustration of the analysis toolchain.

Energy models can be constructed for a processor's instruction set; however this information needs to be constructed, or propagated to a higher level program representation in order to benefit our analysis mechanism. We propose two different techniques (Section 4), for assigning energy to a higher level program representation (LLVM IR). We first propose a mechanism for *mapping* program segments at ISA level to program segments at LLVM IR level. Using this mapping, we can perform a multi level program analysis where we consider the LLVM IR for the structure and semantics of the program and the ISA instructions for the physical effect on the hardware. We also propose an alternative technique, of determining the instruction energy model directly at the LLVM IR level. This is based on empirical data and domain knowledge of the compiler backend and underlying processor.

This paper focuses on static analysis of code for processors that are embedded or deeply embedded. Such processors do not typically feature cache hierarchies. They have small amounts of static-RAM and possibly flash memory available to them. This constrains the application space, but the motivation for analysing software that targets these processors is greater, because these types of embedded systems often have the strictest energy consumption requirements.

The analysis toolchain is illustrated in Figure 1. The static resource consumption analysis mechanism is described in Section 3. Parts of this mechanism perform a symbolic execution of LLVM IR, which is described in Section 2. The techniques described are built into a tool, which can be integrated into the build process, and which statically estimates the energy consumption of an embedded program (and its constituent parts, such as functions) as a mathematical function on several parameters of the input data. Our approach is validated in Section 5 on a number of embedded systems benchmarks, on both xCORE and Cortex-M platforms. Finally, we describe related work in Section 6 and conclude in Section 7.

2. Structure and interpretation of LLVM IR

In this section we describe the core language and an important technique we utilize in the resource consumption analysis mechanism (Section 3), which infers energy formulae given an LLVM IR program.

2.1 The LLVM IR language

LLVM IR is a Static Single Assignment (SSA) based representation. This is used in a number of compilers, and is designed to represent high-level languages. It consists of IR instructions arranged in *basic blocks*. A basic block BB over a CFG has a unique name and is a maximal sequence of instructions, $inst_1$ through $inst_n$, such that all instructions up to $inst_{n-1}$ are not branch or return instructions and $inst_n$ is `br` or `ret`. Basic blocks have a single entry point.

For presentation purposes, we first formalize a simple calculus of LLVM IR, based on the following syntax:

$$
\begin{aligned}
inst = {} & \mathsf{br}\ p\ BB_1\ BB_2 \quad \text{(conditional branch)} \\
\mid {} & x = \mathsf{op}\ a_1..a_n \quad \text{(generic op., no side-effects)} \\
\mid {} & x = \phi\ \langle BB_1, x_1\rangle..\langle BB_n, x_n\rangle \quad \text{(phi functions)} \\
\mid {} & x = \mathsf{call}\ f\ a_1\ ..\ a_n \\
\mid {} & x = \mathsf{memload} \quad \text{(dynamic memory load)} \\
\mid {} & \mathsf{memstore} \quad \text{(dynamic memory store)} \\
\mid {} & \mathsf{ret}\ a
\end{aligned}
$$

We use metavariable names p, f, a, x to describe predicates, function names, generic arguments and variables respectively. The concrete semantics of the instructions are modeled on the actual LLVM IR semantics [35]. Instruction `op` represents any side effect free operation such as `icmp` or `add` in LLVM. LLVM IR has explicit ϕ instructions, which are present since the code is in static single assignment (SSA) form. The ϕ instruction merges the multiple inward data flow paths associated with a variable at the entry of a basic block. It takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. Each pair contains a reference to a predecessor block together with the variable that is propagated to the current block. The only place where a ϕ

can appear is at the beginning of a basic block. Two interesting instructions are memload and memstore. These represent any dynamic memory load and store operation respectively. For instance, `getelementptr` and `load` are some examples of instructions represented by memload. These instructions typically compute pointers dynamically and load data from memory. In our abstract semantics of LLVM IR, we therefore treat variables assigned with values dynamically loaded from memory as unknown (denoted '?'). All call instructions are assumed to eventually return.

2.2 Symbolic evaluation of LLVM IR variables

At the core of our resource consumption analysis mechanism of LLVM IR is a symbolic evaluation function $seval$. Given a block of code BB, and a variable x, $seval(BB, x)$ symbolically executes this block, producing a slice [34] of the block with respect to x. During this static analysis phase, we apply an abstract semantics of LLVM IR, which abstracts away dynamic memory reads and writes i.e., memload and memstore. This has the effect of producing simple expressions, which can be handled by the PUBS solver [3]. The algorithm proceeds by starting at the last assignment of x in the block, and evaluates the assigned expression using this semantics, recursively evaluating all its dependencies until an expression or variable outside the block is reached. For example, given the following snippet:

```
LoopIncrement:
  %postinc = add i32 %i.0, 1
  %exitcond = icmp eq i32 %postinc, %1
  br i1 %exitcond, label %return, label %LoopBody
```

$seval(\ldots,$`%exitcond`$)$ is $($`%i.0`$+1) ==$`%1`, while in the following snippet

```
iftrue2:
  call void @odd()
  br label %LoopIncrement
```

$seval(\ldots,$`%i.0`$)$ would evaluate to `%i.0`, because there are no assignments to `%i.0`.

3. Resource Consumption Analysis for LLVM IR

The techniques described here are used to infer cost relations [3]. Cost relations are recursively defined and closely follow the flow of the program. We need to solve these relations because what we actually want to infer is a closed form formula modeling the cost, parametric to any relevant input arguments to the program. These solvers typically work with simplified control flow graph structures, and therefore we must first perform some simplifications on the control flow graphs, as described in Section 3.3. The analysis then infers block arguments by using symbolic evaluation as described in Section 2.2.

3.1 Inferring block arguments

Block arguments characterize the input data that flows into the block, and is either consumed (killed) or propagated to another block or function. Unfortunately, solving multi-variate cost relations and recurrence relations automatically is still an open problem, and the fewer arguments each relation has, the easier it is to solve these. For this reason, we designed an analysis algorithm to minimize the block arguments before inferring the cost relations.

The algorithm for inferring block arguments is a data flow analysis algorithm. We use a standard means to describe this algorithm, as in [21]. We define a data flow analysis function gen, which, given a basic block, returns the variables of interest in that block:

$$gen(BB) = gen_{blk}(BB) \cup gen_{fn}(BB)$$

The function gen_{blk} returns the input arguments that affect the branching in a block BB, composed of instructions $inst_1$ through $inst_n$. gen_{fn} returns the variables that affect the input to any external calls in the block. gen_{blk} is defined as follows:

$$gen_{blk}(BB) = \begin{cases} ref(seval(BB, p)) & \text{if } inst_n = [\mathsf{br}\ p\ ..] \\ \emptyset & \text{otherwise} \end{cases}$$

The function ref returns all variables referred to in the symbolically evaluated expression given as argument, for example $ref(x > (y + 3))$ returns $\{x, y\}$. We also define function gen_{fn}. This returns all the input arguments that affect the parameters given to the function, and is defined as:

$$gen_{fn}(BB) =$$

$$\bigcup_{k=1}^{n} \begin{cases} \bigcup_{i=1}^{m} ref(seval(BB, a_i)) & \text{if } inst_k \text{ is } [x = \mathsf{call}\ f\ a_1\ ..\ a_m] \\ \emptyset & \text{otherwise} \end{cases}$$

The data flow analysis function $kill$ is defined as:

$$kill(BB) = \bigcup_{k=1}^{n} \begin{cases} \{x\} & \text{if } inst_k \text{ is } x = \mathsf{call}\ \ldots \\ \{x\} & \text{if } inst_k \text{ is } x = \mathsf{op}\ \ldots \\ \{x\} & \text{if } inst_k \text{ is } x = \mathsf{memload}\ \ldots \\ \emptyset & \text{otherwise} \end{cases}$$

Finally, we combine gen and $kill$ by utilizing a transfer function, which is inlined into $args_{in}$ and $args_{out}$. These compute the relevant block arguments utilized by the resource consumption analysis. $args_{in}(BB)$ is defined as the function's arguments if BB is the function's first block. In all other cases, $args_{in}$ and $args_{out}$ are defined as:

$$args_{out}(BB) = \bigcup_{BB' \in next(BB)} phimap_{\langle BB, BB' \rangle}(args_{in}(BB'))$$

$$args_{in}(BB) = (args_{out}(BB) - kill(BB)) \cup gen(BB)$$

where $phimap$ maps variables between adjacent blocks BB and BB' based on the ϕ instructions in BB'.

Functions $args_{in}$ and $args_{out}$ are recomputed until their least fixpoint is found. Finally, the block arguments are found in $args_{in}$. The analysis explained in this section is closely related to live variable analysis. A crucial difference, however, is in the function gen. In our case, this returns a smaller subset of variables than live variable analysis i.e., only the ones that may affect control flow.

3.2 Generating and solving cost relations

In order to generate cost relations we need to characterize the energy exerted by executing the instructions in a single block. We also need to model the continuations of each block. Continuations, expressed as calls to other cost relations, arise from either branching at the end of a block, or from function calls in the middle of a block. For instance, consider the following LLVM IR block:

```
LoopIncrement:
  %postinc = add i32 %i.0, 1
  %exitcond = icmp eq i32 %postinc, %1
  br i1 %exitcond, label %return, label %LoopBody
```

This would translate to the following relation:

$$C_{\mathrm{LI}}(i) = C_0 + C_{\mathrm{ret}}(i+1) \qquad \text{if } i + 1 = a_1$$
$$C_{\mathrm{LI}}(i) = C_1 + C_{\mathrm{LB}}(i+1) \qquad \text{if } i + 1 \neq a_1,$$

where C_{LI}, C_{ret} and C_{LB} characterize the energy exerted when running the blocks `LoopIncrement`, `return` and `LoopBody` respectively. We therefore refer to C_{ret} and C_{LB} as continuations of C_{LI}. Expressing these calls to other cost relations involves evaluating their arguments, which cannot be done without evaluating the program. Instead, by symbolically executing the block, we can express the arguments of the continuation in terms of the input arguments to the block. In order to do so, we perform symbolic evaluation using the function *seval*.

The cost relations, extracted from recursive programs using the techniques discussed in this section, can be automatically transformed to closed form by the PUBS solver. PUBS infers closed form solutions recursively, starting with the inner-most relations, using various techniques such as computing ranking functions and loop invariants. The results of the intermediate steps are then mathematically composed to solve the whole set of given cost relations.

There are cases where the optimized program structures produced by LLVM based compilers prevent the cost relation solvers from finding unique *cover points* in the structure of the cost relations. In order to solve this problem, we need to perform transformations to the call graph upon which we construct our cost relations. This is described in the next section.

3.3 Transformations for control flow graphs

After compilation, nested loop program structures are mangled by compiler optimizations. When the resulting Control Flow Graph (CFG) is directly used to produce CRs, it is usually not possible to infer closed form solutions. For instance PUBS cannot handle complex CFGs, and therefore in order to analyze programs with nested loops, the CFG needs to be simplified. The simplification is done at an early using the following steps:

1. Identify a loop's CFG, A, that has nested loops.
2. Identify the sub-CFG, B, of A corresponding to the inner loop.
3. Extract B out of A, so that B is a separate CFG. This can be thought of as a new function with multiple return points. Hence B's exit edges are removed.
4. In A, in the place where B used to be, keep the continuation to B. Append a continuation to B's exit targets to B's caller in A.

In order to perform the first two steps, we need to identify the loops in the CFG. While LLVM has specific passes to do so, we had better success when using the algorithm described in [33]. As an example, we show how these steps can be used to transform the CFG of a simple insertion sort, as shown in Listing 1. The original CFG of this program, when compiled using `clang` with optimization level `O2` is shown in Figure 2 (left). In this CFG, the nested loops are identified, which also involves identifying their corresponding entries, re-entries, exit and loop headers. Here, blocks `bb1`, `bb2` and `.backedge` form the inner loop. These blocks are hoisted and the exit edge from `.backedge` (dotted) is eliminated. Instead, `.loopexit` is then called after `bb1` "returns" (Figure 2).

The CFG simplifications described in this section preserve the same order of operations when applied to an existing CFG compiled from typical `while` or `for` using `clang` or `xcc`. This means that the program called in the left-side of Figure 2 will consume as much energy as the program in the right-side of Figure 2. The only limitation of this approach is when an induction variable of an outer loop is modified in an inner loop. In this case the transformation cannot occur, however we have not encountered real benchmarks where this takes place.

In order to verify the transformation with respect to energy, let us consider a typical `while` or `for` loop and show that the same sequence of blocks is called after the transformation takes place. We can assume that such a loop has a single header, but may

```
1  void sort(int numbers[], int size) {
2    int i=size, j, temp;
3    while(j = i--)
4      while(j--)
5        if(numbers[j] > numbers[i]) {
6          temp = numbers[i];
7          numbers[i] = numbers[j];
8          numbers[j] = temp;
9        }
10 }
```

Listing 1. This insertion sort demonstrates that certain classes of programs require further analysis or transformation.

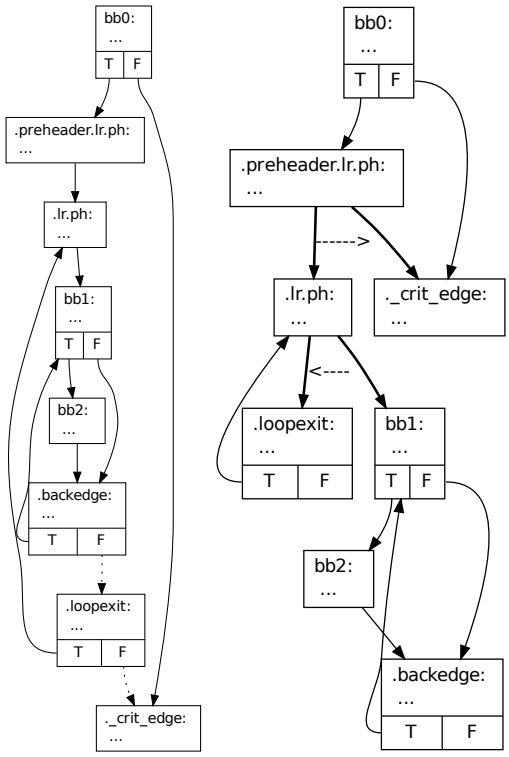


Figure 2. CFG of an insertion sort compiled using `clang` with optimization level `O2` before (left) and after simplification (right).

have multiple exits or reentries. However, the induction variables of the outer loops are not modified in the inner loops. After the transformation takes place on a nested loop structure (B inside A), B is still called from A, however B's exit edges are now removed. The target of B's exit edges will still be called after B completes. This is because we have appended a continuation in A to this target, in Step 4. Hence all blocks will be called in the same sequence. The argument above can be inductively applied to loops with arbitrary nesting levels.

4. Computing energy cost of LLVM IR blocks

The intermediate representation used by LLVM is architecture independent. Any given LLVM IR sequence can be passed to one of many different backends, including ISAs [16]. The exact implementation of the ISA determines the energy consumed by each instruction that is executed. Thus, the conversion to machine code,

together with the processor implementation, affects the energy consumption of an instruction at the LLVM IR level.

For static analysis of LLVM IR to produce useful energy formulae for programs, a method of assigning an energy cost to an LLVM IR segment must be used. Two possible methods are demonstrated in this paper, both having their advantages and disadvantages:

1. *ISA energy model w/mapping*. LLVM IR is mapped to its corresponding ISA instructions and the energy cost is obtained from the ISA level cost model. The advantage is that it is simpler to characterize at ISA level. However, this requires an additional step to correlate LLVM with ISA instructions.
2. *LLVM energy model*. Attributing costs directly to LLVM IR removes the need for a mapping. However, it necessarily simplifies the energy consumption characteristics, reducing accuracy.

In principle, both methods can be explored for both architectures. This paper utilizes an ISA level model for the XMOS processor whereas The Cortex-M is modeled at the LLVM IR level directly.

4.1 XMOS XS1-L ISA level modeling

The aim of ISA level modeling is to associate machine instructions with an energy cost. To achieve this, energy consumption samples must be collected and an appropriate representation of the underlying hardware must be used as a basis for the model. A single-threaded model, such as that defined by Tiwari [28] and expressed in Equation 1, describes the energy of a sequence of instructions, or program.

$$E_{\text{prog}} = \sum_{i \in \text{ISA}} (B_i N_i) + \sum_{i,j \in \text{ISA}} (O_{i,j} N_{i,j}) + \sum_{k \in \text{ext}} E_k \qquad (1)$$

The program's energy, E_{prog}, is first formed from the base cost, B_i, of all instructions, i, in the ISA, multiplied by the occurrences, N_i, of each instruction. For each transition in a sequence of instructions, the overhead, $O_{i,j}$, of switching from instruction i to instruction j, multiplied by the number of times the combinations of i and j, occurs $N_{i,j}$ times. Finally, for a set of k external effects, the cost of each of these effects, E_k is added. For example, these external effects may represent the cache and memory costs, based on the cache hit rate statistics of the program.

The XS1-L architecture implements multi-threading in a hardware pipeline. Even for single-threaded programs, we need to consider the behavior of this multi-threaded pipeline. The power of individual instructions varies by up to $2\times$, with multi-threading introducing up to a $1.6\times$ increase with a $4\times$ performance boost. This means execution time and energy are related in a more complex way than a simpler single-threaded architecture. The model for the XS1-L is built upon existing work of [29] and the more detailed [26], which obtain model data through the energy measurement of specific instruction sequences, and create a representation of some of the processor's internal structure in the model equations. A full description of the XS1-L's energy characteristics and the model is given in [13].

To extend a Tiwari style approach to model the XS1-L processor, two new characteristics must be accounted for: idle time and concurrency. The XS1 ISA has a number of event-driven instructions, which can result in the processor executing no instructions for a period of time, until the event occurs. Furthermore, the multi-threaded pipeline permits only one instruction from a given thread to be present in the pipeline at any one time. These changes are expressed in Equation 2. Here, the energy exerted by running a program depends on a base power, P_{base}, which represents the energy cost when no instructions are executed, multiplied by the number of idle periods, N_{idle}. The clock period of the processor, T_{clk} is also introduced, to allow different clock speeds to be considered. The inter-instruction overhead, previously described in Equation 1 as $O_{i,j}$, is generalized to a constant overhead, O, due to the unpredictability of instruction interaction between threads. For each instruction, the base cost is added to the instruction cost, P_i, which is scaled by the overhead and an additional scaling factor based on the number of active threads, M_t. This is multiplied by the number of occurrences of this instruction at t threads, $N_{i,t}$ and the clock period, T_{clk}. This is done for the varying number of threads, t that may be active in the program over its lifetime.

$$E_{\text{prog}} = P_{\text{base}} N_{\text{idle}} T_{\text{clk}} + \sum_{t=1}^{N_t} \sum_{i \in \text{ISA}} ((M_t P_i O + P_{\text{base}}) \times (N_{i,t} T_{\text{clk}})) \qquad (2)$$

The multi-threaded ISA level model for the XS1-L requires that for each level of concurrency, t, the number of instructions executed at that level should be known, or estimated. If a single threaded program is run on its own on the XS1-L and there are no idle periods, then Equation 2 simplifies to Equation 3, where the idle accounting is removed, and only the first threading level, $t = 1$, is considered.

$$E_{\text{prog}} = \sum_{i \in \text{ISA}} ((M_1 P_i O + P_{\text{base}}) \times (N_i T_{\text{clk}})) \qquad (3)$$

The current analysis effort focuses upon single threaded experiments, thus Equation 3 can be used. Multi-threaded analysis is proposed as future work in Section 7. Temperature variation in the device is not captured in this model. However, prolonged testing of the target hardware showed no significant temperature changes, or associated affects, that would influence the single-threaded tests performed in this work.

4.2 XMOS LLVM IR energy characterization by mapping

To enable the analysis at the LLVM IR level we need a mechanism to propagate the existing energy model at the ISA level up to the LLVM IR level. The mapping technique described in this section creates a fine grained mapping between segments of ISA instructions to LLVM IR instructions, in order to enable the energy characterization of each LLVM IR instruction in a program. A full description of the mapping techniques is given in [8].

Our mapping technique leverages the existing debug mechanism in the XMOS compiler toolchain. This mechanism is originally meant to facilitate the debugging process of an application, particularly when stepping through a program line by line. During the lowering phase of the compilation process, the LLVM IR code is transformed to the specific ISA code by the backend. The debug information (DI) is also stored alongside with the ISA code using the DWARF standard [1], a standardized debugging data format used by many compilers and debuggers to support source level debugging. By tracking this information we can extract an $n : m$ relationship between the two levels, because one source code instruction can be related to many different sequences LLVM IR instructions and therefore many different sequences of ISA instructions. Because this $n : m$ relation complicates static analysis, there is a need for a more fine grained mapping.

To address this issue, we created an LLVM pass that traverses the LLVM IR and replaces the *Source Location Information* with LLVM IR location information. This occurs after the optimization passes, but before the lowering phases. The lowering phases and platform-specific optimizations inside LLVM based compilers do not alter the structure of the program, and therefore the structure of optimized LLVM IR closely resembles the structure of the ISA program. In this way, we can extract a $1 : m$ relationship between the mapping of LLVM IR instructions and ISA instructions. After a mapping is extracted for a particular program, the associated energy

values for the ISA instructions corresponding to a specific LLVM IR instruction are aggregated and then associated with the LLVM IR instruction, and finally to every LLVM IR block.

Although we use the XMOS tool-chain for the mapper tool, the approach is generic and transferable, due to the use of the common LLVM optimizer and code generator, and the use of the DWARF standardized debugging data format, used by many compilers and debuggers to support source layer debugging.

4.3 LLVM IR energy model for ARM

An energy model for ARM Cortex-M series is applied directly at the LLVM IR level, based upon empirical energy measurement data, and knowledge of both the processor architecture and the compiler backend. The Cortex-M3 model is for the most part a simplification of the Tiwari model [28], applied at the LLVM IR level. The processor does not feature a cache, so it is not necessary to model *cache misses* as external effects. The effect of the switching cost between instructions is approximated into the actual instruction cost, rather than assigning a unique overhead for each instruction pairing.

Through analysis of energy measurements for a large set of the target ISA instructions, it was found that LLVM IR instructions, when compiled on this platform, can be segmented into four groups: memory, M, program flow, B, division, D, and all other instructions, G. The LLVM IR syntax described in Section 2 can be related to these groupings. In particular, `br`, `call` and `ret` can be combined into group B; `memload` and `memstore` are members of M; the subset of `op` relating to division make up group D; and finally, ϕ and all remembering members of `op` form group G.

This yields an equation, which accumulates the energy, E_{prog}, consumed by a program based on the number of instructions executed from each group. This is denoted in Equation 4, where E_i is the energy cost of a single instruction in group i, and N_i is the number of instructions executed in that group.

$$E_{\text{prog}} = \sum_{i \in \{M,B,D,G\}} E_i N_i \tag{4}$$

In addition, there are a number of other factors that affect energy, due to the relation between the LLVM IR and the ISA:

1. **Variadic arguments.** LLVM has instructions with variadic arguments. Typically, the number of arguments in the instruction affects the energy consumed in a linear manner.
2. **Data types.** LLVM operations `op` can be performed on values of different data types. If the data type is larger than 32 bits, or is floating point, this will translate into a larger number of ISA instructions on a Cortex-M with no floating point unit.
3. **Predicated instructions.** The Cortex-M processor is capable of executing predicated instruction sequences. In some cases, short LLVM IR blocks originating from ternary expressions in the original source code are directly translated to a number of predicated instructions in the ARM ISA. Therefore, the number of ISA instructions generated could be less than the instructions in LLVM IR, and the static analysis over-approximates the energy consumption of these blocks.

Factors (1) and (2) can be accounted for by parameterizing the LLVM IR energy model. For instance, consider the following call instruction:

```
%6 = call i32 @min(i32 %boptmp88, i32 %boptmp96)
```

Benchmark	L	NL	A	B	C
base64	×		×	×	
mac	×		×		
levenshtein	×	×	×	×	
insertion sort	×	×	×		
matrix multiply	×	×	×		
jpegdct	×	×	×	×	×

Table 1. Benchmark Characteristics.

This translates to a single branch instruction in the ARM ISA, with surrounding register moves to ensure the correct calling convention:

```
1 mov r0, r4    # move arg1 into r0
2 mov r1, r5    # move arg2 into r1
3 bl min        # call min
4 mov r4, r0    # move the result into r4
```

As we can see, the energy consumed by an LLVM `call` instruction is parametric in the number and types of the arguments and return value.

5. Experimental Evaluation

We have selected a series of benchmarks of core algorithmic functions, particularly from the BEEBS [22] and MDH WCET benchmark [9] suites. These are collections of open source benchmarks for deeply embedded systems, where the activities performed in these benchmarks are typical of such systems. Analysing benchmarks of this size and with their particular characteristics is therefore a good means of evaluating our analysis technique in order to demonstrate its usefulness within the embedded systems software space. The benchmarks are single threaded, reflecting the scope of the analysis performed in this paper. Minimal modifications were made to allow integration into our test harness. Table 1 summarizes the characteristics of the benchmarks, and the meaning of the last 5 columns is as follows: (L) contains loops, (NL) contains nested loops, (A) uses arrays and/or matrices, (B) contains bitwise operations, (C) contains loops with complex control flow predicates.

In order to show that our techniques are applicable to multiple languages and platforms, we have ported some of the benchmarks from C to XC. Porting C code to XC typically does not involve rewriting, since the syntax is very similar and they both use the same preprocessors. However, since XC does not provide pointers, some changes need to be made to the benchmarks during the porting process. For the benchmarks that run on the xCORE, we have used the XC compiler, version 13. For Cortex-M benchmarks we have used Clang version 3.5. We proceed by describing the benchmarks. In both cases, the benchmarks are compiled under optimization level `O2`.

Insertion sort. The code of the main function is shown in Figure 1. The energy exerted by the insertion sort partly depends on how many swaps need to take place, and this is dependent on the actual data present inside the array. Since PUBS infers a formula representing an upper bound of the closed form solution, we will be measuring the energy consumed while sorting a reverse-ordered list, and comparing this to the statically inferred formula. Note that the number of iterations in the inner loop depends on an induction variable in the outer loop. This benchmark is parameterized by the length of the list to be sorted, P.

Matrix multiply (BEEBS/MDH WCET). We modified this so that it could work with matrices of various sizes. The matrices are all square, of size P.

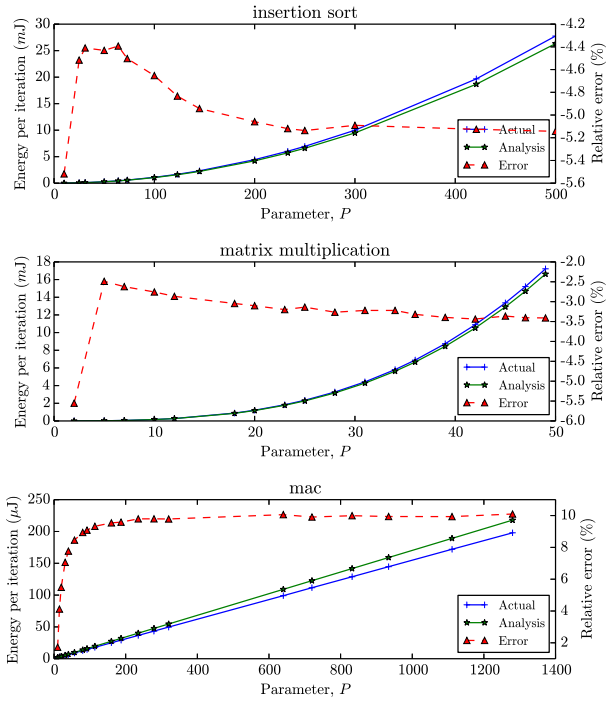


Figure 3. The measurement results and static analysis for the XMOS processor.

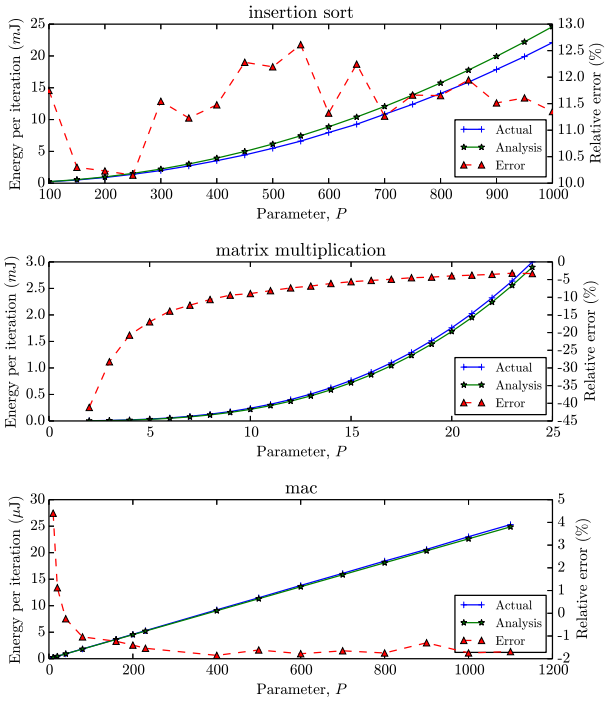


Figure 4. The measurement results and static analysis for the Cortex-M processor.

Base64 encode. Computes the base64 encoding[1] as a string, given an input string of length P.

MAC (MDH WCET). Dot product of two vectors together with sum of squares. Parameterized by the length of the vectors, P.

Jpegdct (MDH WCET). Performs a JPEG discrete cosine transform. Taken from the MDH WCET benchmark suite. This benchmark is not parameterized.

Levenshtein distance (BEEBS). Computes the minimum number of edits to change one string into another. The lengths of the two strings are parameterized with the variables A and B.

5.1 Experimental Setup

For both ARM and XMOS platforms, power measurement data is collected by using instrumented power supplies, a power sense IC and an embedded system running control and data collection software. The implementations differ, but are structurally very similar. Both of these periodically calculate the power (using Equation 5) during a test run, by sampling the voltage on either side of a shunt resistor (V_{bus} and V_{shunt}) to determine the supplied current.

$$P = I \times V_{\text{bus}} \quad \text{where} \quad I = \frac{V_{\text{bus}} - V_{\text{shunt}}}{R_{\text{shunt}}} \tag{5}$$

For the Cortex-M processor, the measurements are taken on an ST Microelectronics STM32VLDISCOVERY board while for the xCORE, a custom XMOS board with an XS1-L based XS1-U16A chip is used.

5.2 Results

The results for the XMOS xCORE and ARM Cortex-M processors are shown in Figures 3 and 4, respectively. These graphs show

[1] Posted by user2859193 on stackoverflow.com/questions/342409

```
1  void function(int A, int B) {
2    int i;
3
4    if(A < B)
5      for(i = 2*A; i >= 0; i--)
6        ...
7    else
8      for(i = B; i >= 0; i--)
9        ...
10 }
```

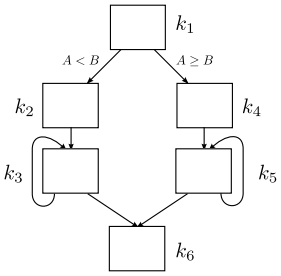


Figure 5. Example program of where the analysis infers a `max` formula, together with its CFG

the insertion sort, matrix multiplication and mac benchmarks, with data series for the static analysis results and actual energy measurements. The static analysis closely fits the empirical results, validating our approach. Table 2 shows the formulae and final errors for all benchmarks. Overall, the final error is typically less than 10% and 20% on the XMOS and ARM platforms respectively, showing that the general trend of the static analysis results can be relied upon to give an estimate of the energy consumption. We explain the sources of error in our results below:

Simple LLVM IR energy model (ARM). For the case of Cortex-M, the errors in the analysis mostly stem from the greatly simplified model of energy consumption in the Cortex-M. The LLVM IR energy model used for the Cortex-M assigns an energy cost to each IR instruction. Therefore, when an IR instruction expands to unexpected, or many ISA level instructions, the energy consumption can be inaccurate. In particular, for base64, ternary operators are heavily used inside its main loop. In LLVM IR, this introduces a

Benchmarks [22]	Formulae		Final error (%)	
	ARM (nJ)	XMOS (nJ)	ARM	XMOS
base64	$158 + 94 \cdot \lfloor \frac{P-1}{3} \rfloor$	$1270 + 734 \cdot \lfloor \frac{P-1}{3} \rfloor$	28.0	1.1
mac	$23P + 14$	$133P + 192$	-1.7	10.1
levenshtein	$47AB + 14A + 31B + 44$	$559AB + 78A + 571 + \max(225B, 180B + 213)$	7.0	0.4
insertion sort	$25P^2 + 11P + 7.1$	$105P^2 + 30P + 75$	11.1	3.0
matrix multiply	$20P^3 + 13P^2 + 97P + 84$	$144P^3 + 200P^2 + 77P + 332$	-3.3	-3.4
jpegdct	54 mJ‡	463 mJ‡	8.5	2.6

‡ This benchmark was not parameteric, thus is not parameterised.

Table 2. Formulae and error values for each benchmark.

number of short conditional blocks inside this loop. These multiple basic blocks in LLVM IR are translated to a smaller number of predicated instructions in the ARM ISA by the compiler, so the static analysis will over approximate the energy consumed.

Measurement error. Measurement errors are introduced from environmental factors such as temperature and power supply fluctuations. The tolerance of the components is also another factor. The test harness contributes error too, as it must call a function repeatedly in order to get its energy measurements. The loop surrounding this function, together with the act of calling can be a significant overhead when the amount of computation inside the function is low. In fact, we can see that in all cases, the relative error converges to a single error result. This is expected because in all of the benchmarks, the parameter controls the number of iterations performed in one or more loops. As the parameter increases, the difference in the constant energy overhead is minimized, with respect to the energy consumption of the function under test. Measurement runs were run numerous times to ensure consistency of results within the expected error margins described above.

Data flow through the processor's execution units. The energy models for the xCORE and ARM assume a random distribution of operand data. In practice, however, operations such as logical tests, bit-manipulation and instructions performed on shorter data types such as `char` will not use the full bit-range of the data path. In cases such as these, energy consumption will be lower, therefore introducing some estimation inaccuracy.

LLVM IR to ISA mapping (xCORE). In the case of the xCORE, the overall results are better than that of the Cortex-M. This is due to a more accurate assignment of energy values to LLVM-IR instructions, which the mapper can produce for each individual program, as described in Section 4.2. Nevertheless, the mapper introduces analysis error. For instance, the mapper does not consider instruction scheduling on the processor, where an instruction fetch stall can happen in some limited scenarios. This can be addressed by performing a further local analysis on the ISA code to determine the possible locations where this happens, and adjusting the energy accordingly. Another problem arises when mapping LLVM IR `phi` instructions to the "corresponding" ISA code. Copy instructions at ISA level, attributed to LLVM IR `phi` instructions inside a loop may be hoisted out of loops. The hoisted ISA cost is thus counted for each loop iteration, leading to an overestimate. This phenomenon was partially addressed by automatically adjusting the energy of `phi` instructions during the mapping.

Static analysis and data dependence. Programs where the behavior and state depends on complex properties of the actual input data are problematic for static resource consumption analysis. An extreme example of such a program would be an interpreter. The execution time of an interpreter not only depends on the size of the program file it is supplied, but also on the program represented in this file. A more typical example would be the euclidean algorithm

```
1  int distances[MAX];
2
3  void sortbysimilarity(char *word, int word_len,
4        char *dictionary[], int dictword_len,
5        int n_strings)
6  {
7      int i = n_strings;
8
9      while(i--) {
10       distances[i] = levenshtein(word,
11               dictionary[i], word_len,
12               dictword_len);
13     }
14     sort(distances, dictionary, n_strings);
15 }
```

Listing 2. Sort by similarity function, demonstrating that the analysis can be composed across multiple functions.

($gcd(a, b)$), where the number of steps taken to execute depends on a relationship between its parameters a and b. Our static analysis technique, however, still manages to compute an approximation – a logarithmic function with base 2, which is dependent on only one of the arguments. Part of the reason why we can analyze programs of this type is that symbolic evaluation of modulus between two variables $x \bmod y$ is defined as an upper bound of $y - 1$, a lower bound of 0 and an approximation of $(y - 1)/2$.

The `levenshtein` cost function for the xCORE processor includes a `max` function, making it a different type of formula from the Cortex-M's cost function. This occurs when a data dependent branch is on the upper bound of the function and the analysis is unable to resolve the branch statically, possibly because the branching is data dependent. An example of this is shown in Figure 5. The analysis cannot statically ascertain the outcome of the $A < B$ expression, so it simply returns the cost function as the maximum of the two possible branches:

$$function = k_1 + \max(k_2 + 2 \cdot k_3 \cdot A, k_4 + k_5 \cdot B) + k_6,$$

where $k_1, ..., k_6$ are the costs of executing the respective basic blocks, as seen in Figure 5.2. The same effect causes `max` to appear in the xCORE's formula since there is a data dependent `if` statement in an inner loop of `levenshtein`.

5.3 Composability

All of the benchmarks so far have consisted of relatively simple code, for which a single function is analyzed. However, the analysis can handle nesting and recursion, in the same way that it can handle functions with multiple basic blocks. In the code in Listing 2, the `levenshtein` and modified `insertion sort` functions are composed into a simple spell checker — for a given string, sort the list of strings by the `sortbysimilarity` to the target string.

In this listing, `dictword_len` is the maximum size of the strings in `dictionary`. Inferring a cost formula for this program does not present any issues as long as it is possible to infer formulas for its

constituent parts. Our techniques construct Cost Relations (CRs) from the program that is being analyzed. An important feature of CRs is their compositionality. Compositionality allows to compute closed form solutions of CRs, composed of multiple relations, by concentrating on one relation at a time. The process starts by solving cost relations that do not depend on any other relations and proceeds by replacing the these cost relations in the equations which call such relations. For instance, for the above program `levenshtein_distance` has an associated energy cost of

$$(A(53B + 16) + 35B + 31) \text{ nJ}, \tag{6}$$

where A and B are the third and fourth arguments to the function. Our modified string sorting routine has a cost of:

$$\left(37A^2 + 14A + 14\right) \text{ nJ}. \tag{7}$$

These functions are systematically combined together so that a cost for `sortbysimilary` is computed. In this case it is

$$\left(530ABC + 157AC + 346BC + 366C^2 + 629C + 210\right) \text{ nJ}, \tag{8}$$

where A is `word_len`, B is `dictword_len` and C is `n_strings`.

6. Related Work

Related work exists in four different areas: energy modeling of processors, mapping low-level program segments to higher level structures, static resource consumption usage analysis and worst-case execution time analysis (WCET).

Energy models of processors for program analysis require energy consumption data in relation to the program's instructions. This data can be collected by simulating the hardware at various levels, including semiconductor [17] and CMOS [5]. Alternatively, higher level representations may be used, such as functional block level [26] that reflects the micro-architecture, direct measurement on a per-instruction basis [28], or profiling the energy consumption of commonly used software blocks [24]. Higher level data collection and modeling efforts are typically quicker to use once the data has been acquired, as there is less computational burden than a low-level simulation. However, the accuracy may be lower, so a suitable trade-off must be met.

Although substantial effort has been devoted to ISA energy modeling, there is not a lot of work done for higher level program representations. This is mostly because precision decreases when moving further away from the hardware. One of the most recently pertinent works for LLVM IR energy modeling is [6]. The authors performed statistical analysis and characterization of LLVM IR code, together with instrumentation and execution on the host machine, to estimate the performance and energy requirements in embedded software. In their case, retrieving the LLVM IR energy model to a new platform requires performing the statistical analysis again. Our LLVM IR energy model takes into consideration types and other aspects of the instructions. Furthermore, our mapping technique requires only the adjustment of the LLVM mapping pass for the new architecture.

Static cost analysis techniques based on setting up and solving recurrence equations date back to Wegbreit's [32] seminal paper, and have been developed significantly in subsequent work [3, 7, 20, 25, 30]. In [18] this approach is applied to the static inference of Java programs' resource consumption as functions of input data sizes, by specializing a generic resource analyzer [10, 20] to Java bytecode analysis [19]. However, this work did not compare the results with measured energy consumptions. In [15] the approach is applied to the energy analysis of XC programs using ISA-level models [13], and the results are compared to actual hardware measurements. Our analysis continues in this line of work but with a number of important differences. First, analysis is performed at the LLVM-IR level and we propose novel techniques for reflecting the ISA-level energy models at the LLVM-IR level. Instead of using a generic resource analyzer (requiring translating blocks to its Horn Clause-based input syntax) and delegating the generation of cost equations to it, we generate the equations directly from the LLVM-IR compiler representation, performing control flow simplifications, and reducing the number of variables modelled by the analysis mechanism. Finally, we study a larger set of benchmarks. Other approaches to cost analysis exist, such as those using dependent types [11], SMT solvers [4], or *size change abstraction* [36].

As discussed in Section 1, energy and time are often correlated to some degree. Techniques such as implicit path enumeration [14] are often used in worst-case execution time analysis of programs. In most cases, programs are assumed to be preprocessed such that no loops are present (e.g. using loop unrolling). Some approaches such as [12] focus on statically predicting cache behavior. WCET analysis is concerned with getting an absolute worst-case timing for hard real-time systems. In practice, for energy consumption analysis we typically are more interested in average cases. Also, most WCET analysis approaches produce absolute timing figures. In our case, we infer energy formulae parameterized by the program's input.

7. Conclusion and Future Work

In this paper we have introduced an approach for estimating the energy consumption of programs based on the LLVM compiler framework. We have shown that this approach can be applied to multiple embedded languages (such as C or XC), compiled using optimization level `O2` with different compilers (such as Clang or XCC). We have also validated this approach for multiple backends, via two target architectures: ARM Cortex M3 and XMOS XS1-L. Our approach is validated by comparing the static analysis to physical measurement taken from the hardware. The results on our benchmarks show that energy estimations using our technique are within 10% and 20% or better in the case of the xCORE and the Cortex-M processors, respectively.

Although the techniques discussed here were initially designed for single threaded programs, these can be adapted to multi-threaded programs. To do so, we need to take the synchronization time into consideration. For example, the XC language has explicit constructs for thread communication using channels, and therefore the blocking communication between threads needs to be modeled. In order to do so, we can analyze the communication throughput of individual threads using techniques discussed in this paper. Using this information we can estimate the time between events happening on channels and hence the utilization of the processor. This, coupled with multi-threaded energy models as discussed in Section 4.1, can be used to analyze multi-threaded programs.

An interesting direction is to further develop the assignment of energy to LLVM IR program segments. In particular, an LLVM IR energy model for the xCORE can be implemented by using the information gathered from the mapping technique together with statistical analysis. The mapping technique used for the xCORE can also be adapted for the ARM case. We aim to further develop our techniques so they can be applied against other embedded processor architectures, such as MIPS, or other ARM variants. In particular, it would be interesting to see how far this technique can be effective with architectures that feature multiple pipelines, register renaming and similar optimizations. Further work could involve energy models applied to short strings of instructions, augmented with appropriate heuristics.

Finally, the static analysis techniques can be improved further. Currently the biggest limitation is solving the cost relations. Cost relations could also be solved numerically, enabling us to analyze more complex programs. An implementation of this can be used when actual formulae are not required.

Acknowledgments

The research leading to these results has received funding from the European Union 7th Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency, and grant agreement 611004, project ICT-Energy. The energy measurement hardware (MAGEEC WAND) used for measuring the energy of the ARM benchmarks was funded by Innovate UK, award 131198. Special thanks are due to Pedro Lopez-Garcia and his team at the IMDEA Software Institute for many fruitful and inspiring discussions. We would like to thank our project partners at Roskilde University and at XMOS. Thanks also go to Samir Genaim for his help on how to best make use of the PUBS solver.

References

[1] The dwarf debugging standard, Oct. 2013. `http://dwarfstd.org/`.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Cost relation systems: A language-independent target language for cost analysis. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 248:31–46, Aug. 2009.

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

[4] D. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. 7460:405–421, 2012.

[5] A. Bogliolo, L. Benini, G. D. Micheli, and B. Ricc. Gate-Level Power and Current Simulation of CMOS Integrated Circuits. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 5(4):473–488, 1997.

[6] C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 333–338, Aug 2011.

[7] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[8] K. Georgiou, S. Kerrison, and K. Eder. A multi-level worst case energy consumption static analysis for single and multi-threaded embedded programs. Technical Report CSTR-14-003, University of Bristol, December 2014.

[9] J. Gustafsson. The Mälardalen WCET benchmarkspast, present and future. *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[10] M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[11] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.

[12] N. D. Jones and M. Müller-Olm, editors. *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, Lecture Notes in Computer Science. Springer, 2009.

[13] S. Kerrison and K. Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):56:1–56:25, Apr. 2015.

[14] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.

[15] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Preproceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, September 2013.

[16] LLVM Project. Writing an LLVM backend. `http://llvm.org/docs/WritingAnLLVMBackend.html`, 2014. Accessed: 2014-03-11.

[17] L. W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975.

[18] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.

[19] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of BYTECODE*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 65–82. Elsevier - North Holland, March 2009.

[20] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP'07)*, Lecture Notes in Computer Science. Springer, 2007.

[21] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[22] J. Pallister, S. J. Hollis, and J. Bennett. BEEBS: open benchmarks for energy measurements on embedded platforms. *CoRR*, abs/1308.5174, 2013.

[23] J. Pallister, S. J. Hollis, and J. Bennett. Identifying Compiler Options to Minimise Energy Consumption for Embedded Platforms. *Computer Journal*, 2013.

[24] G. Qu, N. Kawabe, K. Usami, and M. Potkonjak. Function-level power estimation methodology for microprocessors, 2000. 337786 810-813.

[25] M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press, 1989.

[26] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-level Energy Model Supporting Software Optimizations. In *Proceedings of PATMOS*, 2001.

[27] C. Tice and S. L. Graham. Optview: A new approach for examining optimized code. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '98, pages 19–26, New York, NY, USA, 1998. ACM.

[28] V. Tiwari, S. Malik, and A. Wolfe. *Power analysis of embedded software: a first step towards software power minimization*, pages 222–230. Kluwer Academic Publishers, 1994. 567021.

[29] V. Tiwari, S. Malik, A. Wolfe, and M. T. C. Lee. Instruction level power analysis and optimization of software. In *Proceedings of VLSI Design*, pages 326–328, 1996.

[30] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *15th International Workshop on Implementation of Functional Languages (IFL'03), Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, September 2005.

[31] D. Watt. *Programming XC on XMOS Devices*. XMOS Ltd., 2009.

[32] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[33] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In *Proceedings of SAS*, pages 170–183, 2007.

[34] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[35] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of POPL*, POPL '12, pages 427–440, 2012.

[36] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction (extended version). *CoRR*, abs/1203.5303, 2012.

Attachment D3.3.7

Constraint Specialisation in Horn Clause Verification

Published at the Workshop on Partial Evaluation and Program Manipulation (PEPM 2015)

Constraint Specialisation in Horn Clause Verification

Bishoksan Kafle
Roskilde University, Denmark
kafle@ruc.dk

John P. Gallagher
Roskilde University, Denmark and IMDEA Software Institute, Spain
jpg@ruc.dk

Abstract

We present a method for specialising the constraints in constrained Horn clauses with respect to a goal. We use abstract interpretation to compute a model of a query-answer transformation of a given set of clauses and a goal. The effect is to propagate the constraints from the goal top-down and propagate answer constraints bottom-up. Our approach does not unfold the clauses at all; we use the constraints from the model to compute a specialised version of each clause in the program. The approach is independent of the abstract domain and the constraints theory underlying the clauses. Experimental results on verification problems show that this is an effective transformation, both in our own verification tools (convex polyhedra analyser) and as a pre-processor to other Horn clause verification tools.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords constraint specialisation; query-answer transformation; Horn clauses; abstract interpretation; convex polyhedral analysis

1. Introduction

In this paper, we present a method for specialising the constraints in constrained Horn clauses with respect to a goal. To this end, we first compute a query-answer transformation of a given set of clauses (also called a constraint logic program) with respect to a goal; the aim of the transformation is to simulate the top-down evaluation of the clauses in a bottom-up framework. Then we use abstract interpretation to compute a model of the query-answer transformed program. The idea is to propagate the constraints from the goal top-down and propagate answer constraints bottom-up. Finally we compute a specialised version of each clause in the original program using the constraints from the model without unfolding the clauses at all.

As a result, each clause is further strengthened or removed altogether, preserving the derivability of the goal. Static analysis of the specialised program becomes easier since the implicit constraints in the original clauses are made explicit in the specialised version. In addition to this, since the specialised clauses are more constrained or more specific than the original ones, more specific information will be available for proving the given goal or a failure to prove the goal may be detected at an early stage.

The approach is independent of the abstract domain and the constraints theory. Query-answer transformations, closely related to so-called "magic set" transformations, have been used for Datalog query processing and logic program analysis since the 1980s [3, 16], but have not, to our knowledge, been applied to verification problems. Experimental results on verification problems show that this is an effective transformation, propagating information both backwards from the statement to be proved, and forwards from the Horn clause theory. We show its effectiveness both in our own verification tools and as a pre-processor to other Horn Clause verification tools. In particular, we run our specialisation procedure as a pre-processor both to our *convex polyhedra analyser* and to QARMC [24, 43], a state of the art verification tool. We make the following contributions in this paper:

- we present a method for specialising the constraints in the clauses using *query-answer transformation* and abstract interpretation (see Section 4); and
- we demonstrate the effectiveness of transformation by applying it to Horn clause verification problems (see Section 6).

2. Preliminaries

A constrained Horn clause (CHC) is a first order predicate logic formula of the form $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \rightarrow p(X))$ $(k \geq 0)$, where ϕ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k)$ is the body.

Pure constraint logic programs (CLP) are syntactically and semantically the same as CHC. Unlike CLP, CHCs are not always regarded as executable programs, but rather as specifications or semantic representations of other formalisms. However these are only pragmatic distinctions and the semantic equivalence of CHC and CLP means that techniques developed in one framework are applicable to the other. We follow the syntactic conventions of CLP and write a Horn clause as $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$. In this paper we take the constraint theory to be linear arithmetic with the relation symbols $\leq, \geq, <, >$ and $=$, but the contributions of the paper are independent of the constraint theory.

2.1 Interpretations and models

An interpretation of a set of CHCs is a truth assignment to each atomic formula $p(a_1, \ldots, a_n)$ where p is a predicate and $a_1, \ldots, a_n$ are constants from the constraint theory. An interpretation is represented as a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atomic formula $p(Z_1, \ldots, Z_n)$ where $Z_1, \ldots, Z_n$ are distinct variables and ϕ is a constraint over $Z_1, \ldots, Z_n$. If ϕ is `true` we write $A \leftarrow$ or just A. The constrained fact $A \leftarrow \phi$ is

PEPM '15, January 13–14, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2678015.2682544

shorthand for the set of variable-free facts $A\theta$ such that $\phi\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

Minimal models. A model of a set of CHCs is an interpretation that satisfies each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator (called S_P^D in [28, Section 4]), which is an extension of the standard T_P operator from logic programming, extended to handle the constraint domain D. Furthermore $\mathsf{lfp}(S_P^D)$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, S_P^D(\emptyset), S_P^D(S_P^D(\emptyset)), \ldots$. This sequence provides a basis for abstract interpretation of CHC clauses.

3. Abstract Interpretation

Abstract interpretation [11] is a static program analysis technique which derives sound overapproximations of programs by computing abstract fixed points. Convex polyhedra analysis (CPA) [13] is a program analysis technique based on abstract interpretation [11]. When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \phi$ for each predicate p. The constraint ϕ is a conjunction of linear inequalities, representing a convex polyhedron.

The first application of convex polyhedra analysis to CHCs was by Benoy and King [4]. We summarise briefly the elements of convex polyhedra analysis for CHC; further details (with application to CHC) can be found in [4, 13]. Let $\mathcal{A}$ be the set of convex polyhedra (for some fixed dimension). Let P be a set of CHCs. Suppose there are n predicates in P, say $p_1, \ldots, p_n$, and assume to simplify the discussion that all predicates have the same arity (the dimension of $\mathcal{A}$). The *abstract domain* for P is the set of n-tuples of convex polyhedra $\mathcal{A}^n$. Let the empty polyhedron be denoted $\bot$. Inclusion of polyhedra is a partial order on $\mathcal{A}$ and the partial order $\sqsubseteq$ on $\mathcal{A}^n$ is its pointwise extension. Given an element $\langle d_1, \ldots, d_n\rangle \in \mathcal{A}^n$, define the *concretisation* function γ such that $\gamma(\langle d_1, \ldots, d_n\rangle) = \{\langle p_1(a_1), \ldots, p_n(a_n)\rangle \mid a_i \text{ is a point in } d_i, 1 \leq i \leq n\}$. Construct an *abstract semantic function* $F : \mathcal{A}^n \to \mathcal{A}^n$ satisfying the *safety condition* $S_P^D \circ \gamma \subseteq \gamma \circ F$ which is monotonic with respect to $\sqsubseteq$, where S_P^D is the immediate consequences operator mentioned above. Let the increasing sequence $Y_0, Y_1, \ldots$ be defined as follows. $Y_0 = \bot^n, Y_{n+1} = F(Y_n)$. These conditions are sufficient to establish that if the limit of the sequence exists, say Y, that $M[\![P]\!] = \mathsf{lfp}(S_P^D) \subseteq \gamma(Y)$ [12].

Since $\mathcal{A}^n$ contains infinite increasing chains, the sequence can be infinite. The use of a *widening* operator for convex polyhedra [11, 13] is needed to ensure convergence of the abstract interpretation. Define the sequence $Z_0 = Y_0$, $Z_{n+1} = Z_n \nabla F(Z_n)$ where ∇ is a widening operator for convex polyhedra [13]. The conditions on ∇ ensure that the sequence stabilises; thus for some finite j, $Z_i = Z_j$ for all $i > j$ and furthermore that Z_j is an upper bound for the sequence $\{Y_i\}$. The value Z_j thus represents, via the concretisation function γ, an over-approximation of the least model of P. Furthermore much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds [27]. A threshold is an assertion that is combined with a widening operator to improve its precision. Recently, a technique for deriving more effective thresholds was developed [34], which we have adapted and found to be very effective in experimental studies. In brief, the method collects constraints by iterating the concrete immediate consequence function S_P^D three times starting from the "top" interpretation, that is, the interpretation in which all atomic facts are true.

4. Specialisation by constraint propagation

We next present a procedure for specialising CHCs. In contrast to classical specialisation techniques based on partial evaluation with respect to a goal, the specialisation does not unfold the clauses at all; rather, it computes a specialised version of each clause, in which the constraints from the goal are propagated top-down and answers are propagated bottom-up.

We first make precise what is meant by "specialisation" for CHCs. Let P be a set of CHCs and let A be an atomic formula. The specialisation of P with respect to A is a set of clauses P_A such that for every constraint ϕ over the variables of A, $P \models \forall(\phi \rightarrow A)$ if and only if $P_A \models \forall(\phi \rightarrow A)$. This is a very general definition that allows for many transformations. In practice we are interested in specialisations that eliminate consequences of P that have no relevance to A.

For each clause $H \leftarrow \mathcal{B}$ in P, P_A contains a new clause $H \leftarrow \phi, \mathcal{B}$ where ϕ is a constraint. If the addition of ϕ makes the clause body unsatisfiable, it is the same as removing the clause. Clearly P_A may have fewer consequences than P but our procedure guarantees that it preserves the inferability of (constrained instances of) A. The procedure is summarised as follows: the inputs are a set of CHCs P and an atomic formula A.

1. Compute a *query-answer transformation* of P with respect to A, denoted P^{qa}, containing predicates p^{q} and p^{a} for each predicate p in P.
2. Compute an over-approximation M of the model of P^{qa}.
3. Strengthen the constraints in the clauses in P, by adding constraints from the answer predicates in M.

Next we will explain each step in detail.

4.1 The query-answer transformation

The *query-answer transformation* in CLP was inspired by the magic-set transformation from deductive databases and the language Datalog [3]. Its purpose, both in deductive databases and in subsequent applications in logic program analysis [16] was to simulate goal-directed (*top-down*) computation or deduction in a goal-independent (*bottom-up*) framework.

In the following, for each atom $A = p(t)$, A^{a} and A^{q} represent the atoms $p^{\mathsf{a}}(t)$ and $p^{\mathsf{q}}(t)$ respectively. Given a set of CHCs P and an atom A, the (left-) query-answer clauses for P with respect to A, denoted P_A^{qa} or just P^{qa}, are as follows.

- (Answer clauses). For each clause $H \leftarrow \phi, B_1, \ldots, B_n$ ($n \geq 0$) in P, P^{qa} contains the clause $H^{\mathsf{a}} \leftarrow \phi, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \ldots, B_n^{\mathsf{a}}$.
- (Query clauses). For each clause $H \leftarrow \phi, B_1, \ldots, B_i, \ldots, B_n$ ($n \geq 0$) in P, P^{qa} contains the following clauses:

$$
\begin{array}{l}
B_1^{\mathsf{q}} \leftarrow \phi, H^{\mathsf{q}}. \\
\cdots \\
B_i^{\mathsf{q}} \leftarrow \phi, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \ldots, B_{i-1}^{\mathsf{a}}. \\
\cdots \\
B_n^{\mathsf{q}} \leftarrow \phi, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \ldots, B_{n-1}^{\mathsf{a}}.
\end{array}
$$

- (Goal clause). $A^{\mathsf{q}} \leftarrow \mathsf{true}$.

The clauses P^{qa} encodes a left-to-right, depth-first computation of the query $\leftarrow A$ for CHC clauses P (that is, the standard CLP computation rule, SLD extended with constraints). This is a complete proof procedure, assuming that all clauses matching a given call are explored in parallel. (Note: the incompleteness of standard Prolog

CLP proof procedures arises due to the fact that clauses are tried in a fixed order). It is important to generate the queries and answers in a single set of clauses, since in general the predicates p^{q} and p^{a} are mutually recursive. Independent analyses propagating constraints from head to body of the clauses and propagating constraints from body to head would not in general achieve the same specialisation.

The relationship of the model of the clauses P^{qa} to the computation of the goal $\leftarrow A$ in P is expressed by the following property[1]. An SLD-derivation in CLP is a sequence $G_0, G_1, \ldots, G_k$ where each G_i is a goal $\leftarrow \phi, B_1, \ldots, B_m$, where ϕ is a constraint and $B_1, \ldots, B_m$ are atoms. In a left-to-right computation, G_{i+1} is obtained by resolving B_1 with a program clause. The model of P^{qa} captures the set of atoms that are "called" or "queried" during the derivation, together with the answers (if any) for those calls. This is expressed precisely by Property 1.

PROPERTY 1 (Correctness of query-answer transformation). *Let P be a set of CHCs and A be an atom. Let P^{qa} be the query-answer program for P wrt. A. Then*

(i) if there is an SLD-derivation $G_0, \ldots, G_i$ where $G_0 = \leftarrow A$ and $G_i = \leftarrow \phi, B_1, \ldots, B_m$, then $P^{\mathsf{qa}} \models \forall(\phi|_{\mathsf{vars}(B_1)} \rightarrow B_1^{\mathsf{q}})$;

(ii) if there is an SLD-derivation $G_0, \ldots, G_i$ where $G_0 = \leftarrow A$, containing a sub-derivation $G_{j_1}, \ldots, G_{j_k}$, where $G_{j_i} \leftarrow \phi', B_1, \mathcal{B}'$ and $G_{j_k} = \leftarrow \phi, \mathcal{B}'$, then $P^{\mathsf{qa}} \models \forall(\phi|_{\mathsf{vars}(B_1)} \rightarrow B_1^{\mathsf{a}})$. (This means that the atom B_1 in G_{j_i} was successfully answered, with answer constraint $\phi|_{\mathsf{vars}(B_1)}$).

(iii) As a special case of (ii), if there is a successful derivation of the goal $\leftarrow A$ with answer constraint ϕ then $P^{\mathsf{qa}} \models \forall(\phi \rightarrow A^{\mathsf{a}})$.

Variations such as the following have been used.

- (Refined call predicates). Call predicates of the form $p^{\mathsf{q}}_{i,j}$ could be generated representing calls to the i^{th} atom in the body of clause j [20], giving more fine-grained information on calls.
- (Relaxed answer predicates). In this version the answer clauses are the same as the original clauses of p, and every answer predicate p^{a} is just replaced by p. This can be used where the only interest is in the model of the query predicates, and the motivation is to increase efficiency of analysis of P^{qa}, while possibly losing precision [9].
- (Other computation rules). Left-to-right computation could be replaced by right-to-left or any other order. The success or failure of a goal is independent of the computation rule; hence we could generate answers using other computation rules, or combining computation rules [21]. While different computation rules do not affect the model of the answer predicates, more effective propagation of constraints during program analysis, and thus greater precision, can sometimes be achieved by varying the computation rule.

For each such variation a correctness property can be stated relating the model of the query-answer program to the SLD computation of the given program P and goal A.

4.2 Over-approximation of the model of P^{qa}

The query-answer transformation of P with respect to A is computed. It follows from Property 1(iii) that if A is derivable from P then $P^{\mathsf{qa}} \models A^{\mathsf{a}}$. Abstract interpretation of P^{qa} yields an over-approximation of $M[\![P^{\mathsf{qa}}]\!]$, say M', containing constrained facts for the query and answer predicates. These represent the calls and answers generated during all derivations starting from the goal A. In our experiments we use a convex polyhedra approximation of $M[\![P^{\mathsf{qa}}]\!]$, as described in Section 3.

4.3 Strengthening the constraints in P

We use the information in M' to specialise the original clauses in P. Suppose M' contains constrained facts $p^{\mathsf{q}}(X) \leftarrow \phi^{\mathsf{q}}$ and $p^{\mathsf{a}}(X) \leftarrow \phi^{\mathsf{a}}$. (If there is no constrained fact $p^*(X) \leftarrow \phi^*$ for some p^* then we consider M' to contain $p^*(X) \leftarrow \mathit{false}$).

For each clause

$$p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$$

in P, construct a clause

$$p(X) \leftarrow \phi, \phi_0, \phi_1, \ldots, \phi_n, p_1(X_1), \ldots, p_k(X_k)$$

in P_A, where $p^{\mathsf{a}}(X) \leftarrow \phi_0$, $p_1^{\mathsf{a}}(X) \leftarrow \phi_1, \ldots, p_n^{\mathsf{a}}(X) \leftarrow \phi_n$ are in M'. Here we assume that there is exactly one constrained fact in M' for each predicate $p^a, p_1^a, \ldots, p_n^a$. Disjunctive constraints can be eliminated from the specialised clauses by further transformation and clauses containing the constraint false in the body are eliminated.

Note that wherever M' contains constrained facts $p^{\mathsf{a}}(X) \leftarrow \phi^{\mathsf{a}}$ and $p^{\mathsf{q}}(X) \leftarrow \phi^{\mathsf{q}}$, we have $\phi^{\mathsf{a}} \rightarrow \phi^{\mathsf{q}}$ since the answers for p are always stronger than the calls to p. Thus it suffices to add only the answer constraints to the clauses in P and we can ignore the model of the query predicates. A special case of this is where M' contains a constrained fact $p^{\mathsf{q}}(X) \leftarrow \phi^{\mathsf{q}}$ but there is no constrained fact for $p^{\mathsf{a}}(X)$, or in other words M' contains the constrained fact $p^{\mathsf{a}}(X) \leftarrow \mathit{false}$. This means that all derivations for $p(X)$ fail or loop in P and so adding the answer constraint false for p eliminates looping and failed derivations for p.

Specialisation by strengthening the constraints preserves the answers of the goal with respect to which the query-answer transformation was performed. In particular, we have the following property.

PROPERTY 2. *If P is a set of CHCs and P_A is the set obtained by strengthening the clause constraints as just described, then $P \models A$ if and only if $P_A \models A$.*

The proof of Property 2 is by induction on the length of derivations of A. For each derivation of A using the clauses of P we can construct a derivation of A in P_A and conversely.

The specialisation and analysis are separate in our approach. More complex algorithms intertwining them can be envisaged, though the benefits are not clear. Iteration of our procedure could potentially yield further specialisation.

5. Application to the CHC verification problem

In this section, we discuss the application of our constraint specialisation in Horn clause verification. We assume that there is a distinguished predicate symbol false in P which is always interpreted as false. In practice the predicate false only occurs in the head of clauses; we call clauses whose head is false *integrity constraints*, following the terminology of deductive databases. Thus the formula $\phi_1 \leftarrow \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$ is equivalent to the formula $\mathsf{false} \leftarrow \neg\phi_1 \wedge \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$. The latter might not be a CHC (e.g. if ϕ_1 contains $=$) but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula $X = Y \leftarrow p(X, Y)$ is equivalent to the set of CHCs $\{\mathsf{false} \leftarrow X > Y, p(X, Y),\ \mathsf{false} \leftarrow X < Y, p(X, Y)\}$.

Integrity constraints can be seen as safety properties. For example if a set of CHCs encodes the behaviour of a transition system, the bodies of integrity constraints represent unsafe states. Thus

[1] Note that the model of P^{qa} might not correspond exactly to the calls and answers in the SLD-computation, since the CLP computation treats constraints as syntactic entities through decision procedures and the actual constraints could differ.

proving safety consists of showing that the bodies of integrity constraints are false in all models of the CHC clauses. Figure 1 shows an example set of CHCs taken from [23].

```
c1. false :- A>0,B=0,C=0,D=0,l(B,C,D,A).
c2. l(A,B,C,D) :- -A+D>0,A-G= -1, l_body(B,C,E,F),
                  l(G,E,F,D).
c3. l(A,B,C,D) :- A-D>=0,B+C-3*D>0.
c4. l(A,B,C,D) :- A-D>=0,-B-C+3*D>0.
c5. l_body(A,B,C,D) :- A-C= -1,B-D= -2.
c6. l_body(A,B,C,D) :- A-C= -2,B-D= -1.
```

Figure 1. Example program *t4.pl* [23]

5.1 The CHC verification problem.

To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. If so we say that P is safe. Obviously any model of P assigns false to the bodies of integrity constraints. We restate this property in terms of the logic consequence relation. Let $P \models F$ mean that F is a logical consequence of P, that is, that every interpretation satisfying P also satisfies F.

LEMMA 1. *P has a model if and only if $P \not\models \mathsf{false}$.*

This lemma holds for arbitrary interpretations (only assuming that the predicate false is interpreted as false), uses only the textbook definitions of "interpretation" and "model" and does not depend on the constraint theory.

The verification problem can be formulated deductively rather than model-theoretically. We can exploit proof procedures for constraint logic programming [28] to reason about the satisfiability of a set of CHCs. Let the relation $P \vdash A$ denote that A is derivable from P using some proof procedure. If the proof procedure is sound then $P \vdash A$ implies $P \models A$, which means that $P \vdash \mathsf{false}$ is a sufficient condition for P to have no model, by Lemma 1. This corresponds to using a sound proof procedure to find or check a counterexample. On the other hand to show that P does have a model, soundness is not enough since we need to establish $P \not\models \mathsf{false}$. As we will see in Section 5.2 we approach this problem by using *approximations* to reason about the non-provability of false, applying the theory of abstract interpretation [10] to a complete proof procedure for atomic formulas (the "fixed-point semantics" for constraint logic programs [28, Section 4]). In effect, we construct by abstract interpretation a proof procedure that is *complete* (but possibly not sound) for proofs of atomic formulas. With such a procedure, $P \nvdash \mathsf{false}$ implies $P \not\models \mathsf{false}$ and thus establishes that P has a model.

5.2 Proof Techniques

Proof by over-approximation of the minimal model. It is a standard theorem of CLP that the minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P. That is, $P \models p(v_1, \ldots, v_n)$ if and only if $p(v_1, \ldots, v_n) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that $\mathsf{false} \notin M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where $\mathsf{false} \notin M'$. This technique is called proof by *over-approximation of the minimal model*.

Proof by specialisation. A specialisation of a set of CHCs P with respect to an atom A is the transformation of P to another set of CHCs P' such that $P \models A$ if and only if $P' \models A$. In our context we use specialisation to focus the verification problem on the formula to be proved. More specifically, we specialise a set of CHCs with respect to a "query" to the atom false; thus the specialised CHCs entail false if and only if the original clauses entailed false. The constraint strengthening procedure described in Section 4 is our method of specialisation.

Consider the application of the procedure in Section 4 to the clauses in Figure 1, where the *query-answer transformation* is performed with respect to the atom false. The result is shown in Figure 2. Note that the constraint in clause c4 is strengthened to false, showing that c4 is definitely not used in any derivation of false (and hence can be removed).

```
c1. false :- A>0,B=0,C=0,D=0,l(B,C,D,A).
c2. l(A,B,C,D) :- 2*A-B>=0,-A+D>0,-A+B>=0,3*A-B-C=0,
                  A-G= -1,l_body(B,C,E,F),l(G,E,F,D).
c3. l(A,B,C,D):- A-D>0,D>0,2*A-B>=0,-A+D> -1,
                 -A+B>=0,3*A-B-C=0.
c4. l(A,B,C,D):- false.
c5. l_body(A,B,C,D) :- -A+2*B>=0, 2*A-B>=0,
                       A-C= -1,B-D= -2.
c6. l_body(A,B,C,D) :- -A+2*B>=0,2*A-B>=0,A-C= -2,
                        B-D= -1.
```

Figure 2. Example program *t4.pl* [23] with strengthened constraints

5.3 Analysis of the specialised clauses

Having specialised the clauses with respect to false, it may be that the clauses P_{false} do not contain a clause with head false. In this case safety is proven, since clearly this is a sufficient condition for $P_{\mathsf{false}} \not\models \mathsf{false}$.

If this check fails we still do not know whether P has a model. In this case we can perform the convex polyhedral analysis on the clauses P_{false}. As the experiments later show, safety is often provable by checking the resulting model; if no constrained fact for false is present, then $P_{\mathsf{false}} \not\models \mathsf{false}$. If safety is not proven, there are two possibilities: the approximate model is not precise enough, but P has a model, or there is a proof of false. Refinement techniques could be used to distinguish these, but this is not the topic of this paper.

In summary, our experimental procedure for evaluating the effectiveness of constraint specialisation contains two steps. Given a set of CHCs P with integrity constraints: (1) Compute a specialisation of P with respect to false yielding P_{false}. If P_{false} contains no integrity constraints, then P is safe. (2) If P_{false} does contain integrity constraints, perform a convex polyhedra analysis of P_{false}. If the resulting approximation of the minimal model contains no constrained fact for the predicate false, then P_{false} is safe and hence P is safe. If we find a concrete derivation for false then we conclude that P is unsafe. Otherwise, P is possibly unsafe.

6. Experimental evaluation

Table 1 presents experimental results of applying our constraint specialisation to a number of Horn clause verification benchmarks taken from the repository of Horn clause verification[2] and other sources including [5, 15, 23, 25, 29]. The columns CPA and QARMC present the results of verification using convex polyhedra and QARMC respectively, whereas columns CS + CPA and CS + QARMC show the result of running constraint specialisation followed by CPA or QARMC. The symbol "-" in the table denotes irrelevant. The experiments were carried out on an Intel(R) X5355 quad-core (@ 2.66GHz) computer with 6 GB memory running Debian 5. We set 5 minutes of timeout for each experiment. The specialisation procedure is implemented in 32-bit Ciao Prolog [7] and uses the Parma Polyhedra Library [1].

[2] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/

The results show that constraint specialisation is effective in practice. We report that 109 out of 218, that is 50%, of the problems are solved by constraint specialisation alone. When used as a pre-processor for other verification tools, the results show improvements on both the number of instances solved and the solution time. Using our tool, we report approximately 47% increase in the number of instances solved and twice as much faster in average. Similarly using QARMC, we report 13% increase in the number of instances solved and 5 times faster in average.

	CPA	CS + CPA	QARMC	CS + QARMC
solved (safe/unsafe)	61 (48/13)	162 (144/18)	178 (141/37)	205 (171/34)
unknown / timeout	144/12	49/7	-/40	-/13
total time (secs)	2317	1303	13367	2613
average time (secs)	10.62	5.97	61.31	11.98
%solved	27.98	74.31	81.65	94.04

Table 1. Experiments on a set of 218 (181 safe and 37 unsafe) CHC verification problems

The (perhaps surprising) effectiveness of this relatively simple combination of specialisation and convex polyhedral analysis is underlined by noting that it can solve problems for which more complex methods have been proposed. For example, apart from the many examples from the Horn clause verification benchmarks that require refinement using CEGAR-based approaches, the technique solves the "rate-limiter" and "Boustrophedon" examples presented by Monniaux and Gonnord [40] (Section 5) (directly encoded as Horn clauses); their approach, also based on convex polyhedra, uses bounded model checking to achieve a partitioning of the approximation, while other approaches to such problems use trace-partitioning and look-ahead widening.

7. Related Work

Techniques for strengthening the constraints of logic programs go back at least to the work of Marriott *et al.* on most specific logic programs [39]. In that work the constraints were just equalities between terms and the strengthening was goal-independent. In [19] the idea was similar but it was extended to strengthen constraints while preserving the answers with respect to a goal.

The partial evaluation of (constraint) logic programs also has a long history [17, 18, 30, 32]. The aim is to specialise a program with respect to a goal, but usually unfolding is the key technique for propagating constraints. Global analysis using abstract interpretation was combined with partial evaluation algorithms to propagate constraints bottom-up as well as top-down [22, 31, 33, 35, 36].

Abstract interpretation over the domain of convex polyhedra was introduced by Cousot and Halbwachs [13] and applied to constraint logic programs by Benoy and King [4]. Abstract interpretation over convex polyhedra was incorporated in a program specialisation algorithm by Peralta and Gallagher [41].

The method of widening with thresholds for increasing the precision of widening convex polyhedra was first presented by Halwachs *et al.* [27]. We applied a technique for generating threshold constraints presented by Lakhdar-Chaouch *et al.* [34].

In summary, the basic specialisation techniques that we apply are well known, though we are not aware of previous work combining them in the same way. Our method is a specialisation with respect to a goal but does not perform partial evaluation by unfolding. The aim of our specialisation is to make constraints explicit and propagate constraints as much as possible, thereby making other tools more effective, rather than to produce a more efficient computation of a goal.

Verification of CLP programs using abstract interpretation and specialisation has been studied for some time. Our aim in this paper is not to demonstrate a verification tool but to identify a transformation that benefits CLP verification tools generally.

The idea of improving analysis by applying it to a specialised program was first expressed by Turchin [44] and it was more recently demonstrated using supercompilation [38]. The use of program transformation to verify properties of logic programs was pioneered by Pettorossi and Proietti [42] and Leuschel [37] and continues in recent work by De Angelis *et al.* [14, 15]. Transformations that preserve the minimal model (or other suitable models) of logic programs are applied systematically to make properties explicit.

Much other work on CLP verification exists, much of it based on property abstraction and refinement using interpolation, for example [2, 6, 8, 24, 26, 43]. Our specialisation technique is not directly comparable to these methods, but as we have shown in experiments with QARMC, constraint specialisation can be used as a pre-processor to such tools, increasing their effectiveness.

8. Conclusion and future Work

We introduced a method for specialising the constraints in constrained Horn clauses with respect to a goal using abstract interpretation and *query-answer transformation*. The approach propagates constraints globally, both forwards and backwards, and makes explicit constraints from the original program. This allows better analysis of the transformed program. Furthermore, our approach is independent of the abstract domain and the constraints theory underlying the clauses. Finally, we showed effectiveness of this transformation in Horn clause verification problems.

In the future, we will continue to evaluate its effectiveness in a larger set of benchmarks and as a pre-processor for other existing tools. We also would like to use the specialised version for other purposes, for instance in program debugging since more specific information may make errors in the original program apparent.

Acknowledgments

The research leading to these results has received funding from the EU 7th Framework 318337, ENTRA-Whole-Systems Energy Transparency and the Danish Natural Science Research Council grant NUSA: Numerical and Symbolic Abstractions for Software Model Checking.

References

[1] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[2] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, 2011.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5^{th} ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986.

[4] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In J. P. Gallagher, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, volume 1207 of *Springer-Verlag LNCS*, pages 204–223, August 1996.

[5] D. Beyer. Second competition on software verification - (summary of sv-comp 2013). In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *LNCS*, pages 594–609. Springer, 2013.

[6] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In F. Logozzo and M. Fähndrich, editors, *SAS*, volume 7935 of *LNCS*, pages 105–125. Springer, 2013.

[7] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog system. reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from http://www.clip.dia.fi.upm.es/.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[9] M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *J. Log. Program.*, 25(3): 249–274, 1995. .

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, 1977.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *POPL*, pages 238–252. ACM, 1977.

[12] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[13] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[14] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying programs via iterated specialization. In E. Albert and S.-C. Mu, editors, *PEPM*, pages 43–52. ACM, 2013.

[15] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verimap: A tool for verifying programs through transformations. In E. Ábrahám and K. Havelund, editors, *TACAS*, volume 8413 of *LNCS*, pages 568–574. Springer, 2014. ISBN 978-3-642-54861-1.

[16] S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[17] H. Fujita. An algorithm for partial evaluation with constraints. Technical Report TR-258, ICOT, 1987.

[18] J. P. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.

[19] J. P. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In *Proceedings of Meta90 Workshop on Meta Programming in Logic*. Katholieke Universiteit Leuven, Belgium, 1990.

[20] J. P. Gallagher and D. de Waal. Deletion of redundant unary type predicates from logic programs. In K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.

[21] J. P. Gallagher and D. de Waal. Fast and precise regular approximation of logic programs. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy*. MIT Press, 1994.

[22] J. P. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.

[23] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Failure tabled constraint logic programming by interpolation. *TPLP*, 13(4-5):593–607, 2013.

[24] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. Hsf(c): A software verifier based on Horn clauses - (competition contribution). In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.

[25] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009. ISBN 978-3-642-02657-7.

[26] A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over li+uif. In H. Yang, editor, *APLAS*, volume 7078 of *LNCS*, pages 188–203. Springer, 2011. ISBN 978-3-642-25317-1.

[27] N. Halbwachs, Y. E. Proy, and P. Raymound. Verification of linear hybrid systems by means of convex approximations. In *Proceedings of the First Symposium on Static Analysis*, volume 864 of *LNCS*, pages 223–237, September 1994.

[28] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[29] J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded symbolic execution for program verification. In S. Khurshid and K. Sen, editors, *RV*, volume 7186 of *LNCS*, pages 396–411. Springer, 2011.

[30] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.

[31] N. D. Jones. Combining abstract interpretation and partial evaluation. In P. Van Hentenryck, editor, *Symposium on Static Analysis (SAS'97)*, volume 1302 of *Springer-Verlag LNCS*, pages 396–405, 1997.

[32] H. J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *META*, volume 649 of *LNCS*, pages 49–69. Springer, 1992. ISBN 3-540-56282-6.

[33] L. Lafave and J. P. Gallagher. Partial evaluation of functional logic programs in rewriting-based languages. In N. Fuchs, editor, *Logic Program Synthesis and Transformation (LOPSTR'97)*, Springer-Verlag LNCS, 1998.

[34] L. Lakhdar-Chaouch, B. Jeannet, and A. Girault. Widening with thresholds for programs with complex control graphs. In T. Bultan and P.-A. Hsiung, editors, *ATVA 2011*, volume 6996 of *LNCS*, pages 492–502. Springer, 2011.

[35] M. Leuschel. Advanced logic program specialisation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation - Practice and Theory*, volume 1706 of *LNCS*, pages 271–292. Springer, 1999.

[36] M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst.*, 26(3):413–463, 2004. . URL `http://doi.acm.org/10.1145/982158.982159`.

[37] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *LOPSTR'99*, volume 1817 of *LNCS*, pages 62–81. Springer, 1999.

[38] A. Lisitsa and A. P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008. . URL `http://dx.doi.org/10.1142/S0129054108006066`.

[39] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proc. Fifth International Conference on Logic programming, Seattle, WA*. MIT Press, 1988.

[40] D. Monniaux and L. Gonnord. Using bounded model checking to focus fixpoint iterations. In E. Yahav, editor, *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *LNCS*, pages 369–385. Springer, 2011. ISBN 978-3-642-23701-0.

[41] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In M. Leuschel, editor, *LOPSTR*, volume 2664 of *LNCS*, pages 90–108. Springer, 2002. ISBN 3-540-40438-4.

[42] A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic*, volume 1861 of *LNCS*, pages 613–628. Springer, 2000.

[43] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In M. Hanus, editor, *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007.*, volume 4354 of *LNCS*, pages 245–259. Springer, 2007. ISBN 978-3-540-69608-7. . URL `http://dx.doi.org/10.1007/978-3-540-69611-7_16`.

[44] V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *LNCS*, pages 645–657. Springer, 1980. ISBN 3-540-10003-2. . URL `http://dx.doi.org/10.1007/3-540-10003-2_105`.

Attachment D3.3.8

Probabilistic Output Analysis by Program Manipulation

Published in the 13th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2015

Probabilistic Output Analysis by Program Manipulation

Mads Rosendahl Maja H. Kirkeby*
Computer Science, Roskilde University, Denmark
{madsr,majaht}@ruc.dk

The aim of a probabilistic output analysis is to derive a probability distribution of possible output values for a program from a probability distribution of its input. We present a method for performing static output analysis, based on program transformation techniques. It generates a probability function as a possibly uncomputable expression in an intermediate language. This program is then analyzed, transformed, and approximated. The result is a closed form expression that computes an over approximation of the output probability distribution for the program. We focus on programs where the possible input follows a known probability distribution. Tests in programs are not assumed to satisfy the Markov property of having fixed branching probabilities independently of previous history.

1 Introduction

The aim of a probabilistic output analysis (POA) is to derive a probability distribution for output values from a probability distribution for input to a program. Internal properties of a program can also be analyzed in this way by instrumenting programs with step-counters for complexity analysis [30] or energy consumption measures [23].

When analyzing energy consumption, probability distributions may provide more useful information than boundaries. Wierman et al. states that *"global energy consumption is affected by the average case, rather than the worst case"* [38]. Also in scheduling *"an accurate measurement of a tasks average-case execution time (ACET) can assist in the calculation of more appropriate deadlines"* [18]. For a subset of programs a precise average case execution time can be found using static analysis [13, 15, 33]. In some cases the POA delivers not only an accurate output average but the more descriptive accurate output distribution. In other cases the POA must over approximate the probability distribution and the expected value (average case result) will be approximated safely as a range. Another application area for POA is in temperature management, where worst-case bounds are important [34]. Because POA return distributions it can be used to calculate the probability of energy consumptions above a certain limit, and thereby indicating the risk of over-heating.

The main contribution in this paper is to present a technique for probabilistic analysis where the analysis is seen as a program-to-program translation. This means that the transformation to closed form is a source code program transformation problem and not specific to the analysis. Any necessary approximation in the analysis is also performed at the source code level. The technique also makes it possible to balance the precision of the analysis against the brevity of the result.

The method in this paper is inspired by the techniques used in automatic complexity analysis. Wegbreit's Metric system [37] laid the ground work for many later systems with an aim of deriving least, worst and average case complexity measures. Later works in this area have focused on worst case complexity [2, 24, 30] with advanced systems that can analyze realistic programs. The approach in this paper

*The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

N. Bertrand and M. Tribastone (Eds.): QAPL 2015
EPTCS 194, 2015, pp. 110–124, doi:10.4204/EPTCS.194.8

uses an approach similar to automatic complexity analysis [30] in that we derive the probability distribution without approximations but only in the last phase introduce approximations. We use a simple first-order functional language with restricted recursion for the analysis, but it can be seen as an intermediate language for analysis of programs in other languages. We transform the original program into a program that computes the probability distribution and this program can then be analyzed, transformed, and approximated. It is thus an alternative to deriving cost relations directly from the program [2, 24] or expressing costs as abstract values in a semantics for the language.

As with automatic complexity analysis the aim of probabilistic output analysis is to extract the result as a parameterized expression. The time complexity of a program should be stated as a closed form expression in the input size and for probabilistic output analysis the aim is to find the probability of output values of the program as a function in output values and input size or range. As a small example let us consider the addition `add` of two independent integer values `x` and `y` evenly distributed from 1 to n. It is a tail-recursive program where the output distribution is well-known to be a triangular shaped distribution. The program and the input probability distributions should both be expressed in the same language as they are part of the transformational approach to obtain the output probability distribution.

```
add(x,y) = if (x=0) then y else add(x-1,y+1)
px(x,n) = if (x >= 1 and x <= n) then 1/n else 0
py(y,n) = if (y >= 1 and y <= n) then 1/n else 0
```

Our probabilistic output analysis returns a function describing the probability distribution of the output:

```
padd(z,n) =
  1/(n*n)*max(min(n,z-1) - max(1,z-n) + 1,0)
```

The probability distribution is here a closed form expression parameterized in the output value (`z`) and range of input values (`n`). The analysis can also be used for more complex input distributions and programs but it will not always be able to reduce it to a precise result in closed form. If this is not possible we will approximate the distribution and thus get an over approximation of the extreme cases and a range for the expected value. If input values are not independent we can specify a joint distribution for the values.

2 Probability distributions

The analysis presented here assumes a discrete set of values for input and output. The set will be finite or countable and we will use discrete probability distributions. We consider the input to a program as a discrete random variable and the input probability distribution is then a probability measure that to an event of input having a given value assigns a value between 0 and 1. This is also often referred to as the *probability mass function* in the discrete case, and in the continuous case the equivalent is the *probability density functions*. We will use the phrase *probability distribution* to denote mappings from single values (input or output) to a probability or number between 0 and 1. Distribution will be denoted with an upper case P letter.

Definition 1 (input probability) *For a countable set X an input probability distribution is a mapping $P_x : X \to \{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$, where*

$$\sum_{x \in X} P_x(x) = 1 \, .$$

We define the output probability distribution for a program `p` in a forward manner. It is the *weight* or sum of all probabilities of input values where the program returns the desired value z as output.

Definition 2 (output probability) *Given a program,* $\mathtt{p} : X \to Z$ *and a probability distribution for the input,* P_X*, the output probability distribution,* $P_{\mathtt{p}}(z)$*, is defined as:*

$$P_{\mathtt{p}}(z) = \sum_{x \in X \wedge \mathtt{p}(x)=z} P_X(x) \; .$$

Note that Kozen also uses a similar forward definition [21], whereas Monniaux constructs the inverse and expresses the relationship in a backwards style [25].

Lemma 1 *The output probability distribution,* $P_{\mathtt{p}}(z)$*, satisfies*

$$0 \leq \sum_z P_{\mathtt{p}}(z) \leq 1 \; .$$

The program may not terminate for all input and therefore the sum may be less than one. If we expand the domain Z with an element to denote non-termination, $Z_\perp$, the total sum of the output distribution $P_{\mathtt{p}}(z)$ would be 1.

Approximations of probability distributions. The output analysis cannot necessarily derive the precise probability distribution. Various approaches to approximations of probability distributions have been proposed and can be interpreted as *imprecise probabilities* [1, 9, 11]. Dempster-Shafer structures [17, 3] and P-boxes [9, 12] can be used to capture and propagate uncertainties of probability distributions. There are several results on extending arithmetic operations to probability distributions for both known dependencies between random variables and when the dependency is unknown or only partially known [4, 6, 19, 35, 39]. Algorithms for lifting basic operations on numbers to basic operations on probability distributions can be used as abstractions in static analysis based on abstract interpretation [25]. Our approach uses the P-boxes as bounds of probability distributions. P-boxes are normally expressed in terms of the cumulative probability function but we will here use the probability mass function. We do not, however, use the various basic operations on P-boxes but apply approximations to a probability program such that it forms a P-box.

Definition 3 (over and under approximation) *For a distribution* $P_{\mathtt{p}}$ *an over and under approximation (*$\overline{P}_{\mathtt{p}}$ *and* $\underline{P}_{\mathtt{p}}$*) of the distribution satisfies the conditions:*

$$\overline{P}_{\mathtt{p}} : \forall z. P_{\mathtt{p}}(z) \leq \overline{P}_{\mathtt{p}}(z) \leq 1$$
$$\underline{P}_{\mathtt{p}} : \forall z. 0 \leq \underline{P}_{\mathtt{p}}(z) \leq P_{\mathtt{p}}(z) \; .$$

The aim of the output analysis is to derive as tight approximations $\underline{P}_{\mathtt{p}}$ and $\overline{P}_{\mathtt{p}}$ as possible.

Lemma 2 *Given the definition for over and under approximation they will have boundaries for their total weights as*

$$0 \leq \sum_z \overline{P}_{\mathtt{p}}(z) \leq \infty \qquad 0 \leq \sum_z \underline{P}_{\mathtt{p}}(z) \leq 1 \; .$$

When $\underline{P}_{\mathtt{p}} = \overline{P}_{\mathtt{p}}$ the total weight for each function will be equal to the total weight of $P_{\mathtt{p}}$, according to definition 3. For terminating programs the total weight is 1.

Expected value. Provided that the output from the program is numerical, one may be interested in the average output value of the program or *the expected value* of the output distribution. If the program does not terminate for all input it is not clear how to define the expected value so as part of the further analysis we need a guarantee that the program terminates. If the sum of the $\underline{P}_{\mathtt{p}}$ is 1 then we know that the program terminates for all possible input (*i.e.* input with probability greater than zero).

Lemma 3 *The under approximation of a probability distribution satisfies*

$$\sum_z \underline{P}_{\mathtt{p}}(z) = 1 \Rightarrow \sum_z P_{\mathtt{p}}(z) = 1 \, .$$

A similar implication does not apply to the over approximation. The expected value of the output distribution is defined as the weighted average of the distribution.

Definition 4 (expected value) *The expected value of the output distribution is defined as*

$$E_{\mathtt{p}} = \sum_z z \cdot P_{\mathtt{p}}(z) \, .$$

If we cannot analyze the program precisely, we can use the over approximation to compute an interval for the expected value. We cannot use the approximation $\overline{P}_{\mathtt{p}}$ directly as its weight is not necessarily 1. It can, however, be used to define over and under approximations to the cumulative probability distribution. These two can then be used to calculate a lower and an upper bound for the expected values.

Definition 5 (expected value interval) *For an over approximation of a probability distribution $\overline{P}_{\mathtt{p}}$ we define an over and under accumulation ($F^{\uparrow}$ and $F^{\downarrow}$) and over and under expected value ($E^{\uparrow}$ and $E^{\downarrow}$).*

$$\begin{aligned}
F^{\uparrow}(z) &= \min(\sum_{v \leq z} \overline{P}_{\mathtt{p}}(v), 1) \\
F^{\downarrow}(z) &= \max(1 - \sum_{v \geq z} \overline{P}_{\mathtt{p}}(v), 0) \\
E^{\downarrow} &= \sum_z z \cdot (F^{\uparrow}(z) - F^{\uparrow}(dec(z))) \\
E^{\uparrow} &= \sum_z z \cdot (F^{\downarrow}(z) - F^{\downarrow}(dec(z))) \\
dec(z) &= \max\{v \in Z \mid v < z\} \qquad .
\end{aligned}$$

Notice that an expected value based on an over approximation of the accumulated probability gives an under approximation of the expected value. If the output space Z is integers then the *dec* function will just subtract one from its argument.

Lemma 4 (expected value interval) *For a terminating program the expected value can be approximated by an interval from the over approximation of the probability distribution.*

$$E^{\downarrow} \leq E_{\mathtt{p}} \leq E^{\uparrow} \, .$$

Externalize resource usage. The output analysis can be used to analyze internal properties of the program provided these properties are externalized. As in automatic worst case complexity analysis [30], this may be done by instrumenting the program with step counting information. Similarly we might instrument programs with energy consumption based on low level energy models for operations [23] to be able to analyze programs for average energy consumption.

Usually an operational or denotational semantics of a simple first order functional programming language describes programs as mappings from input values to output values. The time, space or energy required to perform the computation would normally not be part of the semantics. A simple form of resource analysis is to count the number of basic operations that a computation would require. An automatic complexity analysis [30] is then based on a semantics that has been extended (or instrumented)

with step-counting information so that the meaning of a program is a mapping from input values to a tuple of the number of steps and the output value. If we write this semantics as an interpreter in the source language we can convert a program to a step-counting version of the program by partial evaluation. In this way the complexity analysis has been transformed into an output analysis of the program. The aim of the complexity analysis is then to generate an over approximation of the (first component of the) possible output as a function of the size of the possible input. If the semantics is instrumented with other types of resource information we can analyze programs with respect to these properties. Some automatic complexity analysis systems are based on translating programs into cost relations [2] or cost equations [24]. These approaches are then used as approximations of an instrumented semantics that captures the cost of computations.

The functional language we use here may be seen as a meta language for the analysis since we should extend source programs with resource information before they are analyzed. One can also use it as a meta language for analyzing programs in other languages. This can be achieved by having a step-counting interpreter for the other language written in the first order functional language. When analyzing software for embedded systems the programs are often written in simple c-like languages which should then be translated into this meta language.

The challenge of approximation. Analysis of probabilistic behavior introduces some new challenges compared to worst case analysis. It is well known that a function of expected values is not necessarily the same as the expected value of the function. There are a number of other potential pitfalls when making approximations in a probabilistic setting. One might assume that conditions in a program can be assigned a fixed probability of being true independently of previous execution paths in the program. One might also assume that variables have independent probability distributions. An unfortunate effect of using independence as an approximation is that it tends to under approximate the extreme cases. In a throw of two dice the sum of 12 has probability $1/36$ if we can assume independence. If (by some magic) they always showed the same the probability increases to $1/6$. The situation is well-known in the insurance industry and for financial risk management (valuation of derivatives) where one may want to over approximate the risk of extreme event when events are not guaranteed to be independent. One approach to handle such situations is the use of copulas [5] and comonotonicity of probability measures [10].

3 Transformation Based Analysis

Our analysis is based on a small first order functional language with primitive recursion. The first step of the analysis is to translate programs into a new language of probability distribution programs. We then use analysis and transformation techniques to transform the probability distributions into closed form. Failing that, we may over approximate the distribution as discussed in section 5.

Programs (p) are defined as a collection of functions

$$\begin{aligned} f_1(x_1,\ldots,x_n) &= e_1 \\ &\vdots \\ f_n(x_1,\ldots,x_n) &= e_n \quad . \end{aligned}$$

The first function in the program is called externally and for that function we have an input probability distribution P_x specified as a symbolic expression e_x. The language uses a base set D of values for simple expressions, and functions in a program denote mappings from tuples of values to values $D^* \to D$. The

base set of values will not be further restricted here, nor do we specify the exact set of basic operations in the language. Functions are either non-recursive or primitive recursive. The latter will have the form:

$$f(x_1,\ldots,x_n) = \textbf{if } (b(x_1,\ldots,x_n)) \textbf{ then } g(x_1,\ldots,x_n) \textbf{ else } f(e_1\ldots,e_n)$$

Non-recursive functions have right hand sides that are built from simple operations conditional expressions and function calls to non-recursive and primitive recursive functions.

3.1 Probability distribution program

The output distribution program is expressed in a language similar to the original program but extended with an extra class of functions. It contains the original functions of type $D^* \to D$ and probability functions of type $D^* \to [0,1]$. One of these functions will be the output distribution function of type $D \to [0,1]$. The language for probability distribution program uses two new language constructs: Sums over the (possibly) infinite set of all input values in D and a constraint function C. The constraint function eases the handling of boundaries and is defined as

$$C(condition) = \begin{cases} 1 & \text{if } condition = true \\ 0 & \text{otherwise} \end{cases}$$

This definition is related to the indicator function [26] or characteristic function for membership of sets. We also extend the language with a finite product construction which will be used for unfolding primitive recursion.

The first phase constructs the probability distribution program from the original program and a joint input distribution function. The input distribution is defined as a function

$$P_x(x_1,\ldots,x_n) = e_x$$

that computes the probability of each possible input value. If the arguments are independent the function can be written as a product of the probability distribution of each parameter

$$P_x(x_1,\ldots,x_n) = P_{x1}(x_1)\cdots P_{xn}(x_n)$$

The raw form of the probability distribution program is defined as follows. Given an output value, the distribution program sums the probabilities for all input value tuples that the original program maps to the output value. The probability distribution program (P_p) is defined as follows.

$$\begin{aligned} P_\mathrm{p}(z) &= \sum_{x_1}\cdots\sum_{x_n} P_x(x_1,\ldots,x_n)\cdot C(z = f_1(x_1,\ldots,x_n)) \\ P_x(x_1,\ldots,x_n) &= e_x \\ f_1(x_1,\ldots,x_n) &= e_1 \\ &\vdots \\ f_n(x_1,\ldots,x_n) &= e_n \end{aligned}$$

We view a probability distribution program as a program that can be transformed and analyzed. In the next phase function calls are unfolded and in the following phase the result is simplified using rewrite rules.

3.2 Unfolding

In this phase we unfold function calls in the program. We will introduce the central transformation rules for unfolding calls to functions in the original program based on the syntactical structure.

Function calls. Simple calls to functions can be unfolded directly. Calls to primitive recursive function can be composed but each call can be analyzed separately by constructing a joint input distribution function to the call. For such function calls we rewrite the program as follows

$$\begin{aligned}
&\sum_{x_1}\cdots\sum_{x_n} P_x(x_1,\ldots,x_n)\cdot C(z=g(e_1,\ldots,e_n))\\
&=\sum_{u_1}\cdots\sum_{u_n} P_u(u_1,\ldots,u_n)\cdot C(z=g(u_1,\ldots,u_n))\cdot\\
&\quad P_u(u_1,\ldots,u_n)=\sum_{x_1}\cdots\sum_{x_n} P_x(x_1,\ldots,x_n)\cdot C(u_1=e_1)\cdots C(u_n=e_n)\ .
\end{aligned}$$

This rule extends the program with an extra probability function P_u. We assume that the programs do not have unrestricted recursion and therefore we will only generate a bounded number of extra probability functions.

Conditional expressions. For conditional expressions we use the following rule

$$\begin{aligned}
&\sum_{x_1}\cdots\sum_{x_n} P_x(x_1,\ldots,x_n)\cdot C(z=\ \textbf{if}\ (b(x_1,\ldots,x_n))\ \textbf{then}\ g(x_1,\ldots,x_n)\ \textbf{else}\ h(x_1,\ldots,x_n))\\
&=\sum_{x_1}\cdots\sum_{x_n} P_x(x_1,\ldots,x_n)\cdot\\
&\qquad (C(b(x_1,\ldots,x_n))\cdot c(z=g(x_1,\ldots,x_n))+C(\neg b(x_1,\ldots,x_n))\cdot c(z=h(x_1,\ldots,x_n)))\ .
\end{aligned}$$

Unfolding primitive recursion. For primitive recursion we collect the probability of a given result being returned for any number of recursive calls. The condition may never evaluate to true for a certain input (non-termination), and in that situation the sum of output probabilities will be less than 1.

The recursive functions have the form

$$f(x_1,\ldots,x_n)=\ \textbf{if}\ (b(x_1,\ldots,x_n))\ \textbf{then}\ g(x_1,\ldots,x_n)\ \textbf{else}\ f(e_1\ldots,e_n)$$

and they should be analyzed for all input probability distributions we detect at calls to these functions.

The transformation for the primitive recursive form is

$$\begin{aligned}
&\sum_{x_1}\cdots\sum_{x_n} P_x(x_1,\ldots,x_n)\cdot C(z=\ \textbf{if}\ (b(x_1,\ldots,x_n))\ \textbf{then}\ g(x_1,\ldots,x_n)\ \textbf{else}\ f(e_1\ldots,e_n))\\
&=\sum_{x_1}\cdots\sum_{x_n} P_x(x_1,\ldots,x_n)\sum_{i=0}^{\infty}\prod_{j=0}^{i-1} C(\neg b(h(j,x_1,\ldots,x_n)))\cdot C(b(h(i,x_1,\ldots,x_n)))\\
&\qquad\qquad\cdot C(z=g(h(i,x_1,\ldots,x_n)))
\end{aligned}$$

where

$$h(i,x_1,\ldots,x_n)=\ \textbf{if}\ (i=0)\ \textbf{then}\ \langle x_1,\ldots,x_n\rangle\ \textbf{else}\ h(i-1,e_1,\ldots,e_n)\ .$$

In the transformed expression we introduce two variables: i that represents the number of recursive calls, and j that represents all previous recursions for the i under investigation (when i is 0 the term $\prod_{j=0}^{i-1} C(\neg b(h(j,x_1,\ldots,x_n)))$ evaluates to 1). The new function $h(i,x_1,\ldots,x_n)$ describes the evaluation of the expressions $\langle e_1,\ldots,e_n\rangle$, i times. Only when the ith condition is *true* and all previous conditions are *false* can the expression evaluate to a probability above 0.

3.3 Symbolic summation

In the previous phase we unfolded calls to functions in the original program. The aim of this phase is to use algebraic transformation techniques to remove summations. The methods we use are similar to the transformations used in worst case execution time system for solving recurrence equations [31, 24] or symbolic summation techniques in loop bound computations [20]. Some of the central transformation rules we apply in this phase are listed below. In the following transformations the expressions e_1 and e_2 are assumed not to contain the summation variable x.

$$\sum_x C(x = e_1) \cdot f(x) = f(e_1)$$
$$\sum_x C(e_1 \leq x \leq e_2) = (e_2 - e_1 + 1) \cdot C(e_1 \leq e_2)$$
$$\sum_x x \cdot C(e_1 \leq x \leq e_2) = \left(\frac{e_2 \cdot (e_2+1)}{2} - \frac{e_1 \cdot (e_1-1)}{2} \right) \cdot C(e_1 \leq e_2) \ .$$

One could also use computer algebra systems in the reduction process but some of the rules are quite specific to the way we handle the boundaries of summations with the special constraint function. There are a number of rules to combine products of constraint functions and to split intervals into separate expressions.

$$C(e_1 \leq x \leq e_2) \cdot C(e_3 \leq x \leq e_4) = C(\max(e_1, e_3) \leq x \leq \min(e_2, e_4))$$

$$C(\max(e_1, e_2) \leq e_3) = C(e_1 > e_2) \cdot C(e_1 \leq e_3) + C(e_1 \leq e_2) \cdot C(e_2 \leq e_3) \ .$$

There are similar rules for removing the minimum function and for isolating variables in constraints.

There are also rules for symbolic summation of certain infinite summations. If a is an expression where $0 < a < 1$ then we can simplify the expression as follows:

$$\sum_x C(x \geq 0) \cdot a^x = \frac{1}{(1-a)}$$
$$\sum_x C(x \geq 0) \cdot x \cdot a^x = \frac{1}{(1-a)^2} - \frac{1}{(1-a)} \ .$$

This rule is useful when some of the input to the program follows a geometric distribution.

$$P_x(x, n) = C(x \geq 0) \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^x \ .$$

Finally, there are rules for removing finite products and performing standard algebraic simplification transformations.

Max example. As a small example, let us look at the simple non-recursive program `max`, which given two values return the largest. It is chosen because of its simplicity while still producing a non-uniform output distribution. The program is defined as

```
max(x,y) = if (x>y) then x else y
```

The input values are independent and they follow a uniform distribution from 1 to n:

$$P_x(x) = \frac{1}{n} \cdot C(1 \leq x \leq n) \quad \text{and} \quad P_y(y) = \frac{1}{n} \cdot C(1 \leq y \leq n) \ .$$

The output probability program is constructed and simplified using transformation rules for conditional expressions and symbolic summation.

$$\begin{aligned}
P_{\texttt{max}}(z) &= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z = \texttt{if } (x > y) \texttt{ then } x \texttt{ else } y) \\
&= \frac{1}{n^2} \cdot \Big(\quad \sum_y \big(C(1 \le z \le n) \cdot C(1 \le y \le n) \cdot C(y \le (z-1)) \big) \\
&\qquad\qquad + \sum_x \big(C(1 \le x \le n) \cdot C(1 \le z \le n) \cdot C(x \le z) \big) \Big) \\
&= \frac{1}{n^2} \cdot (2z-1) \cdot C(1 \le z \le n) \; .
\end{aligned}$$

The output probability program takes the output value (z) as input and uses the range of input variables (n) as an implicit parameter.

Add example. The recursive addition function was used as an example in the introduction. We shall see how the original program is inserted into the probability formula, expanded and reduced to a closed form function expressing the probability distribution for the output. Recall the program:

```
add(x,y) = if (x=0) then y else add(x-1,y+1)
```

and that we assume independence between the input variables and that both input variables `x` and `y` have a uniform distribution from 1 to a number n. The output probability program is constructed using the rule for primitive recursion.

$$P_{\texttt{add}}(z) = \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \sum_{i=0}^{i-1} \prod_{j=0}^{i-1} C(\neg b(h(j,x,y))) \cdot C(b(h(i,x,y))) \cdot C(z = g(h(i,x,y))$$

where

$$\begin{aligned}
b(x,y) &= x = 0 \\
g(x,y) &= y \\
h(i,x,y) &= \textbf{ if } (i=0) \textbf{ then } \langle x,y \rangle \textbf{ else } h(i-1,x-1,y+1) \\
&= \langle x-i, y+i \rangle
\end{aligned}$$

The simplification process proceeds as follows.

$$\begin{aligned}
&P_{\texttt{add}}(z) = \\
&\quad \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \sum_{i=0}^{i-1} \prod_{j=0}^{i-1} C(\neg(x-j=0)) \cdot C(x-i=0) \cdot C(z = y+i) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \sum_{i=0} C(x=i) \cdot C(z = y+i) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z = y+x) \\
&= \sum_y \frac{1}{n} \cdot C(z-n \le y \le z-1) \cdot \frac{1}{n} \cdot C(1 \le y \le n) \big) \\
&= \frac{1}{n^2} \cdot \max(\min(n, z-1) - \max(1, z-n) + 1, 0) \\
&= \frac{1}{n^2} \cdot \big(C(n < z \le 2n) \cdot (2n - z + 1) \quad + \quad C(1 \le z \le n) \cdot (z-1) \big) \; .
\end{aligned}$$

The probability program computes a triangular shaped probability distribution with a maximum at $n+1$.

3.4 Expected value

When programs produce numeric output values we can use the probability output distribution to compute the expected value or an expected value interval for the output values. The expected value is defined as

$$E_{\mathrm{p}} = \sum_x x \cdot P_{\mathrm{p}}(x) \quad .$$

For the `add` program this gives

$$E_{\mathtt{add}} = \sum_{z=1}^{n} z \cdot \frac{1}{n^2} \cdot (z-1) + \sum_{z=n+1}^{2n} z \cdot \frac{1}{n^2} \cdot (2n-z+1)$$

which, of course, can be reduced further.

4 Composite Types

In the approach we have presented the base domain is a countable set and not necessarily just numbers. We only need to be able to define a probability distribution for values in the domain.

For lists of length $k > 0$ where elements are uniformly distributed over the interval 1 to n we can use the probability function

$$P_L(L) = \frac{1}{n^k} \cdot C(\mathit{length}(L) = k \wedge \forall j : 0 \leq j \leq k-1 \wedge 1 \leq \mathit{hd}(\mathit{tl}^j(L)) \leq n) \ .$$

We assign the probability $1/n^k$ to any list of length k where all elements are in the interval from 1 to n.

If we consider the member function for non-empty lists, it can be written as

```
member(X,L) = if (tl(L)=[] || hd(L)=X) then hd(L)=X
              else member(X,tl(L))
```

The function will follow the pattern of primitive recursion as described earlier and the output probability distribution for the member function is then

$$P_{\mathtt{member}}(z) = \sum_X \sum_L P_X(X) \cdot P_L(L) \cdot C(z = \mathtt{member}(X,L)) \ .$$

We can then use the unfolding rules to simplify the expression further.

The lists were here assumed to contain possibly repeating elements in the list. We could also use a different probability measure to restrict lists to non-repeating lists of values. The example is also analyzed by Wegbreit [37] where he derives the probability as $1-(1-(1/n)^k)$. It is analyzed under the assumption of non-repeating lists but is actually the correct result for repeating lists.

His technique is valid for programs where one can safely assume the Markov property (that probabilities of conditions are fixed). Wegbreit observes that this is not always true even in very simple cases, e.g. in nested conditionals where the outcome of the first condition influences the probability of the outcome for the subsequent condition.

It should be noted that conditions existing inside a recursive structure often invokes dependencies between variables. This occur when there is a *gain of knowledge*: For instance in the `union` function for two repeating lists; if the head of the list is not in the second list, the likelihood of the next element not being in the second list increases slightly.

5 Approximation Techniques

The probability distribution program expresses the probability distribution for output values. Our aim is to transform it into a closed form but this may not always be possible. Failing that, we can instead use approximation techniques to obtain an upper bound for the probability distribution. We have referred to this as the over approximation of the probability distribution, $\overline{P}_{\mathrm{p}}$.

Cumulative distribution functions. Cumulative probabilities will in some cases be more useful and expressive than probability distributions: Cumulative probabilities can be used in both the discrete and the continuous case, and in some cases approximations can be described more precisely using accumulated probabilities than with ordinary distributions. It tends, however, to be more complex to reduce to closed form and thus may require coarser approximations. The bounding of a cumulative distribution was introduced by Ferson [12] as a P-box and can be used to describe imprecise probability distributions.

Definition 6 (cumulative distribution) *Given a program output probability distribution, $P_{\mathrm{p}}(z)$, the cumulative program output probability distribution, $F_{\mathrm{p}}(z)$, is defined as*

$$F_{\mathrm{p}}(z) = \sum_{w \leq z} P_{\mathrm{p}}(w) \; .$$

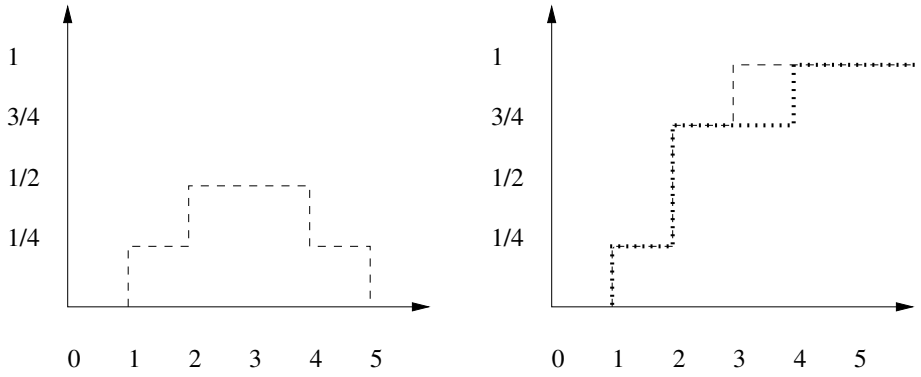
Definition 7 (over and under approximation) *Given a cumulative output probability of a program* p*, F_{p}, the over approximation, $\overline{F}_{\mathrm{p}}$, and the under approximation, $\underline{F}_{\mathrm{p}}$, are defined as*

$$\overline{F}_{\mathrm{p}} : \forall z. F_{\mathrm{p}}(z) \leq \overline{F}_{\mathrm{p}}(z) \qquad\qquad \underline{F}_{\mathrm{p}} : \forall z. \underline{F}_{\mathrm{p}}(z) \leq F_{\mathrm{p}}(z)$$

where for each approximation it must always hold that

$$\forall z. 0 \leq \overline{F}_{\mathrm{p}}(z) \leq 1 \qquad \forall z. 0 \leq \underline{F}_{\mathrm{p}}(z) \leq 1 \; .$$

When we can deduce that a program may return one of two values, but not which one, then the cumulative probability can be used for a more precise description. Such a program could be `if x = 1 then 1 else (if x = 4 then 4 else (if (unanalyzable) then 2 else 3))` and $1 \leq \mathrm{x} \leq 4$ with the probability distribution $P_x(x) = 1/4 \cdot C(1 \leq x \leq 4)$.



Here, the over approximating distribution function will assign $1/2$ for both 2 and 3. In contrast, the over approximating cumulative distribution can express that if the program-output is not 2 it must be 3.

The distributions $\underline{P}_{\mathrm{p}}$ and $\overline{P}_{\mathrm{p}}$ can be used to derive $\underline{F}_{\mathrm{p}}$ and $\overline{F}_{\mathrm{p}}$. However, these may not be as precise as cumulative distributions derived directly.

When approximating cumulative probability distributions the techniques are different from probability mass functions. Instead one may use copulas [5] to over and under approximate dependencies between subexpressions. Copulas are based on the theory of comonotonicity [10] for distributions that may depend on a common (possibly unknown) random variable.

6 Related Work

Probabilistic analysis is related to the analysis of probabilistic programs. Probabilistic analysis is analysis of programs with a normal semantics where the input variables are interpreted over probability distributions. Analysis of probabilistic programs analyzes programs with probabilistic semantics where the values of the input variables are unknown (e.g. flow analysis [28]).

In probabilistic analysis it is important to determine how variables depend on each other, but already in 1976 Denning proposed a flow analysis for revealing whether variables depend on each other [8]. This was presented in the field of secure flow analysis. Denning introduced a lattice-based analysis where she, given the name of a variable, that should be kept secret, deducted which other variables those should be kept secret in order to avoid leaking information. In 1996, Denning's method was refined by Volpano et al. into a type system and for the first time, it was proven sound [36].

Reasoning about probabilistic semantics is a closely related area to probabilistic analysis, as they both work with nested probabilistic influence. The probabilistic analysis work on standard semantic and analyze it using input probability distributions, where a probabilistic semantics allow for random assignments and probabilistic choices [21] and is normally analyzed using an expanded classical analysis or verification method [7].

Probabilistic model checking is an automated technique for formally verifying quantitative properties for systems with probabilistic behaviors. It is mainly focused on Markov decision processes, which can model both stochastic and non-deterministic behavior [14, 22]. It differs from probabilistic analysis as it assumes the Markov property.

In 2000, Monniaux applied abstract interpretation to programs with probabilistic semantics and gained safe bounds for worst case analysis [25]. Pierro et al. introduce a linear mapping structure, a Moore-Penrose pseudo-inverse, instead of a Galois connection. They use the linear structures to compare 'closeness' of approximations as an expression using the average approximation error. Pierro et al. further explores using probabilistic abstract interpretation to calculate the average case analysis [27]. In 2012, Cousot and Monerau gave a general probabilistic abstraction framework [7] and stated that Pierro et al.'s method and many other abstraction methods can be expressed in this new framework.

When analyzing probabilities the main challenge is to maintain the dependencies throughout the program. Schellekens defines this as *Randomness preservation* [33] (or random bag preservation) which in his (and Gao's [15]) case enables tracking of certain data structures and their distributions. They use special data structures as they find these suitable to derive the average number of basic operations. In another approach [37, 29], tests in programs has been assumed to be independent of previous history, also known as the Markov property (the probability of true is fixed). As Wegbreit remarked, this is true only for some programs (e.g. linear search for repeating lists) and others, this is not the case (linear search for non-repeating lists). The Markov property is the foundation in Markov decision processes which is used in probabilistic model-checking [14]. Cousot et al. presents a probabilistic abstraction framework where they divide the program semantics into probabilistic behavior and (non-)deterministic behavior. They handle loops by using a probability function describing the probability of entering the loop in the ith iteration. Monniaux propose another approach for abstracting probabilistic semantics [25]; he first

lifts a normal semantics to a probabilistic semantics where random generators are allowed and then uses an abstraction to reach a closed form. Monniaux's semantic approach uses a backward probabilistic semantics operating on measurable functions. This is closely related to the forward probabilistic semantics proposed earlier by Kozen [21].

An alterntive approach to probabilistic analysis is based on symbolic execution of programs with symbolic values [16]. Such techniques can also be used on programs with infinitely many execution paths by limiting the analysis to a finite set of paths at the expense of tightness of probability intervals [32].

7 Conclusion

Probabilistic analysis of programs has a renewed interest for analyzing programs for energy consumptions. Numerous embedded systems and mobile applications are limited by restricted battery life on the hardware. In this paper we present a technique for extracting a probability distribution for programs from symbolic distributions of the input. It is a static transformation based method which can analyze a first order language with primitive recursion. From the original program and an input probability distribution we generate an output probability distribution program, and transform this program into closed form. We present the essential transformation rules for unfolding calls to the original program and removing infinite sums. The transformed program is then analyzed and approximated using program transformation techniques. The core elements of the analysis have been implemented in a prototype system with the aim of using it to improve energy efficiency of systems. The central challenges of approximating in a probabilistic setting are discussed and we describe some advantages of using cumulative distributions along with copulas to achieve a tighter approximation.

Acknowledgements. This work has benefitted from numerous discussions with Pedro López-García, Alejandro Serrano Mena and other colleagues in Madrid, Bristol and Roskilde.

References

[1] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault & S. Putot (2014): *Static analysis of programs with imprecise probabilistic inputs.* In: *In Verified Software: Theories, Tools, Experiments*, pp. 22–47, doi:10.1007/978-3-642-54108-7_2.

[2] Elvira Albert, Puri Arenas, Samir Genaim & Germán Puebla (2009): *Cost Relation Systems: A Language-Independent Target Language for Cost Analysis.* *Electr. Notes Theor. Comput. Sci.* 248, pp. 31–46, doi:10.1016/j.entcs.2009.07.057.

[3] Mathias Bauer (1996): *Approximations for Decision Making in the Dempster-Shafer Theory of Evidence.* In: *UAI*, Morgan Kaufmann, pp. 73–80. Available at `http://arxiv.org/abs/1302.3557`.

[4] Daniel Berleant & Hang Cheng (1998): *A Software Tool for Automatically Verified Operations on Intervals and Probability Distributions.* *Reliable Computing* 4(1), pp. 71–82, doi:10.1023/A:1009954817673.

[5] Guillem Bernat, Alan Burns & Martin Newby (2005): *Probabilistic timing analysis: An approach using copulas.* *J. Embedded Computing* 1(2), pp. 179–194.

[6] Olivier Bouissou, Eric Goubault, Jean Goubault-Larrecq & Sylvie Putot (2012): *A generalization of p-boxes to affine arithmetic.* *Computing* 94(2-4), pp. 189–201, doi:10.1007/s00607-011-0182-8.

[7] Patrick Cousot & Michael Monerau (2012): *Probabilistic Abstract Interpretation.* In: *ESOP*, *LNCS* 7211, pp. 169–193, doi:10.1007/978-3-642-28869-2_9.

[8] Dorothy E. Denning (1976): *A Lattice Model of Secure Information Flow.* Commun. ACM 19(5), pp. 236–243, doi:10.1145/360051.360056.

[9] S. Destercke & D. Dubois (2009): *The role of generalised p-boxes in imprecise probability models*. In: 6th International Symposium on Imprecise Probability: Theories and Applications, pp. 179–188.

[10] Jan Dhaene, Michel Denuit, Marc J Goovaerts, R Kaas & David Vyncke (2002): *The Concept of Comonotonicity in Actuarial Science and Finance: Theory*. Insurance, mathematics & economics 31(2), pp. 133–161, doi:10.1016/S0167-6687(02)00135-X.

[11] Scott Ferson (2014): *Model uncertainty in risk analysis*. Tech. report, Centre de Recherches de Royallieu, Universite de Technologie de Compiegne.

[12] Scott Ferson, Vladik Kreinovich, Lev Ginzburg, Davis S. Myers, & Kari Sentz (2002): *Constructing Probability Boxes and Dempster-Shafer Structures*. SAND2002-4015, Sandia National Laboratories.

[13] Philippe Flajolet, Bruno Salvy & Paul Zimmermann (1991): *Automatic Average-Case Analysis of Algorithm.* Theor. Comput. Sci. 79(1), pp. 37–109, doi:10.1016/0304-3975(91)90145-R.

[14] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman & David Parker (2011): *Automated Verification Techniques for Probabilistic Systems.* In: SFM, LNCS 6659, pp. 53–113, doi:10.1007/978-3-642-21455-4_3.

[15] Ang Gao (2013): *Modular average case analysis: Language implementation and extension*. Ph.d. thesis, University College Cork.

[16] Jaco Geldenhuys, Matthew B Dwyer & Willem Visser (2012): *Probabilistic symbolic execution*. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 166–176, doi:10.1145/2338965.2336773.

[17] Jean Gordon & Edward H. Shortliffe (1984): *The Dempster-Shafer Theory of Evidence*. In: Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project, p. 21 pp.

[18] Xi Guo, Menouer Boubekeur, J. McEnery & David Hickey (2007): *ACET based scheduling of soft real-time systems: An approach to optimise resource budgeting*. International Journal of Computers and Communications 1(1), pp. 82–86.

[19] Rakowsky Uwe Kay (2007): *Fundamentals of the Dempster-Shafer theory and its applications to system safety and reliability modelling*. International Journal of Reliability, Quality and Safety Engineering 14(06), pp. 579–601, doi:10.1142/S0218539307002817.

[20] Jens Knoop, Laura Kovács & Jakob Zwirchmayr (2011): *Symbolic Loop Bound Computation for WCET Analysis.* In: Ershov Memorial Conference, LNCS 7162, pp. 227–242, doi:10.1007/978-3-642-29709-0_20.

[21] Dexter Kozen (1981): *Semantics of Probabilistic Programs.* J. Comput. Syst. Sci. 22(3), pp. 328–350, doi:10.1016/0022-0000(81)90036-2.

[22] M. Kwiatkowska, G. Norman & D. Parker (September 2010): *Advances and challenges of probabilistic model checking.* In: 48th Annual Allerton Conference on Communication, Control, and Computing, pp. 1691–1698, doi:10.1109/ALLERTON.2010.5707120.

[23] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo & K. Eder (2013): *Energy Consumption Analysis of Programs based on XMOS ISA Level Models*. In: 23rd InternatioFnal Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR, LNCS 8901, pp. 72–90, doi:10.1007/978-3-319-14125-1_5.

[24] Pedro López-García, Luthfi Darmawan & Francisco Bueno (2010): *A Framework for Verification and Debugging of Resource Usage Properties: Resource Usage Verification.* In: ICLP (Technical Communications), LIPIcs 7, pp. 104–113.

[25] David Monniaux (2000): *Abstract Interpretation of Probabilistic Semantics.* In: SAS, LNCS 1824, pp. 322–339, doi:10.1007/978-3-540-45099-3_17.

[26] Carroll Morgan, Annabelle McIver & Karen Seidel (1996): *Probabilistic Predicate Transformers.* ACM Trans. Program. Lang. Syst. 18(3), pp. 325–353, doi:10.1145/229542.229547.

[27] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2006): *Abstract Interpretation for Worst and Average Case Analysis.* In: *Program Analysis and Compilation*, *LNCS* 4444, pp. 160–174, doi:10.1007/978-3-540-71322-7_8.

[28] Alessandra Di Pierro, Herbert Wiklicky, Gabriele Puppis & Tiziano Villa (2013): *Probabilistic data flow analysis: a linear equational approach*. In: *Proc. Fourth Int. Symp. on Games, Automata, Logics and Formal Verification, GandALF*, 119, pp. 150–165, doi:10.4204/EPTCS.119.14.

[29] Hector Soza Pollman, Manuel Carro & Pedro Lopez Garcia (2009): *Probabilistic Cost Analysis of Logic Programs: A First Case Study*. *INGENIARE - Revista Chilena de Ingeniera* 17(2), pp. 195–204.

[30] Mads Rosendahl (1989): *Automatic Complexity Analysis.* In: *FPCA*, pp. 144–156, doi:10.1145/99370.99381.

[31] Mads Rosendahl (2002): *Simple Driving Techniques.* In: *The Essence of Computation*, *LNCS* 2566, pp. 404–419, doi:10.1007/3-540-36377-7_18.

[32] S. Sankaranarayanan, A. Chakarov & S. Gulwani (June 2013): *Static analysis for probabilistic programs: inferring whole program properties from finitely many paths*. In: *PLDI*, ACM., pp. 447–458, doi:10.1145/2462156.2462179.

[33] Michel P. Schellekens (2008): *A modular calculus for the average cost of data structuring.* Springer, doi:10.1007/978-0-387-73384-5.

[34] Lars Schor, Iuliana Bacivarov, Hoeseok Yang & Lothar Thiele (2012): *Worst-Case Temperature Guarantees for Real-Time Applications on Multi-core Systems.* In: *IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 87–96, doi:10.1007/s10836-013-5397-5.

[35] A. Uwimbabazi (2013): *Extended probabilistic symbolic execution*. Master's thesis, University of Stellenbosch.

[36] Dennis M. Volpano, Cynthia E. Irvine & Geoffrey Smith (1996): *A Sound Type System for Secure Flow Analysis.* *Journal of Computer Security* 4(2/3), pp. 167–188, doi:10.3233/JCS-1996-42-304.

[37] Ben Wegbreit (1975): *Mechanical Program Analysis.* *Commun. ACM* 18(9), pp. 528–539, doi:10.1145/361002.361016.

[38] Adam Wierman, Lachlan L. H. Andrew & Ao Tang (2008): *Stochastic Analysis of Power-Aware Scheduling*. In: *Proceedings of Allerton Conference on Communication, Control and Computing*, Urbana-Champaign, IL, pp. 1278–1283, doi:10.1109/ALLERTON.2008.4797707.

[39] Nic Wilson (2000): *Algorithms for Dempster-Shafer Theory*. In: *Handbook of defeasible reasoning and uncertainty management systems*, Springer Netherlands, pp. 421–475, doi:10.1007/978-94-017-1737-3_10.

Attachment D3.3.9

Decomposition by tree dimension in Horn clause verification

Published at the 3rd International Workshop on Verification and Program Transformation (VPT'2015)

Decomposition by tree dimension in Horn clause verification *

Bishoksan Kafle
Roskilde University, Denmark
kafle@ruc.dk

John P. Gallagher
Roskilde University, Denmark
IMDEA Software Institute, Spain
jpg@ruc.dk

Pierre Ganty
IMDEA Software Institute, Spain
pierre.ganty@imdea.org

In this paper we investigate the use of the concept of *tree dimension* in Horn clause analysis and verification. The dimension of a tree is a measure of its non-linearity – for example a list of any length has dimension zero while a complete binary tree has dimension equal to its height. We apply this concept to trees corresponding to Horn clause derivations. A given set of Horn clauses P can be transformed into a new set of clauses $P^{\leq k}$, whose derivation trees are the subset of P's derivation trees with dimension at most k. Similarly, a set of clauses $P^{>k}$ can be obtained from P whose derivation trees have dimension at least $k+1$. In order to prove some property of all derivations of P, we systematically apply these transformations, for various values of k, to decompose the proof into separate proofs for $P^{\leq k}$ and $P^{>k}$ (which could be executed in parallel). We show some preliminary results indicating that decomposition by tree dimension is a potentially useful proof technique. We also investigate the use of existing automatic proof tools to prove some interesting properties about dimension(s) of feasible derivation trees of a given program.

Keywords: Tree dimension, proof decomposition, program transformation, Horn clauses.

1 Introduction

In this paper, we study the role of *tree dimension* in Horn clause analysis and verification. The dimension of a tree is a measure of its non-linearity – for example a list of any length has dimension zero while a complete binary tree has dimension equal to its height. We apply this concept to trees corresponding to Horn clause derivations. A given set of Horn clauses P can be transformed into a new set of clauses $P^{\leq k}$ (whose derivation trees are the subset of P's derivation trees with dimension at most k) and $P^{>k}$ (whose derivation trees have dimension at least $k+1$). Each such set of clauses represents an under-approximation of the original set of clauses and the proof for the original clauses can be constructed from their individual proofs. In order to prove some property of all derivations of P, we systematically apply these transformations, for various values of k, to decompose the proof into separate proofs for $P^{\leq k}$ and $P^{>k}$ (which could be executed in parallel).

We prove each such set of clauses using abstract interpretation [4] over the domain of convex polyhedra [5] as described in [18]. Finally, the preliminary results in a set of Horn clause verification benchmarks show that this is a useful program transformation. This decomposition can also be viewed as refinement where one eliminates possibly infinite sets of program traces. As a result of this, the proof for the remaining part becomes simpler. To motivate readers, we present an example set of constrained Horn clauses (CHCs) P in Figure 1 which defines the Fibonacci function. This is an interesting problem whose dimension depends on the input number and its computations are trees rather than linear sequences. The main contributions of this paper are the following.

*The research leading to these results has been supported by the EU FP7 project 318337, *ENTRA - Whole-Systems Energy Transparency*, the EU FP7 project 611004, *coordination and support action ICT-Energy* and Danish Research Council grant FNU-10-084290.

A. Lisitsa, A.P. Nemytykh, A. Pettorossi (Eds.): 3rd International Workshop
on Verification and Program Transformation (VPT 2015)
EPTCS 199, 2015, pp. 1–14, doi:10.4204/EPTCS.199.1

```
c1. fib(A, A):- A>=0,  A=<1.
c2. fib(A, B) :- A > 1, A2 = A - 2, fib(A2, B2),
           A1 = A - 1, fib(A1, B1), B = B1 + B2.
c3. false:- A>5, fib(A,B), B<A.
```

Figure 1: Example CHCs Fib: it defines a Fibonacci function.

1. We describe how to generate at-most k-dimension program and at-least k-dimension program from a given program using the notion of tree dimension (Section 2);
2. We give a verification algorithm for Horn clauses program based on its proof decomposition (Section 3);
3. We give an alternative way of generating the at-least k-dimension program using the theory of finite tree automata (Section 4);
4. We demonstrate the feasibility of our approach in practice applying it to non-linear Horn clause verification problems (Section 7);
5. We instrument a program with its dimension and use existing automatic verification tools to prove some interesting properties about its dimension (Section 5).

2 Preliminaries

A constrained Horn clause is a first order formula of the form $p(X) \leftarrow \mathscr{C}, p_1(X_1), \ldots, p_k(X_k)$ $(k \geq 0)$ (using Constraint Logic Programming (CLP) syntax), where $\mathscr{C}$ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $\mathscr{C}, p_1(X_1), \ldots, p_k(X_k)$ is the body. A clause is called non-linear if it contains more than one atom in the body ($k > 1$), otherwise it is called linear. A set of Horn clauses is sometimes called a program.

A labeled tree $c(t_1, \ldots, t_k)$ is a tree with its nodes labeled, where c is a node label and $t_1, \ldots, t_k$ are labeled trees rooted at the children of the node and leaf nodes are denoted by c.

Definition 1 (Tree dimension (adapted from [7])) *Given a labeled tree $t = c(t_1, \ldots, t_k)$, the tree dimension of t represented as $dim(t)$ is defined as follows:*

$$dim(t) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{i \in [1..k]} dim(t_i) & \text{if there is a unique maximum} \\ \max_{i \in [1..k]} dim(t_i) + 1 & \text{otherwise} \end{cases}$$

Figure 2 (a) shows a derivation tree t for Fibonacci number 3 and Figure 2 (b) shows its tree dimension. It can be seen that $dim(t) = 1$. This number is a measure of its non-linearity, the smaller the number the closer the tree is to a list. Since it is not a perfect binary tree, the height of t (3) is greater than its dimension.

Given a set of CHCs P and $k \in \mathbb{N}$, we split each predicate H occurring in P into the predicates $H^{\leq d}$ and $H^{=d}$ where $d \in \{0, 1, \ldots, k\}$. Here $H^{\leq d}$ and $H^{=d}$ generate trees of dimension at most d and exactly d respectively.

Definition 2 (At-most-k-dimension program $P^{\leq k}$) *It consists of the following clauses (adapted from [20]):*

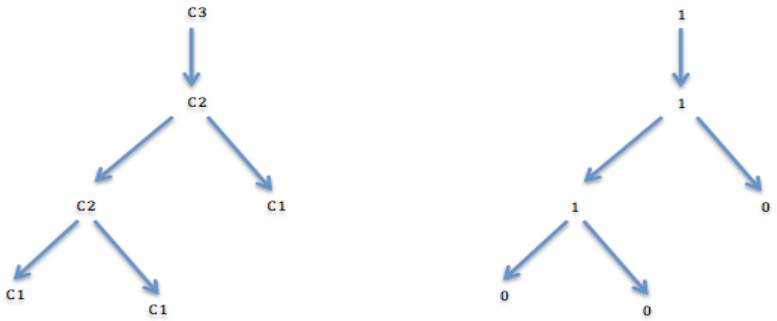


Figure 2: (a) derivation tree of Fibonacci 3 and (b) its tree dimension.

```
%linear clauses
1. fib(0)(A,A) :- A>=0, A=<1.
2. false(0) :- A>5, B<A, fib(0)(A,B).
%epsilon-clauses
3. false[0] :- false(0).
4. fib[0](A,B) :- fib(0)(A,B).
```

Figure 3: Fib$^{\leq 0}$: at-most 0-dimension program of Fib.

1. *Linear clauses:*
   If $H \leftarrow \mathscr{C} \in P$ *, then* $H^{=0} \leftarrow \mathscr{C} \in P^{\leq k}$.
   If $H \leftarrow \mathscr{C}, B_1 \in P$ *then* $H^{=d} \leftarrow \mathscr{C}, B_1^{=d} \in P^{\leq k}$ *for* $0 \leq d \leq k$.
2. *Non-linear clauses:*
   If $H \leftarrow \mathscr{C}, B_1, B_2, \ldots, B_r \in P$ *with* $r > 1$*:*
   - *For* $1 \leq d \leq k$*, and* $1 \leq j \leq r$*:*
     Set $Z_j = B_j^{=d}$ *and* $Z_i = B_i^{\leq d-1}$ *for* $1 \leq i \leq r \wedge i \neq j$*. Then:* $H^{=d} \leftarrow \mathscr{C}, Z_1, \ldots, Z_r \in P^{\leq k}$.
   - *For* $1 \leq d \leq k$*, and* $J \subseteq \{1, \ldots, r\}$ *with* $|J| = 2$*:*
     Set $Z_i = B_i^{=d-1}$ *if* $i \in J$ *and* $Z_i = B_i^{\leq d-2}$ *if* $i \in \{1, \ldots, r\} \setminus J$*. If all* Z_i *are defined, i.e.,* $d \geq 2$ *if* $r > 2$*, then:* $H^{=d} \leftarrow \mathscr{C}, Z_1, \ldots, Z_r \in P^{\leq k}$.
3. ε*-clauses:*
   $H^{\leq d} \leftarrow H^{=e} \in P^{\leq k}$ *for* $0 \leq d \leq k$ *, and every* $0 \leq e \leq d$.

The at-most 0-dimension program of Fib in Figure 1 is depicted in Figure 3 (where the numbers on the first column are not clause identifiers and are there for future reference). In textual form we represent a predicate $p^{\leq k}$ by `p[k]` and a predicate $p^{=k}$ by `p(k)`. Since some programs have derivation trees of unbounded dimension, trying to verify a property for its increasing dimension separately is not a practical strategy. To deal with this, we need some construction which characterises derivation trees of at-least k-dimension. Next we define this construction (*at-least k-dimension program*). For this, we split each predicate H occurring in P into the predicates $H^{>d}$ and $H^{\geq 0}$ where $d \in \{0, 1 \ldots, k\}$. Here $H^{>d}$ generates trees of dimension at-least $d+1$ and $H^{\geq 0}$ generates trees of any dimension.

Definition 3 (At-least k+1-dimension program $P^{>k}$) *In addition to the linear, non-linear and* ε*-clauses from Definition 2 (with each predicate* $H^{\leq k}$ *and* $H^{=k}$ *from* $P^{\leq k}$ *renamed to* $H^{>k}$ *and* $H^{\geq 0}$ *respectively), the at-least k+1-dimension program* $P^{>k}$ *consists of the following clauses:*

```
%linear clauses
fib<0>(A,A) :- A>=0, A=<1.
false<0> :- A>5, B<A,fib<0>(A,B).
%epsilon-clauses
false{0} :- false<0>.
fib{0}(A,B) :- fib<0>(A,B).
%link clauses
false<0> :- false.
fib<0>(A,B) :- fib(A,B).
%original clauses (all clauses)
fib(A, A):- A>=0,  A=<1.
false:- A>5, fib(A,B), B<A.
fib(A, B) :- A > 1, A2 = A - 2, fib(A2, B2),
           A1 = A - 1, fib(A1, B1), B = B1 + B2.
```

Figure 4: Fib$^{>0}$: at-least 1-dimension program of Fib.

1. *Link-clauses:*
   For each $H \leftarrow B \in P$ there is a clause $H^{\geq 0} \leftarrow H \in P^{>k}$.
2. *Original clauses:*
   All clauses in P are also in $P^{>k}$.

The at-least 1-dimension program of Fib in Figure 1 is depicted in Figure 4. In textual form we represent a predicate $p^{>k}$ by p{k} and a predicate $p^{\geq 0}$ by p<k>.

3 Procedure for verification

Given a set of CHCs P (including clauses with false head, also known as *integrity constraints*), the CHC verification problem is to check whether there exists a model of P. This is equivalent to checking whether there is any feasible derivation tree for false; if there is such a derivation then there is no model. We say P is safe if it has a model and unsafe if it has no model. The procedure VERIFY(P) is described in algorithm 1. VERIFY makes use of the procedure SAFE(P) in the Algorithm 1, which is an oracle that returns *safe*, *unsafe* or *unknown*. The oracle is sound: if SAFE(P) returns *safe* (*unsafe*) then P is safe (unsafe). SAFE could be any existing automatic Horn clause solver [12, 19, 18, 17, 6]. When it cannot verify a program within a given time limit, the *unknown* answer is emitted. A given set of Horn clauses P can be transformed into a new set of clauses $P^{\leq k}$ and $P^{>k}$. In order to prove some property of all derivations of P, we systematically apply these transformations, for various values of k, to decompose the proof into separate proofs for $P^{\leq k}$ *(line 4)* and $P^{>k}$ *(line 9)*. If both are safe then P is *safe*. If one of them is unsafe then P is *unsafe*. If an oracle cannot prove whether $P^{\leq k}$ is *safe/unsafe* then we return an *unknown* answer (we assume that the oracle would also return *unknown* for larger values of k). But if it cannot prove whether $P^{>k}$ is *safe/unsafe* then we try the *while loop* in the algorithm 1 with $k = k+1$.

One possible optimisation that we can make in Algorithm 1 is to consider $P^{>k}$ instead of P in the next iteration of the *while loop* if we reach *line 14*. This is because at this stage we have already proven the safety of $P^{\leq k}$.

The soundness of Algorithm 1 is captured by the following lemma and proposition.

Algorithm 1: Verification algorithm for Horn clauses

```
1  Procedure VERIFY (P)
   Input: Set of CHCs P
   Output: safe, unsafe, unknown
2  initialization: k ← 0
3  while true do
4  |  generate P^{≤k}
5  |  r_1 ← SAFE(P^{≤k})
6  |  if r_1 ≠ safe then
7  |  |  return r_1
8  |  end
9  |  generate P^{{k}}
10 |  r'_1 ← SAFE( P^{>k})
11 |  if r'_1 ≠ unknown then
12 |  |  return r'_1
13 |  end
14 |  k ← k + 1
15 end
```

Lemma 1 (Decomposition by dimension) *For all k, program P is safe if and only if both $P^{\leq k}$ and $P^{>k}$ are safe.*

Proposition 1 (Soundness) *If Algorithm 1 returns* safe *then the input program is safe. If it returns* unsafe *then the program is unsafe.*

4 Dimension decomposition using finite tree automata

In this section, we show an alternative method for constructing an at-least k-dimension program, using operations on finite tree automata (FTAs). We first describe the connection between Horn clauses and FTAs and show how to construct an FTA from a set of Horn clauses.

4.1 Trace automata for CHCs

We add identifiers to clauses, whose purpose is to act as constructors of trace trees representing derivations. The identifiers are chosen from a set Σ of ranked function symbols. If P is a set of CHCs, let $\mathsf{id}_P : P \to \Sigma$ be an assignment of function symbols to clauses, such that for every clause $cl \in P$, the arity of $\mathsf{id}_P(cl)$ equals the number of atoms in the body of cl. We allow the same symbol to be assigned by id_P to more than one clause. We can also identify the predicates whose derivations are of interest (the *accepting* predicates in Definition 4).

Definition 4 (Trace FTA for a set of CHCs) *Let P be a set of CHCs, Σ be a set of ranked function symbols and $\mathsf{id}_P : P \to \Sigma$ be a mapping from clauses to function symbols of appropriate arity. Let F be a set of predicates from P called the* accepting *predicates. Define the trace FTA for P as $\mathscr{A}_P^F = (Q, F, \Sigma, \Delta)$ where*

- *Q is the set of predicate symbols of P;*

- $F \subseteq Q$ *is the set of accepting predicate symbols;*
- Σ *is a set of function symbols;*
- $\Delta = \{c(p_1,\ldots,p_k) \to p \mid cl \in P,\ cl = p(X) \leftarrow \mathscr{C}, p_1(X_1),\ldots,p_k(X_k),\ c = \mathsf{id}_P(cl)\}$.

If F is the set of all predicate symbols occurring in the clauses we omit the superscript F from $\mathscr{A}_P^F$.

The set of trees accepted by $\mathscr{A}_P^F$ is written $\mathscr{L}(\mathscr{A}_P^F)$. Elements of $\mathscr{L}(\mathscr{A}_P^F)$ are called the trace trees for P. $\mathscr{L}(\mathscr{A}_P^F)$ is isomorphic to the set of (successful and unsuccessful) derivation trees (for atomic formulas with accepting predicates) constructible from P and from now on we identify trace trees with derivations. We do not define derivation trees formally here, but refer to the notion of an AND-tree in the literature [22, 9].

Example 1 *Let P be the set of CHCs in Figure 1 and let $F = \{\texttt{fib},\texttt{false}\}$. Let id_P map the clauses to c_1,c_2,c_3 respectively. Then $\mathscr{A}_P^F = (Q,F,\Sigma,\Delta)$ where:*

$$\begin{array}{lcllcl} Q & = & \{\texttt{fib},\texttt{false}\} & \Delta & = & \{c_1 \to \texttt{fib},\\ \Sigma & = & \{c_1,c_2,c_3\} & & & c_2(\texttt{fib},\texttt{fib}) \to \texttt{fib},\\ & & & & & c_3(\texttt{fib}) \to \texttt{false}\} \end{array}$$

Figure 2(a) shows a trace tree recognised by this FTA. The tree can also be written $c_3(c_2(c_2(c_1,c_1),c_1))$.

If a mapping $\mathsf{id}_P : P \to \Sigma$ assigns a unique identifier to each clause, that is, id_P is injective, then there is an inverse mapping $\mathsf{id}^{-1} : \mathsf{range}(\mathsf{id}_P) \to P$.

Definition 5 ($\mathsf{chc}_{\mathsf{id}}(\mathscr{A})$) *Given an FTA $\mathscr{A} = (Q,F,\Sigma,\Delta)$ and an injective mapping id such that $\Sigma \subseteq \mathsf{range}(\mathsf{id})$, we can construct a set of CHCs from $\mathscr{A}$, called $\mathsf{chc}_{\mathsf{id}}(\mathscr{A})$, defined as follows:*

$$\mathsf{chc}_{\mathsf{id}}(\mathscr{A}) = \{q(X) \leftarrow \mathscr{C}, q_1(X_1),\ldots,q_n(X_n) \mid \begin{array}{l} c(q_1,\ldots,q_n) \to q \in \Delta,\\ \mathsf{id}^{-1}(c) = q(X) \leftarrow \mathscr{C}, q_1(X_1),\ldots,q_n(X_n)\} \end{array}$$

The set of accepting predicates of $\mathsf{chc}_{\mathsf{id}}(\mathscr{A})$ is defined to be F.

In the definitions we reuse the states in the FTA as predicate symbols in the constructed clauses. In practice we use some injective renaming function from states to predicates in the constructed program. Further discussion of the mappings between CHCs and FTAs can be found in [19]. By construction, the derivations of $\mathsf{chc}_{\mathsf{id}}(\mathscr{A})$ (for the accepting predicates) correspond to the elements of $\mathscr{L}(\mathscr{A})$.

4.2 Construction of the at-least k-dimensional program using FTA operations

In the construction of the at-least k-dimension program $P^{>k}$ in Definition 4, the original program clauses from P are included in the generated clauses. The presence of the original clauses suggests that the "decomposed" verification problem for $P^{>k}$ is as hard as the original problem for P, since it contains the clauses of P as well as others, and so this form might not lend itself to verification.

Thus in the following construction we build $P^{>k}$ based on FTA language difference, and the original clauses are not copied to the at-least k-dimension program. We first define a general FTA-difference for CHCs.

Definition 6 (FTA-difference for CHCs) *Let P and Q be sets of CHCs, F_1 and F_2 their respective accepting predicates and $\mathsf{id}_P : P \to \Sigma$ and $\mathsf{id}_Q : Q \to \Sigma$ their respective identifier assignments, where id_P is injective. Let $\mathscr{A}_P^{F_1}$ and $\mathscr{A}_Q^{F_2}$ be the trace FTAs constructed from P,Q respectively. Then the FTA-difference of P and Q (with their respective accepting predicates) written $P^{F_1} - Q^{F_2}$, is given as $\mathsf{chc}_{\mathsf{id}_P}(\mathscr{A}_P^{F_1} \setminus \mathscr{A}_Q^{F_2})$ where $\setminus$ is the difference of FTAs [3]. The set of accepting predicates is the set of accepting states for the difference FTA.*

The set of derivations for $P^{F_1} - Q^{F_2}$ contains, by construction, those derivations of P^{F_1} that are not derivations of Q^{F_2}. We now apply these notions to the verification procedure based on decomposition. We are given a set of CHCs P, with accepting predicates $F = \{\mathsf{false}\}$. In the program $P^{\leq k}$, the set of accepting predicates is $F^k = \{\mathsf{false}^{\leq k}\}$. Note that we can ignore the derivations for the other predicates of the form $\mathsf{false}^{\leq j}$ or $\mathsf{false}^{=j}$ since $\mathsf{false}^{\leq k}$ by construction accumulates their derivations, for all $j \leq k$.

4.2.1 Assignment of identifiers in the at-most-k-dimension program

Given a program P and the at-most-k-dimension program $P^{\leq k}$, we intend to construct the difference $P^{\{\mathsf{false}\}} - P^{\leq k\{\mathsf{false}\}}$ using Definition 6. In order to do so, we first need to construct the identifier assignment $\mathsf{id}_{P^{\leq k}}$ so as to preserve trace trees from P. This requires the modification of $P^{\leq k}$ to eliminate the ε-clauses, as follows.

Definition 7 (Unfolding of ε-clauses in $P^{\leq k}$) *Let $P^{\leq k}$ be the at-most-k-dimension program obtained from P using Definition 2. Replace each ε-clause of form $H^{\leq d} \leftarrow H^{=e}$ by the set of clauses $H^{\leq d} \leftarrow B$, where $H^{=e} \leftarrow B$ is either a linear or non-linear clause in $P^{\leq k}$.*

The elimination of ε-clauses is an instance of the well-known unfolding transformation which preserves the derivability of atomic formulas. In other words an atom A is derivable from a program P if and only if it is derivable after applying the unfolding transformation [21].

In the following definition, the clause identifiers are chosen for clauses in $P^{\leq k}$ Informally, every clause of $P^{\leq k}$ inherits the clause identifier for the clause in P from which it originates. More precisely we define the clause identifiers for $P^{\leq k}$ as follows.

Definition 8 (Assignment of clause identifiers in $P^{\leq k}$) *Let $P^{\leq k}$ be the at-most-k-dimension program obtained from P using Definition 2, with ε-clauses eliminated according to Definition 7. Each clause of $P^{\leq k}$ is a linear, non-linear or an ε-unfolded-clause. The clause identifiers are assigned in two steps as follows.*

1. *Assign to each linear or non-linear clause the clause identifier from the clause in P from which it is derived in Definition 2.*
2. *Assign to each unfolded ε-clause the clause identifier for the linear or non-linear clause used to unfold it using Definition 7.*

We are now in a position to compare the sets of trace trees for P and $P^{\leq k}$ using their respective FTAs.

Lemma 2 *Let P be a set of CHCs and let $\mathsf{id}_P : P \to \Sigma$ be an injective function assigning clause identifiers to P. Let $F_1 = \{\mathsf{false}\}$. Let $k \geq 0$ and let $P^{\leq k}$ be the at-most-k-dimension program obtained from P using Definition 2 with ε-clauses unfolded using Definition 7 and let $F_2 = \{\mathsf{false}^{\leq k}\}$. Then $\mathscr{L}(\mathscr{A}^{F_2}_{P^{\leq k}}) = \{t \mid t \in \mathscr{L}(A^{F_1}_P), dim(t) \leq k\}$.*

The proof is by induction on derivations in $P^{\leq k}$ and uses the correspondence of the clause identifiers as set up in Definition 8.

Theorem 1 *Let P be a set of CHCs and let $\mathsf{id}_P : P \to \Sigma$ be an injective function assigning clause identifiers to P. Let $k \geq 0$ and let $P^{\leq k}$ be the at-most-k-dimension program obtained from P using Definition 2 with ε-clauses unfolded using Definition 7. Then false is derivable from $P - P^{\leq k}$ if and only if $\mathsf{false}^{>k}$ is derivable from $P^{>k}$.*

```
c1. fib(0)(A,A) :- A>=0, A=<1.
c3. false(0) :- A>5, B<A, fib(0)(A,B).
c3. false[0] :- A>5, B<A, fib(0)(A,B).
c1. fib[0](A,B) :-A>=0, A=<1.
```

Figure 5: Fib$^{\leq 0}$ after unfolding ε-clauses and assigning clause identifiers.

Thus we have shown a different method of constructing the at-least k-dimension program $P^{>k}$, namely by taking the difference of P with $P^{\leq k}$, which contains only derivations (for its accepting predicates) that have dimension greater than k.

Details on difference construction can be found in [19]. We construct the difference of two FTAs by (1) standardising apart the predicate names; (2) forming the union of the two FTAs; (3) determinising the union; (4) removing from the determinised FTA all states (and transitions that contain them) that contain an accepting state of the second FTA. Note that the set of states of the determinised FTA is a subset of the powerset of the original states. Note that determinisation of FTAs is often considered prohibitively complex even for small FTAs. We use a recent optimised FTA determinisation algorithm [10], returning a compact form of the determinised called product form, which can be used directly in constructing the resulting clauses.

Example 2 *We illustrate this through an example using Fib$^{\leq 0}$ (Figure 3). The clauses 1 and 2 in Fib$^{\leq 0}$, will have c_1 and c_3 as identifiers since they were derived respectively from the clauses c_1 and c_3 in Fib (Figure 1). By unfolding ε-clauses (clauses 3 and 4) using respectively clauses 2 and 1 in Figure 3, we obtain* `false[0] :- A>5, B<A, fib(0)(A,B)` *and* `fib[0](A,B) :-A>=0, A=<1`. *They will have identifiers c_3 and c_1 respectively. Therefore, the clauses in Fib$^{\leq 0}$ will have the identifiers assigned as shown in Figure 5.*

After assigning identifiers to each of the clauses in Fib$^{\leq 0}$, we can construct an FTA corresponding to it using Definition 4, and obtain the FTA shown in Figure 6: as before we represent a predicate $p^{\leq k}$ by `p[k]` and a predicate $p^{=k}$ by `p(k)`.

$$Q = \{\texttt{fib(0)},\ \texttt{false(0)},\ \texttt{false[0]},\ \texttt{fib[0]}\} \qquad \Delta = \{c_1 \rightarrow\ \texttt{fib(0)},\ c_3(\ \texttt{fib(0)}) \rightarrow\ \texttt{false(0)},\ c_3(\ \texttt{fib(0)}) \rightarrow\ \texttt{false[0]},\ c_1 \rightarrow\ \texttt{fib[0]}\}$$
$$F = \{\texttt{false[0]}\}$$
$$\Sigma = \{c_1, c_3\}$$

Figure 6: FTA (Q, F, Σ, Δ) corresponding to Fib$^{\leq 0}$.

The difference FTA between $\mathscr{A}_{Fib}^{\{\mathsf{false}\}}$ and $\mathscr{A}_{Fib^{\leq 0}}^{\{\mathsf{false}^{\leq 0}\}}$ accepts trees rooted at false which have dimension greater than 0. The determinised FTA (DFTA) constructed as explained above is shown in the Figure 7. DFTA states are sets of predicates, and we represent a set using square brackets instead of curly brackets in the code, e.g. `[fib(0), fib[0], fib]`. Furthermore the product form referred to above contains set of DFTA states, such as `[[fib(0), fib[0], fib], [fib]]`.

We can generate a new program from this DFTA together with the original program Fib following the approach taken in [19] obtaining the program in Figure 8. It should be noted that the derivation trees rooted at false have dimension at-least 1. Now verification of the original program Fib is decomposed into verifying the program in Figure 3 (where `false[0]` is replaced by `false` and the program in Figure 8.

```
c1 -> [fib(0), fib[0], fib].
c2([[fib(0), fib[0], fib], [fib]],
                      [[fib(0), fib[0], fib], [fib]]) -> [fib].
c3([[fib]]) -> [false].
```

Figure 7: Transitions of the determinised FTA.

```
fib_0(A,A) :- A>=0, A=<1.
fib(A,B) :-  A>1,C=A-2,D=A-1,B=E+F,fib_1(C,F), fib_1(D,E).
false :- A>5, B<A, fib(A,B).
fib_1(A,B) :- fib_0(A,B).
fib_1(A,B) :-  fib(A,B).
```

Figure 8: At-least 1-dimension program of Fib produced using the difference of FTAs

5 Program instrumentation with dimension

The dimension of successful derivations in a set of CHCs is not always obvious from the text of the clauses. In some cases a bound on the dimension is clear from the form of the clauses; for instance all derivations using a set of linear clauses clearly have dimension zero. But consider the well known 91-function of McCarthy[1], represented in Figure 9 using Horn clauses.

Although it is possible to construct derivation trees of arbitrary dimension using the clauses in Figure 9, the dependencies between the two recursive calls to `mc91` imply that no *successful* derivation has dimension greater than 2. We now show how to establish this using a transformation to instrument the clauses with dimension information, and then use automatic verification tools to establish properties of the dimension.

Definition 9 (Dimension-instrumented clauses) *Let P be a set of CHCs. Define the set P_{dim} of CHC as follows.*

- *For each predicate p of arity m define a predicate p' of arity $m+1$.*
- *For each clause in P of the form*

$$p(X) \leftarrow \mathscr{C}, p_1(X_1), \ldots, p_n(X_n)$$

construct a clause

$$p'(X,K) \leftarrow \mathscr{C}, p_1^{'}(X_1,K_1), \ldots, p_n^{'}(X_n,K_n), dim_n(K_1,\ldots,K_n,K)$$

in P_{dim}, where $K_1,\ldots,K_n,K$ are variables added as the final argument for their respective predicates, and $dim_n(K_1,\ldots,K_n,K)$ is defined according to the rules in Definition 1 for determining the dimension of a tree.

[1] http://en.wikipedia.org/wiki/McCarthy_91_function

```
mc91(N,X) :-  N > 100,  X = N-10.
mc91(N,X) :-  N =< 100,  Y = N+11,
              mc91(Y,Y2), mc91(Y2,X).
```

Figure 9: McCarthy's 91-function defined as Horn clauses

```
fib(A, A, K):-  A>=0,  A=<1, dim0(K).
fib(A, B, K) :- A > 1, A2 = A - 2, fib(A2, B2, K1),
           A1 = A - 1, fib(A1, B1, K2), B = B1 + B2, dim2(K1, K2, K).
dim0(K):-K=0.
dim2(K1, K2, K3):-K1>=K2+1, K3=K1.
dim2(K1, K2, K3):-K2>=K1+1, K3=K2.
dim2(K1, K2, K3):- K1=K2, K3 = K1+1.
```

Figure 10: Fib program instrumented with its dimension

Example 3 *The dimension-instrumented version of the McCarthy 91-function contains the following clauses.*

```
mc91(N,X,K) :-  N > 100,  X = N-10, dim0(K).
mc91(N,X,K) :-  N =< 100,  Y = N+11,
     mc91(Y,Y2,K1), mc91(Y2,X,K2), dim2(K1,K2,K).
dim0(K):-K=0.
dim2(K1, K2, K3):-K1>=K2+1, K3=K1.
dim2(K1, K2, K3):-K2>=K1+1, K3=K2.
dim2(K1, K2, K3):- K1=K2, K3 = K1+1.
```

Using the instrumented program we can try to prove information about the dimension, such as upper or lower bounds or other relationships between the dimension and other predicate arguments. It follows from the undecidability result of Gruska [14] on context-free grammars, that the problem of determining whether the dimension of set of CHC is bounded by a constant is, in general, undecidable.

Example 4 *To establish that the upper bound of successful derivations is 2, for facts* `mc91(X,Y)`*, we add the following integrity constraint to the dimension-instrumented clauses.*

```
false :- K > 2, mc91(X,Y,K).
```

The clauses together with the integrity constraint are given to an automatic solver for Horn clauses [12, 19], which are able to prove the safety of the clauses and thus establish the upper bound of 2.

In the next example, we show that the dimension can depend on the values of other predicate arguments.

Example 5 *The dimension-instrumented version of the* Fib *clauses is shown in Figure 10. The property to be proved is that the dimension of* Fib *is lesser or equal to the half of its input value, expressed by the integrity constraint* `false:- fib(A,B, K), 2*K -1>=A`*. Again, this property is established by applying a Horn clause solver to prove the safety of the clauses together with the integrity constraint.*

Example 6 *We present the well known counting change example taken from [1, Chapter 1]. The Figure 11 shows its CLP encoding and the Figure 12 shows the dimension-instrumented version in CLP. The property of interest is to relate the number of different coins (counts) with the program dimension. We can establish that the dimension is at most the number of different coins as expressed by the integrity constraint* `false :- B>=1, K > B, cc(A, B, C, K)`.

In general, verifying whether a program has a certain dimension is as challenging as proving any other properties of the program. But in some cases the knowledge of program dimension is useful for proving other program properties. For instance, using the knowledge that the McCarthy 91-function has dimension at most 2 would allow us to restrict the proof of any program property relating to successful derivations to the program $P^{[2]}$ where P is the set of clauses for the McCarthy 91-function.

```
% base case: that is a hit
cc(0, Y, 1) :- Y>0.
% base case: that is a miss
cc(X, _, 0) :- X<0.
cc(_, Y, 0) :- Y=<0.
%inductive case
cc(X, Y, Z) :- X>0, kinds_of_coins(Y,A),
                    X1 = X-A, cc(X1, Y, Z1),
                    Y1 = Y-1, cc(X, Y1, Z2), Z = Z1 + Z2.
kinds_of_coins(A,B) :- A >= 1, B >= 1.
```

Figure 11: Counting change example encoded as CLP clauses

```
cc(0, Y, 1,K) :- Y>0, dim0(K).
cc(X, _, 0,K) :- X<0, dim0(K).
cc(_, Y, 0,K) :- Y=<0, dim0(K).
cc(X, Y, Z,K) :-
        X>0, kinds_of_coins(Y,A, K0), X1 = X-A,
        cc(X1, Y, Z1,K1), Y1 = Y-1, cc(X, Y1, Z2,K2),
        Z = Z1 + Z2, dim3(K0, K1,K2,K).
kinds_of_coins(A,B, K) :- A >= 1, B >= 1, dim0(K).
dim3(K0, K1,K2,K):-
        dim2(K0, K1, K3), dim2(K3,K2, K).
%predicates dim0(K) and dim2(K1, K2, K) are defined as above
```

Figure 12: Counting change example instrumented with its dimension

6 Related Work

The notion of dimension of a tree has a long history in science (starting with Geology) which has been detailed by Esparza *et al.* [8]. However, the use of dimension for program verification is more recent. Ganty and Iosif used it [11] for computing summaries of programs with procedures whose variables (global, local and parameters) take their value from the set of integers. Roughly speaking, the method they define first computes procedure summaries for all derivation trees of dimension 0, then they compute summaries for derivation trees of dimension 1 reusing the summaries computed for dimension 0 and so on.

Decomposition can be compared to refinement techniques based on automata [15, 16, 19] in which the aim is to eliminate sets of program traces that have been shown to be safe. Proof of the safety of a given dimension or dimensions of a set of clauses allows those dimensions to be eliminated, focusing the proof on the remaining dimensions. Our decomposition technique offers a very precise and practical approach to checking and eliminating infinite sets of traces.

7 Experimental results

We carried out an experiment on a set of 16 non-linear CHC verification problems taken from the repository[2] of software verification benchmarks. Our aim in the current paper is not to make a systematic comparison with other verification techniques; these are exploratory experiments to establish whether dimension-based decomposition is practical. The results are summarized in Table 1. Columns **Program**, **Result**, **Time** and **dim(k)** respectively represent a program, its verification result using our approach, time in seconds taken to generate the programs and solve it and a value of a proof decomposition parameter k.

Table 1: Experimental results on non-linear CHC verification problems

Program	**Result**	**Time(s)**	**dim(k)**
addition	safe	4	0
bfprt	safe	4	0
binarysearch	safe	4	0
countZero	safe	3	0
floodfill	safe	3	0
identity	safe	4	0
merge	safe	5	0
palindrome	safe	3	0
fib	safe	4	0
mc91	safe	4	0
revlen	safe	4	0
running	unsafe	6	1
triple	unsafe	-	-
buildheap	unsafe	-	-
parity	unsafe	4	0
remainder	unsafe	4	0
avg. time(s)		4	

For the safety check (the procedure *SAFE* in Algorithm 1) we use the verification procedure described in [18] which uses abstract interpretation over the domain of convex polyhedra, with a timeout of 5 minutes. The symbol "-" in Table 1 denotes that we were unable to solve these problems within the given time. Our approach solves 14 out of 16 problems with an average time of 4 seconds (over the solved problems). Our previous approach based on refinement with finite tree automata described in [19] solves 1 more additional problem, that is, *triple* than our current approach. These examples were also run on QARMC [13] which solves all the problems (much faster).

Most of the problems are solved when we decompose the proof with the value of $k = 0$. This indicates that separating the proofs for linear programs eases the verification task. The splitting induced as a result of separating a set of traces has an effect on delaying join and widening operations during convex polyhedra analysis which increases its precision. In addition to this, some of the case base proofs (for example conditionals) becomes a normal proof without conditionals due to proof separation and the process of finding invariants becomes easier.

8 Conclusion and future work

We presented a program transformation approach to Horn clause verification using the notion of *tree dimension* to decompose the verification problem by separating dimensions. We presented one algorithm based on this idea which yielded preliminary results on set of non-linear Horn clause verification benchmarks, showing that the approach is feasible and this transformation is useful both for proving safety of a program as well as for finding bugs.

Other ideas of program verification based on tree-dimension are worth investigating, including proof by induction based on tree dimension, and further investigation of proof strategies that could exploit knowledge of dimension bounds (such as those discussed in Section 5).

[2] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Eldarica/RECUR/

Although it is formulated in the context of Datalog, it is known from Afrati *et al.* [2] that a set of CHC of bounded dimension can be turned into an equivalent set of linear CHC. The exact complexity of their procedure is still open.

References

[1] Harold Abelson & Gerald J. Sussman (1996): *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press.

[2] Foto N. Afrati, Manolis Gergatsoulis & Francesca Toni (2003): *Linearisability on datalog programs*. *Theor. Comput. Sci.* 308(1-3), pp. 199–226, doi:10.1016/S0304-3975(02)00730-2.

[3] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison & M. Tommasi (2007): *Tree Automata Techniques and Applications*. Available on: `http://www.grappa.univ-lille3.fr/tata`. Release October, 12th 2007.

[4] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In Robert M. Graham, Michael A. Harrison & Ravi Sethi, editors: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, ACM, pp. 238–252, doi:10.1145/512950.512973. Available at `http://dl.acm.org/citation.cfm?id=512950`.

[5] Patrick Cousot & Nicolas Halbwachs (1978): *Automatic Discovery of Linear Restraints Among Variables of a Program*. In Alfred V. Aho, Stephen N. Zilles & Thomas G. Szymanski, editors: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, ACM Press, pp. 84–96, doi:10.1145/512760.512770. Available at `http://dl.acm.org/citation.cfm?id=512760`.

[6] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations*. In Erika Ábrahám & Klaus Havelund, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, *Lecture Notes in Computer Science* 8413, Springer, pp. 568–574, doi:10.1007/978-3-642-54862-8_47.

[7] Javier Esparza, Stefan Kiefer & Michael Luttenberger (2007): *On Fixed Point Equations over Commutative Semirings*. In: *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, *LNCS* 4393, Springer, pp. 296–307, doi:10.1007/978-3-540-70918-3_26.

[8] Javier Esparza, Michael Luttenberger & Maximilian Schlund (2014): *A Brief History of Strahler Numbers*. In: *LATA '14, 8th Int. Conf. on Language and Automata Theory and Applications*, *LNCS* 8370, Springer, p. 113, doi:10.1007/978-3-319-04921-2_1.

[9] J. P. Gallagher & L. Lafave (1996): *Regular Approximation of Computation Paths in Logic and Functional Languages*. In: *Partial Evaluation*, *LNCS* 1110, Springer, pp. 115–136, doi:10.1007/3-540-61580-6_7.

[10] John P. Gallagher, Mai Ajspur & Bishoksan Kafle (2014): *An Optimised Algorithm for Determinisation and Completion of Finite Tree Automata*. Technical Report 145, Roskilde University, Denmark. Available from http://arxiv.org/pdf/1511.03595v1.pdf.

[11] Pierre Ganty, Radu Iosif & Filip Konečný (2013): *Underapproximation of Procedure Summaries for Integer Programs*. In: *TACAS '13: Proc. 19th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* 7795, Springer, pp. 247–261, doi:10.1007/978-3-642-36742-7_18.

[12] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution)*. In Cormac Flanagan & Barbara König, editors: *TACAS*, *LNCS* 7214, Springer, pp. 549–551, doi:10.1007/978-3-642-28756-5_46.

[13] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[14] Jozef Gruska (1971): *A Few Remarks on the Index of Context-Free Grammars and Languages*. *Information and Control* 19(3), pp. 216–223, doi:10.1016/S0019-9958(71)90095-7.

[15] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2009): *Refinement of Trace Abstraction*. In: *Static Analysis, 16th International Symposium, SAS 2009*, *LNCS* 5673, Springer, pp. 69–85, doi:10.1007/978-3-642-03237-0_7.

[16] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2013): *Software Model Checking for People Who Love Automata*. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, *LNCS* 8044, Springer, pp. 36–52, doi:10.1007/978-3-642-39799-8_2.

[17] Krystof Hoder & Nikolaj Bjørner (2012): *Generalized Property Directed Reachability*. In Alessandro Cimatti & Roberto Sebastiani, editors: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, *Lecture Notes in Computer Science* 7317, Springer, pp. 157–171, doi:10.1007/978-3-642-31612-8_13.

[18] Bishoksan Kafle & John P. Gallagher (2015): *Constraint Specialisation in Horn Clause Verification*. In: *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015*, ACM, pp. 85–90, doi:10.1145/2678015.2682544.

[19] Bishoksan Kafle & John P. Gallagher (2015): *Tree Automata-Based Refinement with Application to Horn Clause Verification*. In: *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, *LNCS* 8931, Springer, pp. 209–226, doi:10.1007/978-3-662-46081-8_12.

[20] Michael Luttenberger (2011): *An Extension of Parikh's Theorem beyond Idempotence*. *CoRR* abs/1112.2864. Available at `http://arxiv.org/abs/1112.2864`.

[21] Alberto Pettorossi & Maurizio Proietti (1999): *Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs*. *J. Log. Program.* 41(2-3), pp. 197–230, doi:10.1016/S0743-1066(99)00029-1.

[22] Robert F. Stärk (1989): *A Direct Proof for the Completeness of SLD-Resolution*. In Egon Börger, Hans Kleine Büning & Michael M. Richter, editors: *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, *Lecture Notes in Computer Science* 440, Springer, pp. 382–383, doi:10.1007/3-540-52753-2_52.

Attachment D3.3.10

Convex polyhedral abstractions, specialisation and property-based predicate splitting in Horn clause verification

Published at the 1st Workshop on Horn Clauses for Verification and Synthesis (HCVS 2014)

Convex polyhedral abstractions, specialisation and property-based predicate splitting in Horn clause verification*

Bishoksan Kafle
Roskilde University
Denmark
kafle@ruc.dk

John P. Gallagher
Roskilde University
Denmark
IMDEA Software Institute
Madrid, Spain
jpg@ruc.dk

We present an approach to constrained Horn clause (CHC) verification combining three techniques: abstract interpretation over a domain of convex polyhedra, specialisation of the constraints in CHCs using abstract interpretation of query-answer transformed clauses, and refinement by splitting predicates. The purpose of the work is to investigate how analysis and transformation tools developed for constraint logic programs (CLP) can be applied to the Horn clause verification problem. Abstract interpretation over convex polyhedra is capable of deriving sophisticated invariants and when used in conjunction with specialisation for propagating constraints it can frequently solve challenging verification problems. This is a contribution in itself, but refinement is needed when it fails, and the question of how to refine convex polyhedral analyses has not been studied much. We present a refinement technique based on interpolants derived from a counterexample trace; these are used to drive a property-based specialisation that splits predicates, leading in turn to more precise convex polyhedral analyses. The process of specialisation, analysis and splitting can be repeated, in a manner similar to the CEGAR and iterative specialisation approaches.

1 Introduction

In this paper we explore the use of techniques used in constraint logic program (CLP) analysis and specialisation, for the purpose of CHC verification. Pure CLP is syntactically and semantically the same as CHC. Unlike CLP, CHCs are not always regarded as executable programs, but rather as specifications or semantic representations of other formalisms. However these are only pragmatic distinctions and the semantic equivalence of CHC and CLP means that techniques developed in one framework are applicable to the other.

Relevant concepts from CLP include the approximation of the minimal model of a CLP program using abstract interpretation, specialisation of a CLP program with respect to a goal and model-preserving transformation of CLP programs. Relevant concepts drawn from the CHC verification literature include finding a model of a set of CHCs, property-based abstraction, counterexample generation, and refinement of property-based abstraction using interpolants.

The results shown in the paper are preliminary and much research remains to be done in exploiting the many connections and possibilities for cross-fertilisation between CLP and CHC. The contributions of this paper are:

*The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement no. 318337, ENTRA - Whole-Systems Energy Transparency and the Danish Natural Science Research Council grant NUSA: Numerical and Symbolic Abstractions for Software Model Checking.

Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko,
Valerio Senni (Eds.): First Workshop on
Horn Clauses for Verification and Synthesis (HCVS 2014)
EPTCS 169, 2014, pp. 53–67, doi:10.4204/EPTCS.169.7

- to demonstrate that abstract interpretation over convex polyhedra is capable of deriving sophisticated invariants, and when used in conjunction with specialisation for propagating constraints it can frequently solve challenging verification problems;
- to investigate the problem of refinement of polyhedral abstractions, drawing ideas from counterexample-guided refinement.

In Section 2 we define the basic notation and concepts needed for the verification procedure. Section 3 reviews the technique of abstract interpretation over convex polyhedra, applied to CLP/CHC, along with the important enhancement of this technique using widening thresholds. In Section 4 a procedure for specialisation of CHCs is described, based on query-answer transformations and abstract interpretation. A simple but surprisingly effective verification tool-chain combining specialisation with abstract interpretation is introduced. Section 5 explains how to use a (spurious) counterexample from a failed verification attempt to construct a property-based specialisation using interpolants. Experimental results and related works are reported in Section 6 and Section 7 respectively. Finally in Section 8 we conclude and discuss possible extensions and improvements.

2 Preliminaries

A CHC is a first order predicate logic formula of the form $\forall(\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k) \rightarrow H(X))$ $(k \geq 0)$, where ϕ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $B_1, \ldots, B_k, H$ are predicate symbols, $H(X)$ is the head of the clause and $\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k)$ is the body. Sometimes the clause is written $H(X) \leftarrow \phi \wedge B_1(X_1), \ldots, B_k(X_k)$ and in concrete examples it is written in the form `H :- `ϕ`, `B_1`(`X_1`),...,`B_k`(`X_k`)`. In examples, predicate symbols start with lowercase letters while we use uppercase letters for variables.

In this paper we take the constraint theory to be linear arithmetic with the relation symbols $\leq, \geq$, $<, >$ and $=$. There is a distinguished predicate symbol false which is interpreted as false. In practice the predicate false only occurs in the head of clauses; we call clauses whose head is false *integrity constraints*, following the terminology of deductive databases. Thus the formula $\phi_1 \leftarrow \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$ is equivalent to the formula $\mathsf{false} \leftarrow \neg\phi_1 \wedge \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$. The latter might not be a CHC (e.g. if ϕ_1 contains $=$) but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula `X=Y :- p(X,Y)` is equivalent to the set of CHCs `false :- X>Y, p(X,Y)` and `false :- X<Y, p(X,Y)`. Integrity constraints can be seen as safety properties. For example if a set of CHCs encodes the behaviour of a transition system, the bodies of integrity constraints represent unsafe states. Thus proving safety consists of showing that the bodies of integrity constraints are false in all models of the CHC clauses. Figure 1 shows an example set of CHCs (taken from [6]), modeled over reals containing an integrity constraint, and in this example the problem is to prove that the body of the first clause is unsatisfiable.

2.1 The CHC verification problem.

To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. Obviously any model of P assigns false to the bodies of integrity constraints. We restate this property in terms of the derivability of the predicate false. Let $P \models F$ mean that F is a logical consequence of P, that is, that every interpretation satisfying P also satisfies F.

Lemma 1. *P has a model if and only if $P \not\models \mathsf{false}$.*

```
c1. false:- N>0,I=0,A=0,B=0, l(I,A,B,N).
c2. l(I,A,B,N):-I < N, l_body(A,B,A1,B1), I1 = I+1, l(I1,A1,B1,N).
c3. l(I,A,B,N):- I >=N, A + B > 3 * N.
c4. l(I,A,B,N):- I >=N, A + B < 3 * N.
c5. l_body(A0,B0,A1,B1):- A1 = A0+1, B1 = B0+2.
c6. l_body(A0,B0,A1,B1):- A1 = A0+2, B1 = B0+1.
```

Figure 1: Example program *t4.pl* [6]

Proof. Writing $I(F)$ to mean that interpretation I satisfies F, we have:

$$\begin{aligned} P \not\models \mathsf{false} \quad &\equiv \quad \text{there exists some interpretation } I \text{ such that } I(P) \text{ and } \neg I(\mathsf{false}) \\ &\quad\ \ \textit{by definition of the } \models \textit{ relation} \\ &\equiv \quad \text{there exists some interpretation } I \text{ such that } I(P) \\ &\quad\ \ (\textit{since } \neg I(\mathsf{false}) \textit{ is true by defn. of } \mathsf{false}) \\ &\equiv \quad P \text{ has a model.} \end{aligned}$$

□

This lemma holds for arbitrary interpretations (only assuming that the predicate false is interpreted as false), uses only the textbook definitions of "interpretation" and "model" and does not depend on the constraint theory.

The verification problem can be formulated deductively rather than model-theoretically. We can exploit proof procedures for constraint logic programming [24] to reason about the satisfiability of a set of CHCs. Let the relation $P \vdash A$ denote that A is derivable from P using some proof procedure. If the proof procedure is sound then $P \vdash A$ implies $P \models A$, which means that $P \vdash \mathsf{false}$ is a sufficient condition for P to have no model, by Lemma 1. This corresponds to using a sound proof procedure to find or check a counterexample. On the other hand to show that P does have a model, soundness is not enough since we need to establish $P \not\models \mathsf{false}$. As we will see in Section 2.3 we approach this problem by using *approximations* to reason about the non-provability of false, applying the theory of abstract interpretation [11] to a complete proof procedure for atomic formulas (the "fixed-point semantics" for constraint logic programs [24, Section 4]). In effect, we construct by abstract interpretation a proof procedure that is *complete* (but possibly not sound) for proofs of atomic formulas. With such a procedure, $P \not\vdash \mathsf{false}$ implies $P \not\models \mathsf{false}$ and thus establishes that P has a model.

2.2 Representation of Interpretations

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \mathscr{C}$ where A is an atomic formula $p(Z_1,\ldots,Z_n)$ where $Z_1,\ldots,Z_n$ are distinct variables and $\mathscr{C}$ is a constraint over $Z_1,\ldots,Z_n$. If $\mathscr{C}$ is true we write $A \leftarrow$ or just A. The constrained fact $A \leftarrow \mathscr{C}$ is shorthand for the set of variable-free facts $A\theta$ such that $\mathscr{C}\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

Minimal models. A model of a set of CHCs is an interpretation that satisfies each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$

can be computed as the least fixed point (lfp) of an immediate consequences operator (called S_P^D in [24, Section 4]), which is an extension of the standard T_P operator from logic programming, extended to handle the constraint domain D. Furthermore $\mathsf{lfp}(S_P^D)$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, S_P^D(\emptyset), S_P^D(S_P^D(\emptyset)), \ldots$. This sequence provides a basis for abstract interpretation of CHC clauses.

2.3 Proof Techniques

Proof by over-approximation of the minimal model. It is a standard theorem of CLP that the minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P. That is, $P \models p(v_1, \ldots, v_n)$ if and only if $p(v_1, \ldots, v_n) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that $\mathsf{false} \notin M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where $\mathsf{false} \notin M'$. This technique is called proof by *over-approximation of the minimal model*.

Proof by specialisation. A specialisation of a set of CHCs P with respect to an atom A is the transformation of P to another set of CHCs P' such that $P \models A$ if and only if $P' \models A$. Specialisation is usually viewed as a program optimisation method, specialising some general-purpose program to a subset of its possible inputs, thereby removing redundancy and pre-computing statically determined computations. In our context we use specialisation to focus the verification problem on the formula to be proved. More specifically, we specialise a set of CHCs with respect to a "query" to the atom false; thus the specialised CHCs entail false if and only if the original clauses entailed false.

3 Abstract Interpretation over Convex Polyhedra

Convex polyhedron analysis (CPA) [12] is a program analysis technique based on abstract interpretation [11]. When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \mathscr{C}$ for each predicate p. The constraint $\mathscr{C}$ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CLP was by Benoy and King [4]. Since the domain of convex polyhedra contains infinite increasing chains, the use of a *widening* operator for convex polyhedra [11, 12] is needed to ensure convergence of the abstract interpretation. Furthermore much research has been done on improving the precision of widening operators. One techniques is known as widening-upto, or widening with thresholds [23]. A threshold is an assertion that is combined with a widening operator to improve its precision.

Recently, a technique for deriving more effective thresholds was developed [27], which we have adapted and found to be effective in experimental studies. The thresholds are computed by the following method. Let S_P^D be the standard immediate consequence operator for CHCs mentioned in Section 2.2. That is, if I is a set of constrained facts, $S_P^D(I)$ is the set of constrained facts that can be derived in one step from I. Given a constrained fact $p(Z) \leftarrow \mathscr{C}$, define $\mathsf{atomconstraints}(p(Z) \leftarrow \mathscr{C})$ to be the set of constrained facts $\{p(Z) \leftarrow C_i \mid \mathscr{C} = C_1 \wedge \ldots \wedge C_k, 1 \leq i \leq k)\}$. The function atomconstraints is extended to interpretations by $\mathsf{atomconstraints}(I) = \bigcup_{p(Z) \leftarrow \mathscr{C} \in I} \{\mathsf{atomconstraints}(p(Z) \leftarrow \mathscr{C})\}$.

Let $I_\top$ be the interpretation consisting of the set of constrained facts $p(Z) \leftarrow \mathsf{true}$ for each predicate p. We perform three iterations of S_P^D starting with $I_\top$ (the first three elements of a "top-down" Kleene sequence) and then extract the atomic constraints. That is, thresholds is defined as follows.

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(S_P^{D(3)}(I_\top))$$

A difference from the method in [27] is that we use the concrete semantic function S_P^D rather than the abstract semantic function when computing thresholds. The set of threshold constraints represents an attempt to find useful predicate properties and when widening they help to preserve invariants that might otherwise be lost during widening. See [27] for further details. Threshold constraints that are not invariants are simply discarded during widening.

4 Specialisation by constraint propagation

We next present a procedure for specialising CHC clauses. In contrast to classical specialisation techniques based on partial evaluation with respect to a goal, the specialisation does not unfold the clauses at all; rather, we compute a specialised version of each clause in the program, in which the constraints from the goal are propagated top-down and answers are propagated bottom-up. The implementation is based on query-answer transformations and abstract interpretation over convex polyhedra.

Let P be a set of CHCs and let A be an atomic formula. For each clause $H \leftarrow \mathscr{B}$ in P we compute a new clause $H \leftarrow C, \mathscr{B}$ where C is a constraint, yielding a program P_A *specialised* for A. If the addition of C makes the clause body unsatisfiable, it is the same as removing the clause from P_A. Clearly P_A may have fewer consequences than P but our procedure guarantees that it preserves the inferability of (constrained instances of) A. That is, for every constraint C over the variables of A, $P \models \forall(C \rightarrow A)$ if and only if $P_A \models \forall(C \rightarrow A)$.

The procedure is as follows: the inputs are a set of CHCs P and an atomic formula A.

1. Compute a *query-answer transformation* of P with respect to A, denoted P_A^{qa}, containing predicates p^{q} and p^{a} for each predicate p in P.
2. Compute an over-approximation of the model of P_A^{qa}, expressed as a set of constrained facts $p^*(X) \leftarrow C$, where $*$ is q or a. We assume that each predicate p^* has exactly one constrained fact in the model (where C is possibly *false* or a disjunction).
3. For each clause $p(X) \leftarrow \mathscr{B}$ in P, let the model of p^{a} be $p^{\mathsf{a}}(X) \leftarrow C^{\mathsf{a}}$ (where X is the same tuple of variables in $p(X)$ and $p^{\mathsf{a}}(X)$).
4. Replace the clause $p(X) \leftarrow \mathscr{B}$ in P by $p(X) \leftarrow C^{\mathsf{a}}, \mathscr{B}$ in P_A.

Note that if for some predicate p, C^{a} is false, then all the clauses for p are removed in P_A as their bodies are unsatisfiable. We now explain each step in turn.

4.1 The query-answer transformation

The query-answer transformation was inspired by – but is a generalisation of – the magic-set transformation from deductive databases [3]. Its purpose, both in deductive databases and in subsequent applications in logic program analysis [15] was to simulate goal-directed (*top-down*) computation or deduction in a goal-independent (*bottom-up*) framework. Let us define the transformation.

Given a set of CHCs P and an atom A, the query-answer program for P wrt. A, denoted P_A^{qa}, consists of the following clauses. For an atom $A = p(t)$, A^{a} and A^{q} represent the atoms $p^{\mathsf{a}}(t)$ and $p^{\mathsf{q}}(t)$ respectively.

- (Answer clauses). For each clause $H \leftarrow C, B_1, \ldots, B_n$ $(n \geq 0)$ in P, P_A^{qa} contains the clause $H^{\mathsf{a}} \leftarrow C, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \ldots, B_n^{\mathsf{a}}$.

- (Query clauses). For each clause $H \leftarrow C, B_1, \ldots, B_i, \ldots, B_n$ ($n \geq 0$) in P, P_A^{qa} contains the following clauses:
  $B_1^{\mathsf{q}} \leftarrow C, H^{\mathsf{q}}.$
  $\ldots$
  $B_i^{\mathsf{q}} \leftarrow C, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \ldots, B_{i-1}^{\mathsf{a}}.$
  $\ldots$
  $B_n^{\mathsf{q}} \leftarrow C, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \ldots, B_{n-1}^{\mathsf{a}}.$
- (Goal clause). $A^{\mathsf{q}} \leftarrow \mathsf{true}$.

The program P_A^{qa} encodes a left-to-right, depth-first computation of the query $\leftarrow A$ for CHC clauses P (that is, the standard CLP computation rule, SLD extended with constraints). This is a complete proof procedure, assuming that all clauses matching a given call are explored in parallel. (Note: the incompleteness of standard Prolog CLP proof procedures arises due to the fact that clauses are tried in a fixed order).

The relationship of the model of the program P_A^{qa} to the computation of the goal $\leftarrow A$ in P is expressed by the following property[1]. An SLD-derivation in CLP is a sequence $G_0, G_1, \ldots, G_k$ where each G_i is a goal $\leftarrow C, B_1, \ldots, B_m$, where C is a constraint and $B_1, \ldots, B_m$ are atoms. In a left-to-right computation, G_{i+1} is obtained by resolving B_1 with a program clause.

Property 1 (Correctness of query-answer transformation)**.** *Let P be a set of CHCs and A be an atom. Let P_A^{qa} be the query-answer program for P wrt. A. Then*

(i) if there is an SLD-derivation $G_0, \ldots, G_i$ where $G_0 = \leftarrow A$ and $G_i = \leftarrow C, B_1, \ldots, B_m$, then $P_A^{\mathsf{qa}} \models \forall(C|_{\mathsf{vars}(B_1)} \rightarrow B_1^{\mathsf{q}})$;

(ii) if there is an SLD-derivation $G_0, \ldots, G_i$ where $G_0 = \leftarrow A$, containing a sub-derivation $G_{j_1}, \ldots, G_{j_k}$, where $G_{j_i} \leftarrow C', B_1, B'$ and $G_{j_k} = \leftarrow C, B'$, then $P_A^{\mathsf{qa}} \models \forall(C|_{\mathsf{vars}(B_1)} \rightarrow B_1^{\mathsf{a}})$. (This means that the atom B_1 in G_{j_i} was successfully answered, with answer constraint $C|_{\mathsf{vars}(B_1)}$).

(iii) As a special case of (ii), if there is a successful derivation of the goal $\leftarrow A$ with answer constraint C then $P_A^{\mathsf{qa}} \models \forall(C \rightarrow A^{\mathsf{a}})$.

4.2 Over-approximation of the model of the query-answer program $P_{\mathsf{false}}^{\mathsf{qa}}$

The query-answer transformation of P with respect to false is computed. It follows from Property 1(iii) that if false is derivable from P then $P_{\mathsf{false}}^{\mathsf{qa}} \models \mathsf{false}^{\mathsf{a}}$. Convex polyhedral analysis of $P_{\mathsf{false}}^{\mathsf{qa}}$ yields an overapproximation of $M[\![P_{\mathsf{false}}^{\mathsf{qa}}]\!]$, say M', containing constrained facts for the query and answer predicates. These represent the calls and answers generated during all derivations starting from the goal false.

4.3 Strengthening the constraints in P

We use the information in M' to specialise the original clauses in P. Suppose M' contains constrained facts $p^{\mathsf{q}}(X) \leftarrow C^{\mathsf{q}}$ and $p^{\mathsf{a}}(X) \leftarrow C^{\mathsf{a}}$. If there is no constrained fact $p^*(X) \leftarrow C^*$ for some p^* then we consider M' to contain $p^*(X) \leftarrow \mathit{false}$. The clauses in P with head predicate p can be *strengthened* using the constraints C^{q} and C^{a}. Namely, for every clause $p(X) \leftarrow \mathscr{B}$ in P (assuming that the constrained facts are renamed to have the same variables X) the conjunction $C^{\mathsf{q}} \wedge C^{\mathsf{a}}$ are added to the body $\mathscr{B}$. The addition of C^{q} corresponds to propagating constraints "top-down" (via the calls) while the addition of

[1] Note that the model of P_A^{qa} might not correspond exactly to the calls and answers in the SLD-computation, since the CLP computation treats constraints as syntactic entities through decision procedures and the actual constraints could differ.

```
c1. false:- N>0,I=0,A=0,B=0, l(I,A,B,N).
c2. l(A,B,C,D) :- 2*A+ -1*B>=0,-1*A+1*D>0,-1*A+1*B>=0,3*A+ -1*B+ -1*C=0,
                  1*A+ -1*E= -1,l_body(B,C,F,G),l(E,F,G,D).
c3. l(A,B,C,D):- 3*A+ -3*D>0,1*D>0,2*A+ -1*B>=0,-3*A+3*D> -3,
                -1*A+1*B>=0,3*A+ -1*B+ -1*C=0.
c4. l(A,B,C,D):- false.
c5. l_body(A,B,C,D) :- -1*A+2*B>=0, 2*A+ -1*B>=0,
                       1*A+ -1*C= -1,1*B+ -1*D= -2.
c6. l_body(A,B,C,D) :- -1*A+2*B>=0,2*A+ -1*B>=0,1*A+ -1*C= -2,1*B+ -1*D= -1.
```

Figure 2: Example program *t4.pl* [6] with strengthened constraints

C^{a} represented propagation "bottom-up" (via the answers). Furthermore, note that $C^{\mathsf{a}} \rightarrow C^{\mathsf{q}}$ since the answers for p are always stronger than the calls to p. Thus it suffices to add the constraint C^{a} to $\mathscr{B}$.

Specialisation by strengthening the constraints preserves the answers of the goal with respect to which the query-answer transformation was performed. In particular, in our application we have the following property.

Property 2. *If P is a set of CHCs and P_{false} is the set obtained by strengthening the clause constraints as just described, then $P \models \mathsf{false}$ if and only if $P_{\mathsf{false}} \models \mathsf{false}$.*

The result of strengthening the constraints in Figure 1, using the query-answer program with respect to the goal `false`, is shown in Figure 2. Note that the constraint in clause `c4` is strengthened to `false`.

4.4 Analysis of the model of the specialised clauses

It may be that the clauses P_{false} do not contain a clause with head false. In this case safety is proven, since clearly $P_{\mathsf{false}} \not\models \mathsf{false}$. If this check fails, the convex polyhedral analysis is now run on the clauses P_{false}. As the experiments later show, safety is often provable by checking the resulting model; if no constrained fact for false is present, then $P_{\mathsf{false}} \not\models \mathsf{false}$. If safety is not proven, there are two possibilities: the approximate model is not precise enough, but P has a model, or there is a proof of false. To distinguish these we proceed to try to refine the clauses by splitting predicates.

5 Safety Check and Program Refinement

This section outlines a procedure for safety check, counterexample analysis and refinement. Refinement is considered when a proof of safety or an existence of a real counterexample (that is, a proof of false cannot be established.

Safety check and counterexample analysis The absence of a constrained fact for predicate false in the over-approximation proves that the given set of CHCs is safe. If safety can not be shown, our implementation of the convex polyhedron analysis produces a derivation tree for false as a trace term which we define formally below. For our program in Figure 1, the set of constrained facts representing the approximate model is shown below.

```
f1. l_body(A,B,C,D) :- 1*B+ -1*D>= -2,-1*B+1*D>=1,-1*A+2*B>=0, 2*A+ -1*B>=0,
                       1*A+1*B+ -1*C+ -1*D= -3.
```

```
f2. false :- true.
f3. l(A,B,C,D) :- 1*D>0,2*A+ -1*B>=0,-1*A+1*B>=0,-3*A+3*D> -3,
                  3*A+ -1*B+ -1*C=0.
```

Since there is a constrained fact for false, the shortest derivation for it is found, using clause c1 followed by clause c3. This will be represented as a *trace term* c1(c3), which is formally defined below. The idea of trace terms to capture the shape of derivations was introduced by Gallagher and Lafave [18].

AND-trees and trace terms. Each CHC is associated with an identifier, as shown in Figure 1. These identifiers are treated as constructors whose arity is the number of non-constraint atoms in the clause body. The following definitions of derivations and trace terms is adapted from [18].

An *AND-tree* is a tree each of whose nodes is labelled by an atom and a clause, such that

1. each non-leaf node is labelled by a clause $A \leftarrow C, A_1, \ldots, A_k$ and an atom A, and has children labelled by $A_1, \ldots, A_k$,
2. each leaf node is labelled by a clause $A \leftarrow C$ and an atom A.

We assume that the variables in node labels are renamed appropriately, details are not given here. Any finite derivation corresponds to an AND-tree, and each AND-tree T can be associated with a trace term $\mathsf{tr}(T)$ defined as:

1. c_j, if T is a single leaf node labelled by the clause of form $A \leftarrow C$ with identifier c_j; or
2. $c_i(\mathsf{tr}(T_1), \ldots, \mathsf{tr}(T_n))$, if T is labelled by the clause with identifier by c_i, and has subtrees $T_1, \ldots, T_n$.

A trace-term uniquely defines an AND-tree (up to renaming of variables). The set of constraints of an AND-tree, represented as $\mathsf{constr}(T)$ is

1. C, if T is a single leaf node labelled by the clause of form $A \leftarrow C$; or
2. $C \cup \bigcup_{i=1..n}(\mathsf{constr}(T_i))$ if T is labelled by the clause $A \leftarrow C, A_1, \ldots, A_k$ and has subtrees $T_1, \ldots, T_n$.

We say that an AND-tree T is satisfiable if $\mathsf{SAT}(\mathsf{constr}(T))$. Let T be an AND-tree whose root is labelled by atom A. Define $\mathsf{proj}(T)$ to be $\mathsf{constr}(T)|_{\mathsf{vars}(A)}$.

Interpolants. Given two sets of constraints C_1, C_2 such that $C_1 \cup C_2$ is unsatisfiable, a (Craig) interpolant is a constraint I with (1) $C_1 \subseteq I$, (2) $I \cup C_2$ is unsatisfiable and (3) I contains only variables common to C_1 and C_2. We implemented the algorithm from [31] for interpolants for linear constraints.

Given an AND-tree T where $\neg\mathsf{SAT}(\mathsf{constr}(T))$, we can construct an interpolant for each non-root node of T, also known as tree interpolants. Let T' be a sub-tree of T, whose root is labelled with A'. Then the interpolant I associated with A' is defined as above where $C_1 = \mathsf{constr}(T')$ and $C_2 = \mathsf{constr}(T) \setminus C_1$, and the interpolants of subtree of T' together with the constraints at the root of T' implies I. Note that by construction of the AND-tree, the only variables in common between C_1 and C_2 (and hence in I) are the variables in A', the label of T'. More details on tree interpolation can be found in [8].

The set $\mathsf{interpolant}(T)$ is the set of constrained facts $A \leftarrow I$, for all non-root nodes of T labelled by atom A with interpolant I as defined above.

Counterexample checking. Given a trace term, let T be the corresponding AND-tree. We report that the CHCs have no model if $\mathsf{SAT}(\mathsf{constr}(T))$, and our procedure terminates. For our example it can be verified that $\mathsf{SAT}(\mathsf{constr}(\mathtt{c1(c3)}))$ does not hold, so the trace c1(c3) is a false alarm. We now use the interpolants to split predicates and try to get a more precise approximation of the model.

From the trace term `c1(c3)` in the running example we derive $\mathsf{interpolant}(\texttt{c1(c3)}) = \{I\}$ where $I = \texttt{l(A,B,C,D)} \leftarrow \texttt{A} + -3 * \texttt{B} + \texttt{C} + \texttt{D} =< 0$.

We then split the constrained facts in the approximation of the model, using the corresponding interpolants and their negations. In the example we split constrained fact `f3` by strengthening its constraint with I and $\neg I$ respectively. Fioravanti *et al*. use a related technique for splitting clauses [16]. Strengthening first with I we get

```
l(A,B,C,D):- D>0,2*A+ -1*B>=0,-1*A+1*B>=0,-3*A+3*D> -3,
             3*A+ -1*B+ -1*C=0,A+ -3*B+C+D=<0
```

which after simplification becomes

```
l(A,B,C,D) :- -4*A+4*B+ -1*D>=0,1*D>0,-3*A+3*D> -3,2*A+ -1*B>=0,
              3*A+ -1*B+ -1*C=0.
```

We follow the same step with $\neg I$ and obtain the following set of constrained facts.

```
l(A,B,C,D) :- -4*A+4*B+ -1*D>=0,1*D>0,-3*A+3*D> -3,2*A+ -1*B>=0,
              3*A+ -1*B+ -1*C=0.
l(A,B,C,D) :- 4*A+ -4*B+1*D>0,-1*A+1*B>=0,-3*A+3*D> -3,2*A+ -1*B>=0,
              3*A+ -1*B+ -1*C=0.
```

These together with `f1` and `f2` give us a new set of constrained facts, which forms the input to the refinement phase of our procedure.

Refinement by Predicate Splitting. Refinement consists of obtaining a specialised set of CHCs from a given set of constrained facts and input set of CHCs. We do this by using polyvariant specialisation (PS) based on the method of multiple specialisation [32] with a property-based abstract domain based on the given set of constrained facts. PS is a program specialisation which introduces several new predicates corresponding to specialised versions of the same predicate. Polyvariant specialisation brings the expressive power of disjunctive predicates into the analysis [17]. Space does not permit a more detailed description. For our running example we obtain a split of the predicate `l` into `l_1` and `l_3` , and the specialised program is as follows.

```
false :-        1*A>0,1*B=0,1*C=0,1*D=0,l_3(B,C,D,A).
l_3(A,B,C,D) :- 2*A+ -1*B>=0,-1*A+1*B>=0,-1*A+1*D>0,4*A+ -4*B+1*D>0,
                3*A+ -1*B+ -1*C=0, A+ -1*E= -1,l_body_2(B,C,F,G),l_1(E,F,G,D).
l_3(A,B,C,D) :- 4*A+ -4*B+1*D>0,3*A+ -3*D>0,-1*A+1*B>=0,-3*A+3*D> -3,
                3*A+ -1*B+ -1*C=0.
l_1(A,B,C,D) :- 2*A+ -1*B>=0,-1*A+1*B>=0,-1*A+1*D>0,3*A+ -1*B+ -1*C=0,
                1*A+ -1*E= -1,l_body_2(B,C,F,G),l_1(E,F,G,D).
l_1(A,B,C,D) :- 3*A+ -3*D>0,2*A+ -1*B>=0,1*D>0,-1*A+1*B>=0,-3*A+3*D> -3,
                3*A+ -1*B+ -1*C=0.
l_body_2(A,B,C,D) :- 2*A+ -1*B>=0,-1*A+2*B>=0,1*A+ -1*C= -1,1*B+ -1*D= -2.
l_body_2(A,B,C,D) :- 2*A+ -1*B>=0,-1*A+2*B>=0,1*A+ -1*C= -2,1*B+ -1*D= -1.
```

The next iteration continues with this specialised program. The intention of splitting and PS is to guarantee progress of refinement, that is, a counterexample once eliminated never occurs again. Our procedure does not guarantee progress, that is, the same spurious counterexamples might appear in subsequent iterations, but in practice we find the polyvariant specialisation usually eliminates the given counterexample. The large number of constants in the above examples are derived during invariants computation. In the next iteration, our example terminates with a real counter example, thus proving our example program unsafe (over the real numbers).

Toolchain. Our verification procedure is summarised in Figure 3, which is divided into three parts, an *abstractor* (inside green dotted box), followed by a *safety check* and *counterexample analyser* and *refiner* (inside red box). It should be noted that the tools inside the green and red boxes produce new set of CHCs by specialisation.

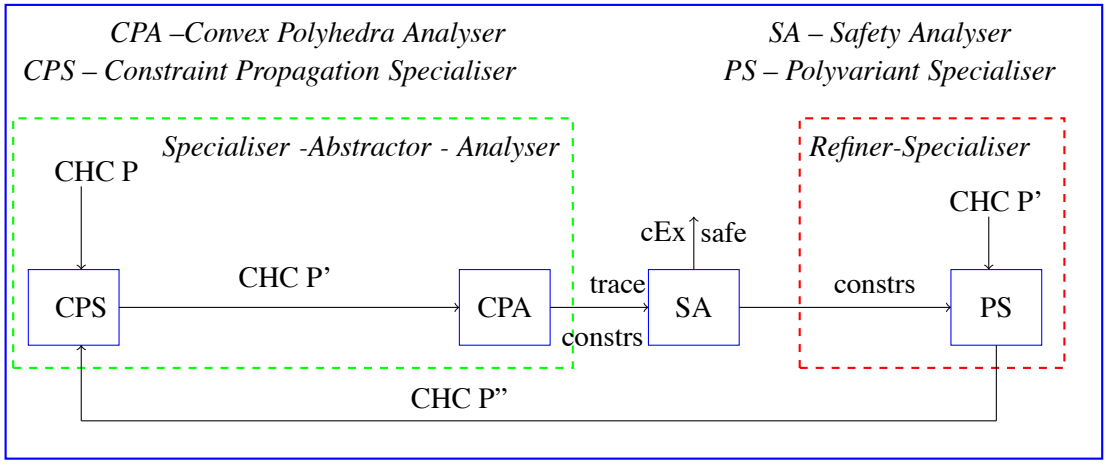


Figure 3: *CHC verification toolchain*.

The effects of CPA and PS in our procedure complement each other and the CPA model gets more accurate during refinement which allows generation of better specialised programs. In essence, it marries the effectiveness of CPA with PS.

6 Experiments

Table 1 presents the results of applying our toolchain depicted in Figure 3 to a number of benchmark programs taken from the repository of Horn clause benchmarks in SMT-LIB2[2] and other sources including [19, 26, 22, 5, 14]. The experiments were carried out using a computer, Intel(R) X5355 having 4 processors (each @ 2.66GHz) and total memory of 6 GB. Debian 5 (64 bit) is the Operating System running in it and we set 2 minutes of timeout for each experiment. Our tool-chain is implemented in 32-bit Ciao Prolog [9][3] and the Parma Polyhedra Library [1][4] for this purpose.

In Table 1, columns Program, "n" , Result and time (sec) respectively represent the benchmark program, the number of refinement iterations necessary to verify a given property, the results of verification and the time (in seconds) to verify them. Value 0 in column "n" means that no refinement is necessary, whereas value greater than 0 indicates the actual number of iterations necessary and value "-" means that these programs are beyond the reach of our current tool within the given time limit. Problems marked with (*) were not handled by our tool-chain since their solution generates numbers which do not fit in 32 bits, the limit of our Ciao Prolog implementation. Problems such as systemc-token-ring.01-safeil.c contain complicated loop structure with large strongly connected components in the predicate dependency graph and our convex polyhedron analysis tool is unable to derive the required invariant.

[2] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/
[3] http://ciao-lang.org/
[4] http://bugseng.com/products/ppl/

Program	n	Result	time (secs)	Program	n	Result	time (secs)
MAP-disj.c.map.pl	0	safe	1.0	jaffex1c.pl	0	safe	0.01
MAP-forward.c.map.pl	0	safe	1.0	jaffex1a.pl	0	safe	0.01
t1.pl	0	safe	0.01	qrdcmp.smt2	0	safe	118.0
t1-a.pl	0	safe	0.01	choldc.smt2	0	safe	19.0
t2.pl	0	safe	0.01	lop.smt2	0	safe	39.0
t3.pl	0	safe	1.0	pzextr.smt2	0	safe	40.0
t4.pl	1	unsafe	1.0	qrsolv.smt2	0	safe	18.0
t5.pl	0	safe	0.01	tridag.smt2	0	safe	13.0
MAP-disj.c-scaled.pl	0	safe	1.0	systemc-pc-sfifo_1	0	unsafe	12.0
INVGEN-id-build	0	safe	1.0	loops-terminator	0	unsafe	0.01
INVGEN-nested5	0	safe	1.0	loops-for-bounded	3	unsafe	5.0
INVGEN-nested6	0	safe	117.0	TRACER-testabs15	0	safe	1.0
INVGEN-nested8	0	safe	1.0	INVGEN-apache-esc-abs	0	safe	2.0
INVGEN-svdsomeloop	0	safe	3.0	DAGGER-barbr.map.c	0	safe	119.0
INVGEN-svd1	2	safe	13.0	systemc-token-ring.01-safeil.c	-	?	-
INVGEN-svd4	0	safe	5.0	sshs3-srvr1a-safeil.c(*)	-	?	-
loops-count-up-down	0	unsafe	1.0	sshs3-srvr1b-safeil.c	-	?	-
loops-sum04	8	unsafe	2.0	amebsa.smt2	-	?	-
dfpp12.pl	0	safe	0.01	bandec.smt2(*)	-	?	-
TRACER-testloop27	1	unsafe	1.0	TRACER-testloop28	-	?	-
TRACER-testloop8	0	unsafe	0.01	crank.smt2	-	?	-
jaffex1b.pl	0	safe	0.01	pldi12.pl	-	?	-
jaffex1d.pl	0	safe	0.01	loops-sum01	-	?	-

Table 1: Experimental results on CHC benchmark problems

The results of our procedure in a larger set of benchmarks obtained from previous sources are summarised in Table 2. Though our tool-chain is not optimized at all, the overall result shows that it compares favourably with other advanced verification tools like HSF [20], VeriMAP [14], TRACER [25] etc. in both time and the number of problems solved, and thus showing the effectiveness of our approach.

	without refinemet	with refinement
solved (safe/unsafe)	160 (142/18)	181 (158/23)
unknown/ timeout	49/7	-/35
total time	1293	3410
average time (secs)	5.98	18.73

Table 2: Experimental results on 216 CHC verification problems, where "-" means not relevant.

7 Related Work

Verification of CLP programs using abstract interpretation and specialisation has been studied for some time. The use of an over-approximation of the semantics of a program can be used to establish safety properties – if a state or property does not appear in an over-approximation, it certainly does not appear in the actual program behaviour. A general framework for logic program verification through abstraction was described by Levi [29].

The use of program transformation to verify properties of logic programs was pioneered by Pettorossi and Proietti [30] and Leuschel [28]. Transformations that preserve the minimal model (or other suitable models) of logic programs are applied systematically to make properties explicit. For example, if a program can be transformed to one containing a clause $A \leftarrow true$ then A is a consequence of the program.

Recent work by De Angelis *et al*. [13, 14] applies a specialisation approach to the Horn clause verification problem as discussed here, namely, with integrity constraints expressing the properties to be proved. Both our approach and theirs repeatedly apply specialisations preserving the property to be proved. However the difference is that their specialisation techniques are based on unfold-fold transformations, with a sophisticated control procedure controlling unfolding and generalisation. Our specialisations are restricted to strengthening of constraints or polyvariant splitting based on local conditions. Their test for success or failure is a simple syntactic check, whereas ours is based on an abstract interpretation to derive an over-approximation.

Counterexample guided abstraction refinement (CEGAR) [10] has been successfully used in verification to automatically refine (predicate) abstractions to reduce false alarms but not much has been explored in refining abstractions in the convex polyhedral domain. See [7, 21] for more details about the use of interpolation in refinement. A number of tools implementing predicate abstraction and refinement are available, such as HSF [20] and BLAST [2]. TRACER [19] is a verification tool based on CLP that uses symbolic execution.

Informally one can say that approaches differ in where the "hard work" is performed. In the work of De Angelis *et al*. the specialisation procedure is the core, whereas in the CEGAR approaches the refinement step is crucial, and interpolation plays a central role. In our approach, by contrast, most of the hard work is done by the abstract interpretation, which finds useful invariants as well as propagating constraints globally. The main problem is to find effective ways of refining polyhedral abstractions. Finding the most effective balance between specialisation, abstraction and refinement techniques is a matter of ongoing research.

8 Conclusion and Future works

We described an iterative procedure for Horn clause verification which interleaves abstract interpretation with specialisation. A specialised set of CHCs is produced first by strengthening the constraints in the given clauses using the results of the abstract interpretation. Then the procedure terminates if an abstract interpretation of the resulting program is sufficient to verify the required properties, otherwise, a polyvariant specialisation guided by an abstract counterexample is performed using the inferred constraints as well as interpolated constraints.

In the future, we would like to find a way of ensuring progress of refinement, maybe using the powerset polyhedra domain, and also interface our toolchain with SMT solvers for satisfiability checking and interpolant generation.

References

[1] R. Bagnara, P. M. Hill & E. Zaffanella (2008): *The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems*. *Science of Computer Programming* 72(1–2), pp. 3–21. Available at http://dx.doi.org/10.1016/j.scico.2007.08.001.

[2] T. Ball, V. Levin & S. K. Rajamani (2011): *A decade of software model checking with SLAM*. *Commun. ACM* 54(7), pp. 68–76. Available at http://doi.acm.org/10.1145/1965724.1965743.

[3] F. Bancilhon, D. Maier, Y. Sagiv & J. Ullman (1986): *Magic Sets and other strange ways to implement logic programs*. In: *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*. Available at http://dx.doi.org/10.1145/6012.15399.

[4] F. Benoy & A. King (1996): *Inferring Argument Size Relationships with CLP(R)*. In J. P. Gallagher, editor: *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, *Springer-Verlag Lecture Notes in Computer Science* 1207, pp. 204–223. Available at http://dx.doi.org/10.1007/3-540-62718-9_12.

[5] D. Beyer (2013): *Second Competition on Software Verification - (Summary of SV-COMP 2013)*. In N. Piterman & S. A. Smolka, editors: *TACAS*, *Lecture Notes in Computer Science* 7795, Springer, pp. 594–609. Available at http://dx.doi.org/10.1007/978-3-642-36742-7_43.

[6] D. Beyer, T. A. Henzinger, R. Majumdar & A. Rybalchenko (2007): *Path invariants*. In J. Ferrante & K. S. McKinley, editors: *PLDI*, ACM, pp. 300–309. Available at http://doi.acm.org/10.1145/1250734.1250769.

[7] N. Bjørner, K. L. McMillan & A. Rybalchenko (2013): *On Solving Universally Quantified Horn Clauses*. In F. Logozzo & M. Fähndrich, editors: *SAS*, *Lecture Notes in Computer Science* 7935, Springer, pp. 105–125. Available at http://dx.doi.org/10.1007/978-3-642-38856-9_8.

[8] R. Blanc, A. Gupta, L. Kovács & B. Kragl (2013): *Tree Interpolation in Vampire*. In K. L. McMillan, A. Middeldorp & A. Voronkov, editors: *LPAR*, *Lecture Notes in Computer Science* 8312, Springer, pp. 173–181. Available at http://dx.doi.org/10.1007/978-3-642-45221-5_13.

[9] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García & G. Puebla (1997): *The Ciao Prolog system. Reference manual*. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM). Available from http://www.clip.dia.fi.upm.es/.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu & H. Veith (2003): *Counterexample-guided abstraction refinement for symbolic model checking*. *J. ACM* 50(5), pp. 752–794. Available at http://doi.acm.org/10.1145/876638.876643.

[11] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In R. M. Graham, M. A. Harrison & R. Sethi, editors: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, ACM, pp. 238–252. Available at http://dl.acm.org/citation.cfm?id=512950.

[12] P. Cousot & N. Halbwachs (1978): *Automatic Discovery of Linear Restraints Among Variables of a Program*. In A. V. Aho, S. N. Zilles & T. G. Szymanski, editors: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, ACM Press, pp. 84–96. Available at http://dl.acm.org/citation.cfm?id=512760.

[13] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *Program verification via iterated specialization*. *Sci. Comput. Program.* 95, pp. 149–175. Available at http://dx.doi.org/10.1016/j.scico.2014.05.017.

[14] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations*. In E. Ábrahám & K. Havelund, editors: *TACAS*, *Lecture Notes in Computer Science* 8413, Springer, pp. 568–574. Available at http://dx.doi.org/10.1007/978-3-642-54862-8_47.

[15] S. Debray & R. Ramakrishnan (1994): *Abstract Interpretation of Logic Programs Using Magic Transformations*. *Journal of Logic Programming* 18, pp. 149–176. Available at http://dx.doi.org/10.1016/0743-1066(94)90050-7.

[16] F. Fioravanti, A. Pettorossi & M. Proietti (2002): *Specialization with Clause Splitting for Deriving Deterministic Constraint Logic Programs*. In: *In Proc. IEEE Conference on Systems, Man and Cybernetics, Hammamet*, IEEE Press. Available at http://dx.doi.org/10.1109/ICSMC.2002.1167971.

[17] F. Fioravanti, A. Pettorossi, M. Proietti & V. Senni (2013): *Controlling Polyvariance for Specialization-based Verification*. *Fundam. Inform.* 124(4), pp. 483–502. Available at http://dx.doi.org/10.3233/FI-2013-845.

[18] J. P. Gallagher & L. Lafave (1996): *Regular Approximation of Computation Paths in Logic and Functional Languages*. In O. Danvy, R. Glück & P. Thiemann, editors: *Partial Evaluation*, *Springer-Verlag Lecture Notes in Computer Science* 1110, pp. 115–136. Available at http://dx.doi.org/10.1007/3-540-61580-6_7.

[19] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard & P. J. Stuckey (2013): *Failure tabled constraint logic programming by interpolation*. *TPLP* 13(4-5), pp. 593–607. Available at http://dx.doi.org/10.1017/S1471068413000379.

[20] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea & A. Rybalchenko (2012): *HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution)*. In C. Flanagan & B. König, editors: *TACAS*, *LNCS* 7214, Springer, pp. 549–551. Available at http://dx.doi.org/10.1007/978-3-642-28756-5_46.

[21] A. Gupta, C. Popeea & A. Rybalchenko (2011): *Solving Recursion-Free Horn Clauses over LI+UIF*. In H. Yang, editor: *APLAS*, *Lecture Notes in Computer Science* 7078, Springer, pp. 188–203. Available at http://dx.doi.org/10.1007/978-3-642-25318-8_16.

[22] A. Gupta & A. Rybalchenko (2009): *InvGen: An Efficient Invariant Generator*. In A. Bouajjani & O. Maler, editors: *CAV*, *Lecture Notes in Computer Science* 5643, Springer, pp. 634–640. Available at http://dx.doi.org/10.1007/978-3-642-02658-4_48.

[23] N. Halbwachs, Y. E. Proy & P. Raymound (1994): *Verification of Linear hybrid systems by means of convex approximations*. In: *Proceedings of the First Symposium on Static Analysis*, *LNCS* 864, pp. 223–237. Available at http://dx.doi.org/10.1007/3-540-58485-4_43.

[24] J. Jaffar & M. Maher (1994): *Constraint Logic Programming: A Survey*. *Journal of Logic Programming* 19/20, pp. 503–581. Available at http://dx.doi.org/10.1016/0743-1066(94)90033-7.

[25] J. Jaffar, V. Murali, J. A. Navas & A. E. Santosa (2012): *TRACER: A Symbolic Execution Tool for Verification*. In P. Madhusudan & S. A. Seshia, editors: *CAV*, *Lecture Notes in Computer Science* 7358, Springer, pp. 758–766. Available at http://dx.doi.org/10.1007/978-3-642-31424-7_61.

[26] J. Jaffar, J. A. Navas & A. E. Santosa (2011): *Unbounded Symbolic Execution for Program Verification*. In S. Khurshid & K. Sen, editors: *RV*, *Lecture Notes in Computer Science* 7186, Springer, pp. 396–411. Available at http://dx.doi.org/10.1007/978-3-642-29860-8_32.

[27] L. Lakhdar-Chaouch, B. Jeannet & A. Girault (2011): *Widening with Thresholds for Programs with Complex Control Graphs*. In T. Bultan & P.-A. Hsiung, editors: *ATVA 2011*, *Lecture Notes in Computer Science* 6996, Springer, pp. 492–502. Available at http://dx.doi.org/10.1007/978-3-642-24372-1_38.

[28] M. Leuschel & T. Massart (1999): *Infinite State Model Checking by Abstract Interpretation and Program Specialisation*. In A. Bossi, editor: *LOPSTR'99*, *Lecture Notes in Computer Science* 1817, Springer, pp. 62–81. Available at http://dx.doi.org/10.1007/10720327_5.

[29] G. Levi (2000): *Abstract Interpretation Based Verification of Logic Programs*. *Electr. Notes Theor. Comput. Sci.* 40, p. 243. Available at http://dx.doi.org/10.1016/S1571-0661(05)80052-0.

[30] A. Pettorossi & M. Proietti (2000): *Perfect Model Checking via Unfold/Fold Transformations*. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv & P. J. Stuckey, editors: *Computational Logic*, *Lecture Notes in Computer Science* 1861, Springer, pp. 613–628. Available at http://dx.doi.org/10.1007/3-540-44957-4_41.

[31] A. Rybalchenko & V. Sofronie-Stokkermans (2010): *Constraint solving for interpolation*. J. Symb. Comput. 45(11), pp. 1212–1233. Available at http://dx.doi.org/10.1016/j.jsc.2010.06.005.

[32] W. H. Winsborough (1989): *Path-Dependent Reachability Analysis for Multiple Specialization*. In E. L. Lusk & R. A. Overbeek, editors: *NACLP*, MIT Press, pp. 133–153. Available at http://dblp.uni-trier.de/db/conf/slp/slp89.html#Winsborough89.