UNIVERSIDAD POLITÉCNICA DE
MADRID
FACULTAD DE INFORMÁTICA

Un Marco Unificado para Análisis de Recursos y Tiempo de Ejecución, Validación Dinámica y Pruebas Unitarias

(A Unified Framework for Resource and Execution Time Analysis, Run-Time Checking and Unit-Testing)

Tesis Doctoral

Edison Fernando Mera Menéndez
Ingeniero Matemático
Homologado en España a
Licenciado en Matemáticas

Noviembre 2010

Departamento de Inteligencia Artificial
Facultad de Informática

Un Marco Unificado para Análisis de Recursos y Tiempo de Ejecución, Validación Dinámica y Pruebas Unitarias

Candidato: **Edison Fernando Mera Menéndez**

Ingeniero Matemático
Escuela Politécnica Nacional, Ecuador
Homologado en España a
Licenciado en Matemáticas

Director: **Pedro López García**

Doctor en Informática
Licenciado en Informática
Universidad Politécnica de Madrid

Madrid, Noviembre 2010

UNIVERSIDAD POLITECNICA DE MADRID

Tribunal nombrado por el Magfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día. . . . . de. . . . . de 201. . .

Presidente: _____

Vocal: _____

Vocal: _____

Vocal: _____

Secretario: _____

Suplente: _____

Suplente: _____

Realizado el acto de defensa y lectura de la Tesis el día . . . . . de. . . . . de 201. . . en la Facultad de Informática.

EL PRESIDENTE LOS VOCALES

EL SECRETARIO

A mi familia

Agradecimientos

Quiero dar las gracias especialmente a mi director de tesis, Pedro López, y al director del Grupo de Investigación CLIP, Manuel Hermenegildo, quienes siempre han creído en mí en los momento más críticos del desarrollo de la presente tesis, por todo el tiempo y recursos que siempre me han brindado para llevar a buen término el desarrollo del presente trabajo, y sus oportunos consejos tanto en el ámbito científico y profesional, como en el personal. A Francisco Bueno, quien fue la primera persona del grupo CLIP con la que tuve contacto y a través de la cual me integré formalmente en el grupo. A Daniel Cabeza, Manuel Carro y Germán Puebla, quienes sobre todo al inicio de mis estudios tuvieron el tiempo y la paciencia para resolver mis dudas. También a otros investigadores del grupo con quienes he compartido tiempo y jornadas de trabajo, como son Elvira Albert, Puri Arenas, Samir Genaim, Susana Muñoz, Damiano Zanardini, Jaime Lipton y Nik Swoboda.

Quisiera también mencionar a mis compañeros de doctorado, con algunos de los cuales inicié mis estudios, y otros que se fueron incorporando posteriormente, quienes siempre me han brindado la ayuda y soporte necesario, como son Pablo Chico, Jesús Correas, Jose Manuel Gómez, Dragan Ivanovic, José Morales, Claudio Ochoa, Diana Ramírez, David Trallero, Teresa Trigo, Claudio Vaucheret, Miguel Gómez-Zamalloa y en particular, a Jorge Navas, quien fue coautor de uno de los artículos publicados. A los miembros y exmiembros del personal administrativo y de soporte, Astrid Beascoa, Rosa Padilla, Alberto García Pañoso y Juan Cespedes. En el tiempo que he estado aquí he tenido la oportunidad de conocer algunos investigadores externos, como John Gallager, María García de la Banda, Pawel Pietrzak y Peter J. Stuckey. Pido disculpas si he dejado de mencionar a alguna persona, ya que no dispongo de espacio suficiente para referirme a todos aquellos que de alguno u otro modo han influido en el desarrollo de la presente tesis.

Por último, quiero agradecer a mis padres que me han dado todo, incluso lo que les hacía falta para que yo pudiera llegar hasta aquí.

Sinopsis

Hemos desarrollado un marco general para inferir automáticamente cotas superiores e inferiores del uso que un programa lógico hace de los recursos en general, dadas como funciones de los tamaños de los datos de entrada. Este permite el tratamiento de recursos independientes de la plataforma (o *definidos por el usuario/dependientes de la aplicación*), tales como los bits enviados o recibidos por una aplicación a través de un *socket*, el número de llamadas a un predicado, archivos que se dejan abiertos, accesos a una base de datos, así como otros dependientes de la plataforma, como tiempo de ejecución o consumo de energía. El trabajo incluye un análisis global paramétrico respecto a los recursos y el tipo de aproximación (cotas superiores e inferiores). El usuario puede definir los parámetros del análisis para un recurso mediante aserciones, así como asociar costes a las operaciones básicas del programa que afectan el uso de dicho recurso. El análisis estático global infiere el uso del recurso para todas las partes del programa. Las aserciones pueden definirse a diferentes niveles de abstracción. Por ejemplo, pueden asociar funciones del uso de recursos para diferentes tipos de programas a nivel del código fuente y pueden describir también cómo se actualiza el valor de dichos recursos en las cabezas de los predicados o en la preparación de un literal en el cuerpo de dichos predicados. En este caso, el analizador usa una función de coste definida en la aserción para actualizar el uso del recurso mientras se analizan las cabezas de las cláusulas o los literales del cuerpo.

Para los recursos dependientes de la plataforma, como el tiempo de ejecución, realizamos una única vez por plataforma un perfilado que calcula los parámetros asociados a operaciones básicas, a nivel de código fuente o *bytecode*. Hemos aplicado el marco general a tiempo de ejecución de dos maneras y experimentado con la información suministrada a nivel de fuente y de *bytecode*. En el primer enfoque, el análisis estático devuelve un vector de recursos independiente de la plataforma relacionado con las operaciones de bajo nivel de la ejecución del programa. Dichas operaciones deben verse reflejadas en el lenguaje de alto nivel. El perfilador calcula las constantes que aparecen

en las funciones de recursos de la plataforma dada. A continuación usamos aserciones para definir el tiempo de ejecución como un recurso compuesto de los recursos independientes de la plataforma y los resultados del perfilado. En el segundo enfoque, en la etapa de perfilado se calculan las constantes y funciones que acotan el tiempo de ejecución de cada instrucción de la máquina abstracta. A continuación, en la etapa de estimación de recursos se emplea la información de dichos tiempos para inferir cotas del tiempo de ejecución dependientes de la plataforma. También el resultado puede mejorarse introduciendo aserciones a nivel de *bytecode*.

Además, dado que no podemos verificar todas las propiedades estáticamente, presentamos un marco unificado para verificación estática, validación dinámica (o en tiempo de ejecución) y pruebas unitarias. Hemos diseñado métodos para compilar validaciones dinámicas de (parte de) las aserciones que no pueden ser verificadas estáticamente. Las pruebas unitarias permiten poner a prueba las validaciones dinámicas y (parte de) las pruebas verificadas estáticamente se eliminan. Además de las propiedades relacionadas con recursos, podemos tratar otras como la ausencia de fallo, el determinismo y las de estado (o funcionales) como los tipos de los argumentos de entrada/salida.

Una contribución importante es que para todas las tareas hemos usado un lenguaje de aserciones unificado, el cual permite definir recursos y propiedades relacionadas y verificables con ayuda de los resultados del análisis que es lo suficientemente poderoso, general y extensible como para expresar una gran variedad de propiedades interesantes de los programas.

Entre las aplicaciones del presente trabajo tenemos la verificación del consumo de recursos, depuración del rendimiento, certificación de propiedades para código móvil, control de granularidad en computación paralela/distribuida y especialización de programas orientada por los recursos. El marco para pruebas unitarias y en tiempo de ejecución se ha aplicado con éxito en la validación de la adecuación al estándar ISO-Prolog y en la detección de varios errores en el código fuente del sistema `Ciao`. El marco completo ha sido integrado con éxito en el sistema `Ciao`/`CiaoPP`.

UNIVERSIDAD POLITÉCNICA DE
MADRID
FACULTAD DE INFORMÁTICA

A Unified Framework for Resource and Execution Time Analysis, Run-Time Checking and Unit-Testing

PhD Thesis

Edison Fernando Mera Menéndez
November 2010

Artificial Intelligence Department
Computer Science School

A Unified Framework for Resource and Execution Time Analysis, Run-Time Checking and Unit-Testing

PhD Candidate: **Edison Fernando Mera Menéndez**

Mathematical Engineer
Escuela Politécnica Nacional, Ecuador
Convalidated in Spain to
Graduate in Mathematics

Advisor: **Pedro López García**

Doctor in Computer Science
Graduate in Computer Science
Universidad Politécnica de Madrid

Madrid, November 2010

To my family

Acknowledgements

First of all, I want to express my gratitude to my thesis advisor, Pedro López, and to the director of the CLIP Research Group, Manuel Hermenegildo, who always have supported me, even in the hardest moments of the development of this thesis, for all their time, energy and resources that have allowed me to finish this thesis, and also for their good suggestions in the scientific, professional and also personal field. I would also like to express my gratitude to Francisco Bueno, who was the first person of the CLIP group that I met, and who introduced me formally as member of the group. To Daniel Cabeza, Manuel Carro, and Germán Puebla, who mainly at the beginning of my studies had the time and patience to solve my doubts. Also to other researchers of the CLIP group who shared time with me: Elvira Albert, Samir Genaim, Damiano Zanardini, Puri Arenas, Susana Muñoz, Jaime Lipton and Nik Swoboda.

I would like to mention my doctoral fellows, some of them I started my studies with, and others started its studies further, who always gave me support and help when needed: Pablo Chico, Jesús Correas, Jose Manuel Gómez, Dragan Ivanovic, José Morales, Claudio Ochoa, Diana Ramírez, David Trallero, Teresa Trigo, Claudio Vaucheret, Miguel Gómez-Zamalloa and in particular, Jorge Navas, who was coauthor in one of the published papers. To the members and former members of the administrative and support staff, Astrid Beascoa, Rosa Padilla, Alberto García Pañoso and Juan Cespedes.

During these years I have had the opportunity to meet many researchers around the world such as Pawel Pietrzak, María García de la Banda, Peter J. Stuckey and John Gallager. My apologizes if I forget to mention any person, since I do not have enough space here to thank all people who in one way or another have influenced the development of this thesis.

Finally, thank you very much to my parents, who gave me everything, even what they did not have to make me reach this point.

Abstract

We have developed a general framework for automatically inferring both upper- and lower-bounds on the usage that a logic program makes of resources in general. Such bounds are given as functions of input data sizes. Our approach gives support for platform-independent (or *user-defined/application-dependent*) resources, such as bits sent or received by an application over a socket, number of calls to a predicate, number of files left open, number of accesses to a database, as well as platform-dependent resources, such as execution time or energy. The framework includes a global analysis that is parametric with respect to resources and the type of approximation (lower- and upper-bounds). The user can define the parameters of the analysis for a particular resource using assertions. It is also possible to associate basic cost functions with elementary program operations that affect the usage of such resource. Then, the global static analysis can infer the cost of all the procedures in the program. The assertions can be defined at different abstraction levels. For example, they can associate resource usage functions to different program constructs at source code level, can also describe how predicates update the value for those resources in the clause heads or in the preparation of the execution of its body literals. In such a case, a cost function defined in the assertion is used by the analyzer to update the resource usage when the clause heads or the body literals are analyzed.

For platform-dependent resources (e.g., execution time) we perform a one-time profiling phase that computes parameters associated to basic operations, at source or bytecode-level. We have applied the general framework to execution time estimation in two ways and experimented with information supplied at source and bytecode-levels. In the first approach, the compile-time cost bounds analysis gives a vector of platform-independent resources that corresponds to particular low-level operations related to program execution. Such operations must be reflected in the high-level language. The

profiling phase determines the values of the constants appearing in the resource usage functions, for a given platform. Then, we use assertions to define the platform-dependent resource (execution time) as a composition of the basic platform-independent resources and the values of the constants resulting from the profiling phase. In the second approach, the one-time profiling stage calculates constants and functions bounding the execution time of each abstract machine instruction. Then, the compile-time resource usage estimation phase uses the instruction timing information (which is platform-dependent) to infer bounds on execution time. To improve precision, resource usage assertions can also be given at bytecode level.

Furthermore, since not all properties can be verified statically, we have developed a framework that unifies static verification, run-time checking and unit testing. In that sense, we have designed methods for compiling run-time checks for (parts of) assertions which cannot be verified at compile-time. Unit tests also provide test cases for the run-time checks, and (parts of) unit tests that can be verified at compile-time are eliminated. In addition to those resource-related properties, we support other properties like non-failure, determinism and state (or functional) properties like types of input/output arguments on calls or successes.

A key contribution of this work is that we preserve the use of a unified assertion language for all tasks. Such language is used to define resources and resource-related properties that can be verified based on the results of the analysis and is powerful, general and extensible enough to express a large class of interesting properties.

Applications of this work include resource usage verification, performance debugging, certification of resource usage properties in mobile code, resource and granularity control in parallel/distributed computing and resource-oriented specialization. The unit-testing and run-time checking framework has been effectively applied to the validation of ISO Prolog compliance and to the detection of different types of bugs in the `Ciao` source. The overall unified framework has been successfully integrated in the `Ciao/CiaoPP` system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is well recognized the importance of inferring information about the costs of computations for a variety of applications. These costs are usually expressed as execution steps and, sometimes, execution time or memory.

However, there are certain applications where it is interesting to be able to infer the usage that a program does of a wide variety of resources, including resources that are application-dependent, and thus, must be definable by the user. Applications of this analysis includes (but are not limited to) resource usage verification and debugging, certification of resource consumption in mobile code, proof carrying code, resource/granularity control in parallel/distributed computing, or resource-oriented specialization. Examples of interesting resources are bits sent or received by an application over a socket, number of calls to a procedure, number of files left open, number of accesses to a database, energy consumed, monetary units spent, disk space used, etc. Thus, a challenge in cost analysis is to develop a general analysis framework which is parametric to the type of resource inferred, and specially, offers support for user-defined resources.

A particularly interesting resource is execution time. This can be easily argued if we focus our attention on safety critical embedded and real-time applications, which have become pervasive in areas of high economic impact, like transportation, consumer electronics or mobile telephony. For example,

in control systems of automobiles, aircrafts and trains, or systems for highly remote operation (satellite, space, etc.), the computation must be correctly performed within its time constraints (and also with an adequate use of resources). Thus, it is of crucial importance to ensure ahead of time, that these constraints are met for all possible executions, which in turn depend on the safe static estimation of execution time. Of course, this estimation must, at the same time, be as accurate as possible because of cost effectiveness reasons, or hardware requirements.

The problem of resource usage inference becomes more complex for execution time, and in general, for resources that are dependent on characteristics of the platform where programs are executed. For example, the execution time is closely related to the number of execution steps (a platform-independent feature), but also on platform characteristics such as type of processor, clock frequency, cache behavior, etc. There are proposals to obtain accurate estimates of the number of execution steps as a function of input data size, however, it is not straightforward to turn such steps into execution time. There are also works on *worst case execution times* (WCET), in the context of high-level imperative programming languages. Although the WCET approach takes into account the low level platform characteristics mentioned above, it only provides numeric upper-bounds on execution time (i.e., a constant numeric value, but not a function on input data sizes), and often requires annotating loops manually to express an upper-bound on the number of iterations. Thus, another challenge is to develop a safe and accurate execution time analysis that brings together the best of these two approaches, taking into account the low-level platform-dependent factors, and the high-level estimation of the number of iterations as a function on input data sizes.

Finally, one of the most promising applications of resource analysis is the automatic verification of resource usage related properties. However, not all the properties can be verified statically. Thus, some run-time checking is needed. A way of performing (at least) an (incomplete) verification of

those properties, consists of generating unit-tests for the program. Such tests express the input data, the expected output, the number of times that the unit-tests should be repeated, as well as other commands or properties related with the tests. Although there are several approaches for unit-testing, none of them integrates compile-time verification, run-time checking and unit-testing. This is also another challenge to be addressed.

State of the Art

Cost analysis has been studied for several declarative languages [9, 29, 21, 22, 23], in order to infer the complexity in terms of execution steps of a given program. In the context of functional languages, there are systems like ACE [40], which can automatically extract upper-bounds on execution steps for a subset of functional programming. The system is based on program transformation. The original program is transformed into a step-counting version and then into a composition of a cost bound and a measure function. Rosendahl defines in [61] another automatic upper-bound analysis based on an abstract interpretation of a step-counting version. The analysis measures both execution time and execution steps. However, size measures cannot automatically be inferred and the experimental section shows few details about the practicality of the analysis. In [52], a complexity analysis is presented, based in the abstract interpretation of a modified version that count the number of execution steps.

In the context of logic programming, [22, 21] present a method for automatically inferring functions which captures an upper-bound on the number of resolution steps or reductions that a procedure will execute as a function of the size of its input data. In [43, 41], the method of [22, 41] was fully automated in the context of a practical compiler and in [23, 13, 41] a similar approach was applied in order to also obtain lower-bounds, which are specially relevant in parallel execution. This approach captures well the fact that program execution cost in general depends on properties of the input

data (such as input data sizes), so that the inferred resource usage bounds are given as *functions on the size (or values) of input data*.

Regarding the particular case where the resource to be inferred is execution time, there are works on the so called *worst case execution time* (WCET) in the context of imperative languages and for different application domains, that have produced precise timing models (see, e.g., [68, 63, 7, 27] and its references). In particular [38, 7] proposes a portable WCET analysis for Java. It considers platform-independent features and takes advantage of the abstract machine-based implementation of such a language. However these and related methods do not infer cost functions of input data sizes but rather absolute maximum execution times, and they generally require the manual annotation of loops to express an upper-bound on the number of iterations. Moreover, most of the approaches are highly tied to either a particular language or architecture, and none of them estimate lower-bounds.

Moreover, none of the mentioned approaches provide a general analysis framework which offers support for user-defined resources, such as number of accesses to a database, number of SMS sent or monetary units spent, as well as the traditional execution time or memory. This is an interesting challenge, since, as already mentioned, for certain current applications, it is very useful or necessary to infer the usage that a program does of a wide variety of resources.

Regarding the run-time checking of assertions in logic programs, to our knowledge, there is not too much work in this area. Previous work has concentrated mostly on the static (i.e., compile-time) checking of properties (which comprises *static verification* and *static debugging* [12, 33, 54, 55, 34]), as well as on techniques for reducing at compile-time the number of checks that have to be performed dynamically (i.e., at run-time): any assertions present in the program are verified (or falsified) as much as possible during the compilation phase, since compile-time checking is preferable to run-time checking –always incomplete as a means of verification. However the existence in all practical programs of data only known at run-time and the

rich nature of the properties considered, make a certain degree of run-time checking inevitable –a reasonable price to pay in return for property expressiveness.

Regarding unit-testing, there are previous works in this area, e.g., [8], [69], or the framework included in SWI-Prolog [67], called `plunit`, which also runs on SICStus Prolog and provides a portable testing framework. In the SWI-Prolog unit-testing framework, unit-test specifications are written in the same source code module or in a different file with the same name as the module being tested, but the framework does not allow to write unit-test specifications in the same module that contains the predicates being tested. Finally, none of the above mentioned unit-testing frameworks smoothly integrate compile-time verification, run-time checking and unit-testing, unlike our approach, in wich only *test drivers* need to be added because the existing run-time assertion checking machinery is used as a checker for the cases defined by the unit tests.

Thesis Objectives

The overall objective of this thesis is the development of a general framework for resource analysis, integrated with unit-testing, run-time checking and static verification, and provide instantiations of the framework for execution time estimation. In order to fulfil such an objective, the thesis will provide solutions to the challenges previously mentioned, focusing on the following tasks or sub-objectives:

1. Implementing and integrating all the developed techniques in an advanced program development environment, namely the `CiaoPP`/ `Ciao` system, in order to assess their feasibility and practicality.

2. Defining a static general resource analysis which will be parametric with respect to resources and type of approximation (lower- and upper-bounds), and will be adaptable to any execution platform. It should

support a wide variety of resources, including user-defined resources such as number of bits sent or received by an application over a socket, number of calls to a predicate, or number of accesses to a database. This will require the extension of the `CiaoPP` assertion language for defining such resources and expressing related properties.

3. Using the framework for the particular case where the resource is *execution time*. This will require defining cost models, which are parameterized with respect to the execution platform, and performing an assessment of them regarding the trade-off between efficiency and accuracy.

4. Developing a framework for run-time checking of properties that could not have been verified at compile-time. The framework should be able to deal with a wide range of properties, including resource usage related properties.

5. Extending as little as possible the `CiaoPP` assertion language in order to define unit-tests, developing a driver that allows running them (using the run-time assertion checking machinery previously mentioned), and integrating it into the `CiaoPP` static verification framework so that those unit-tests that can be proven statically be eliminated.

6. Integrating unit-testing with run-time checking so that unit-tests can provide test cases for run-time checks, and any existing assertions be also checked during unit testing, even if they were not conceived as tests.

Main Contributions

The results obtained in this Thesis have been published and presented in international forums, most of them classified as first class conferences. Such publications are co-authored with other researchers, and in all of them, the

contribution of the candidate has been relevant, as it is shown by the fact that authors are sorted by its degree of contribution and the candidate is the first author in almost all papers. The main contributions of this thesis are enumerated here:

- Development of a general framework for automatically inferring both upper- and lower-bounds on the usage that a logic program makes of resources in general (see Figure 1.1). Such bounds are given as functions of input data sizes. Examples of resources that can be analyzed by using our framework are execution time, execution steps, memory, energy consumption, as well as other *user-defined resources*, like the number of bits sent or received by an application over a socket, number of calls to a predicate, number of files left open, number of accesses to a database, monetary units spent, disk space used, etc.
- The framework includes a global analysis which is parametric with respect to resources and type of approximation (lower- and upper-bounds). The user can define the parameters of the analysis for a particular resource by means of assertions. This allows to associate basic cost functions with elementary operations of programs, expressing how they affect the usage of a particular resource. The global static analysis can then infer the resource usage of all the procedures in the program. The description of this framework has been done in collaboration with Jorge Navas and Manuel Hermenegildo, and published in the *23rd International Conference on Logic Programming (ICLP'07)* [51].
- The assertions can be defined at different levels of abstraction (source and bytecode). For example, they can associate resource usage functions to different program constructs at the source code level. In particular, assertions can describe how predicates in general update the value for those resources that are applicable to clause heads (such as the number of arguments passed or resolution steps). In such a case, a cost function defined in the assertion will be used by the analyzer in order to update the resource usage when clause heads are analyzed. It

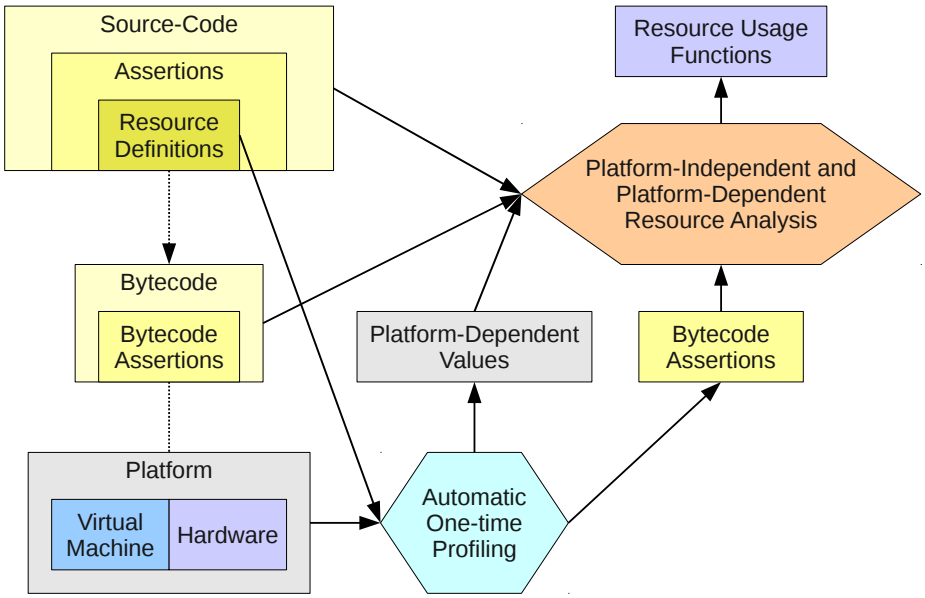

Figure 1.1: Resource Analysis.

is also possible to express how the preparation of the execution of body literals updates a particular resource, as for example, the number of unifications performed. Resource usage assertions can also be given at the bytecode level.

- The approach gives support for platform-independent (e.g., application dependent) resources, such as bits sent or received by an application over a socket, number of calls to a predicate, number of files left open, number of accesses to a database, as well as platform-dependent resources, such as execution time or energy.
- The assertion language also gives support for defining compound resources as a linear combination of other basic resources.
- For platform-dependent resources, the approach performs a one-time

profiling phase that computes platform-dependent parameters associated to basic program operations, at source or bytecode level.

- We have applied the general framework to execution time estimation following two different approaches and experimented with resource usage information supplied at source and bytecode levels.
  - In the first approach, the one-time profiling phase determines the values of certain constants occurring in the resource usage functions for a particular platform. These constants allow to statically computing time bound functions for procedures and to predict with a significant degree of accuracy the execution times of such procedures in the given platform. For each procedure, the compile-time cost bounds analysis gives a vector of platform-independent resources, each one corresponding to a particular low-level operation related to program execution, provided that such operation can be observed in the high level language. The execution time (i.e., the platform-dependent resource) of the procedure is given as a function of input data sizes, and is the linear combination of such platform-independent resources with the vector of (platform-dependent) constants obtained in the one-time profiling phase. We have made use of the facility to define compound resources (previously commented) in order to integrate this process smoothly with the general framework. This work has been done in collaboration with Germán Puebla, Manuel Carro, and Manuel Hermenegildo, and have been presented as a poster in the *22nd International Conference on Logic Programming (ICLP'06)* [48], in the *16th Workshop on Logic Programming Environments* (co-located with ICLP'06) [47], and published in the *Ninth International Symposium on Practical Aspects of Declarative Languages (PADL'07)* [49].
  - In the second approach, applicable to logic programs running on a bytecode-based abstract machine (in most cases the WAM

or a variant), we take advantage of the fact that abstract machines provide a certain separation between platform-dependent and platform-independent concerns in compilation. Many of the differences between architectures are encapsulated in the specific abstract machine implementation and the bytecode is left largely architecture independent. We apply a one-time profiling stage to calculate constants and functions bounding the execution time of each abstract machine instruction. The compile-time resource usage estimation phase uses the instruction timing information in order to infer platform-dependent bounds on actual execution time, given as functions of input data size. This work has been done in collaboration with Manuel Carro and Manuel Hermenegildo, and published in the *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)* [45].

- We have developed a framework that unifies static verification, run-time checking and unit testing. A key contribution is that we preserve the use of a unified assertion language for all of these tasks. In this sense, we have designed methods for compiling run-time checks for (parts of) assertions which cannot be verified at compile-time. Unit tests also provide test cases for the run-time checks. Finally, (parts of) unit tests that can be verified at compile-time are eliminated. But this is not limited to resource-related properties, because this task fall in a more generic framework, which is the run-time checking of computational properties. For example, in addition to the resource related properties, we can deal with properties like non-failure, determinism, and the state (or functional) properties like types of input/output arguments on call or success. This work has been done in collaboration with Manuel Hermenegildo, and published in the *25th International Conference on Logic Programming (ICLP'09)* [46].

- All the developed methods and techniques, including the general re-

source usage analysis, its particularization to execution time estimation, and the unified framework for run-time checking, static verification and unit-testing have been implemented and integrated in the `Ciao/CiaoPP` system. The experimental results are encouraging, in particular the implementation has been effectively used for the verification of ISO Prolog compliance and to the detection of different types of bugs in the `Ciao` system source code.

Impact of the Conferences of the Publications

As shown in Table 1.1, publications are classified according to the following ranking databases: the **CORE**[*] listings, the **CiteSeer**[†] impact listings (see also the upgraded **CiteSeerX**[‡] listing) and the **CS Conference Rankings**[§]. Each of these databases (except CORE) maps venues to a number between 0 and 1 (or 0 and 100%) which corresponds to the position of the corresponding venue divided by the total number of ranked venues (the lower the position the better). CORE classifies venues, instead, into four discrete ranking categories: A+, A, B and C. In order to have a numerical figure with which to compare to the other databases and be able to compute an average value, we have mapped CORE's A+ to top 10%, A to 33%, B to 66% and C to 100%. We obtain an overall numerical ranking for each publication as the **average** of all *available* rankings for the corresponding venue (some venues do not appear in all ranking databases). Finally, publications are classified according to this average. Publications with average ranking 0-33% are considered as **first level**, 33-66% are considered as **second level** and the rest are considered as **third level**. In the table, we report the individual rankings available for the corresponding venue, as well as the global average position, in the form of a percentage.

[*] `http://www.core.edu.au/`

[†] `http://citeseer.ist.psu.edu/impact.html`

[‡] `http://citeseerx.ist.psu.edu/stats/venues`

[§] `http://www.cs-conference-ranking.org/conferencerankings/alltopics.html`

Conference or Workshop	Ranking Database				Average Position	Level
	CORE	Citeseer		CS Conf		
		Position	Impact	Rankings		
ICLP [46, 48, 51]	A	118/1221 (9%)	1.47	0.97	top 15%	First
PADL [49]	A	351/1221 (28%)	0.87	0.88	top 24%	First
PPDP [45]	B	421/1221 (34%)	0.75	-	top 50%	Second
WLPE [47]	Satellite Workshop of ICLP (not ranked)					Workshop

Table 1.1: Impacts of the conferences of the publications.

Publication	Cites	Per Year
ICLP'07[51]	32	10.67
PADL'07[49]	15	3.00
ICLP'09[46]	6	6.00
PPDP'08[45]	4	0.50
ICLP'06[48]	1	0.25
WLPE'06[47]	0	0.00

Table 1.2: Impacts of the publications.

Quality Indexes

Table 1.3 shows the Quality Indexes of the candidate, which were calculated using the Publish or Perish tool ¶.

In addition to the various simple statistics (number of papers, number of citations, and others), Publish or Perish calculates the following citation

¶ Harzing, A.W. (2009) *Publish or Perish, *version (2.8.3644), available at www.harzing.com/pop.htm

Papers:	6	Cites/paper:	9.67	h-index:	4	AWCR:	40.28
Citations:	58	Cites/author:	14.2	g-index:	6	AW-index:	6.35
Years:	5	Papers/author:	1.43	hc-index:	4	AWCRpA:	10.12
Cites/year:	11.6	Authors/paper:	4.33	hI-index:	1	e-index:	6.40
				hI,norm:	2	hm-index:	1.03

Table 1.3: Quality indexes of candidate's papers.

metrics [‖]:

h-index **Hirsch's h-index**: Proposed by J.E. Hirsch in his paper An index to quantify an individual's scientific research output, arXiv:physics/0508025 v5 29 Sep 2005. It aims to provide a robust single-number metric of an academic's impact, combining quality with quantity.

g-index **Egghe's g-index**: Proposed by Leo Egghe in his paper Theory and practice of the g-index, Scientometrics, Vol. 69, No 1 (2006), pp. 131-152. It aims to improve on the h-index by giving more weight to highly-cited articles.

e-index **Zhang's e-index**: Publish or Perish also calculates the e-index as proposed by Chun-Ting Zhang in his paper The e-index, complementing the h-index for excess citations, PLoS ONE, Vol 5, Issue 5 (May 2009), e5429. The e-index is the (square root) of the surplus of citations in the h-set beyond h2, i.e., beyond the theoretical minimum required to obtain a h-index of 'h'. The aim of the e-index is to differentiate between scientists with similar h-indices but different citation patterns.

hc-index **Contemporary h-index**: Proposed by Antonis Sidiropoulos, Dimitrios Katsaros, and Yannis Manolopoulos in their paper Generalized h-index for disclosing latent facts in citation networks, arXiv:cs.DL/0607066 v1 13 Jul 2006. It aims to improve on the h-index by giving more weight to recent articles, thus rewarding academics who maintain a steady level of activity.

AWCR / AW-index **Age-weighted citation rate (AWCR) and AW-index**: The AWCR measures the average number of citations to an entire body of work, adjusted for the age of each individual paper. It was inspired by Bihui Jin's note The AR-index: complementing the

[‖]Taken from the Publish or Perish website

h-index, ISSI Newsletter, 2007, 3(1), p. 6. The Publish or Perish implementation differs from Jin's definition in that we sum over all papers instead of only the h-core papers.

hI-index **Individual h-index (original)**: The Individual h-index was proposed by Pablo D. Batista, Monica G. Campiteli, Osame Kinouchi, and Alexandre S. Martinez in their paper Is it possible to compare researchers with different scientific interests?, Scientometrics, Vol 68, No. 1 (2006), pp. 179-189. It divides the standard h-index by the average number of authors in the articles that contribute to the h-index, in order to reduce the effects of co-authorship.

hI,norm **Individual h-index (PoP variation)**: Publish or Perish also implements an alternative individual h-index that takes a different approach: instead of dividing the total h-index, it first normalizes the number of citations for each paper by dividing the number of citations by the number of authors for that paper, then calculates the h-index of the normalized citation counts. This approach is much more fine-grained than Batista et al.'s; we believe that it more accurately accounts for any co-authorship effects that might be present and that it is a better approximation of the per-author impact, which is what the original h-index set out to provide.

hm-index **Multi-authored h-index**: A further h-like index is due to Michael Schreiber and first described in his paper To share the fame in a fair way, hm modifies h for multi-authored manuscripts, New Journal of Physics, Vol 10 (2008), 040201-1-8. Schreiber's method uses fractional paper counts instead of reduced citation counts to account for shared authorship of papers, and then determines the multi-authored hm index based on the resulting effective rank of the papers using undiluted citation counts.

Chapter 2

General User Definable Resource Bound Analysis

2.1 Introduction

In this chapter we propose an analyzer which allows automatically inferring both upper- and lower-bounds on the usage that a logic program makes of *user-definable resources*.

In our approach, a resource is a user-defined, application-dependent notion which associates a basic resource usage function with elementary operations in the base language and/or to some predicates in libraries. In this sense, each *resource* is essentially a user-defined *counter*. The user gives a name (such as, e.g., `bits_received`) to the counter and then defines via assertions how each elementary operation in the program (e.g., unifications, calls to builtins, external calls, etc.) increments or decrements that counter. The use of resources obviously depends in practice on the sizes or values of certain inputs to programs or predicates. Thus, in the assertions describing elementary operations the counters may be incremented or decremented not only by constants but also by amounts that are *functions* of input data sizes or values. Correspondingly, the objective of our method is to statically derive from these elementary assertions and the program text functions that

yield upper- and lower-bounds on the amount of those resources that each of the predicates in the program (and the program as a whole) will consume or provide. The input to these functions will also be the sizes or value ranges of the topmost input data to the program or predicate being analyzed.

2.1.1 Related Work

As mentioned before in Section 1, most previous work is specific to the analysis of execution steps. Debray *et al.* presents in [21, 22] a semi-automatic analysis which infers upper-bounds on the number of execution steps. These bounds are functions on the sizes or value ranges of input data. This seminal work applies to a large class of logic programs and presents techniques in order to deal with the generation of multiple solutions via backtracking. The authors also show how other specific analyses could be developed, e.g., time or memory. This approach was later fully automated and extended to inferring upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [23, 36]. Our method builds on this work but generalizes it in order to deal with a much more general class of user-defined resources, allowing thus the coverage of an unlimited number of analyses within a single implementation. Grobauer presents in [31] a method for automatically extracting cost recurrences from first-order DML programs. The main feature is the use of dependent types to describe a size measure that abstracts from data to data size. In [52], and inspired by [5] and [44], a complexity analysis is presented for Horn clauses, also fully automating the necessary calculations. In [39], Igarashi *et al.* presents a method for modeling problems such as memory management, lock primitive usage, etc., and a type-based method is proposed as solution to the inference problem. In [64] a cost model is presented for inferring cost equations for recursive, polymorphic, and higher-order functional programs. While it is claimed that the approach can be modified in order to infer a reduced set of resources such as execution time, execution steps, or memory, no details are given. Worst case execution time (WCET) estimation has been studied for imperative

languages and for different application domains (see, e.g., [63, 7, 27] and its references). However these and related methods again concentrate only on execution time. Also, they do not infer resource usage functions of input data sizes but rather absolute maximum execution times, and they generally require the manual annotation of loop iteration bounds. In [16] a method is presented for reserving resources before their actual use. However, the programmer (or program optimizer) needs to annotate the program with "acquire" and "consume" primitives, as well as provide loop invariants and function pre and post-conditions. Interesting type-based related work has also been performed in the GRAIL system [3], also oriented towards resource analysis, but it has concentrated mainly on ensuring memory bounds.

In comparison with previous work our approach allows dealing with a class of resources which is open, in the ample sense that such resources are in fact defined by programmers using an assertion language, which we also consider itself an important contribution of our work. Another important contribution of our work because of its impact in the scalability and automation of the analysis is that our approach allows defining the resource usage of external predicates, which can be used for modular composition. In addition, assertions also allow describing by hand the usage of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating.

2.2 Worked Example

Consider a client application in Figure 2.1 that sends a data buffer through a socket and receives another (possibly transformed) data buffer. In this section, we will provide an overview of our approach through that program.

A *resource* is a user-defined, application dependent notion which associates a basic cost function with some user-selected predicates in the program. This is expressed by adding annotations using our assertion language

```
:- pred client(Opts, IBuf, OBuf)
   : list(gnd) * list(byte) * var.

client([Host,Port],IBuf,OBuf) :-
   connect(Host,Port,Stream),
   exch_buffer(IBuf,Stream,OBuf),
   close(Stream).

exch_buffer([],_,[]).
exch_buffer([B|Bs],Id,[B0|Bs0]) :-
   exch_byte(B,Id,B0),
   exch_buffer(Bs,Id,Bs0).

:- head_cost(ub,bits_received,0).
:- literal_cost(ub,bits_received,0).
```

```
/* SOCKET LIBRARY */

:- trust pred connect(Host,Port,S)
    : atm * num * var
   => atm * num * atm
   + cost(ub,bits_received,0).

:- trust pred close(Stream)
    : atm => atm
   + cost(ub,bits_received,0).

:- trust pred exch_byte(B,Id,B0)
    : byte * atm * var
   => byte * atm * byte.
   + cost(ub,bits_received,8).
```

Figure 2.1: A simple client application.

(Section 2.3.1) to the code. The objective of the analysis is to approximate the usage that the program makes of the resource. In the example, assume that we would like to obtain an upper-bound on the number of bits received by the application that we will call `bits_received`. We assume that the program receives 8 bits each time that `exch_byte/3` is called. This fact is reflected by the user by adding the assertion `'cost(ub,bits_received,8)'` which will increment the counter associated with the upper-bound on the number of bits received by 8. Similarly, we assume that open and close socket connections (`connect/3` and `close/1`) do not imply any exchange of bits, as indicated by the `'cost(ub,bits_received,0)'` assertion. In addition, the types and modes of the socket operations must be given to the analysis by other analyses or by user-provided assertions. In this example, we assume that the analysis does not have access to the code of the socket operations and hence, user annotations are required. For now, we will skip the assertions `':- head_cost(ub,bits_received,0)'` and `':- literal_cost(ub,bits_received,0)'`. The rest of this section describes the main steps applied by the analyzer to approximate the number of bits received of the program depicted in Figure 2.1.

Step 1: Size metrics and mode inference. In the first step, the approach needs to infer for each argument in the program the notion of size

metrics. For instance, length of a list, depth of a term, size of a term, etc. In addition, the analysis also needs to infer if each argument is input or output in order to perform properly the size and resource usage analyses described in Section 2.3.2, 2.3.3, and 2.3.4. Input/output and size metrics information can be required by the language (typed language), given by the user (via assertions), or, as in our implementation, inferred automatically via analysis. In the example this information is asserted by the user in case of the socket library and inferred from the language for the predicates `client/3` and `exch_buffer/3`.

Step 2: Inference of data dependencies and size relationships. In the second step, the analysis yields *argument dependency graph*s for the clauses within a strongly connected component, through a dataflow analysis. These graphs will be used for inferring *size relationships* for each literal argument between the input and output head arguments of every clause. In the example, assume the `exch_buffer/3` predicate. The analysis will infer from the first clause that the size of the third argument is 0, i.e. empty list, if the first argument is also an empty list. We denote this size relationship by the equation $\Psi^3_{exch\_buffer}(0, \_) = 0$. Note that the size of the third argument does not depend on the second argument. We denote this by using the do not care symbol '_'. Similarly, the analysis will infer from the second recursive clause the following equation: $\Psi^3_{exch\_buffer}(x, \_) = \Psi^3_{exch\_buffer}(x - 1, \_) + 1$. The recurrence equation system shown above must be approximated by a recurrence solver in order to obtain a closed form solution. In this case, our analysis yields the solution $\Psi^3_{exch\_buffer}(x, \_) = x$, i.e. the size of the third argument is proportional to the size of the first argument.

Step 3: Resource usage analysis. In this step, the analysis will use the size metrics, modes, the data dependencies, and the size relationships inferred in previous steps, and also the user-defined resource-related assertions in order to infer a resource usage equation for each clause and further simplify the resulting obtaining upper/lower bound closed form solutions. The re-

source analysis will statically derive safe upper/lower bounds on the amount of resources that each of the predicates consumes or provides. The result given by our analysis for an upper-bound on the number of bits received by `exch_buffer/3` is $\mathsf{Cost}(exch\_buffer, ub, bits\_received, \langle 0, \_\rangle) = 0$ and $\mathsf{Cost}(exch\_buffer, ub, bits\_received, \langle x, \_\rangle) = 8 + \mathsf{Cost}(exch\_buffer, ub, bits\_received, \langle x-1, \_\rangle)$. Again, this equation system is solved by a recurrence solver, resulting in the closed form
$\mathsf{Cost}(exch\_buffer, ub, bits\_received, \langle x, \_\rangle) = 8 \times x$
Note that since we know from the user-defined assertions that `connect/3` and `close/1` do not receive any bits, then
$\mathsf{Cost}(client, ub, bits\_received, \langle \_, n, \_\rangle) = 8 \times n$

2.3 A Framework for Inference of Resource Usage

We can now describe our framework for inferring upper- and lower-bounds on the usage that a program makes of a set of user-definable resources. Our basic approach is as follows. Given a predicate call p, let $\Phi(p, r, \overline{n})$ denote the exact units of resource r consumed or produced during the computation of p for a vector of argument sizes $\overline{n}$. An expression $\mathsf{Cost}(p,\mathtt{ap},r,\overline{n})$ is determined (at compile-time) that approximates $\Phi(p, r, \overline{n})$ with approximation `ap`. For assuring the correctness of our approach, we must always generate resource usage bound functions such as $\mathsf{Cost}(p,\mathtt{ap},r,\overline{n})$ that hold the following conditions:

- If the analysis computes an upper-bound approximation, i.e., $\mathtt{ap} = \mathtt{ub}$, then:
$$\Phi(p, r, \overline{n}) \leq \mathsf{Cost}(p,\mathtt{ub},r,\overline{n}) \tag{2.1}$$
- Conversely, if the analysis computes a lower-bound $\mathtt{ap} = \mathtt{lb}$, then:
$$\mathsf{Cost}(p,\mathtt{lb},r,\overline{n}) \leq \Phi(p, r, \overline{n}) \tag{2.2}$$

Note that the analysis can always generate trivial upper- and lower-bounds, ∞ and $-\infty$, in those cases where it cannot infer resource equations or find a closed form.

Certain program information such as, for example, input/output modes and size metrics for predicate arguments is first automatically inferred by other abstract interpretation-based analyzers and then provided as input to the size and resource analysis. The techniques involved in inferring this information are beyond the scope of this work —see, e.g., [36] and its references for some examples. Based on this information, our analysis first finds bounds on the size of input arguments to the calls in the body of the predicate being analyzed, relative to the sizes of the input arguments to this predicate, using the inferred metrics.

The size of an output argument in a predicate call depends in general on the size of the input arguments in that call. For this reason, for each output argument we infer an expression which yields its size as a function of the input data sizes. Argument sizes are described in terms of size metrics. Typical size metrics are the actual value of a number, the length of a list, the size (number of constant and function symbols) of a term, etc. To this end, and using the input-output argument information, data dependency graphs are used to set up recurrence equations whose solution yields size relationships between input and output arguments of predicate calls. This information regarding argument sizes and other such as resource-related assertions are then used to set up another set of recurrence equations whose solution provides resource usage bound functions. Both the size and resource usage recurrence equations must be solved by a recurrence equation solver. Although the operation of such solvers is beyond the scope of the work our implementation does provide a table-based solver which covers a reasonable set of recurrence equations such as first-order and higher-order linear recurrence equations in one variable with constant and polynomial coefficients,* divide and conquer recurrence

*Note that it is always possible to reduce a system of linear recurrence equations to a single linear recurrence equation in one variable.

equations, etc. In addition, the system allows the use of external solvers (such as, e.g. [4], Mathematica, Matlab, etc.). Note also that, since we are computing upper/lower bounds, it suffices to compute upper/lower bounds on the solution of a set of recurrence equations, rather than an exact solution. This allows obtaining an *approximate* closed form when the exact solution is not possible.

In further sections, we will describe each main component of our framework. In Section 2.3.1 we will first present the assertion language proposed for defining resources and annotating elementary operations. Section 2.3.2 shows how size relationships among program variables are determined, Section 2.3.3 describes how the resource usage bound functions are inferred, and finally, Section 2.3.4 shows how users can define resources using our assertion language.

2.3.1 The Resource Assertion Language

We start by describing the assertion schema. This language is used for describing resources and providing other input to the resource analysis, and is also the language in which the resource analysis produces its output. This assertion language is used additionally to state resource-related specifications which can then be proved or disproved based on the results of analysis following the scheme of [36] allowing finding bugs, verifying the program, etc.

The rules for the assertion language grammar are listed in Figure 2.2. In this grammar, Var corresponds to variables written in the syntax for variables of the underlying logic programming language (i.e., normally non-empty strings of characters which start with a capital letter or underscore). Similarly, Num is any valid number and $Pred\_name$ any valid name for a predicate in the underlying programming language, normally non-empty strings of characters which start with a lower-case letter or are quoted. $State\_prop$ corresponds to other *state properties* such as modes and types, and $Comp\_prop$ stands for any other valid *computational property*, see [36] and its references.

Predicates can be annotated with zero or more assertions. These as-

⟨*program_assrt*⟩	::=	`:-` ⟨*status_flag*⟩ ⟨*pred_assrt*⟩.
	|	`:-` `head_cost`(⟨*approx*⟩,Res_name,Δ^H).
	|	`:-` `literal_cost`(⟨*approx*⟩,Res_name,Δ^L).
⟨*status_flag*⟩	::=	`trust` | `check` | `true` | `checked` | `false` | ϵ
⟨*pred_assrt*⟩	::=	`pred` ⟨*pred_desc*⟩ ⟨*pre_cond*⟩ ⟨*post_cond*⟩ ⟨*comp_cond*⟩.
⟨*pred_desc*⟩	::=	Pred_name | Pred_name(⟨*args*⟩)
⟨*args*⟩	::=	Var | Var, ⟨*args*⟩
⟨*pre_cond*⟩	::=	`:` ⟨*state_props*⟩ | ϵ
⟨*post_cond*⟩	::=	`=>` ⟨*state_props*⟩ | ϵ
⟨*comp_cond*⟩	::=	`+` ⟨*comp_props*⟩ | ϵ
⟨*state_prop*⟩	::=	`size`(Var,⟨*approx*⟩,⟨*sz_metric*⟩,⟨*arith_expr*⟩) | State_prop
⟨*state_props*⟩	::=	⟨*state_prop*⟩ | ⟨*state_prop*⟩, ⟨*state_props*⟩
⟨*comp_prop*⟩	::=	`size_metric`(Var,⟨*sz_metric*⟩) | ⟨*cost*⟩ | Comp_prop
⟨*comp_props*⟩	::=	⟨*comp_prop*⟩ | ⟨*comp_prop*⟩, ⟨*comp_props*⟩
⟨*cost*⟩	::=	`cost`(⟨*approx*⟩,Res_name,⟨*arith_expr*⟩)
⟨*approx*⟩	::=	`ub` | `lb` | `oub` | `olb`
⟨*sz_metric*⟩	::=	`value` | `length` | `term_size` | `depth` | `void`
⟨*arith_expr*⟩	::=	$-$ ⟨*arith_expr*⟩ | ⟨*arith_expr*⟩ `!` | ⟨*quantifier*⟩ ⟨*arith_expr*⟩
	|	⟨*arith_expr*⟩ ⟨*bin_op*⟩ ⟨*arith_expr*⟩
	|	`exp`(⟨*arith_expr*⟩,*Num*) | `log`(*Num*, ⟨*arith_expr*⟩)
	|	Num | ⟨*sz_metric*⟩(Var)
⟨*bin_op*⟩	::=	`+` | `-` | `*` | `/`
⟨*quantifier*⟩	::=	$\sum$ | $\prod$

Figure 2.2: Syntax of the resource assertion language.

sertions can refer to properties of the execution states when the predicate is called (`calls`), properties of the execution states when the predicate terminates execution (`success`), and properties which refer to the whole computation of the predicate, rather than the input-output behavior (`comp`, which herein will be used only for resource-related properties). The assertion schema also provides syntactic sugar which allows defining the `calls`, `success`, and `comp` parts together in a more compact way via `pred` assertions. For brevity, the ⟨*state_props*⟩ fields can also be written using "star notation" (see the examples). In addition, there may be a set of global `head_cost` and `literal_cost` declarations, one for each resource and approximation direction. The *Res_name* fields determine which resource the assertion refers to. These *Res_name*s are user-provided identifiers which give a name to each particular resource that needs to be tracked. Resources do not need to be

declared in any other way –the set of resources that the system is aware of is simply the set of such names that appear in assertions which are in the scope. The $\langle approx \rangle$ fields state whether $\langle arith\_expr \rangle$ is providing an upper-bound or a lower-bound (with `oub` meaning it is a "big O" expression, i.e., with only the order information, and `olb` meaning it is an Ω asymptotic lower-bound).

The first and most fundamental use of assertions in our context is to describe how the execution of some predicates increments or decrements the usage of the resources (counters) defined in the program. The purpose of analysis is then to infer the resource usage of all predicates in the program. The `head_cost`($\langle approx \rangle$, Res_name, $\Delta^{\mathtt{H}}$) declarations are used to describe how predicates in general *update* the value for those resources that are applicable to predicate heads (such as counting the number of arguments passed or total execution steps –see Section 2.3.3). The definition of $\Delta^{\mathtt{H}} : clause\_head \times \overline{n} \rightarrow arith\_expr$ is provided by means of a user-defined (or imported) predicate, written in the source language, that takes the head itself and the size of its input arguments and will be called by the analyzer when the clause head is analyzed. This code gets loaded into the compiler in a similar way to, e.g., macro expansion code. The `literal_cost`($\langle approx \rangle$, Res_name, $\Delta^{\mathtt{L}}$) declarations describe how predicate bodies *update* the value of certain resources which are applicable to body literals (such as, for example, number of unifications). In this case, $\Delta^{\mathtt{L}} : body\_lit \times \overline{n}_i \rightarrow arith\_expr$ is also user (or library) provided code which will be executed when the body literals of different predicates are analyzed, note that such predicate will take as input argument the literal itself and the sizes of the input arguments of such i-th literal in the body of the clause being analyzed. The `cost`($\langle approx \rangle$, Res_name, $\langle arith\_expr \rangle$) comp-type properties are included in `comp` or `pred` assertions and used to provide the actual resource usage bound functions for each builtin or external (e.g., defined in another language) predicate used in the program. Such assertions have different status. If an assertion has been proved to be true it has a status `true`. Assertions can also be used to provide information to the analyzer in order to increase its precision or to

describe predicates defined in other modules. These assertions have a `trust` status. Assertions with a `check` status are the ones used to specify the *intended* semantics of a predicate. Additionally, the system verifies whether that intended semantics is consistent with results of the analysis. If it is validated, assertions with `checked` status are used. However, if it has been detected to be false then the system will use assertions with `false` status. Otherwise, if our system cannot validate it statically because of the undecidability of the property, the `check` status remains. As mentioned previously, the aim of the analysis is to derive functions that describe the resource usage (as well as argument size relations) for the rest of the predicates in the program. Note however that it is also possible to provide `comp` or `pred` assertions for some of those predicates and this can also be used to guide the analysis. In particular, the analysis will compute the most precise expression between the resource usage function provided by the assertion ($\langle cost \rangle$) and the resource usage function inferred by analysis. Additionally, size metrics (`size_metric`(Var,$\langle sz\_metric \rangle$)) information can be provided by users if needed, but note that in practice size metrics can often be derived automatically from the inferred types.

Assertions can also be used, via the *pre_cond* and *post_cond* fields, to declare relationships between the data sizes of the inputs and outputs of predicates, which are needed by our analysis, as will be described later. These assertions are also used to label predicate arguments as input or output, as well as to provide types or size (`size`(Var,$\langle approx \rangle$,$\langle sz\_metric \rangle$,$\langle arith\_expr \rangle$)) information. In the same way as with the $\langle cost \rangle$ properties, for user-defined predicates these other assertions can be provided by the user or inferred by analysis. Again, analysis will compute the most precise of the two.

2.3.2 Size Analysis

We will give the intuition behind the data dependency-based method for inferring bounds on the sizes of output arguments in the head of a predicate as a function of the sizes of input arguments to the predicate. Besides this,

as a result of the size analysis, we have bounds on the size of each input argument to body literals in a clause as a function of the size of the input arguments to the head of that clause. The size of the input arguments to body literals will be used later to infer functions which give bounds on the resource usage of body literals in terms of the sizes of the input arguments to the head. We adopt the approach of Debray *et al.* [21, 22] for the inference of upper-bounds on argument sizes and [23] for lower-bounds. For the sake of brevity, we will only consider the inference of upper-bounds in this work, and refer the reader to [23] for the inference of lower bounds.

The size of an input is defined in terms of metrics. By *size metrics* we refer to a total function that, given a term, returns an arithmetic expression or an undefined value $\perp$, possibly in terms of other input argument sizes. One of the difference with respect to Debray's approach is that our analysis is parametric on size metrics, which can be defined by the user through `size_metric` and `size` assertions. For concreteness, several size metrics are defined in our system. We define here a $\mathsf{size}(\langle sz\_metric \rangle, t)$ operation which returns the size of a term t under the metric $\langle sz\_metric \rangle$ for those predefined metrics:

- If size metrics is the integer value, then:

$$\mathsf{size}(value, t) = \begin{cases} n & \text{if } t \text{ is an integer } n \\ \ominus(\mathsf{size}(value, t_1), \ldots, \mathsf{size}(value, t_n)) & \text{if } t = \ominus(t_1, \ldots, t_n) \\ \perp & \text{otherwise.} \end{cases}$$

  where $\ominus$ is an evaluable arithmetic operator.

- If size metrics is the length of a list, then:

$$\mathsf{size}(length, t) = \begin{cases} 0 & \text{if t = [ ]} \\ 1 + \mathsf{size}(length, T) & \text{if t = } [H \mid T] \\ \perp & \text{otherwise.} \end{cases}$$

- If size metrics is the size of a term, then:

$$\mathsf{size}(\mathit{term\_size}, t) = \begin{cases} 1 & \text{if } t \text{ is a constant} \\ 1 + \sum_{i=1}^{n} \mathsf{size}(\mathit{term\_size}, t_i) & \text{if } t = f(t_1, \ldots, t_n) \\ \bot & \text{otherwise.} \end{cases}$$

- If size metrics is the depth of a term, then:

$$\mathsf{size}(\mathit{depth}, t) = \begin{cases} 0 & \text{if } t \text{ is a constant} \\ 1 + \mathsf{max}\{\mathsf{size}(\mathit{depth}, t_i)\} & \text{if } t = f(t_1, \ldots, t_n) \\ \bot & \text{otherwise.} \end{cases}$$

Some examples:
$\mathsf{size}(\mathit{length}, [X, Y]) = 2$,
$\mathsf{size}(\mathit{length}, [X|Y]) = \bot$,
$\mathsf{size}(\mathit{value}, 3 + 7) = 10$,
$\mathsf{size}(\mathit{term\_size}, f(g(a), b) = 4$, and
$\mathsf{size}(\mathit{depth}, f(2, f(3, \mathit{nil}, \mathit{nil}), \mathit{nil}) = 2$.

Since our approach assumes the general case in which the input program is not normalized, sometimes we need to establish size relationships as the size difference between two terms. This relationship is provided by the function $\mathsf{diff}(\langle sz\_metric \rangle, t_1, t_2)$ operation, which returns an approximation of the size difference between two terms t_1 and t_2 under the metric $\langle sz\_metric \rangle$. We define it again for our predefined metrics:

- If size metrics is the integer value, then:

$$\mathsf{diff}(\mathit{value}, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ \bot & \text{otherwise.} \end{cases}$$

- If size metrics is the length of a list, then:

$$\mathsf{diff}(\mathit{length}, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ \mathsf{diff}(\mathit{length}, t, t_2) - 1 & \text{if } t_1 = [\_|t] \text{ for some term t} \\ \bot & \text{otherwise.} \end{cases}$$

- If size metrics is the size of a term, then:

$$\mathsf{diff}(\mathit{term\_size}, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ \mathsf{sz}(t_1(i)) - \mathsf{size}(\mathit{term\_size}, t_1) & \text{if } t_1 = f(s_1, \ldots, s_n) \\ & s_i \equiv t_2, \exists i, 1 \leq i \leq n \\ \bot & \text{otherwise.} \end{cases}$$

  where $\mathsf{sz}(t_1(i))$ is a symbolic expression that represents the size of the i-th argument position of the t_1 term.

- If size metrics is the depth of a term, then:

$$\mathsf{diff}(\mathit{depth}, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ \mathsf{max}\{\mathsf{diff}(\mathit{depth}, s_i, t_2)\} - 1 & \text{if } t_1 = f(s_1, \ldots, s_n) \\ \bot & \text{otherwise.} \end{cases}$$

Thus,
$\mathsf{diff}(\mathit{length}, [A, B|T], T) = -2$,
$\mathsf{diff}(\mathit{length}, T, [H|T]) = \bot$,
$\mathsf{diff}(\mathit{value}, X, X) = 0$,
$\mathsf{diff}(\mathit{term\_value}, X, f(X)) = \bot$, and
$\mathsf{diff}(\mathit{depth}, f(2, X, Y), X) = -1$

A directed acyclic graph called *argument dependency graph*, $G = (V, E)$, is used to represent the data dependency between argument positions in a clause body, and between them and those in the clause head. Nodes in V denotes argument positions. There is an edge from a node n_1 to a node n_2,

$(n_1, n_2) \in E$ if the variable bindings generated by n_1 are used to construct the term occurring at n_2. The node n_1 is said to be a *predecessor* of the node n_2. We will assume a predec function that takes an argument dependency graph, a literal, and a parameter position and returns its nearest predecessor in the graph.

Using the size and diff functions and the argument dependency graph for every clause, the analysis will traverse each strongly-connected component in reverse topological order in order to set up size relations for expressing the size of each argument position in terms of the sizes of its predecessors for every clause. Let $\mathsf{sz}(i)$ denote the size of the term occurring at an argument position i. For convenience, we will omit the argument $\langle sz\_metric \rangle$ in the size and diff functions in the rest of the chapter. Then, the size relationships can be obtained as follows:

- *Output arguments.* Let $l_1, \ldots, l_n$ denote the input argument positions of the literal L, $\mathcal{I}_\perp$ the set of integer numbers augmented with the special symbol $\perp$, denoting "undefined", and let $\Psi_p^k : \mathcal{I}_\perp \times \ldots \times \mathcal{I}_\perp \mapsto \mathcal{I}_\perp$ be a function that represents the size of the k-th (output) argument position of the predicate p of literal L in terms of the size of its input argument positions. Assume that i is an output argument position in a clause. Then the following size relation is set up:

$$\mathsf{sz}(i) \leq \Psi_p^i(\mathsf{sz}(l_1), \ldots, \mathsf{sz}(l_n))$$

  1. If L is recursive, then $\Psi_p^i(\mathsf{sz}(l_1), \ldots, \mathsf{sz}(l_n))$ is a symbolic expression.
  2. Otherwise, if L is non-recursive then the function Ψ_p^i has been recursively computed, and thus we replace $\Psi_p^i(\mathsf{sz}(l_1), \ldots, \mathsf{sz}(l_n))$ by the (explicit) expression resulting from the application of the function Ψ_p^i to $\mathsf{sz}(l_1), \ldots, \mathsf{sz}(l_n)$.

- *Input arguments.* Assume now that i is an input argument position in a body literal, and i' the term occurring at an argument position i. Let

$\mathsf{predec}(i)$ be the set of predecessors of i in the argument dependency graph. We have the following possibilities:

1. Compute $\mathsf{size}(i')$. If $\mathsf{size}(i') \neq \bot$ then set up the size relation: $\mathsf{sz}(i) \leq \mathsf{size}(i')$.
2. If $\exists r \in \mathsf{predec}(i)$ such that the size metrics corresponding to r and i are the same and $d = \mathsf{diff}(r, i) \neq \bot$, then set up the size relation: $\mathsf{sz}(i) \leq \mathsf{sz}(r) + d$.
3. If $\mathsf{size}(i')$ can be expanded using the definition of the size function, then expand $\mathsf{size}(i')$ one step and recursively compute $\mathsf{size}(t_i)$ for the appropriate subterms t_i of i'. If each of these recursive size computations have a defined result, then use them to compute the size relation for $\mathsf{size}(i')$.
4. Otherwise, $\mathsf{sz}(i) = \bot$.

Size relations can be propagated to transform a size relation corresponding to an input argument in a body literal or an output argument in the clause head into a function in terms of the sizes of the input arguments of the head. The basic idea here is to repeatedly substitute size relations for body literals into size relations for head arguments. This is the purpose of the normalization algorithm described in [21]. However, for recursive clauses, we need to solve the symbolic expression due to recursive literals into an explicit function first.

Example 2.3.1. Consider again the program described in Figure 2.1. We will denote by *pred_name* the name of a predicate, and by $pred\_name_i^j$ the i-th argument position in the j-th literal with predicate name *pred_name* in the body of a clause. If there is only one body literal with predicate name *pred_name* in the body of a clause then we omit the superscript j and write simply $pred\_name_i$. Let $head_i$ denote the i-th argument position in the clause head.

Size relation equations for `exch_buffer/3`:

$$
\begin{array}{lcl}
\mathsf{sz}(exch\_byte_1) & \leq & \mathsf{size}(B) = 1 \\
\mathsf{sz}(exch\_byte_2) & \leq & \mathsf{sz}(head_2) + \mathsf{diff}(Id, Id) = \mathsf{sz}(head_2) \\
\mathsf{sz}(exch\_byte_3) & \leq & \Psi^3_{exch\_byte}(\mathsf{sz}(exch\_byte_1), \mathsf{sz}(exch\_byte_2)) = 1 \\
\mathsf{sz}(exch\_buffer_1) & \leq & \mathsf{sz}(head_1) + \mathsf{diff}([B|Bs], Bs) = \mathsf{sz}(head_1) - 1 \\
\mathsf{sz}(exch\_buffer_2) & \leq & \mathsf{sz}(head_2) + \mathsf{diff}(Id, Id) = \mathsf{sz}(head_2) \\
\mathsf{sz}(exch\_buffer_3) & \leq & \Psi^3_{exch\_buffer}(\mathsf{sz}(exch\_buffer_1), \mathsf{sz}(exch\_buffer_2)) \\
\mathsf{sz}(head_3) & \leq & \mathsf{sz}(exch\_buffer_3) + 1
\end{array}
$$

Normalized size relation equations for the output argument of the head:

$$
\begin{array}{rcl}
\mathsf{sz}(head_3) \leq & \Psi^3_{exch\_buffer}(\mathsf{sz}(exch\_buffer_1), \mathsf{sz}(exch\_buffer_2)) + 1 \\
\leq & \Psi^3_{exch\_buffer}(\mathsf{sz}(head_1) - 1, \mathsf{sz}(head_2)) + 1
\end{array}
$$

Closed form for the output argument of the head:

$$
\begin{array}{rcl}
\Psi^3_{exch\_buffer}(0, y) = & 0 \\
\Psi^3_{exch\_buffer}(x, y) = & \Psi^3_{exch\_buffer}(x - 1, y) + 1
\end{array}
$$

$$\Psi^3_{exch\_buffer}(x, y) = x$$

Figure 2.3: Size relation equations for `exch_buffer/3`.

The Figure 2.3 shows all equations needed to establish the size of the output arguments of the head. First, the system sets up the size relation for the input/output arguments of the body literals. Then, the system sets up the size relation for the output arguments of the head and obtains its normalized form. Note that the code of `exch_byte/3` is not available. Therefore, the analysis takes the size of its third argument from its type definition, assuming that the size of the output argument is 1. Thus, the system establishes the recurrence equation for the output argument ($head_3$) in the head (since it belongs to a recursive predicate). Then, it obtains the boundary condition $\Psi^3_{ex\_buf}(0, y) = 0$ from the non-recursive clause, and using it, obtains a closed form function by calling the recurrence equation solver (variables x and y represent $\mathtt{sz}(head_1)$ and $\mathtt{sz}(head_2)$ respectively).

2.3.3 Resource Usage Analysis

In order to infer the resource usage functions all predicates in the program are processed in a single traversal of the call graph in reverse topological order. Consider such a predicate p defined by clauses $\mathtt{C}_1, \ldots, \mathtt{C}_m$. Assume that $\overline{n}$ is a tuple such that each element corresponds to the size of an input argument position to predicate p. Then, the resource usage expressed in units of resource r with approximation $\mathtt{ap}$ of a call to p, for an input of size $\overline{n}$, can be expressed as:

$$\mathsf{Cost}(p, \mathtt{ap}, r, \overline{n}) = \bigodot(\mathtt{ap})_{1 \leq k \leq m}\{\mathsf{Cost}_{clause}(\mathtt{C}_k, p, \mathtt{ap}, r, \overline{n})\} \qquad (2.3)$$

where $\bigodot(\mathtt{ap})$ is a function that takes an approximation identifier $\mathtt{ap}$ and returns a function which applies over all $\mathsf{Cost}_{clause}(\mathtt{C}_k, p, \mathtt{ap}, r, \overline{n})$, for $1 \leq k \leq m$. For example, if $\mathtt{ap}$ is the identifier for approximation "upper-bound" (`ub`), then a possible conservative definition for $\bigodot(\mathtt{ap})$ is the $\sum$ function. In this case, and since the number of solutions generated by a predicate that will be demanded is generally not known in advance, a conservative upper-bound on the computational cost of a predicate is obtained by assuming that all solutions are needed, and that all clauses are executed, thus the cost of the predicate is assumed to be the sum of the costs of all of its clauses. However, it is straightforward to take mutual exclusion into account, which is inferred by CiaoPP [36, 42] and is available to our analysis, to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses. If $\mathtt{ap}$ is the identifier for approximation "lower-bounds" (`lb`), then $\bigodot(\mathtt{ap})$ is the *min* function.

Let us see now how to compute the resource usage of clauses. Consider a clause $\mathtt{C}_k$ of predicate p of the form $\mathtt{H}^k :- \mathtt{L}^k_1, \ldots, \mathtt{L}^k_l$ where $\mathtt{L}^k_i$, $1 \leq i \leq l$, is a literal (either a predicate call, or an external or builtin predicate), and $\mathtt{H}^k$ is the clause head. Because of backtracking, the number of times a literal

will be executed depends on the number of solutions of the previous literals. Assume that $\overline{n}_i$ is a tuple with the sizes of all the input arguments to literal $\mathtt{L}_i^k$, given as functions of the sizes of the input arguments to the clause head, that is, $\overline{n}_i = \psi_i(\overline{n})$. Note that these $\overline{n}_i$ size relations have previously been computed during size analysis for all input arguments to literals in the bodies of all clauses. $\mathsf{Sols}_{\mathtt{L}_j^k}$ is the number of solutions literal $\mathtt{L}_j^k$ can generate.

Then, $\mathsf{Cost}_{clause}(\mathtt{C}_k, \mathtt{ap}, r, \overline{n})$, the resource usage expressed in units of resource r with approximation $\mathtt{ap}$ of clause C of predicate p, is given by the expression $\mathsf{Cost}_{clause}(\mathtt{C}_k, \mathtt{ap}, r, \overline{n}) = \mathsf{solver}(\mathsf{Cost}(\mathtt{C}_k, \mathtt{ap}, r, \overline{n}))$. That is, it is expressed as the solved form function of the following expression which, in general, for recursive clauses yields a recurrence equation:

$$\mathsf{Cost}(\mathtt{C}_k, \mathtt{ap}, r, \overline{n}) = \ \delta(\mathtt{ap}, r)(\mathtt{H}^k, \overline{n}) + \sum_{i=1}^{lim(\mathtt{ap},\mathtt{C}_k)} (\prod_{j \prec i} \mathsf{Sols}_{\mathtt{L}_j^k}(\psi_j(\overline{n})))(\beta(\mathtt{ap}, r)(\mathtt{L}_i^k, \psi_i(\overline{n})) + \mathsf{Cost}_{lit}(\mathtt{L}_i^k, \mathtt{ap}, r, \psi_i(\overline{n}))) \tag{2.4}$$

where $lim(\mathtt{ap}, \mathtt{C}_k)$ is a function that takes an approximation identifier $\mathtt{ap}$ and a clause $\mathtt{C}_k$ and returns the index of a literal in the clause body. For example, if $\mathtt{ap}$ is the identifier for approximation "upper-bound" (`ub`), then $lim(\mathtt{ap}, \mathtt{C}_k) = l$ (the index of the last body literal). If $\mathtt{ap}$ is the identifier for approximation "lower-bounds" (`lb`), then $lim(\mathtt{ap}, \mathtt{C}_k)$ is the index for the rightmost body literal that is guaranteed not to fail. $\delta(\mathtt{ap}, r)$ is a function that takes an approximation identifier $\mathtt{ap}$ and a resource identifier r and returns a function $\Delta^{\mathtt{H}} : \mathit{clause\_head} \times \overline{n} \rightarrow \mathit{arith\_expr}$ which takes a clause head, its input data sizes and returns an arithmetic resource usage expression $< \mathit{arith\_expr} >$ as defined in Figure 2.2. Thus, $\delta(\mathtt{ap}, r)(\mathtt{H}^k, \overline{n})$ represents $\Delta^{\mathtt{H}}(\mathtt{H}^k, \overline{n})$. Note that in $\Delta^{\mathtt{H}}$ we can also take into account additional costs in trying to reach a given clause $\mathtt{C}_k$ due to it will be tried only if clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{k-1}$ fail to yield a solution, or due to the indexing scheme used in the language implementation. On the other hand, $\beta(\mathtt{ap}, r)$ is a function that takes an approximation identifier $\mathtt{ap}$ and a resource identifier r and returns a function $\Delta^{\mathtt{L}} : \mathit{body\_lit} \times \overline{n}_i \rightarrow \mathit{arith\_expr}$ which takes the i-th body literal,

its input data sizes, and returns also an arithmetic resource usage expression $< arith\_expr >$. In this case, $\beta(\mathtt{ap}, r)(\mathtt{L}_i^k, \psi_i(\overline{n}))$ represents $\Delta^{\mathtt{L}}(\mathtt{L}_i^k, \overline{n}_i)$. Section 2.3.4 illustrates different definitions of the functions $\delta(\mathtt{ap}, r)$ and $\beta(\mathtt{ap}, r)$ in order to infer different resources. $\mathsf{Sols}_{\mathtt{L}_j}$ is the number of solutions that literal $\mathtt{L}_j$ can generate, where $j \prec i$ denotes that $\mathtt{L}_j$ precedes $\mathtt{L}_i^k$ in the literal dependency graph for the clause. The inference of upper-bounds on the number of solutions given a literal is far from being trivial. We take the approach of [21]. Finally, $\mathsf{Cost}_{lit}(\mathtt{L}_i^k, \mathtt{ap}, r, \psi_i(\overline{n}))$ is:

- If $\mathtt{L}_i^k$ is recursive (i.e., calls a predicate q which is in the strongly-connected component of the call graph being analyzed), then the expression $\mathsf{Cost}_{lit}(\mathtt{L}_i^k, \mathtt{ap}, r, \psi_i(\overline{n}))$ is replaced by a symbolic expression $\mathsf{Cost}(q, \mathtt{ap}, r, \psi_i(\overline{n}))$.

- If $\mathtt{L}_i^k$ is not recursive, assume that it is a call to q (where q can be either a predicate call, or an external or builtin predicate), then q has been already analyzed, i.e., the (closed form) resource usage function for q has been recursively computed as γ and $\mathsf{Cost}_{lit}(\mathtt{L}_i^k, \mathtt{ap}, r, \psi_i(\overline{n}))$ can be expressed explicitly in terms of the function γ, and it is thus replaced with $\gamma(\psi_i(\overline{n}))$.

Note that in both cases, if there is a resource usage assertion for q, `'cost(ap,r,`$\langle arith\_expr \rangle$`)'`, then $\mathsf{Cost}_{lit}(\mathtt{L}_i^k, \mathtt{ap}, r, \psi_i(\overline{n}))$ is replaced by the most precise, greatest lower-bound if $\mathtt{ap} = \mathtt{ub}$ or least upper-bound if $\mathtt{ap} = \mathtt{lb}$, between the arithmetic resource usage expression in closed form and its closed form resource usage function inferred previously by the analysis, provided they are not incompatible, in which case an error is flagged.
It can be proved by induction on the number of literals in the body of clause `C` that:

1. If clause `C` is not recursive, then expression (2.4) results in a closed form function of the sizes of the input argument positions in the clause head;

2. If clause C is simply recursive, then expression (2.4) results in a recurrence equation in terms of the sizes of the input argument positions in the clause head;

3. If clause C is mutually recursive, then expression (2.4) results in a recurrence equation which is part of a system of equations for mutually recursive clauses in terms of the sizes of the input argument positions in the clause head.

If these recurrence equations can be solved, including approximating the solution in the direction of ap, then $\mathsf{Cost}(p, \mathtt{ap}, r, \overline{n})$ can be expressed in a closed form, which is a function of the sizes of the input argument positions in the head of predicate p (and hence $\mathsf{Cost}_{clause}(\mathtt{C}, \mathtt{ap}, r, \overline{n}) = \mathsf{solver}(\mathsf{Cost}(p, \mathtt{ap}, r, \overline{n}))$). Thus, after the strongly-connected component to which p belongs in the call graph has been analyzed, we have that expression (2.3) results in a closed form function of the sizes of the input argument positions in the clause head.

Finally, note that our analysis is parametrized by the functions $\delta(\mathtt{ap}, r)$ and $\beta(\mathtt{ap}, r)$ whose definitions can be given by means of assertions of type head_cost and literal_cost respectively, as shown in Figure 2.2. These functions make our analysis parametric with respect to any resource of interest defined by users.

2.3.4 Defining the Parameters (Functions) of the Analysis

In this section we explain and illustrate with examples how the functions that make our resource analysis parametric, namely, δ (which includes the definition of $\Delta^{\mathtt{H}}$), and β (which includes the definition of $\Delta^{\mathtt{L}}$) are written in practice in our system. Again, we assume that we are interested in computing upper-bounds on the different resources.

Assume for example that the resource we want to measure is an upper-bound on the number of resolution steps performed by a program. This is

achieved by providing the following `head_cost` assertion and definition of the `delta_one/2` predicate:

```
:- head_cost(ub,steps,delta_one).
delta_one(_,1).
```

In order to simplify the process of defining interesting and useful $\Delta^{\mathtt{H}}$ and $\Delta^{\mathtt{L}}$ functions, our implementation provides a library with predicates that perform syntactic operations on clauses, such as, for example, getting the number of arguments in a clause head or body literal, get a clause head, get a clause body, accessing an argument of a clause head or body literal, getting the main functor and arity of a term in a certain position, etc. In this context it is important to remember that the different $\Delta^{\mathtt{H}}$ and $\Delta^{\mathtt{L}}$ function definitions perform syntactic matching on the program text.

Assume now that the resource we want to measure is the number of argument passings that occur during clause head matching in a program (as an approximation to the number of unifications performed by the program). This is achieved by the following code:

```
delta_num_args(H,N) :- functor(H,_,N).
```

As another example, if we are interested in decomposing arbitrary unifications performed while unifying a clause head with the literal being solved into simpler steps, we can define a resource `num_unifs`, and a `head_cost` assertion which counts the number of function symbols, constants, and variables in each clause head as follows:

```
:- head_cost(ub,num_unifs,
             delta_num_unifs).

num_fun_vars(0,_H,0).
num_fun_vars(N,H,S) :-
    N > 0,
    arg(N,H,Arg),
    nfun_vars(Arg,S1),
    N1 is N-1,
    num_fun_vars(N1,H,S2),
    S is S1 + S2.
```

```
delta_num_unifs(H,S) :-
    functor(H,_,N),
    num_fun_vars(N,H,S).

nfun_vars(Arg,1) :-
    var(Arg).
nfun_vars(Arg,1) :-
    atomic(Arg).
nfun_vars(Arg,S) :-
    nonvar(Arg),
    functor(Arg,_,N),
    num_fun_vars(N,Arg,S1),
    S is S1 + 1.
```

If, in addition to the number of unifications performed while unifying a clause head, we are also interested in the cost of term creation for the literals in the body of clauses, we can define a resource `terms_created`, and define `literal_cost` assertion which keeps track of the number of function symbols, and constants in body literals:

```
:- literal_cost(ub,
    terms_created,
    beta_terms_created).

beta_terms_created(L,S) :-
     functor(L,_,N),
     num_fun(N,L,S).

num_fun(0,_L,0).
num_fun(N,L,S) :-
     N > 0,
     arg(N,L,Arg),
     nfun(Arg,S1),
     N1 is N-1,
     num_fun(N1,L,S2),
     S is S1 + S2.
```

```
:- head_cost(ub,
    terms_created,
    delta_terms_created).

delta_terms_created(_L,0).

nfun(Arg,0) :-
     var(Arg).
nfun(Arg,1) :-
     atomic(Arg).
nfun(Arg,S) :-
     nonvar(Arg),
     functor(Arg,_,N),
     num_fun(N,Arg,S1),
     S is S1 + 1.
```

Note that in this case we also define a `head_cost` assertion which returns 0 for every clause head.

More interestingly, our implementation provides a library with predicates that perform semantic checks of properties. These properties are inferred by

the available analyzers. Some of the analyses are always performed as part of the resource analysis, as mode and type analysis, and others are performed on demand, depending on the properties that need to be checked in the $\Delta^{\mathtt{H}}$ and $\Delta^{\mathtt{L}}$ function definitions or depending on the type of approximation to be performed by the resource analysis.

Assume now that we want to differentiate the counting of unifications where one of the terms being unified is a variable and thus behave as an "assignment," and the counting of full unifications, i.e., when both terms being unified are not variables, and thus unification performs a "test" or produces new terms, etc.

For this purpose, we can define a resource, as for example `vo_unif` which counts the number of variables in the clause head which correspond to "output" argument positions through a `head_cost` assertions. This describes a component of the execution time that is directly proportional to the number of cases where both a goal argument and the corresponding head argument are variables. This should boil down to assignment (maybe with trailing). This is achieved by the following code:

```
:- head_cost(ub,vo_unif,
             delta_vo_unif).
delta_vo_unif(H,S) :-
    functor(H,_,N),
    num_vo_unif(N, H, S).

num_vo_unif(0,_H,0) :- !.
num_vo_unif(N,H,S) :-
    arg(N,H,Arg),
    free(Arg),
    !,
    nvo_unif(Arg,S1),
    N1 is N-1,
    num_vo_unif(N1, H, S2),
    S is S1 + S2.
```

```
num_vo_unif(N,H,S) :-
    N1 is N-1,
    num_vo_unif(N1,H,S).

nvo_unif(Arg,1) :-
    var(Arg).
nvo_unif(Arg,0) :-
    atomic(Arg).
nvo_unif(Arg,S) :-
    nonvar(Arg),
    functor(Arg,_, N),
    num_vo_unif(N,Arg,S1),
    S is S1 + 1.
```

Similarly, we could define resources for counting:

- The number of variables in the clause head which correspond to input

argument positions,

- the number of function symbols and constants in the clause head which appear in output arguments, or

- the number of function symbols and constants in the clause head which appear in input arguments.

Example 2.3.2. Consider the same program defined in Figure 2.1 and the size relations computed in Example 2.3.1. We now show the corresponding resource usage equations for each clause for the resource `bits_received`, denoted by `bits` for brevity, inferred automatically by our system. Although the functions $\delta(\mathtt{ap}, r)(\mathtt{H}, \overline{n})$ and $\beta(\mathtt{ap}, r)(\mathtt{L}_i, \overline{n}_i)$ take as arguments a clause head `H` and a body literal $\mathtt{L}_i$ respectively, in our examples we will only write the predicate name of `H` and $\mathtt{L}_i$ for the sake of simplicity. Since the program is analyzed in a single traversal of the call graph in reverse topological order, the system starts by analyzing the predicate `exch_buffer/3`. Note that the resource usage for external predicates (whose code is not available) `connect/3`, `exch_byte/3` and `close/1` is already given by "trust" assertions which express that:

$$\begin{aligned} \mathsf{Cost}(\mathit{connect}, \mathtt{ub}, \mathtt{bits}, \langle \_, \_ \rangle) &= 0 \\ \mathsf{Cost}(\mathit{exch\_byt}, \mathtt{ub}, \mathtt{bits}, \langle \_, \_ \rangle) &= 8 \\ \mathsf{Cost}(\mathit{close}, \mathtt{ub}, \mathtt{bits}, \langle \_ \rangle) &= 0 \end{aligned}$$

For the recursive clause of `exch_buffer/3`, the system sets up the following recurrence equation, where n represents the length of the first argument to this predicate (note that the system infers the "length" size metric for this argument and that $n > 0$):

$$\begin{aligned}\mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n, \_\rangle) = \; & \overbrace{\delta(\mathtt{ub}, \mathtt{bits})(exch\_buffer)}^{0} + \\ & \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(exch\_byte)}^{0} + \overbrace{\mathsf{Cost}(exch\_byte, \mathtt{ub}, \mathtt{bits}, \langle \_, \_\rangle)}^{8} + \\ & \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(exch\_buffer)}^{0} + \mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n-1, \_\rangle) \\ & = 8 + \mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n-1, \_\rangle)\end{aligned}$$

For the non-recursive clause of `exch_buffer/3` the system infers:

$$\mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle 0, \_\rangle) = 0$$

which can be used as boundary condition for solving the previous recurrence equation, yielding the following closed form resource usage function:

$$\mathsf{Cost}(exch\_buf, \mathtt{ub}, \mathtt{bits}, \langle n, \_\rangle) = 8 \times n$$

Now, the `client/3` predicate is analyzed, and the system sets up the following expression for its only clause (where k is the length of the input buffer, i.e., the second argument to this predicate):

$$\begin{aligned}\mathsf{Cost}(client, \mathtt{ub}, \mathtt{bits}, \langle \_, k\rangle) = \; & \overbrace{\delta(\mathtt{ub}, \mathtt{bits})(client)}^{0} + \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(connect)}^{0} + \\ & \overbrace{\mathsf{Cost}(connect, \mathtt{ub}, \mathtt{bits}, \langle \_, \_\rangle)}^{0} + \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(exch\_buffer)}^{0} + \\ & \overbrace{\mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle k, \_\rangle)}^{8\times k} + \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(close)}^{0} + \\ & \overbrace{\mathsf{Cost}(close, \mathtt{ub}, \mathtt{bits}, \langle \_\rangle)}^{0} = 8 \times k\end{aligned}$$

2.4 Experimental Results

To study the feasibility of the approach we have completed a prototype implementation of the analyzer. It is written in the Ciao language and uses a

```
:-pred merge/1: list(atm).

merge(FileList) :-
  open('all_files',write,OS),
  merge_files(FileList,OS),
  close(OS).

merge_files([],_OS).
merge_files([F|Fs],OS):-
  open(F,read,IS),
  read(IS,E),
  write(OS,E),
  close(IS),
  merge_files(Fs,OS).


:-head_cost(ub,leftopen,0).
:-literal_cost(ub,leftopen,0).

:-head_cost(lb,leftopen,0).
:-literal_cost(lb,leftopen,0).
```

```
/* FILE LIBRARY */

:-trust pred open(FileN,Mode,
          Stream):
      atm * atm * var =>
      atm * atm * int
      + ( cost(ub,leftopen,1),
          cost(lb,leftopen,1),
          not_fails).

:-trust pred close(Stream): int
      + ( cost(ub,leftopen,-1),
          cost(lb,leftopen,-1),
          not_fails).

:-trust pred read(Stream,Data):
      int * var =>
      int * list(atm)
      + ( cost(ub,leftopen,0),
          cost(lb,leftopen,0),
          not_fails).

:-trust pred write(Stream,Data):
      int * list(atm)
      + ( cost(ub,leftopen,0),
          cost(lb,leftopen,0),
          not_fails).
```

Figure 2.4: An application that merges the content of a set of files.

number of modules and facilities from CiaoPP, the Ciao preprocessor (including difference equation processing). We have also written a Ciao language extension (a "package" in Ciao terminology) which when loaded into a module allows writing the resource-related assertions and declarations proposed herein.† We have then used this prototype to analyze a set of representative benchmarks which include definitions of resources using this language and used the system to infer the resource usage bound functions.

First, we show the actual resource for which bounds are being inferred by the analysis for a given benchmark together with a brief description. In

†The system also supports adding resource assertions specifying expected resource usages which the implemented analyzer will then verify or falsify using the results of the implemented analysis.

```
:-entry insert/3:
   bst * num * var.
insert(nil,E,tree(E,nil,nil)).
insert(tree(N,L,R),E,
       tree(N,L,R)):-
  N == E,!.
insert(tree(N,L,R),E,
       tree(N,NL,R)):-
  E < N, !,
  insert(L,E,NL).
insert(tree(N,L,R),E,
       tree(N,L,NR)):-
  E > N,
  insert(R,E,NR).

:-regtype bst/1.
bst(nil).
bst(tree(Node,Left,Rigth)):-
  num(Node),
  bst(Left),bst(Rigth).

:-head_cost(ub,heap_usage,
     heap_usage_function).
:-literal_cost(ub,heap_usage,
     heap_usage_function).

heap_usage_function(LitInfo,
                    Cost):-
  get_literal(LitInfo,Head),
  get_modes(LitInfo,Modes),
  usage_func(Modes,Head,1,0,
             Cost).
```

```
usage_func([],_,_,Cost,Cost).
usage_func([in|Modes],Head,Ind,
           Acc,Cost):-
     NInd is Ind + 1,
     usage_func(Modes,Head,NInd,
                Acc,Cost).
usage_func([out|Modes],Head,Ind,
           Acc,Cost):-
     arg(Index,Head,Term),
     term_heap_usage(Term,Cost),
     NAcc is Acc + Cost,
     NInd is Ind + 1,
     usage_func(Modes,Head,NInd,
                NAcc,Cost).

term_heap_usage(Term,4):-
     var(Term),!.
term_heap_usage(Term,4):-
     atm(Term),!.
term_heap_usage(Term,N):-
     functor(Term,F,_A),
     Term =.. [F|Ts],
     term_heap_usage_(Ts,N1),
     N is N1 + 4.

term_heap_usage_([],0).
term_heap_usage_([T|Ts],N):-
     term_heap_usage(T,N1),
     term_heap_usage_(Ts,N2),
     N is N1 + N2.
```

Figure 2.5: Insertion in a binary search tree

addition, we also show the size metric used for the relevant arguments. While any of the resources defined in a given benchmark could then be used in any of the others we show only the results for the most natural or interesting resource for each one of them. We have tried to use a relatively wide range of resources: number of bytes sent by an application, number of calls to a particular predicate, robot arm movements, number of files left open in a kernel code, number of accesses to a database, heap memory usage, etc. We also cover a significant set of complexity functions such as constant, poly-

```
:- pred hanoi(N,A,B,C): num * elem * elem * elem.
:- trust comp hanoi(N,A,B,C) + (size_metric(A,void),
                                 size_metric(B,void),
                                 size_metric(C,void)).

hanoi(1,A,_,C):-
        move_disk(A,C),
        !.
hanoi(N,A,B,C) :-
        N1 is N - 1,
        hanoi(N1,A,C,B),
        move_disk(A,C),
        hanoi(N1,B,A,C).


:- trust pred move_disk(A,B): elem * elem
      + cost(ub,energy,3).

:- head_cost(ub,energy,0).
:- literal_cost(ub,energy,0).
```

Figure 2.6: The Towers of Hanoi program using robotic arms

nomial, and exponential using relevant data structures in Prolog programs such as lists, trees, etc.

- `bst` is the program shown in Figure 2.5 and we measure the heap usage in terms of number of bytes as a function on the depth of the input argument.

- `client` is the program depicted in Figure 2.1 and we measure the number of bits received by the application as a function on the length of the input argument.

- `color_map` performs map coloring and we measure the number of unifications as a function that depends on the term size of one of the input arguments.

- `fib` computes the Fibonacci function and infers the number of arithmetic operations depending on the integer value of the input argument.

- `hanoi` is the program shown in Figure 2.6 and we want to measure the number of robot movements as a function that depends on the integer value of the input argument.

- `eight_queen` plays the 8-queens game and we measure the number of queens movements as a function on the length of the input argument.

- `eval_polynom` evaluates a polynomial function and we measure the floating point unit time usage as a function on the length of the list of coefficients.

- `grammar` represents a simple sentence parser and we measure the number of phrases generated by the parser as a function on the term size of the input argument.

- `insert_stores` is a database transaction that adds a new entry into the `STORE` relation. We measure the number of updates as a function on the relation size, i.e. number of records.

- `merge` is the same program illustrated in Figure 2.4 and we measure the number of files left open as a function that depends on the length of the list of files.

- `perm`: performs a permutation of a list and we measure the number of WAM instructions as a function on the input list length.

- `power_set` generates the powerset of a list and we measure the number of output elements as a function that depends on the input list length.

- `qsort` implements the quicksort algorithm and we measure the number of lists parallelized as a function on the input list length.

- `send_files` is a program that sends the content of a set of files through a stream. We measure the number of bytes read as a function on the input list length.

Program	Usage Function	Exact Function	Time
`bst`	$\lambda x.20 \cdot x + 16$	$\lambda x.20 \cdot x + 16$	184
`client`	$\lambda x.8 \cdot x$	$\lambda x.8 \cdot x$	186
`color_map`	104691	31686	176
`eight_queen`	19173961	19173961	304
`eval_polynom`	$\lambda x.2.5x$	$\lambda x.2.5x$	44
`fib`	$\lambda x.2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3$	$\lambda x.2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3$	116
`grammar`	24	16	227
`hanoi`	$\lambda x.2^x - 1$	$\lambda x.2^x - 1$	100
`insert_stores`	$\lambda n, m.n + k$ $\lambda n, m.n$	$-$ $-$	292
`merge`	$\lambda x.x$	$\lambda x.x$	180
`perm`	$\lambda x.46 \sum_{i=1}^{x} \cdot \frac{x!}{(i-1)!} + 32 \sum_{i=1}^{x} \cdot \frac{x!}{i!} + 4 \cdot x!$	$-$	98
`power_set`	$\lambda x.\frac{1}{2} \cdot 2^{x+1}$	$\lambda x.\frac{1}{2} \cdot 2^{x+1}$	119
`qsort`	$\lambda x.4 \cdot 2^x - 2x - 4$	$\lambda x.2 \cdot x^2$	144
`send_files`	$\lambda x, y.x \cdot y$	$\lambda x, y.x \cdot y$	179
`subst_exp`	$\lambda x, y.2xy + 2y$	$\lambda x, y.x \cdot y$	153
`zebra`	30232844295713061	6869	292

Table 2.1: Accuracy and efficiency in milliseconds of the analysis.

- `subst_exp` substitutes a list of variables in a mathematical expression. We measure the number of replacements as a function on the list length and also the term size of the input arguments.

- `zebra` is based on the classic zebra puzzle, and we measure the number of resolution steps as a function on the term size of the input.

The results from the analysis of these benchmarks are shown in Table 2.1. For brevity, we report only results for upper-bounds analysis. The column

Usage Function shows the actual resource usage function (which depends on the size of the input arguments) inferred by the analysis, given as a lambda term. The column Exact Function shows the exact resource usage function, given also as a lambda term. Finally, the column labeled Time shows the resource analysis times in milliseconds, on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 5.0. Note that these times do not include other analyses such as types, modes, etc.

2.5 Chapter Conclusions

We have presented a static analysis that infers upper- and lower-bounds on the usage that a logic program makes of a quite general notion of user-definable resources, and shown several useful applications. The inferred bounds are in general functions of input data sizes. We have also presented the assertion language which is used to define such resources. The analysis then derives the related (upper- and lower-bound) resource usage functions for all predicates in the program. Our preliminary experimental results are encouraging because they show that interesting resource bound functions can be obtained automatically and in reasonable time, at least for our benchmarks. While clearly further work is needed to assess scalability we are cautiously hopeful in the sense that our approach allows defining via assertions the resource usage of external predicates, which can then be used for modular composition. These includes also predicates for which the code is not available or which are written in a programming language that is not supported by the analyzer. In addition, assertions also allow describing by hand the usage of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. Our expectation is that the automatic analysis will be able to do the bulk of the work for large applications, even if the cost of some specially complex predicates may still

need to be given by the user. In particular, for the examples in Table 2.1 all results where obtained automatically. Finally, we expect the applications of our analysis to be rather interesting, including resource consumption verification and debugging (including for mobile code), resource control in parallel/distributed computing, and resource-oriented specialization.

Chapter 3

Applying the Framework to Execution Time Estimation

3.1 Introduction

Predicting statically the running time of programs has many applications ranging from task scheduling in parallel execution to proving the ability of a program to meet strict time constraints in real-time systems. A starting point in order to attack this problem is to infer the computational complexity of such programs. This is one of the reasons why the development of static analysis techniques for inferring cost-related properties of programs has received considerable attention.

However, as mentioned previously, in most cases such cost properties are expressed using platform-independent metrics (like number of *reductions* or *execution steps*). Such platform-independent cost information has been shown to be quite useful in various applications. This includes, for example, scheduling parallel tasks [41, 43, 32]. In a typical scenario, these tasks will be executed in a single parallel machine, where all processors are typically identical. Therefore, the deduced number of reductions can actually be used as a relative measure in order to compare to a first degree of approximation the amount of work under the tasks. However, in distributed execution

and other mobile/pervasive computation scenarios, where different platforms come into play with each platform having different computing power, it becomes necessary to express costs in metrics that can be later instantiated to different architectures so that actual running time can be compared using the same units. This applies also to heterogeneous parallel computing platforms. Moreover, although the number of *execution steps* is a measure that has the advantage of being platform independent, it is not straightforward to translate such steps into execution time.

With this in mind, we define a framework for automatically inferring both upper- and lower-bounds on execution times that are in general functions that depend on input data sizes. The framework has been implemented as part of the `CiaoPP` [36] system, and includes a global static analysis which is an instantiation (to execution time estimation) of the general resource usage analysis defined in Chapter 2. It also combines compile-time analysis with a one-time, program-independent profiling stage of a given platform in order to determine the values of certain parameters for that platform. These parameters calibrate a cost model which, from then on, is able to compute statically time bound functions for procedures and to predict with a significant degree of accuracy the execution times of such procedures in that concrete platform. We define several cost models for execution time estimation (parameterized with respect to the execution platform), and perform an assessment of them regarding the trade-off between accuracy and efficiency. Such models are evaluated and the advantages and disadvantages of them shown.

In the following sections we present our framework showing two approaches for two kinds of cost models (high- and low-level cost models). The first approach is described in Section 3.2, were we define several cost models based on high-level (source) language characteristics in which we can measure certain aspects that are platform-independent. Then, using profiling techniques, we calibrate the platform-dependent parameters for a specific platform. Such models are evaluated in order to see the best trade-off between simplicity and precision of the model. Although promising exper-

imental results were obtained, the predicted execution times were not very precise.

The second approach is defined in Section 3.3, where we define a low-level cost model applicable to logic programs running on a bytecode-based abstract machine. In this case, the one-time, program-independent profiling stage calculates constants or functions bounding the execution time of each abstract machine instruction. As expected, this model predicts more precise execution times than the former ones based on high-level syntactic characteristics of programs.

In addition to cost analysis, the implementation of profilers in declarative languages has also been considered by various authors, with the aim of helping to discover why a part of a program does not exhibit the expected performance. Debray [20] showed the basic considerations to have in mind when profiling Prolog programs: handling backtracking and failure. Ducassé [25] designed and implemented a trace analyzer for Prolog which can be applied to profiling. Sansom and Peyton Jones [62] focused on profiling of functional languages using a semantic approach and highlighted the difficulty in profiling such kind of languages. Jarvis and Morgan [59] showed how to profile lazy functional programs. Brassel et al. [10] solved part of the difficulty in profiling when considering special features in functional logic programs, like sharing, laziness and non-determinism. We will use also profiling but, since our aim is to *predict* performance, profiling will in our case be aimed at calibrating the values for some constants that appear in the cost functions, and which will be instrumental to forecast execution times for a given platform and cost model. Therefore we will not use profiling with just some fixed input arguments, but with a set of programs and input arguments which we hope will be representative enough to derive meaningful characteristics of an execution platform.

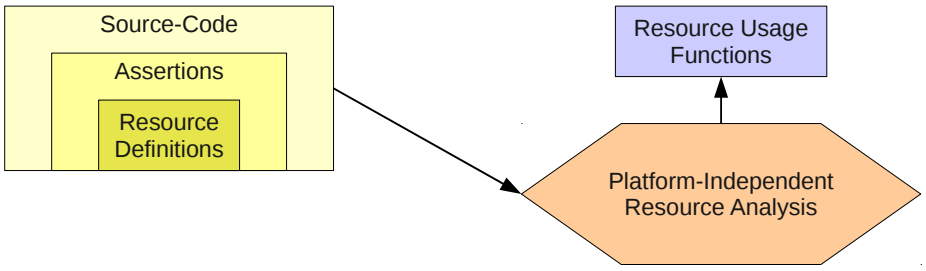

Figure 3.1: Source-Level/Platform-Independent Resource Analysis.

3.2 Source Code-Based (High-Level) Model

In this section we present a framework which combines static cost analysis with profiling techniques in order to infer functions which yield upper- and lower-bounds on *execution times* of program procedures [48, 47, 49]. This framework is based on the analysis of high-level syntactic characteristics of the program clause text such as sizes of terms in heads, sizes of terms in bodies, or number of arguments. In this approach, platform-independent cost functions are first inferred which are parameterized by certain constants (see Figure 3.1). These constants aim at capturing the execution time of certain low-level operations on each platform. For each execution platform, the value of such constants is determined experimentally once and for all by running a set of special-purpose synthetic benchmarks and measuring their running times with a profiling toolkit that we have also developed. Once these constants are determined, they are fed into the model with the objective of predicting with a certain accuracy execution times (see Figure 3.2).

We have studied a relatively large number of cost models, involving different sets of constants in order to explore experimentally which of the models produces the most precise results, i.e., which parameters model and predict

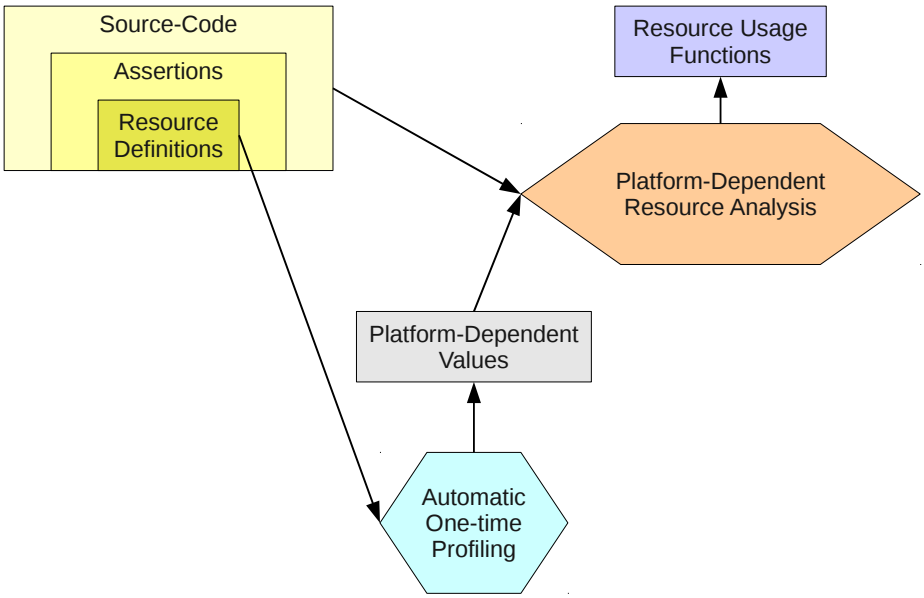

Figure 3.2: Source-Level/Platform-Dependent Resource Analysis.

best the actual execution times of procedures. In doing this we have taken into account the trade-off between simplicity of the cost models (which implies efficiency of the cost analysis and also simpler profiling) and the precision of their results. With this aim, we have started with a simple model and explored several possible refinements.

Note that although we have developed the framework for execution time estimation, it can also be applied to the estimation of other platform-dependent and -independent resources, provided that the appropriated source-level based resources are defined. Thus, what we are proposing is a general approach (see Figure 3.3).

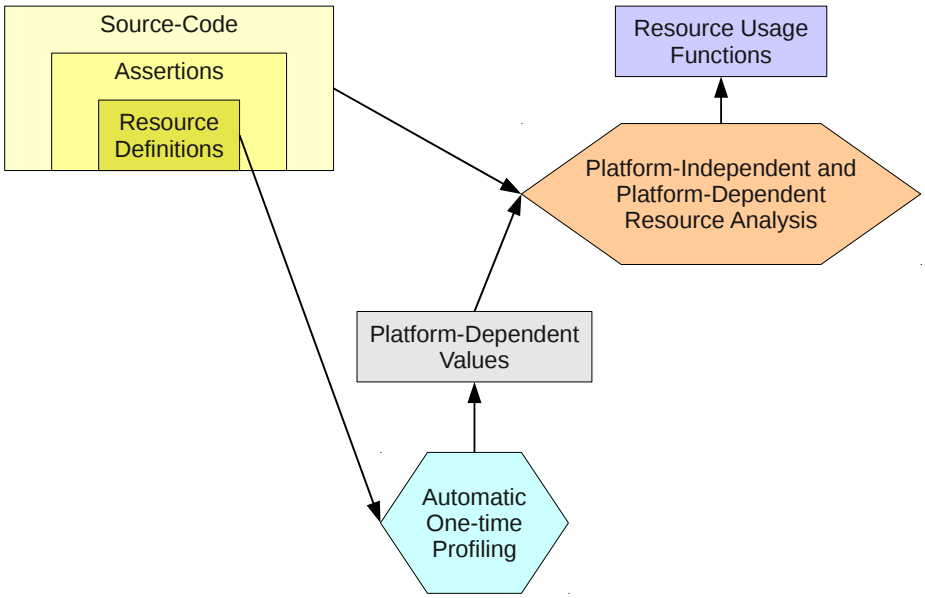

Figure 3.3: Source-Level Resource Analysis.

3.2.1 Proposed Platform-Dependent Cost Models

In this section we describe how to particularize the general resource usage expressions defined in Chapter 2 in order to estimate bounds on the execution time of clauses and predicates using platform-dependent cost models. For simplicity, the discussion that follows is focused on the estimation of upper-bounds on execution times. We refer the reader to [23] for details on lower-bounds cost analysis. We also use the terms *resource usage function* and *cost function* indistinctly. Thus, we instantiate the function $\delta(\mathtt{ap}, r)$ appearing in expression (2.4), Section 2.3.3 (Chapter 2), to a function $\Delta^{\mathtt{H}}$ that represents the time needed to resolve a given literal against the corresponding clause head $\mathtt{H}^k$, but also the cost associated with selecting alternatives, the cost coming from setting up the body literals for execution, allocating

activation records, etc. In the following, we will still refer to $\Delta^{\mathtt{H}}$ as the *clause head cost function* (but understanding that it now includes all these costs). Similarly, we instantiate $\beta(ap, r)$ appearing in the same expression (2.4) to a function $\Delta^{\mathtt{L}}$, representing the time needed to prepare the arguments just before calling the given literal in the clause body. To simplify our discussion we will assume that $\Delta^{\mathtt{L}} = 0$, although in practice we use a function that takes into account the mentioned costs. We will consider different definitions for $\Delta^{\mathtt{H}}$, each of them yielding a different cost model. These cost models make use of a vector of platform-dependent constants, together with a vector of platform-independent resource usage metrics, each one corresponding to a particular low-level operation related to program execution. Examples of such low-level operations considered by the cost models are unifications where one of the terms being unified is a variable and thus behave as an "assignment," or full unifications, i.e., when both terms being unified are not variables, and thus unification performs a "test" or produces new terms, etc.

Thus, we generalize $\Delta^{\mathtt{H}}_{\Omega}$ to be a function parameterized by the cost model, so that:

$$\Delta^{\mathtt{H}}_{\Omega} = \mathtt{time}(\Omega) \tag{3.1}$$

where $\mathtt{time}(\Omega)$ returns the time associated to a resolution step, including the aforementioned additional overheads. The parameter $\Omega = (\omega_1, \ldots, \omega_v)$ is a vector denoting which characteristics we want to take into account: every ω_i looks at a different indicator (i.e., a platform-independent resource) of the execution time. The family of cost models we will study assumes that $\mathtt{time}(\Omega)$ is defined as follows:

$$\mathtt{time}(\Omega) = \mathtt{time}(\omega_1) + \cdots + \mathtt{time}(\omega_v),\ v > 0 \tag{3.2}$$

where each $\mathtt{time}(\omega_i)$ contributes with the part of the execution time which depends on the feature ω_i. We also assume that:

$$\mathtt{time}(\omega_i) = K_{\omega_i} \times I(\omega_i) \tag{3.3}$$

where K_{ω_i} is a platform-dependent constant and $I(\omega_i)$ is a platform-independent cost function. I.e., K_{ω_i} expresses the cost of each unit of $I(\omega_i)$ in terms of time. Equation (3.2) can be written in vector notation as

$$\texttt{time}(\Omega) = \overline{K}_\Omega \bullet \overline{I(\Omega)} \tag{3.4}$$

where $\overline{K}_\Omega = (K_{\omega_1}, \ldots, K_{\omega_v})$ and $\overline{I(\Omega)} = (I(\omega_1), \ldots, I(\omega_v))$ are vectors of platform-dependent constants and of platform-independent cost functions, respectively.

A cost model, of which we have tested several, is given by a particular definition of the parameter Ω. Every cost model is defined by the program characteristics taken into account by it. While a large number of indicators can be used, we have identified some of them as specially interesting. We list them below, giving a mnemonic to every ω_i and explaining the meaning of each $I(\omega_i)$.

In what follows we will say that an argument of a literal is an *output argument* if the term being passed by the calling literal is known to be a variable at run-time, and an *input argument* if it is not a variable. Run-time arguments can be classified as either input or output using well-known techniques for mode analyses (in our case, those provided by `CiaoPP`).

$I(steps) = 1$ Every successful *head traversal* has a constant weight in the execution. I.e., in equation (3.3), we have:

$$\texttt{time}(steps) = K_{steps}$$

$I(vounif) =$ *the number of variables in the clause head which correspond to "output" argument positions.* This describes a component of the execution time that is directly proportional to the number of cases where both a goal argument and the corresponding head argument are variables. This should boil down to assignment (maybe with trailing):

$$\texttt{time}(vounif) = K_{vounif} \times I(vounif)$$

$I(viunif) =$ *the number of variables in the clause head which correspond to "input" argument positions.* This component corresponds to the number of non-variable goal arguments which are unified with a variable in the head. The unification for such arguments is also similar to an assignment with a small, constant cost. We assume that the cost of creating the input argument is constant. Given these assumptions:

$$\texttt{time}(viunif) = K_{viunif} \times I(viunif)$$

$I(gounif) =$ *The number of function symbols and constants in the clause head which appear in output arguments.* We are capturing here the size of the terms that are created when a variable in a goal is unified with a non-variable in the clause head:

$$\texttt{time}(gounif) = K_{gounif} \times I(gounif)$$

$I(giunif) =$ *The number of function symbols and constants in the clause head which appear in input arguments.* We assume that there is a component of the execution time which depends on the number of arguments in which neither the goal nor the clause head arguments are variables. For each of these arguments, we take into account the number of symbols in the clause head:

$$\texttt{time}(giunif) = K_{giunif} \times I(giunif)$$

$I(nargs) = arity(H)$ we are assuming that there is a component of the execution time that depends on the number of arguments in the clause head:

$$\texttt{time}(nargs) = K_{nargs} \times I(nargs) \tag{3.5}$$

This component is obviously redundant with respect to the previous ones, but we have included it as a statistical control: the experiments

should show (and do show) that it is irrelevant when the others are used.

Clearly, other components can be included (such as whether activation records are created or not) but our objective is to see how far we can go with the components outlined above.

Let $\Phi(p,\texttt{time},\overline{n})$ denote the execution time of a call to predicate p for an input of size $\overline{n}$, and let $\Phi(p,\texttt{time}_\Omega,\overline{n})$ be a function which estimates it according to cost model Ω. We have that:

$$\Phi(p,\texttt{time}_\Omega,\overline{n}) = \overline{K}_\Omega \bullet \overline{\Phi}(p,I_\Omega,\overline{n}) \tag{3.6}$$

where $\texttt{time}_\Omega$ is the resource identifier for the execution time estimated using model Ω, i.e., using $\Delta^{\texttt{H}}_\Omega = \texttt{time}(\Omega)$ as head cost function. $\overline{K}_\Omega$ and $\overline{\Phi}(p,I_\Omega,\overline{n})$ are vectors of the form:

$\overline{K}_\Omega = (K_{\omega_1}, \ldots, K_{\omega_v})$,

$\overline{\Phi}(p,I_\Omega,\overline{n}) = (\Phi(p,\omega_1,\overline{n}), \ldots, \Phi(p,\omega_v,\overline{n}))$,

where $\Phi(p,\omega_i,\overline{n})$ is the resource usage function that gives the platform-independent resource identified as ω_i, i.e., using $\Delta^{\texttt{H}}_{\omega_i} = I(\omega_i)$ as head cost function.

Equation (3.6) gives the basis for computing values for constants K_{ω_i} via profiling (as will be explained in Section 3.2.3). Also, it provides an approach to obtain the cost of a procedure expressed in a platform-dependent resource usage metric from another cost expressed in a platform-independent resource usage metric. In this approach, the compile-time cost bounds analysis gives a vector of platform-independent resources that corresponds to particular low-level operations related to program execution (such operations must be reflected in the high-level language.). The profiling phase determines the values of the constants appearing in the resource usage functions, for a given platform. Then, assertions are used to define the platform-dependent resource (execution time) as a composition of the basic platform-independent

resources and the values of the constants resulting from the profiling phase. This is illustrated in Figure 3.2.

3.2.2 Dealing with Builtins

In this section we present our approach to the cost analysis of programs which call builtins, or more generally, predicates whose code is not available to the analyzer (external predicates). We will refer to all of them as builtins for brevity. We assume that a cost function is available (expressed via `trust` assertions [35]) for each such predicate. This cost function can be a constant in simple cases but more generally it will be a function that depends on sizes of the (input) arguments of the predicate. As an example, the cost of arithmetic predicates (such as `=:=/2`, `=\=/2`, or `>/2`) is approximated by a function that depends on the size (and types) of the arithmetic expressions that will appear as arguments.

Note that this is a significant change with respect to the cost analysis proposed in [21] since one of the simplifying assumptions made in that analysis was to not count calls to certain builtin as resolution steps (which meant that they were simply ignored in the cost analysis). While such an assumption made sense for inferring *number of resolution steps*, the assumption is not realistic for estimating *execution times*, since the time involved in executing such builtins is not negligible in general and thus has to be taken into account.

We have modeled this by assuming that each builtin contributes with a new component of the cost model to the execution time as expressed in Equation (3.2). Then, a new $\texttt{time}(\omega_i)$ is added for each builtin predicate `b/n` as follows:

$$\texttt{time}(b/n) = K_{b/n} \times I(b/n)$$

We now consider in more detail the case of arithmetic operators and discuss several possibilities. For the sake of accuracy, every arithmetic operator can be dealt with separately: let $\odot/n$ be an arithmetic operator. As usual,

the execution time due to the total number of times that this operator is evaluated is given by:

$$\texttt{time}(\odot/n) = K_{\odot/n} \times I(\odot/n)$$

where $K_{\odot/n}$ approximates the time taken by the evaluation of the arithmetic operator $\odot/n$. $I(\odot/n)$ could be the number of times that the arithmetic operator is evaluated. With these assumptions, Equation (3.6) (in Section 3.2.1) also holds for programs that perform calls to builtin predicates, say, for example, a builtin b/n, by introducing b/n and $\odot/n$ as new cost components of Ω.

Alternatively, $I(\odot/n)$ can be a cost function defined as:

$$I(\odot/n) = \sum_{a \in S} \texttt{EvCost}(\odot/n, a)$$

where S is the set of arithmetic expressions appearing in the clause body which will be evaluated; and $\texttt{EvCost}(\odot/n, a)$ represents the cost corresponding to the operator $\odot/n$ in the evaluation of the arithmetic term a, i.e.:

$$\texttt{EvCost}(\odot/n, A) = \begin{cases} 0 & \text{if } \texttt{atomic}(A) \vee \texttt{var}(A) \\ 1 + \sum_{i=1}^{n} \texttt{EvCost}(\odot/n, A_i) & \text{if } A = \odot(A_1, ..., A_n) \\ \sum_{i=1}^{m} \texttt{EvCost}(\odot/n, A_i) & \text{if } A = \hat{\odot}(A_1, ..., A_m), \hat{\odot} \neq \odot \end{cases}$$

For simplicity we can make the assumption that the cost of evaluating the arithmetic term t to which a variable appearing in A will be bound at execution time is zero (i.e., to ignore the cost of evaluating t). This can be a good approximation if in most cases t is a number and thus no evaluation of a complex expression is needed for it. This is the case in our simple benchmarks and our experimental results show good time predictions for arithmetic builtin predicates using just the simple cost model. On the other hand, a more refined cost model which assumes that cost is a function on

the size of t will be needed for programs which evaluate symbolic arithmetic expressions.

Note that the simple models that we have discussed ignore the possible optimizations that the compiler might perform. We can take into account those performed by source-to-source transformation by placing our analyses in the last stage of the front-end, but at some point the language the compiler works with would be different enough as to require different considerations in the cost model.

3.2.3 Calibrating Constants via Profiling

In order to compute values for the platform-dependent constants which appear in the different cost models proposed in Section 3.2.1, our calibration schema takes advantage of the relationship between the platform-dependent and -independent cost metrics expressed in Equation (3.6). In this sense, the calibration of the constants appearing in $\overline{K}_\Omega$ is performed by solving systems of linear equations (in which such constants are treated as variables).
Based on this expression, the calibration procedure consists of:

1. Using a selected set of calibration programs which aim at isolating specific aspects that affect execution time in general cases. For these calibration programs it holds that $\Phi(p, \omega_i, \overline{n})$, i.e., the function giving the cost (in units of resource ω_i) of predicate `p` for an input of size $\overline{n}$, is known for all $1 \leq i \leq v$. This can be done by using any of the following methods:

   - The analyzers integrated in the `CiaoPP` system infer the exact cost function, i.e., both upper- an lower-bounds are the same: $\texttt{Cost}(p,\texttt{lb},\omega_i,\overline{n}) = \texttt{Cost}(p,\texttt{ub},\omega_i,\overline{n}) = \Phi(p,\omega_i,\overline{n})$
   - $\Phi(p,\omega_i,\overline{n})$ is computed by a profiler tool, or
   - $\Phi(p,\omega_i,\overline{n})$ is supplied by the user together with the code of program `p` (i.e., the cost function is not the result from any automatic

analysis but rather `p` is well known and its cost function can be supplied in a trust assertion).

2. For each benchmark p in this set, automatically generating a significant amount m of input data for it. This can be achieved by associating with each calibration program a data generation rule.

3. For each generated input data d_j, computing a pair $(\overline{C}_{p_j}, T_{p_j})$, $1 \leq j \leq m$, where:

   - T_{p_j} is the j-th observed execution time of program p with this generated input data.
   - $\overline{C}_{p_j} = \overline{\Phi}(p, I_\Omega, \overline{n}_j)$, where $\overline{n_j}$ is the size of the j-th input data d_j.

4. Using the set of pairs $(\overline{C}_{p_j}, T_{p_j})$ to set up the equation:

$$\overline{C}_{p_j} \bullet \overline{K}_\Omega = T_{p_j} \tag{3.7}$$

   where $\overline{K}_\Omega$ is considered a vector of variables.

5. Setting up the (overdetermined) system of equations resulting from putting together all the Equations (3.7) corresponding to all the calibration programs.

6. Solving the above system of equations using the least squares method (see, e.g., [65]). A solution to this system gives values to the vector $\overline{K}_\Omega$ and hence, to the constants K_{ω_i} which are the elements composing it.

7. Calculating the constants for builtins and arithmetic operators by performing repeated tests in which only the builtin being tested is called, accumulating the time, and dividing the accumulated time by the number of times the repeated test has been performed.

3.2.4 Assessment of the Calibration of Constants

We have assessed both the constant calibration process and the prediction of execution times using the previously proposed cost models in two different platforms:

- "intel" platform: Dell Optiplex, Pentium 4 (Hyper threading), 2GHz, 512MB RAM memory, Fedora Core 4 operating System with Kernel 2.6.
- "ppc" platform: Apple iMac, PowerPC G4 (1.1) 1.5GHz, 1GB RAM memory, with Mac OS X 10.4.5 Tiger.

Equation (3.7) is, in general, overdetermined, and we plan to find an approximation which is "best" in some sense, by using the least squares method. We used the Householder transformation [37], which decomposes the $m \times n$ matrix $C = \{\overline{C}_{p_j}\}$ into the product of two matrices Q and U such that $C = Q \bullet U$, where Q is an orthonormal matrix (i.e., $Q^T \bullet Q = I$, the $m \times m$ identity matrix) and U an upper triangular $m \times n$ matrix. Then, multiplying both sides of Equation (3.7) by Q^T and simplifying we can get:

$$U \bullet K = Q^T \bullet T = B$$

where, for clarity, we denote $K = \overline{K}_\Omega$, $T = T_{p_j}$ and $Q^T \bullet T = B$. We can take advantage of the structure of U and define V as the first n rows of U, n being the number of columns of C and b the first n rows of B, then K can be estimated solving the following upper triangular system, where $\hat{K}$ stands for the estimate for K:

$$V \bullet \hat{K} = Q^T \bullet T = b$$

Since this method is being used to find an approximate solution, we define the residual of the system as the value $R = T - C\hat{K}$.

Let $RSS = R \bullet R$ be the residual square sum, and let $MRSS = \frac{RSS}{m-n}$ be the mean of residual square sum, where m and n are the number of rows and columns of the matrix C respectively, and finally let $S = \sqrt{MRSS}$ be

Model	Components
A	steps nargs giunif gounif viunif vounif
B	steps giunif gounif viunif vounif
C	steps giunif gounif vounif
D	steps

Table 3.1: List of cost models being applied.

Plat.	Model	S (μs)	$\overline{K}_\Omega$
intel	A	6.2475	(21.27, 9.96, 10.30, 8.23, 6.46, 5.69)
	B	9.3715	(26.56, 10.81, 8.60, 6.17, 6.39)
	C	13.7277	(27.95, 11.09, 8.77, 7.40)
	D	68.3088	108.90
ppc	A	4.7167	(41.06, 5.21, 16.85, 15.14, 9.58, 9.92)
	B	5.9676	(43.83, 17.12, 15.33, 9.43, 10.29)
	C	16.4511	(45.95, 17.55, 15.59, 11.82)
	D	116.0289	183.83

Table 3.2: Values (in nanoseconds) for vector constants in several cost models, sorted by S, the standard error of the model.

the estimation of the standard error of the model, S. In order to evaluate experimentally which models generate the best approximation of the observed time, we have compared the values of $MRSS$ (or S) for several proposed models.

Table 3.1 shows the considered models. Table 3.2 shows the estimated values for the vector K using the calibration programs in Table 3.3, as well as the standard error of the model, sorted from the best to the worst model. Note that the estimation of K has to be done only once per platform. In the case of the intel platform it took 15.62 seconds and in ppc 17.84 seconds, repeating the experiment 250 times for each program.

Program	Error (%)			
Model	A	B	C	D
Environment creation	20	16	12	73
Predicates with no arguments	10	6	2	85
Traverse a list without last call optimization	20	20	11	80
Traverse a list with last call optimization	53	50	32	88
Program (unifying deep terms) for which $I(giunif)$ is known	16	18	18	474
Program (unifying deep terms) for which $I(gounif)$ is known	0	4	2	409
Program (unifying flat terms) for which $I(giunif)$ is known	16	18	18	472
Program (unifying flat terms) for which $I(gounif)$ is known	5	10	8	386
Program for which $I(viunif)$ is known	9	11	36	735
Program for which $I(vounif)$ is known	1	2	11	227
Unify two list element by element	34	29	20	26
Predicate with many arguments	17	16	9	159

Table 3.3: Calibration programs used to estimate the constants and the estimation error.

Our approach has been tested on the programs used in the calibration process itself for the considered models. Table 3.3 shows the error incurred in when an observed value is compared against an estimated value using the models in Table 3.1. It can be observed that the simpler models incur in significant errors while the more complex ones are more accurate (understandable since these calibrators exercise just particular implementation aspects and are thus expected to deviate from any "normal" behaviour).

In the same way that we approximate $\Phi(p, r, \overline{n})$, either by computing the upper-bound [21] $\mathsf{Cost}(p,\mathtt{ub},\mathsf{r},\overline{n})$ defined in Equation (2.1) (Section 2.3, Chapter 2), or the lower-bound [23] $\mathsf{Cost}(p,\mathtt{lb},\mathsf{r},\overline{n})$ defined in Equation (2.2), we can also compute bounds on execution times as the confidence intervals defined by our linear model:

$$C^{\mathtt{ub}} = \overline{\mathtt{Cost}}(p,\mathtt{ub},I_\Omega,\overline{n})$$

$$\mathsf{Cost}(p,\mathtt{ub},\mathtt{time}_\Omega,\overline{n}) = \overline{K}_\Omega \bullet C^{\mathtt{ub}} + Z_{\alpha/2}\sqrt{(C^{\mathtt{ub}})^T(V^TV)^{-1}(C^{\mathtt{ub}})}$$

$$C^{\mathtt{lb}} = \overline{\mathtt{Cost}}(p,\mathtt{lb},I_\Omega,\overline{n})$$

$$\mathsf{Cost}(p,\mathtt{lb},\mathtt{time}_\Omega,\overline{n}) = \overline{K}_\Omega \bullet C^{\mathtt{lb}} - Z_{\alpha/2}\sqrt{(C^{\mathtt{lb}})^T(V^TV)^{-1}(C^{\mathtt{lb}})}$$

Where $Z_{\alpha/2}$ is the optimal dispersion value of $1-\alpha$ level, assuming that the estimated values follow a Gaussian distribution (in our case $\alpha = 0.001$ and $Z_{\alpha/2} = Z_{0.0005} = 3.29$). However, even considering that such expressions are correct from the statistical point of view, they could define a very complex algebraic expression. We give below other approximation, which is a bit less accurate, but appropriate if we want simpler expressions.

Consider first the covariance matrix as $E = S^2(V^TV)^{-1} = \{E_{ij}\}$. In order to calculate the standard deviation of $\hat{K}$, we use the E matrix, and then, $StdDev(K_{\omega_i}) = \sqrt{E_{ii}}$. This expression allow us to compute bounds for $\overline{K}_\Omega$, including both lower-bounds $\overline{K}_\Omega^{\mathtt{lb}}$ and upper-bounds $\overline{K}_\Omega^{\mathtt{ub}}$, and are defined as follows:

$$StdDev(\overline{K}_\Omega) = (StdDev(K_{\omega_1}),\ldots,StdDev(K_{\omega_v}))$$

$$\overline{K}_\Omega^{\mathtt{lb}} = \overline{K}_\Omega - Z_{\alpha/2} \times StdDev(\overline{K}_\Omega) \tag{3.8}$$

$$\overline{K}_\Omega^{\mathtt{ub}} = \overline{K}_\Omega + Z_{\alpha/2} \times StdDev(\overline{K}_\Omega)$$

Then, we can compute the bounds for execution time as:

$$\mathsf{Cost}(p,\mathtt{ub},\mathtt{time}_\Omega,\overline{n}) \approx \overline{K}_\Omega^{\mathtt{ub}} \bullet \overline{\mathtt{Cost}}(p,\mathtt{ub},I_\Omega,\overline{n}) \tag{3.9}$$

$$\mathsf{Cost}(p,\mathtt{lb},\mathtt{time}_\Omega,\overline{n}) \approx \overline{K}_\Omega^{\mathtt{lb}} \bullet \overline{\mathtt{Cost}}(p,\mathtt{lb},I_\Omega,\overline{n}) \tag{3.10}$$

3.2.5 Assessment of the Prediction of Execution Times

We have tested the proposed cost models in a set of programs not used in the calibration process in order to assess how well their execution time is predicted, without performing any runtime profiling on them. We have performed experiments with the 63 possible cost models resulting from selecting one or more of the components described in Section 3.2.1. For space reasons we only show the three most accurate cost models (according to a global accuracy comparison that will be presented later) plus the steps model, which, despite its simplicity, has a special interest, as we will also see later. Experimental results are shown in Table 3.4, where the analyzers integrated in the `CiaoPP` system infer the exact platform-independent cost function for all the programs in that table, which means that the upper- and lower-bound are the same, i.e., $\texttt{Cost}(p,\mathsf{lb},\omega_i,\overline{n}) = \texttt{Cost}(p,\mathsf{ub},\omega_i,\overline{n}) = \Phi(p,\omega_i,\overline{n})$.

The first three rows for each test program show the three more accurate predictions along with the model used. The fourth row shows the prediction obtained by the cost model *steps*, which assumes that the execution time is directly proportional to the number of resolution steps performed. Note that $\Phi_{clause}(C, steps, \overline{n})$ gives the number of resolution steps performed by clause `C`. The row tagged as **Obs.** corresponds to the actual measured timings, and the last row details the analysis time (roughly the same in all benchmarks, and which includes mode, type, and cost analysis).

The first column is the program name, the second is the cost model Ω (= vector of characteristics taken into account) and the third and fourth are the timing estimations corresponding to the "intel" and "ppc" platforms. These are computed by using the average value of the constant $\overline{K}_\Omega$ as estimated in Table 3.2 with the formula:

$$\textbf{Estimate}_{\mathtt{P}} = \overline{K}_\Omega \bullet \overline{\Phi}(p,I_\Omega,\overline{n})$$

Deviations respect to the measured values are also shown between parenthesis in the column $\textbf{Estimate}_{\mathtt{P}}$.

The observed execution times have been measured by running the pro-

Prog.	Model	$\mathbf{Estimate_P}$		Prog.	Model	$\mathbf{Estimate_P}$	
		intel	ppc			intel	ppc
		(μs) (%)	(μs) (%)			(μs) (%)	(μs) (%)
evpol	A	90 (44)	77 (23)	palind	A	132 (18)	180 (5)
	B	85 (38)	75 (26)		B	101 (9)	164 (5)
	C	82 (35)	70 (33)		C	87 (24)	142 (19)
	D	90 (45)	85 (13)		D	167 (43)	282 (52)
	Obs.	58	97		Obs.	110	172
	T_{ca}(s)	2.0	4.5		T_{ca}(s)	2.0	4.7
hanoi	A	319 (31)	399 (4)	powset	A	538 (59)	728 (17)
	B	243 (3)	359 (7)		B	405 (28)	658 (7)
	C	206 (14)	301 (25)		C	324 (5)	535 (14)
	D	341 (38)	539 (34)		D	449 (38)	757 (21)
	Obs.	235	384		Obs.	308	615
	T_{ca}(s)	2.2	4.9		T_{ca}(s)	2.1	4.6
nrev	A	131 (68)	179 (26)	append	A	50 (75)	69 (24)
	B	101 (39)	164 (16)		B	39 (44)	63 (15)
	C	83 (18)	135 (3)		C	31 (22)	51 (5)
	D	144 (80)	244 (59)		D	55 (85)	92 (56)
	Obs.	69	139		Obs.	25	54
	T_{ca}(s)	2.0	4.7		T_{ca}(s)	1.9	4.4

Table 3.4: Experiments on example programs.

grams with input data of a fixed size. We generated randomly 10 input data sets of such fixed size, and for each data set we run 5 times every program. The observed execution time for such input size was computed as the average of all runs.

Table 3.5 compares the overall accuracy of the four cost models already shown in Table 3.4, for the two considered platforms. The last column shows the global error and it is an indicator of the amount of deviation of the execution times estimated by each cost model with respect to the

Platform	Model Error (%)			
	A	B	C	D
intel	53.17	31.06	21.48	58.45
ppc	18.72	14.66	19.44	43.04

Table 3.5: Global comparative of the accuracy of cost models.

observed values. As global error we take the square mean of the errors in each example being considered in Table 3.4. By considering both platforms in combination we can conclude that the more accurate cost model is $\Omega = (steps, giunif, gounif, viunif, vounif)$. This cost model has an overall error of 14.66 % in the ppc platform and 31.06 % in the intel platform. In the latter architecture the model $\Omega = (steps, giunif, gounif, vounif)$ appears to be the best. This is in line with the intuition that taking into account a comparatively large number of lower-level operations should improve accuracy. However, such components should contribute significantly to the model in order to avoid noise introduction. It is also interesting to see that including $nargs$ in the cost model does not further improve accuracy, as expected, since nargs is not independent from the four components giunif, gounif, viunif, vounif. In fact, including this component results in a less precise model in both platforms, due to the noise introduced in the model. Also, the cost model steps deserves special mention, since it is the simplest one and, at least for the given examples, the error is smaller than we expected and better than more complex cost models not shown in the tables.

The disparity in the accuracy for both platforms can be attributed to a number of reasons, among them the difference in the internal architectures (number of registers, orthogonality in their usage, etc.), which make predicting execution characteristics in intel processors harder. The weight of some constants can also differ from the calibration programs to the benchmarks due to, e.g., the state of the internal processor pipelines and state of registers. In our experience, the ppc architecture offers a more homogeneous behavior performance-wise.

Overall we believe that the results are encouraging in the sense that our combined framework predicts with an acceptable degree of accuracy the execution times of programs and paves the way for even more accurate analyses by including additional parameters.

3.2.6 Applications

The experimental results presented in Section 3.2.5 show that the proposed framework can be relevant in practice for estimating platform dependent cost metrics such as execution time. We believe that execution time estimates can be very useful in several contexts. As already mentioned, in certain mobile/pervasive computation scenarios different platforms come into play, with each platform having different capabilities. More concretely, the execution time estimates could be useful for performing resource/granularity control in parallel/distributed computing. This belief is based on previous experimental results, where it appeared from the sensitivity of the results observed in such experiments, that while it is not essential to be absolutely precise in inferring the best time estimates for a query, the number of reductions by itself was too rough a measure and the current time estimation approach could presumably improve on previous results.

One of the good features of our approach is that we can translate platform-independent cost functions (which are the result of the analyzer) into platform-dependent cost functions (using the relationship in expression (3.6)). A possible application for taking advantage of this feature is mobile code safety and in particular Proof-Carrying Code (PCC), a general approach in which the code supplier augments the program with a certificate (or proof). Consider a scenario where the producer sends a certificate with a platform-independent cost function (i.e., where the cost is expressed in a platform-independent metric) together with a calibration program. The calibration program includes a fixed set of calibration benchmarks. Then, the consumer runs (only once) the calibration program and computes the values for the constants appearing in the cost functions. Using these constants, the

consumer can obtain platform-dependent cost functions [32].

Another application of the proposed approach is resource-oriented specialization. The proposed cost models, which include low-level factors for CLP programs, are more refined cost models than previously proposed ones and thus can be used to better guide the specialization process. The inferred cost functions can be used to develop automatic program transformation techniques which take into account the size of the resulting program, its run time and memory usage, and other low-level implementation factors. In particular, they can be used for performing self-tuning specialization in order to compare different specialized version according to their costs [19].

The use of a source-level characterization of the execution profile, which undoubtedly carries some lack of accuracy with it, can be applied not only to different architectures, but also to different compilation / execution schemes. By identifying a rich enough cost model, and using the calibration programs under a given execution model (and architecture), predictions about this execution model / architecture can be made. The advantage lies in that instrumenting the low-level representation used by the execution algorithm (e.g., WAM code & emulator, C code / assembler, or interpreters or virtual machines for other bytecode representations) is not needed: $\overline{K}_\Omega$ should get instantiated to the cost (or an approximation thereof) of every identified *basic* feature in the execution model under study.

3.2.7 Section Conclusions

We have developed a framework which allows estimating execution times of procedures of a program in a given execution platform. The method proposed combines compile-time (static) cost analysis with a one-time profiling of the platform in order to determine the values of certain constants. These constants calibrate a cost model from which time cost functions for a given platform can be computed statically. The approach has been implemented and integrated in the `CiaoPP` system. To the best of our knowledge, this is the first combined framework for estimating statically and accurately execution

time bounds based on static automatic inference of upper- and lower-bound complexity functions plus experimental adjustment of constants. We have performed an experimental assessment of this implementation for a wide range of different candidate cost models and two execution platforms. The results achieved show that the combined framework predicts the execution times of programs with a reasonable degree of accuracy. We believe this is an encouraging result, since using a one-time profiling for estimating execution times of other, unrelated programs is clearly a challenging goal.

Also, we argue that the work presented in this section presents an interesting trade-off between accuracy and simplicity of the approach. At the same time, there is clearly room for improving precision by using more refined cost models which take into account additional (lower level) factors. Of course, these models would also be more difficult to handle since on one hand they would require computing more constants and on the other hand they may require taking into account factors which are not observable at source level.

3.3 Bytecode-Based (Low-Level) Model

In this section we propose a new analysis framework which, in order to improve the accuracy of the time predictions, on the one hand it takes into account lower level factors and on the other hand, it makes the model richer by directly taking into account the inherently variable cost of certain low-level operations [45].

Regarding the choice of this lower level, rather than trying for example to model directly the characteristics of the physical processor, as in WCET, and given that most popular logic programming implementations are based on variations of the Warren abstract machine (WAM) [66, 1], we chose to model cost at the level of abstract machine instructions. Abstract machines have been used as a basic implementation technique in several programming paradigms (functional, logic, imperative, and object-oriented) [24] with the

advantage that they provide an intermediate layer that separates to a certain extent the many low-level details of real (hardware) machines from the higher-level language, while at the same time making compilation easier. This property can be used to facilitate the design of our framework.

Within this setting, we present a new framework for the static estimation of execution times of programs. The basic ideas in our approach follow:

1. Measure the execution time of each of the instructions in a lower-level L_B (bytecode) language (or approximate it with a function if it depends on the value of an argument) in some specific abstract machine implementation when executed on a given processor / O.S.

2. Make the information regarding instruction execution time available to the timing analyzer. This is, in our proposal, done by means of *cost assertions* (written in a suitable assertion language) which are stored in a module accessible to the compiler/analyzer.

3. Given a concrete program P written in the source language L_H, compile it into L_B and record the relationship between P and its compiled counterpart.

4. Automatically analyze program P, taking into account the instruction execution time (determined in item 1 above) to infer a cost function C_P. This function is an expression which returns (bounds on) the actual execution time of P for different input data sizes for the given platform.

Points (1) and (2) are performed in a one-time profiling phase, independent from program P, while the rest are performed once for each P in the static (compile-time) cost analysis phase. We would like to point out that, in general, the basic ideas underlying our work can be applied to any language L_H as long as (i) cost estimation can be derived for programs written in L_H, (ii) the translation of L_H to some other (usually lower-level) language L_B is accessible, and (iii) the execution time of the instructions in L_B can be

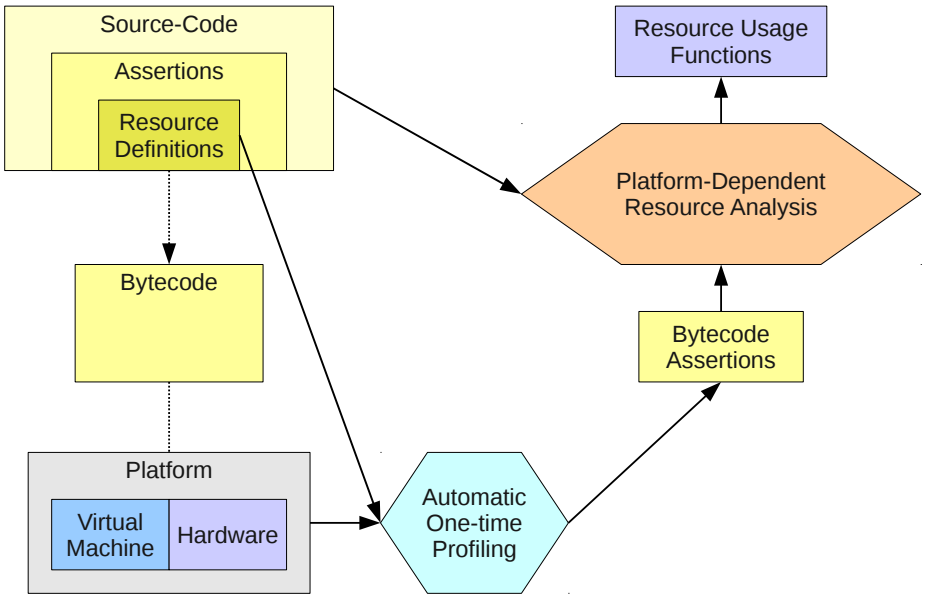

Figure 3.4: Bytecode-Level/Platform-Dependent Resource Analysis.

timed accurately enough. We will, however, focus herein on logic languages, so that we assume L_H to be a Prolog-like language and L_B some variant of the WAM bytecode.

The proposed framework is illustrated in Figure 3.4, and has been implemented as part of the `CiaoPP` [36] system in such a way that any abstract machine properly instrumented can be analyzed.

To the best of our knowledge, this is the first attempt at providing a timing analysis producing upper- and lower-bound time *functions* based on the cost of lower-level machine instructions.

Note that although the initial objective was to developed a framework for execution time estimation, it can also be applied to the estimation of platform-independent (bytecode related) resources, such as the number of times the bytecodes are executed (as illustrated in Figure 3.5).

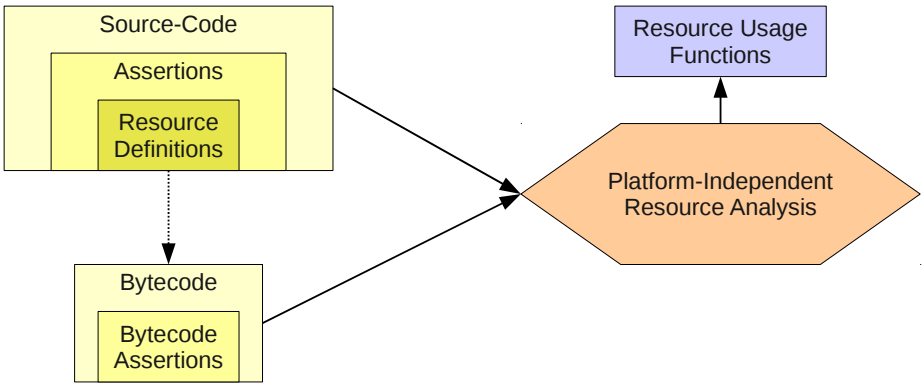

Figure 3.5: Bytecode-Level/Platform-Independent Resource Analysis.

In fact, as a more general contribution of this thesis, we propose a framework based on bytecode-level cost models which can be used to infer both, platform-dependent and independent resources (see Figure 3.6).

3.3.1 Mappings Between Program Segments and Bytecodes

Let $OpSet = \{b_1, b_2, \ldots, b_n\}$ be the set of instructions of the abstract machine under consideration. We assume that each instruction is defined by a numeric identifier and its arity, i.e., $b_i \equiv f_i/n_i$, where f_i is its identifier and n_i the arity. Each program is compiled into a sequence of expressions of the form $f(a_1, a_2, \ldots, a_n)$ where f is the instruction name and the a_i's are its arguments. For conciseness, we will use I_i to refer to such expressions. The sequences of expressions into which a program is compiled are generally encoded using bytecodes. In the following we will often refer to sequences of abstract machine instructions or sequences of bytecodes simply as "bytecodes."

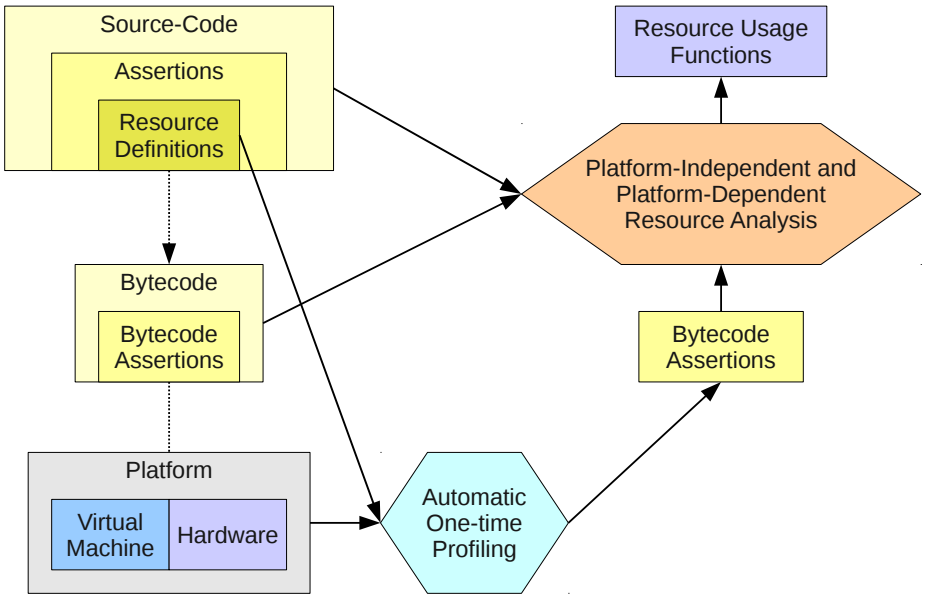

Figure 3.6: Bytecode-Level Resource Analysis.

Let `C` be a clause `H :- L`$_1, \ldots,$ `L`$_m$. Let $E(\texttt{C})$ be a function that returns the sequence of bytecodes resulting from the compilation of clause `C`:

$$E(\texttt{C}) =< I_1, I_2, \ldots, I_p >$$

Let $E(\texttt{C}, \texttt{H})$ be a function that maps the clause head `H` to the sequence of bytecodes in $E(\texttt{C})$ starting from the beginning up to the first `call/execute` instruction or to the end of the sequence $E(\texttt{C})$ if there are no more `call/execute` instructions (i.e., to the end of the bytecode sequence resulting from the compilation of clause `C`). Let $E(\texttt{C}, \texttt{L}_\texttt{i})$ be the function that maps literal `L`$_i$ of clause `C` to the sequence of bytecodes in $E(\texttt{C})$ which start at the `call` bytecode instruction corresponding to this literal and up to the next `call/execute` instruction or to the end of the sequence $E(\texttt{C})$ if there are no more `call/execute` instructions. If $\uplus$ represents the concatenation

of sequences of bytecodes, then:

$$E(\mathtt{C}) = E(\mathtt{C},\mathtt{H}) \biguplus (\biguplus_{i=1}^{m} E(\mathtt{C},\mathtt{L_i}))$$

Note that functions $E(\mathtt{C},\mathtt{H})$ and $E(\mathtt{C},\mathtt{L_i})$ do not necessarily return the bytecodes that one would normally associate to the clause head `H` and literal $\mathtt{L_i}$ respectively. Instead, the definition of those functions associates the instructions corresponding to argument preparation for a given call with the (success of the) *previous* call (or head). This is to cater for the fact that, in the context of backtracking, the WAM argument preparation occurs only one time per call to a literal, even if such call is retried more times before failing definitively. As a result, the cost of argument preparation for a given `call/execute` instruction needs to be associated with the previous literal to that `call/execute`, in order not to count it every time the call is retried.

Table 3.6 shows how predicate `append`/3 is compiled into bytecodes, and identifies the result of calling the $E(\mathtt{C},\mathtt{H})$ and $E(\mathtt{C},\mathtt{L_i})$ functions for each clause head and body literal. $\mathtt{H}^1$ represents the head of the first clause (C_1), and $\mathtt{H}^2$ and $\mathtt{L}_1^2$ the head of the second (recursive) clause (C_2) and the first literal in such clause body (the only body literal).

3.3.2 Modeling the Execution Time of Instructions

We define a function $t(I)$ (the *timing model*), which takes a bytecode instruction I and returns another function which estimates the execution time for it depending on the input data sizes of the bytecode. This is similar to the approach described in [6], where, however, $t(I)$ was a constant.

In many cases we can assume that the time to execute a bytecode is constant. However there are some instructions for which this does not hold because their definitions involve loops. In many of these cases the timing model consists of an initial constant time t_0 plus another additional constant time t_{iter} to cater for the cost of each iteration, and a simple linear model can be used: $t_0 + n \times t_{iter}$. Consider for example the `unify_void` n instruction,

$E(\mathtt{C_1}, \mathtt{H^1})$	`append([], X, X).`		
	append/3/1	try_me_else append/3/2	get_value(V0,A2)
		allocate	deallocate
		get_constant([],A0)	proceed
		get_variable(V0,A1)	
$E(\mathtt{C_2}, \mathtt{H^2})$	`append([X|Xs], Y, [X|Zs]) :-`		
	append/3/2	trust_me	get_variable(V6,A2)
		allocate	set_variable(V7)
		get_variable(V0,A0)	set_variable(V8)
		set_variable(V1)	get_list(V1,V8)
		set_variable(V2)	set_variable(V9)
		set_variable(V3)	unify_variable(V7,V9)
		get_list(V1,V3)	unify_variable(V6,V8)
		set_variable(V4)	put_value(V2,A0)
		unify_variable(V2,V4)	put_value(V5,A1)
		unify_variable(V0,V3)	put_value(V7,A2)
		set_variable(V5,A1)	deallocate
$E(\mathtt{C_2}, \mathtt{L_1^2})$	`append(Xs, Y, Zs).`		
		execute append/3	

Table 3.6: Sequences of bytecodes assigned to clause heads and body literals of the clauses C_1 and C_2 of predicate append by the functions $E(\mathtt{C}, \mathtt{H})$ and $E(\mathtt{C}, \mathtt{L})$.

which pushes n new unbound cells on the heap [1], and whose execution time is a linear function on n. In some other cases instructions have different execution times depending on the (fixed) values a given argument can take from some finite set. In such cases, execution time is an arbitrary function on the argument. Specific constants are assigned to each possible argument value by means of profiling (Section 3.3.4).

Since the cost of a given instruction is different when it succeeds and when it fails, we will have two costs for each instruction that can fail: one for the success case and another for the failure case. Finally, and besides lower-level

factors such as cache behavior, there are some additional variable factors (such as, e.g., the length of dereferencing chains) which may affect execution times. These factors are in principle not impossible to cater for via a combination of static and dynamic analysis, but, given the additional complication involved, we will ignore them herein and explore what kind of precision of timing prediction can be achieved with this first level of approximation.

Another factor that we are not taking into account at this moment is garbage collection (GC). GC makes programs run slower, which, at profiling time, increases the (estimated) cost of every instruction. Therefore, turning it off at profile time (which gives a smaller estimation of instruction cost) is safe when finding out lower-bounds: if the program whose execution time is to be predicted is run with GC turned on, then it would run slower w.r.t. an execution with GC turned off (as it was when profiling), and the estimated bounds will still be lower-bounds, albeit more conservative. An inverse reasoning applies to upper-bounds, and the technique herein presented is equally valid. However, for the sake of simplicity, we have taken all the measurements (both for profiling and executions to be predicted) with GC disconnected.

3.3.3 Estimating the Execution Time of Clauses and Predicates

In this section we describe how to particularize the general resource usage expressions defined in Chapter 2 in order to estimate bounds on the execution time of predicates. For simplicity, the discussion that follows is focused on the estimation of upper-bounds on execution times (as it was done in the previous section). We also use the terms *resource usage function* and *cost function* indistinctly. Thus, in order to estimate upper-bounds on execution time, we particularize $\delta(\mathtt{ap}, r)(\mathtt{H}^k, \overline{n})$ in expression (2.4) of Section 2.3.3 (Chapter 2) for the case when the resource r is execution time and the approximation `ap` is `ub` as follows:

$$\delta(\mathtt{ap}, r)(\mathtt{H}^k, \overline{n}) = \gamma(\mathtt{H}^k, \overline{n}) + \sum_{I \in E(\mathtt{C}_k, \mathtt{H}^k)} t(I)(\overline{n})$$

which represents the execution time needed to resolve the head $\mathtt{H}^k$ of the clause $\mathtt{C}_k$ with the literal being solved. $\gamma(\mathtt{H}^k, \overline{n})$ denotes the execution time necessary to determine that clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{k-1}$ will not yield a solution and that $\mathtt{C}_k$ must be tried: the function γ obviously takes into account the type and cost of the indexing scheme being used in the underlying implementation. Similarly, we particularize $\beta(\mathtt{ap}, r)(\mathtt{L}_i^k, \psi_i(\overline{n}))$ as follows:

$$\beta(\mathtt{ap}, r)(\mathtt{L}_i^k, \psi_i(\overline{n})) = \sum_{I \in E(\mathtt{C}_k, \mathtt{L}_i^k)} t(I)(\overline{n}_i), \;\; i = 1, \cdots, m$$

which represents the time needed to prepare the call to literal $\mathtt{L}_i^k$ in the body of the clause $\mathtt{C}_k$, with $E(\mathtt{C}_k, \mathtt{L}_i^k)$ and $t(I)$ defined as in Sections 3.3.1 and 3.3.2 respectively.

Note that our approach allows defining via assertions the execution time of external predicates, which can then be used for modular composition [51]. This includes also predicates for which the code is not available or which are even written in a programming language that is not supported by the analyzer. In addition, assertions also allow describing by hand the execution time of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. The description of the assertion language is summarized in Figure 4.2, and we refer the reader to [51] for details.

3.3.4 Estimating Instruction Execution Times via Profiling

In this section we will see how data regarding the expected execution time of each instruction in the abstract machine (Section 3.3.2) can be accurately measured in a realistic environment.

```
while (op != END) { /* WAM emulation loop */
  ...
  record_profile_info(op);
  /* op is the current bytecode */
  switch(op) {
    ...
  }
  ...
  op = get_next_op();
}
```

Figure 3.7: A simple WAM emulation loop instrumented.

3.3.5 Instruction Profiling

Profiling aims at calculating a function $t(I)$ for each bytecode instruction I. An approach is to instrument the WAM implementation so that time measures are taken and recorded at appropriate points in the execution [38]. In practice, a number of issues have to be taken into account in order to obtain accurate enough measurements. These include the selection of the places where the instrumentation code will be inserted, how to minimize the effects of such instrumentation on the execution (not only execution time but also, e.g., cache behavior), and how to work around the complex instruction scheduling performed by modern processors, which may lead to large variance in the results, especially since we aim at measuring very small fragments of code.

A first approximation is to add profiling-related calls in designated parts of the bytecode interpreter main loop. Figure 3.7 shows a piece of code illustrating this. The operation `record_profile_info(op)` records the start time for the bytecode `op`. The end time is processed when the next opcode is fetched. The data for each bytecode is maintained in memory during execution (and in raw form in order to impact execution as little as possible) and later saved to an external file.

A benchmark-based analysis is also proposed in [38], which describes how

the instrumented code can be reused effectively on various platforms without modifying it, and how the execution time of a specific set of bytecodes can be measured.

However, the methods mentioned above have drawbacks. For example, the first one (instrumenting the main loop) depends on the existence of very precise, non-intrusive, low-overhead timing operations which, unfortunately, are not always available in all platforms. Portable O.S. calls, besides having a typically high associated overhead, are in general not accurate enough for our purposes. Even if a very fast timing operation is available (which is not the case in platforms such as mobile and embedded devices), its introduction may affect the behavior of the machine being analyzed if the abstract machine loop is very optimized. For example, if the new code changes register and variable allocation, program behavior will be affected in unforeseen ways.

We will, however, use an instrumented loop like that of Figure 3.7 to count the number of bytecodes executed in a calibration step.

3.3.6 Measuring Time Accurately

In order to do portable time measurements in platforms where high resolution timing is difficult or impossible to achieve, workarounds have to be used. The approach that we have followed is based on using synthetic benchmarks which on purpose repeatedly execute the instructions under estimation for a large enough time, and later divide the total execution time by the number of times the instructions were executed. A complication in this process is that it is in general not possible to run a single instruction repeatedly within the abstract machine, since the resulting sequence would not be legal and may "break" the abstract machine, run out of memory, etc. In general, more complex sequences of instructions must be constructed and repeated instead.

Therefore, the approach we have followed involves designing a set of *legal* programs which cover all the bytecode instructions, relate the execution time of these programs with the individual instruction execution times with a system of equations, and solving such a system.

Programs	Instructions	Trace
c1_5 :- c1_5_0.	00 : `execute 01`	00 : `execute` 01
c1_5_0 :- c1_5_1.	01 : `execute 02`	01 : `execute` 02
c1_5_1 :- c1_5_2.	02 : `execute 03`	02 : `execute` 03
c1_5_2 :- c1_5_3.	03 : `execute 04`	03 : `execute` 04
c1_5_3 :- c1_5_4.	04 : `execute 05`	04 : `execute` 05
c1_5_4 :- c1_5_5.	05 : `execute 06`	05 : `execute` 06
c1_5_5.	06 : `proceed`	06 : `proceed`
c1_0 :- c1_0_0.	01 : `execute` 02	01 : `execute` 02
c1_0_0.	02 : `proceed`	02 : `proceed`

Table 3.7: Programs used in order to get the execution time of the `execute` instruction.

3.3.7 Getting Instruction Execution Time

We now discuss how to set up calibration programs in order to get the cost of bytecodes. In this section, and in order to simplify the discussion, we deal with those bytecodes whose execution time is bound by a constant. In the following section we extend our technique to manage instructions whose execution time is unbound.

Let $C_i, i = 0, 1, \ldots, n$ be a set of synthetic calibration programs, each of them returning the execution time of a block of code. Each C_i, which we will refer to as calibrator, is generated in such a way that it repeats such block a given number of times, say r. Let us assume, for example, that we want to calibrate the WAM instruction "`execute`" when it does not fail and that we want to repeat its execution 5 times (i.e., $r = 5$). Table 3.7 shows a set of programs which can be used to calibrate this WAM instruction. Columns **Instructions** and **Trace** show the WAM code as generated by the compiler and the sequence of instructions executed when running the program starting from the first clause respectively. In general, in our approach, rather than a concrete program, calibrators are program generation templates which take r as an input and return, e.g., the programs in Table 3.7 for that value

of r. The actual calibration program includes an entry point which calls the programs in Table 3.7 and returns the value of the execution time of the `execute` instruction, subtracting the time spent in the entry calls (e.g., `c1_5` for Table 3.7). In this case the calibration time is easy to compute as the difference between the execution time of `c1_5` and `c1_0` divided by r. The result of the calibration should ideally be invariant with respect to r; in practice this is however not true due, among other factors, to timing imprecision. Thus, r needs to be determined for each case: it has to be a large enough value to ensure stability of the time measured by the calibrator for the particular platform and the method used to measure time, but not excessively large, as this would make calibration impractical.

In some cases we cannot isolate the behavior of only one bytecode. This is specially the case in the calibrators of instructions which alter the program flow, such as `call`, `proceed`, `trust_me`, `try_me_else`, `retry_me_else`, `allocate`, `deallocate`. It is also the case when measuring the cost of failure for any of the instructions which can fail (generally the `get_` and `unify_` instructions). All these instructions need to be always executed together with other bytecodes in order to make the calibration program legal. As a result, and due to interactions between the costs of the different instructions, the equations are not as easy to configure in all cases as the simple case for the `execute` instruction above.

As a simple example, the calibrator that returns the cost of `call` and the `proceed` instructions uses the programs in Table 3.8 (where we have turned off the optimization of register / variable allocation in the compiler for simplicity). In order to be able to separate the cost of `call` and `proceed` an idea might be to find a calibrator that isolates the cost of `proceed` by itself and subtract from the value given by the calibrator for `call` and `proceed` and obtain the cost of `call`. However, that is in general not possible since in all legal calibrators `proceed` and `call` must always appear combined with other bytecodes. In general we need to set up a system of equations in which the known values are the costs given by our calibrators and the unknown values

Programs	Instructions	Trace	
c2_5 :-	00 : `allocate`	00 : `allocate`	06 : `call 09`
c_5,	01 : `call 09`	01 : `call 09`	09 : `proceed`
c_5,	02 : `call 09`	09 : `proceed`	07 : `deallocate`
c_5,	03 : `call 09`	02 : `call 09`	08 : `execute 09`
c_5,	04 : `call 09`	09 : `proceed`	09 : `proceed`
c_5,	05 : `call 09`	03 : `call 09`	
c_5,	06 : `call 09`	09 : `proceed`	
c_5.	07 : `deallocate`	04 : `call 09`	
	08 : `execute 09`	09 : `proceed`	
		05 : `call 09`	
c_5.	09 : `proceed`	09 : `proceed`	
c2_0 :-	00 : `allocate`	00 : `allocate`	
c_0,	01 : `call 04`	01 : `call 04`	
c_0.	02 : `deallocate`	04 : `proceed`	
	03 : `execute 04`	02 : `deallocate`	
		03 : `execute` 04	
c_0.	04 : `proceed`	04 : `proceed`	

Table 3.8: Programs used to get the execution time of the `call` and `proceed` instructions.

are the costs of the individual bytecodes. Such equations can be configured automatically, by executing the calibration programs in a special version of the WAM with the bytecode dispatch loop instrumented as in Figure 3.7 so that the profiler keeps an account of the executed bytecodes.

Let c_i, $0 \leq i \leq n$, be the time calibrator C_i has returned, and let β_j, $0 \leq j \leq m$, $m \geq n$, be the cost of a bytecode B_j, distinguishing between the case of a fail or a success in the execution of such bytecode. In other words, $B_j \in I \times \{fail, success\}$, where I is the set of all possible bytecodes and *fail* and *success* represent the failure or success of the execution of a bytecode.

We can then set up the following system of equations:

$$\begin{array}{rcl} c_1 & = & a_{11}\beta_1 + a_{12}\beta_2 + \cdots + a_{1m}\beta_m \\ c_2 & = & a_{21}\beta_1 + a_{22}\beta_2 + \cdots + a_{2m}\beta_m \\ & \cdots & \\ c_n & = & a_{n1}\beta_1 + a_{n2}\beta_2 + \cdots + a_{nm}\beta_m \end{array} \tag{3.11}$$

which we can rewrite such using matrix notation:

$$C = AB \tag{3.12}$$

where $B = (\beta_i)$ is the vector of execution times for the bytecodes. In order to obtain B we ideally need to configure as many calibrators as bytecodes. Finding a solution to this system of equations requires, in principle, independence among the equations (i.e., there is no other linear independent equation but those in expression (3.11)), and to have as many equations as variables. However, that is not always possible due to dependencies between the number of times a bytecode is executed. For example, in the WAM under analysis, the following invariant holds:

Proposition 1. *For any program, the number of times* `retry_me_else` *is called plus the number of times* `trust_me` *is called is equal to the number of failures.*

This holds since a failure always causes backtracking to the next choice point, which always implies executing either a `retry_me_else` or a `trust_me` instruction. As the coefficients a_{ij} in the equation above are precisely the number of times every bytecode is executed, it turns out that, for a given execution, some coefficients are dependent on some other coefficients, therefore breaking the initial independence assumption: the system of equations is underdetermined and it does not have a unique solution.

For this reason, since the coefficients a_{ij} where obtained by summarizing legal programs (i.e., the calibrators), and they will be affected by the linear dependency mentioned above, the undetermined system (3.12) will not have

a unique solution. However, note that when several bytecodes in a block must be executed together (because of constraints in the WAM compilation and execution scheme) knowing the execution time of each of them in isolation is not needed: knowing the total execution time of the whole block is enough. This intuitive idea can be formalized and generalized with the following result:

Proposition 2. *Given a set of n calibration programs C_i, that define n linear independent equations with β_i variables (corresponding to the m bytecodes, with both success and failure cases included), if we have that for all programs there exist $m-n$ linear independent relationships between the number of bytecodes that are always fulfilled, then the estimated execution time is invariant with respect to the choice of any arbitrary element of the solution set of such linear system.*

Proof : Let B be an arbitrary solution of $C = AB$. Let X be a vector which represents the number of times each bytecode has been executed for a given program. The estimated execution time is $E = X^T B$, i.e., the sum of the time for each bytecode multiplied by the number of times it has been executed.

By linear algebra, and considering that each calibrator defines a linear independent equation, we have that the range of A is n, and the kernel (or nullspace) of A is given by the set of all λ such that $A\lambda = 0$, a vector space of dimension $m - n$ (0 represents the null vector of dimension n). In other words, we have that:

$$C = AB = AB + 0 = AB + A\lambda = A(B + \lambda) \tag{3.13}$$

Then, $B + \lambda$ is a solution of Equation (3.12), and it is also a representative of the solution set of such equation system. What we should prove now is that $X^T(B + \lambda) = X^T B$, that is, canceling common terms and transposing the equations:

$$\lambda^T X = 0 \tag{3.14}$$

On the other hand, we have a set of $m - n = k$ linear dependencies between the number of bytecodes executed of the form:

$$\begin{aligned} 0 &= v_{11}x_1 + v_{12}x_2 + \cdots + v_{1m}x_m \\ 0 &= v_{21}x_1 + v_{22}x_2 + \cdots + v_{2m}x_m \\ &\cdots \\ 0 &= v_{k1}x_1 + v_{k2}x_2 + \cdots + v_{km}x_m \end{aligned}$$

Or, rewriting them using matrices:

$$0 = VX \tag{3.15}$$

The result of multiplying an arbitrary vector d by V is a vector $\mu^T = (dV)$ and for the equation above, it follows that $\mu^T X = 0$.

But note that the rows of A were obtained executing a program that meets the linear dependencies too, that is, $\mu^T A^T = 0$. Transposing, we have:

$$A\mu = 0 \tag{3.16}$$

For this reason, we can see that as λ, μ is a member of the kernel of A, and considering the uniqueness of the kernel of a matrix, and that μ is an element of a space of dimension $m - n$, we can choose μ such that $\lambda = \mu$, that is, we can express λ as the product $(dV)^T$, as result of basic theorems of linear algebra. Therefore, we have that:

$$\lambda^T X = \mu^T X = (dV)X = d(VX) = d(0) = 0 \tag{3.17}$$

□

Then, the method we follow to select a representative solution B is simply to complete the equation systems with $m - n$ arbitrary equations in order to make them become determined. Such equations should be selected in such a way that the β_i values be positive, for example, by setting the cost to 0 as the time of the bytecodes that are faster, avoiding negative solutions.

3.3.8 Dealing with Unbound Instructions

We now consider the case of bytecode instructions whose execution time depends on the specific values that certain parameters can take at run time. In such cases the accuracy of our analysis can be increased by taking advantage of static term-size analysis and the addition of cost-related assertions for such instructions. Such assertions make it possible to introduce ad-hoc functions giving the size of the input parameters of the bytecode.

In fact, our system is able to deal with several metrics (e.g., value, length, size, depth, ...) as shown in [22, 21, 23], but for brevity, in the following paragraphs we will describe an example unifying lists.

Let us take, the instruction `unify_variable(V, W)` and let us assume that we want to calculate an upper-bound for its execution time upon success and for the case where the two arguments to unify are lists of numbers. We assume that an upper-bound to the execution time is proportional to the number of iterations necessary to scan the lists. The timing model for such instruction is thus $K_1 + K_2 * length(V)$, because if the instruction succeeds, the length of both V and W should be equal. The value of constants K_1 and K_2 is calculated by setting up two benchmarks which unify lists of different length l_1 and l_2. If the cost of `unify_variable` for these two list lengths is, respectively, B_1 and B_2, then we set up the following system of linear equations:

$$\begin{aligned} B_1 &= K_1 + K_2 \times l_1 \\ B_2 &= K_1 + K_2 \times l_2 \end{aligned} \tag{3.18}$$

Note that B_1 and B_2 can be added to the system of equations (3.12) to get its values in one step, and later, we solve K_1 and K_2 in the system of linear equations (3.18).

3.3.9 Experimental Results

In order to evaluate the techniques presented so far we need to choose a concrete bytecode language and an implementation of its abstract machine

to execute and profile with.

In 1983, David H. D. Warren designed an abstract machine for the execution of Prolog consisting of a memory architecture and an instruction set [66, 1]. This design became known as the *Warren Abstract Machine* (WAM), currently is the de facto standard target for most publicly available Prolog compilers, using the architecture suggested by Warren or a derivate of it.

In order to evaluate the feasibility of the approach we have chosen a relatively simple WAM design, which is quite close to the original WAM definition. It is based on [14], but has been ported from Java to C/C++ to achieve similar performance of other Prolog systems.

The use of a relatively simple abstract machine allows evaluating the technique while avoiding the many practical complications present in modern implementations, such as having complex instructions resulting from merging other, simpler ones, or specializations of instruction and argument combinations. This of course does not preclude the application of our technique to the more complex cases.

In our concrete abstract machine, we have considered 42 equations for 43 bytecodes, differentiating the success and failure cases. As we have seen in Proposition 1, there exists a linear relationship between the number of bytecodes that a program will call which can be stated as:

$$\begin{aligned} 0 = \quad & x_{30} + x_{38} - x_{13} - x_{15} - x_{17} - x_{22} - x_{41} \\ & - x_{43} - x_{49} - x_{50} - x_{51} - x_{52} - x_{53} \end{aligned}$$

where the x_i represent the number of times the bytecode tagged as β_i has been executed for any program being analyzed (see Tables 3.9 and 3.11).

By Proposition 2, we are free to select any arbitrary solution of the linear system. The proposed solution has been found by setting arbitrarily the cost of fail to zero. Then, our set of linear equations, discarding those whose

calibrators are composed only with one bytecode, is as follows:

$$\begin{array}{ll}
0 = \beta_{13} & c_{29} = \beta_{01} + \beta_{17} + \beta_{30} \\
c_{01} = \beta_{01} + \beta_{07} & c_{34} = \beta_{01} + \beta_{23} + \beta_{30} + \beta_{35} \\
c_{07} = \beta_{07} + \beta_{24} & c_{36} = \beta_{01} + 2\beta_{28} + \beta_{30} + \beta_{37} \\
c_{09} = \beta_{09} + \beta_{24} & c_{37} = \beta_{17} + \beta_{38} \\
c_{11} = \beta_{01} + \beta_{11} + 2\beta_{28} + \beta_{30} & c_{38} = \beta_{07} + \beta_{24} + \beta_{39} \\
c_{13} = \beta_{01} + \beta_{13} + \beta_{30} & c_{40} = \beta_{01} + \beta_{23} + \beta_{30} + \beta_{41} \\
c_{15} = \beta_{15} + \beta_{38} & c_{42} = \beta_{01} + 2\beta_{27} + \beta_{30} + \beta_{52} \\
c_{17} = \beta_{17} + \beta_{30} & c_{43} = \beta_{01} + \beta_{27} + \beta_{28} + \beta_{30} + \beta_{49} \\
c_{19} = \beta_{19} + \beta_{33} & c_{46} = \beta_{01} + 2\beta_{28} + \beta_{30} + \beta_{50} \\
c_{20} = \beta_{20} + \beta_{33} + \beta_{43} & c_{49} = \beta_{01} + 2\beta_{19} + 2\beta_{27} + \beta_{30} + 2\beta_{31} + 2\beta_{33} + \beta_{51} \\
c_{22} = \beta_{01} + \beta_{22} + \beta_{23} + \beta_{30} & c_{51} = \beta_{01} + 2\beta_{20} + \beta_{30} + 2\beta_{31} + \beta_{53}
\end{array} \tag{3.19}$$

Solving this linear system we get:

$$\begin{array}{ll}
\beta_{01} = c_{29} - c_{17} & \beta_{30} = -c_{29} + c_{17} + c_{13} \\
\beta_{07} = -c_{29} + c_{17} + c_{01} & \beta_{35} = c_{34} - c_{23} - c_{13} \\
\beta_{09} = -c_{29} + c_{17} + c_{09} - c_{07} + c_{01} & \beta_{37} = c_{36} - 2c_{27} - c_{13} \\
\beta_{11} = -2c_{27} - c_{13} + c_{11} & \beta_{38} = c_{37} - c_{29} + c_{13} \\
\beta_{13} = 0 & \beta_{39} = c_{38} - c_{07} \\
\beta_{15} = -c_{37} + c_{29} + c_{15} - c_{13} & \beta_{41} = c_{40} - c_{23} - c_{13} \\
\beta_{17} = c_{29} - c_{13} & \beta_{49} = c_{43} - c_{27} - c_{26} - c_{13} \\
\beta_{19} = c_{19} - c_{32} & \beta_{50} = c_{46} - 2c_{27} - c_{13} \\
\beta_{20} = -c_{44} - c_{32} + c_{20} & \beta_{51} = c_{49} - 2c_{30} - 2c_{26} - 2c_{19} - c_{13} \\
\beta_{22} = -c_{23} + c_{22} - c_{13} & \beta_{52} = c_{42} - 2c_{26} - c_{13} \\
\beta_{24} = c_{29} - c_{17} + c_{07} - c_{01} & \beta_{53} = c_{51} + 2c_{44} + 2c_{32} - 2c_{30} - 2c_{20} - c_{13}
\end{array} \tag{3.20}$$

The leftmost column of Tables 3.9 and 3.11 summarizes the calibration data for the instructions of our WAM implementation. For brevity, we actually only show those being used in the examples tested, although we have calibrated all of them. In the second column there is a tag that is the variable name in the linear equations system.

In the examples we deal with a subset of Prolog which only has operations on integers, atoms, lists, and terms. Likewise, we obviate issues like mod-

ules or syntactic sugar which can be dealt with at the Prolog level. A few additional built-in predicates are required to have a minimal functionality including `write/1`, `consult/1`, etc. They are profiled separately and their timing is given to the system through assertions. This is also a valid solution in order to be able to analyze larger programs.

The experiments were made on the following representative platforms:

- Ultra**Sparc**-T1, 8 cores x 1GHz (4 threads per core), 8GB of RAM, SunOS 5.10.
- **Intel** Core Duo 1.66GHz, 2GB of RAM, Ubuntu Linux 7.04.
- Nokia **N810**. 400MHz processor, 128MB of RAM, Internet Tablet OS, Maemo Linux based OS2008 51.3

In order to reduce noise in the data because of spurious results, we have repeated each experiment 20 times and present the lowest results. In the calibration step 1000 repetitions were made (i.e., $r = 1000$). When possible, the tests were performed with the machines in single-user mode, stopping unnecessary processes. System tasks such as garbage collection, which, as mentioned before, is not considered in our model at the moment, were turned off.

Tables 3.9 and 3.11 show the timing model for this WAM and the architectures studied. In the benchmarks used the `is/2` instruction is compiled into basic operations over pairs of numbers. The table shows the corresponding instructions named `arith_*`. We also have separated the cost of the instructions `put_constant`, `get_constant` when they are called for an atom or an integer. Note however, that their cost is very similar in most cases, but this will still help to reduce errors in the estimation. For the `unify_variable` instruction we have also included calibrations for several cases depending on the type and size of the input arguments in order to increase precision. In other cases, as mentioned in 3.3.8, the execution time of this instruction is not bounded by any a-priori known constant. Since,

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
allocate	β_{01}	29	366	1055
arith_add	β_{02}	29	489	1438
arith_div	β_{03}	29	580	1541
arith_mod	β_{04}	29	641	1553
arith_mul	β_{05}	28	519	1468
arith_sub	β_{06}	28	519	1438
call	β_{07}	11	183	261
cut	β_{08}	13	183	581
deallocate	β_{09}	7	305	142
execute	β_{12}	15	152	574
get_level	β_{18}	28	213	1054
get_list	β_{19}	20	275	763
get_struct	β_{20}	52	642	1766
get_value	β_{21}	43	488	1457
get_variable	β_{23}	43	549	1658
proceed	β_{24}	17	61	699
set_variable	β_{33}	29	213	850
trust_me	β_{38}	29	336	973
try_me_else	β_{39}	30	457	1132
unequal	β_{40}	21	244	1021

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
get_constant_atom	β_{14}	38	518	1211
get_constant_int	β_{16}	28	396	1157
put_a_constant_atom	β_{25}	20	122	594
put_a_constant_int	β_{26}	20	122	506
put_constant_atom	β_{27}	37	274	1085
put_constant_int	β_{28}	37	274	997
put_value	β_{29}	21	183	910
retry_me_else	β_{30}	33	336	999
set_constant_atom	β_{31}	26	213	861
set_constant_int	β_{32}	25	183	767
unify_variable(nvar,var)	β_{42}	35	396	1309
unify_variable(var,nvar)	β_{43}	35	397	1309
unify_variable(int,int)	β_{44}	32	275	1179
unify_variable(atm,atm)	β_{46}	44	427	1413
unify_variable(str(1),str(1))	β_{47}	77	885	2560
unify_variable(list(1),list(1))	β_{45}	96	1068	3291
unify_variable(list(100),list(100))	β_{48}	4062	42511	217975

Table 3.9: Timing model for the WAM instructions. Cost of bytecodes when they succeed.

Platform	Timing Model (ns)
Intel	$44 + 40.62 * length(X)$
N810	$427 + 425.11 * length(X)$
Sparc	$1413 + 2179.75 * length(X)$

Table 3.10: Timing model given by a linear function, for `unify_variable(X,Y)` when the arguments are lists of integers, and the instruction does not fail.

as also shown in Section 3.3.8, in our implementation it is possible to use functions instead of constants as timing model for a given instruction, in this table we include in the calibrations two data points for the case when the arguments are lists of integers, and for lists of size (length) 1 and 100 (β_{45} and β_{48} in Table 3.9). The value for an empty list is the same as for unifying any two equal atoms, i.e., β_{46} in Table 3.9. Table 3.10 shows the resulting

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
fail	β_{13}	0	0	0
get_constant_atom	β_{15}	32	457	1256
get_constant_int	β_{17}	26	366	1169
get_value	β_{22}	25	244	1106
unequal	β_{41}	11	61	651
unify_variable	β_{43}	121	1065	3867

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
unify_variable(const1, const2), const1 $\neq$ const2	β_{49}	41	154	697
unify_variable(int,int)	β_{50}	122	1035	3830
unify_variable(list(1),list(1))	β_{51}	338	3227	12229
unify_variable(atm,atm)	β_{52}	127	1126	4282
unify_variable(str(1),str(1))	β_{53}	223	2381	9239

Table 3.11: Timing model for the WAM instructions. Cost of bytecodes when they fail.

#	Program	Data size
1	append(+A,+,-)	x=length(A)=150
2	evalpol(+A,+X,-)	x=length(A)=100
3	fib(+N,-)	x=N=16
4	hanoi(+N,+,+,+,-)	x=N=8
5	nreverse(+L,-)	x=length(L)=83
6	palindro(+A,-)	x=length(A)=9
7	powset(+A,-)	x=length(A)=11

#	Program	Data size
8	list_diff(+L,+D,-)	x=length(L)=65 y=length(D)=65
9	list_inters(+L,+D,-)	x=length(L)=65 y=length(D)=65
10	substitute(+A,+B,-)	x=term_size(A)=67 y=term_size(B)=80
11	derive(+E,+,-)	x=term_size(E)=75

Table 3.12: List of program examples used in the experimental assessment.

timing model for `unify_variable` using these values to fit our linear model for this instruction.

Using the timing model shown in Tables 3.9, 3.10, and 3.11, we have performed some experiments with a series of programs on the three platforms (Intel, N810, and Sparc) in order to assess the accuracy of our technique for estimating execution times. The results of these experiments are shown in Tables 3.13 (Intel), 3.14 (N810), and 3.15 (Sparc).

Column **Pr. No.** lists the program identifiers, whose association with the programs and the input data sizes used is shown in Table 3.12. Column **Cost App.** indicates the type of approximation of the automatically inferred cost functions which estimate execution times (as a function on input data size): upper-bound (**U**), lower-bound (**L**), or exact (**E**). Such cost functions are shown in column **Cost Function** for the three different platforms considered in our experiments. The variables x and y represent the sizes of the

Pr. No.	Cost. App.	Intel Cost Function	Est. (μs)	Prf. (μs)	Obs. (μs)	D. (%)	Pr.D. (%)
1	E	$0.73x + 0.21$	110	110	113	-2.4	-2.4
2	E	$0.69x + 0.19$	69	69	71	-2.3	-2.3
3	E	$0.69 \cdot 1.6^x + 0.21(-0.62)^x - 0.72$	1525	1525	1576	-3.3	-3.3
4	E	$-0.0042 \cdot 2^x + 0.73x \cdot 2^x - 0.86$	1501	1501	1589	-5.7	-5.7
5	E	$0.37x^2 + 0.49x + 0.12$	2569	2569	2638	-2.7	-2.7
6	E	$0.36 \cdot 2^x + 0.37x \cdot 2^x - 0.24$	1875	1875	2027	-7.8	-7.8
7	E	$0.91 \cdot 2^x + 0.87x - 0.6$	1868	1868	1931	-3.3	-3.3
8	L	$0.66x + 0.2$	43	68	81	-67.2	-17.8
	U	$0.78xy + 1.7x + 0.4$	3414	3569	3640	-6.4	-2.0
9	L	$0.83x + 0.2$	54	79	91	-54.6	-14.8
	U	$0.78xy + 1.7x + 0.4$	3414	3694	4011	-16.2	-8.2
10	L	$2x$	135	142	124	8.6	13.7
	U	$1.4xy + 1.4y + 6.1x + 4.1$	7922	2937	2858	120.6	2.7
11	L	$2.9x$	216	138	111	72.3	22.5
	U	$3x + 3$	226	216	162	34.0	29.5

Table 3.13: Observed and estimated execution time with cost functions, Intel platform (microseconds).

input arguments to the programs which are relevant for the inference of the cost functions. The type of approximation directly depends on the one used by the resource analysis described in Section 2.3.3 and particularized in Section 3.3.3 for estimating the number of executed instructions (as a function on input data size). The value **E** means that the lower- and upper-bound cost functions are the same, and thus, since the analysis is safe, this means that the exact cost function was inferred. Using the cost functions shown in column **Cost Function**, and in order to assess their accuracy, we have also estimated execution times for particular input data for each program and compared them with the observed execution times. These execution times are shown in columns **Est.** and **Obs.** respectively. Column **D.** shows the

Pr. No.	Cost. App.	N810 Cost Function	Est. (μs)	Prf. (μs)	Obs. (μs)	D. (%)	Pr.D. (%)
1	E	$7.8x + 2.7$	1169	1169	1037	12.0	12.0
2	E	$7.8x + 2.7$	786	786	641	20.6	20.6
3	E	$8.3 \cdot 1.6^x + 2.5(-0.62)^x - 8.4$	18333	18333	14496	23.7	23.7
4	E	$0.74 \cdot 2^x + 7.8x \cdot 2^x - 10$	16095	16095	16144	-0.3	-0.3
5	E	$3.9x^2 + 5.7x + 1.6$	27247	27247	28381	-4.1	-4.1
6	E	$4.4 \cdot 2^x + 3.9x \cdot 2^x - 2.9$	20167	20167	20416	-1.2	-1.2
7	E	$9.5 \cdot 2^x + 10x - 6$	19517	19517	19653	-0.7	-0.7
8	L	$7.3x + 2.8$	474	744	640	-30.4	15.1
	U	$8.2xy + 19x + 5.5$	35849	37162	29266	20.4	24.1
9	L	$8.7x + 2.8$	569	839	732	-25.4	13.7
	U	$8.2xy + 19x + 5.5$	35849	38076	29907	18.2	24.4
10	L	$21x$	1399	1475	1068	27.3	32.9
	U	$15xy + 15y + 64x + 43$	85893	30375	25543	153.3	17.4
11	L	$29x$	2190	1423	854	108.7	53.3
	U	$30x + 30$	2306	2193	1342	56.8	51.1

Table 3.14: Observed and estimated execution time with cost functions, Nokia N810 platform (microseconds).

relative harmonic difference * between the estimated and the observed time. The source of inaccuracies in the execution time estimations of our technique come mainly from two sources: the timing model (which gives the execution time estimation of bytecodes, as shown in Tables 3.9 and 3.11)) and the static analysis (see Section 3.3.3, which estimates the number of times that the bytecodes are executed, depending on the input data size). Since we are interested in identifying the source(s) of inaccuracies, we have also introduced the column **Prf.** It shows the result of estimating execution times using the timing model and assuming that the static analysis was perfect and obtained a function which provides the *exact* number of times that the

*$rel\_harmonic\_diff(x, y) = (x - y)(1/x + 1/y)/2$.

Pr. No.	Cost. App.	Sparc Cost Function	Est. (μs)	Prf. (μs)	Obs. (μs)	D. (%)	Pr.D. (%)
1	E	$26x + 7.4$	3906	3906	4670	-18.0	-18.0
2	E	$25x + 7.1$	2543	2543	2985	-16.1	-16.1
3	E	$26 \cdot 1.6^x + 7.8(-0.62)^x - 27$	56828	56828	59120	-4.0	-4.0
4	E	$1.2 \cdot 2^x + 26x \cdot 2^x - 33$	53504	53504	63156	-16.7	-16.7
5	E	$13x^2 + 17x + 4.3$	90973	90973	109849	-19.0	-19.0
6	E	$13 \cdot 2^x + 13x \cdot 2^x - 8.5$	66400	66400	78980	-17.4	-17.4
7	E	$32 \cdot 2^x + 32x - 22$	66224	66224	78151	-16.6	-16.6
8	L	$24x + 7.1$	1574	2458	2991	-68.7	-19.7
	U	$27xy + 62x + 14$	118269	123733	129951	-9.4	-4.9
9	L	$30x + 7.1$	1940	2824	3394	-58.9	-18.5
	U	$27xy + 62x + 14$	118269	127378	133703	-12.3	-4.8
10	L	$68x$	4545	4821	4634	-1.9	4.0
	U	$48xy + 48y + 207x + 140$	277175	101779	111829	103.8	-9.4
11	L	$95x$	7104	4628	4038	59.6	13.7
	U	$98x + 98$	7454	7147	6081	20.5	16.2

Table 3.15: Observed and estimated execution time with cost functions, Sparc platform (microseconds).

bytecodes are executed. This obviously represents the case in which all loss of accuracy must be assigned to the timing model. The "perfect" cost model is obtained from an actual execution by instrumenting the profiler so that it records the number of times each instruction is executed for the application and the particular input data. Column **Pr.D.** shows the relative harmonic difference between **Prf.** and the observed execution time **Obs.**

The upper part of Tables 3.13, 3.14, and 3.15, up to the double line corresponds to examples where an exact cost function for the number of executed bytecodes was automatically inferred by the static analysis (note that, as expected, the values **Est.** and **Prf.** are the same). We can see that with an exact static analysis, the estimated execution times **Est.** are quite

Model	A	B	C	D	WAM based
Deviation (%)	51.17	31.06	21.48	58.45	4.72

Table 3.16: Comparison between the higher level models and the abstract machine-based model, on the Intel platform.

precise, which in turn supports the accuracy of our timing model.

It is particularly interesting to compare these results with those which were obtained using a variety of higher-level models in [49]. Table 3.16 provides the standard deviation of the four high-level models of [49] as well as that of the abstract machine-based model presented in this section, for the Intel platform and our set of benchmarks. It can be observed that the results obtained with the abstract machine-based model are more than five times better on the same platform than those obtained using the higher-level models.

With the abstract machine-based model, and for this type of programs we believe that the remaining error comes simply from the accumulated loss of accuracy of the bytecode instruction profiling and expect that making the *timing* model more precise will increase precision even further.

The lower part of Tables 3.13, 3.14, and 3.15 shows programs for which there is no unique value for $\Phi(p, time, \overline{n})$, where $\Phi(p, time, \overline{n})$ (as described in 2.3) denotes the cost (in time units) of computing a call to program p for an input of size $\overline{n}$ on a given platform. The reason is that for such programs, the number of instructions executed does not only depend on the input data sizes, but also depends on other characteristics of the input data (e.g., their actual values). Thus, for a given data size, there are actual lower- and upper-bounds for the cost of the program calls. For this reason, the two observed execution times shown in column **Obs.** for each program have been obtained by running the program with the input data, of the size specified in Table 3.12, that yield the actual lower- and upper-bounds to the execution times for such size. In this case, the static analysis infers approximations to such actual lower- and upper-bound cost functions (**L** and **U** respectively).

These predictions are understandably much less accurate in these cases than those in the first part of the table, but still reasonable. In any case, lower-bounds and upper-bounds tend to be reasonably smaller or bigger than the observed execution times respectively. In general, for the programs in the lower part of the tables with big (absolute) values for **D.**, the (absolute) value for **Pr.D.** is reasonably small. This means that, in those cases, most of the inaccuracy in the estimation of execution times (**Est.**) comes from the static analysis, which does not approximate actual lower- and upper-bound cost functions accurately enough, and that the timing model used for predicting the execution time of bytecodes is reasonably precise. Thus, we believe that using a better static analysis for inferring cost functions which take into account other characteristics of the input data, besides their sizes, would significantly improve the predictions. In any case, there is always a reasonable slack in the precision of the estimations due to the timing measurements and the timing model. In most cases the best approximation is given by the combination of "upper approximation of cost execution" with "mean of bytecode instruction execution time," even if this combination still sometimes produces inaccurate results for this class of programs. This is, in any case, quite understandable since, to start with, no exact cost function was deduced for them.

For nondeterministic programs involving significant search the predicted and the actual execution times turn out to be very different. This is ultimately due to the lack of accuracy of the automatic cost analysis when dealing with such programs, which point to an issue to be solved, i.e., that of finding out reliably the number of solutions of a goal. We are not showing any example of this kind, since they add little interesting information regarding the pursue of accuracy.

3.3.10 Section Conclusions and Future Work

We have developed a framework for estimating upper- and lower-bounds on the execution times of logic programs running on a bytecode-based abstract

machine. We have shown that working at the abstract machine level allows taking into account low-level issues without having to tailor the analysis for each architecture and platform, and allows obtaining more accurate estimates than with previous approaches, including comparatively accurate upper- and lower-bound estimations of execution time.

Although the framework has been presented in the context of logic programs, we believe the technique can easily be applied to other languages. This adaptation of the approach, while certainly not trivial, to some extent would actually imply some simplification, since backtracking does not need to be taken into account. For example, analyses have been recently developed for Java bytecode [2] which infer the number of execution steps using similar techniques to those used in logic programming [22, 21, 23]. Such analyses could be adapted, following the techniques presented herein, to take into account the bytecode timing information and would then be able to estimate actual execution time for Java programs. Appropriate cost models for Java bytecode are already being developed in [60].

We believe that the more accurate execution time estimates that can be obtained with our technique can be very useful in several contexts including parallelism, compilation, real-time applications, pervasive systems, etc. More concretely, increased timing precision can improve the effectiveness of resource/granularity control in parallel/distributed computing. This belief is based on previous experimental results, where it appeared that, even if improved precision in timing estimates is not essential, it does yield increased speedups. Also, the inferred cost functions can be used to develop automatic program optimization techniques. For example, they can be used for performing self-tuning specialization which compares statically the estimated execution time of different specialized versions [18].

Given that our experimental results are encouraging with respect to actually being able to find more accurate upper- and lower-bounds to program execution times, the approach may eventually also be used for verification (or falsification) of timing constraints, as in, for example, real-time systems,

which was not possible in an accurate way with previous approaches. In fact, our approach (which can be adapted to take also into account destructive assignment, as in [50]) can potentially be used to solve a common problem in current WCET static analysis, where only constant WCET bounds (i.e., non dependent on input data sizes) are inferred. These bounds are not always appropriate since the WCET of a given program often depends on several input parameters, and using an absolute bound, covering all possible situations (i.e., all possible values or sizes of input), produces only a very gross over approximation [28]. Substituting the estimated costs of the bytecodes by the actual worst-case costs of the instructions and using our approach, the WCET is expressed as a cost function parameterized by the size or values of input arguments, providing tighter WCET approximations. On the other hand, WCET work has produced more accurate (but, unfortunately, non-freely available) timing models which take into account many low-level parameters (such as cache behavior, pipeline state, etc.) which we have abstracted away in our work. It is clear that a combination of both techniques might be very useful in practice.

Chapter 4

Unit-Testing, Run-Time and Compile-Time Checking

4.1 Introduction

In this chapter we present an approach that unifies *unit testing* with *run-time verification* within an overall framework that also comprises *static verification* and *static debugging* [12, 33, 54, 55, 34]. This novel framework for program development is aimed at finding bugs in programs or validating them with respect to (partial) specifications given in terms of *assertions* (using the concept of *abstractions* as over-/under-approximations of program semantics). A novel and expressive language of assertions allows describing quite general program properties [53, 56, 17, 11].

The previous work in this context cited above has concentrated mostly on the static (i.e., compile-time) checking of such assertions as well as on techniques for reducing at compile-time the number of checks that have to be performed dynamically (i.e., at run time): any assertions present in the program are verified (or falsified) to the extent possible during the compilation phase, since compile-time checking is always preferable to run-time checking –always incomplete as a means of verification. However the existence in all practical programs of data only known at run-time and the rich nature of the

properties considered make a certain degree of run-time checking inevitable –a reasonable price to pay in return for property expressiveness.

In this work we concentrate instead on the run-time portion of the model. Our aim is to a) develop effective implementation techniques for run-time checking that integrate seamlessly into our combined compile-time/run-time framework and b), based on this, to also develop *well-integrated* facilities for unit testing. To this end, we have first developed an implementation of run-time checks, as an evolution of the approach sketched in [55], based on transforming the program into a new one which preserves the semantics of the original program and at the same time checks during its execution the assertions. Such transformation allows checking preconditions and postconditions, including conditional postconditions, i.e., postconditions that must hold only when certain preconditions hold. It also allows checking properties at arbitrary program points (i.e., in literal positions in clause bodies) as well as certain computational properties (properties that are not specific to a program point but rather to whole computations, such as, for example, determinism, non-failure, or use of resources –steps, time, memory, etc.).

Our transformation also addresses to some extent one of the main drawbacks of run-time checking (in addition to incompleteness): the overhead introduced during execution of the program. The proposed transformation reduces run-time overhead by avoiding meta-interpretation whenever possible and by using special features of the low-level language when appropriate. Also, run-time checks can be compiled inline as opposed to calling a library, saving (meta-)call overhead.

Another relevant issue addressed by our transformation is being able to provide messages to the user which are as informative as possible when a violation of the safety policy is found, i.e., when a run-time check fails. To this end, the transformation saves appropriate information at source code level in the transformed file. Depending on the level of code instrumentation selected, increasingly more accurate information about the assertions is saved, and, thus, presented, offering different trade-offs between information level and

program size.

With respect to *testing*, we propose a minimal extension to the assertion language in order to be able to define *unit tests* [26]. The resulting language can express for example the input data for performing such unit tests, the expected output, the number of times that the unit tests should be repeated, etc. In contrast to previous work in this area (e.g., [8], [69], or the unit test framework recently included in SWI-Prolog [67]), a key contribution of our approach is that these unit tests blend in with the assertion language and reuse the overall framework. In particular, as mentioned before, only *test drivers* need to be added because the existing run-time assertion checking machinery is used as a checker for the cases defined by the unit tests.

An advantage of our approach is that the unit test specifications can be encapsulated in the same module that contains the predicates being tested, or placed in a separate file containing the tests for the module or modules of the application. This contrasts with, e.g., the `plunit` unit testing of SWI-Prolog, where unit test specifications are written in the same source code module or in a dedicated file with the same name as the module being tested.

Both the run-time check generation and the unit testing approaches proposed have been implemented within the `CiaoPP/Ciao` system. We provide some experimental results which illustrate the implementation trade-offs involved. As mentioned before, thanks to the `CiaoPP/Ciao` machinery only the (parts of) assertions which cannot be verified at compile-time are converted into run-time checks. Since in our approach unit tests are also assertions, static analysis can also eliminate parts of or whole unit tests. At the same time, the tight integration also allows using the unit test drivers to exercise run-time checks corresponding to those parts of assertions that could not be checked at compile-time, even if they were not conceived as tests. Figure 4.1 shows the resulting `Ciao/CiaoPP`'s unified framework.

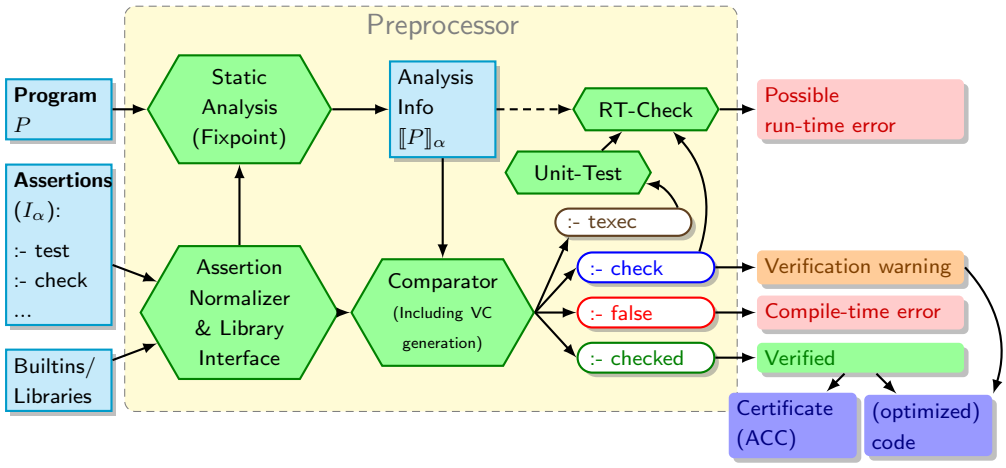

Figure 4.1: The `Ciao` unified assertion framework (`CiaoPP`'s verification/testing architecture).

4.2 The Ciao Assertion Language

Assertions are linguistic constructions which allow expressing properties of programs. In the `Ciao` assertion language, assertions are always instances of some *assertion schema*. Such schemas allow talking about preconditions, (conditional) postconditions, whole executions, program points, etc. Each schema in turn contains one or two logic formulae which are (intuitively) used to say things such as "X is a list of integers," "Y is ground," "$p(X)$ does not fail," etc. In this approach the user has a high degree of freedom for defining these logic formulae for the properties considered of interest. For space considerations, we will focus on a subset of the `Ciao` assertion language. In particular, although the language has assertions specifically designed for expressing properties related to the declarative semantics, in this work we will focus on the *operational* semantics of programs, more specifically, in assertions referring to *execution states* and *computations* (see [56, 11] for a detailed description of the full language). Also, although the assertion language incorporates significant syntactic sugar, we will use only the (unfortunately more verbose) raw forms. An execution state $\langle G \mid \theta \rangle$ consists of the current goal G

program-assert	::=	`:-` *predicate-assert* `.` | *prog-point-assert*
predicate-assert	::=	*pred-assert* | *status pred-assert*
		| `entry` *pred-cond* | `exit` *pred-cond*
		| `texec` *pred-cond* `+` *exec-formula*
pred-assert	::=	`calls` *pred-cond*
		| `success` *pred-cond* `=>` *state-formula*
		| `comp` *pred-cond* `+` *comp-formula*
pred-cond	::=	*Pred* | *Pred* : *state-formula*
Pred	::=	*Pred-name*`(`*args*`)`
args	::=	*Var* | *Var*`,`*args*
state-formula	::=	`(`*state-formula* `,` *state-formula*`)`
		| `(`*state-formula* `;` *state-formula*`)`
		| `compat(`*State-prop*`)` | *State-prop*
comp-formula	::=	`(`*comp-formula* `,` *comp-formula*`)`
		| `(`*comp-formula* `;` *comp-formula*`)`
		| *Comp-prop*
exec-formula	::=	`(`*exec-formula* `,` *exec-formula*`)` | *Exec-prop*
status	::=	`check` | `true` | `checked` | `trust` | `false`
prog-point-assert	::=	*status*`(`*state-formula*`)`

Figure 4.2: Syntax of the assertion language.

and the current constraint store θ which contains information on the values of variables. The operational semantics is given in terms of *derivations*, which are sequences of reductions between such execution states. By *computation* we mean the (sorted) execution tree containing all possible derivations of a goal from a calling state. Figure 4.2 shows the grammar of the raw form assertion (sub)language (including the extensions for unit testing that will be described later).

Predicate Assertions: They refer to properties of a particular predicate. In the schemas below a concrete assertion will include concrete values in place of the symbols *Pred*, *Precond* and *Postcond*. In all schemas *Pred* is a *predicate descriptor*, i.e., a predicate symbol applied to distinct free variables, and *Precond* and *Postcond* are logic formulas about execution states, that we call *state-formulae*. An atomic *state-formula* is constructed with a *state property predicate* (e.g., `list(X)` or `X > 3`) which expresses properties about (the values) of variables. A *state-formula* can also be a conjunction or disjunction of *state-formulae*. Standard (C)LP syntax is used, so that the comma should be interpreted as conjunction (e.g., "`( list(X), list(Y) )`"), and the semicolon as disjunction (e.g., "`( list(X) ; int(X) )`").

- *Describing success states:*

  `:- success` *Pred* [`:` *Precond*] `=>` *Postcond*`.`

  Interpretation: in any call to *Pred*, if *Precond* succeeds in the calling state and the computation of the call succeeds, then *Postcond* should also succeed in the success state.

  Example 4.2.1. The following assertion expresses that for any call to predicate `qsort/2` with the first argument bound to a list of numbers, if the call succeeds, then the second argument should also be bound to a list of numbers:

  ```
  :- success qsort(A,B) : list(A,num) => list(B,num).
  ```

  □

  If *Precond* is omitted, the assertion is equivalent to:
  `:- success` *Pred* `: true =>` *Postcond*`.`
  and it is interpreted as "for any call to *Pred* which succeeds, *Postcond* should succeed in the success state."

- *Describing admissible calls:* `:- calls` *Pred* `:` *Precond*`.`

  Interpretation: in all calls to *Pred*, the formula *Precond* should succeed in the calling state.

Example 4.2.2. The following assertion expresses that in all calls to predicate `qsort/2`, the first argument should be bound to a list of numbers:

```
:- calls qsort(L,R) : list(L,num).
```

□

The set of all `call` assertions is considered *closed* in the sense that they must cover all valid calls.

- *Describing properties of the computation:*

  `:- comp` *Pred* [`:` *Precond*] `+` *comp-formula*`.`

  Interpretation: for any call to *Pred*, if *Precond* succeeds in the calling state, then *comp-formula* should also succeed for the computation of *Pred*.

  Example 4.2.3. Let the assertion:

  ```
  :- comp qsort(L,R) : ( list(L,num), var(R) ) + not_fails.
  ```

  where the atom `not_fails` is implicitly interpreted as `not_fails(qsort(L,R))`, i.e., it is as if it executed $\langle qsort(L,R) \mid \theta \rangle$ and checked that it does not fail. □

In addition, other assertion schemas such as `entry` and `exit` assertions can be used to refer to external calls to the module.*

Program-Point Assertions: The program points considered are the places in a program in which a new literal may be added, i.e., before the first literal (if any) of a clause, between two literals, and after the last literal (if any) of a clause. Program-point assertions are literals appearing at the corresponding program point and which are of the form: `check`(*state-formula*).

*Note that in `CiaoPP` the `pred` assertions of exported predicates can be used optionally instead of `entry` and `exit` assertions to define the module interface.

where *state-formula* is a logic formula about execution states (see the grammar in Figure 4.2). The resulting assertion should be interpreted as "whenever computation reaches a state originated at the program point in which the assertion is, *state-formula* should succeed."

Status: Independently of the schema used, each assertion has a flag (`check`, `trust`, `true`, etc.), the assertion "status," which determines whether the assertion is to be checked, to be trusted, has already been proved correct by analysis, etc. Again for simplicity we use only the `check` status herein (which is assumed by default when no flag is present).

The Logic Formulae: We allow conjunctions and disjunctions in the formulae, and choose to write them down, for simplicity, in the usual CLP syntax. Thus, logic formulae about execution states can be:

- An atom of the form $p(t_1, \ldots, t_n)$ with $n \geq 0$, where p/n is a *property predicate* (e.g., `list(X)` or `X > 3`).
- An expression of the form ($F1$, $F2$) where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the comma should be interpreted as conjunction (e.g., "`( list(X), list(Y) )`").
- An expression of the form ($F1$; $F2$) where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the semicolon should be interpreted as disjunction (e.g., "`( list(X) ; int(X) )`").

4.3 Run-Time Checking of Assertions

In this section we first focus on run-time checking of predicate assertions, and then we comment on the approach for program-point assertions, since the later is much simpler than the former. Our run-time checking system

step one	step two
p :- *entry-checks*, *exit-preconditions-checks*, *ext-comp-checks*(**p1**), *exit-postconditions-checks*. % p renamed to p1 % within module	**p1** :- *calls-checks*, *success-preconditions-checks*, *comp-checks*(*call_stack*(**p2**, *locator*)), *success-postconditions-checks*. **p2** :- body_0. ... **p2** :- body_n.

Figure 4.3: The *transforming procedure definitions* scheme for run-time checking.

is composed of a set of transformations, to be performed by the preprocessor, and a library containing a number of primitives that the transformed programs will call.

We start by discussing two possible approaches regarding the source-to-source transformations to be performed in order to implement run-time checking schemes. In the first kind of transformation, that we refer to as "transforming calls," the run-time checks are placed before and after any call to predicates which are affected by assertions. In the second kind of transformation, that we call "transforming procedure definitions," the original predicate is rewritten so that it performs the run-time checks itself, each time it is called, and calls to it are left unchanged. Figure 4.3 illustrates this approach for a predicate `p`. In this transformation the original predicate `p` is renamed to `p2` and a new definition of `p`, which performs the run-time checks, is added by following two steps. "Step one" (first column of the figure) is used to add any run-time checks corresponding to, e.g., `entry` and `exit` assertions before and after a call to a new predicate `p1`.

The objective of this first transformation is to separate external calls from internal ones. Then `p1` is defined so that it calls predicate `p2` and performs

all run-time checks corresponding to each type of (kernel) predicate-level assertions, i.e., `calls`, `success`, or `comp` in the right place. In this kind of transformation, calls to `p` are left unchanged.

Clearly, each scheme has advantages and disadvantages, specially when considering a program consisting of several modules. When transforming calls, additional run-time checking code will be introduced in all modules that call the predicate affected by a given assertion. This will likely result in a larger code size than in the transforming procedure definitions approach, since a program can easily see a large number of assertions from, e.g., libraries. Also, if a given file containing an assertion is modified, all the modules using it will have to be recompiled. The big advantage of the transforming calls approach is that if no run-time assertion checking is required in a given module, only that module needs to be recompiled, whereas in the transforming procedure definitions approach all the modules containing procedures with run-time checks and which are used by the given module need recompilation. Thus, for libraries, in the transforming calls approach only one version of each file is compiled whereas in the transforming procedure definitions approach typically two versions of the libraries are kept in the system, one with run-time checks and the other one without. Both approaches allow mixing modules with and without run-time checks. Another potential advantage of the transforming calls approach is that it makes it easier for certain kinds of analysis and specialization algorithms (specially those which are not multivariant) to analyze and optimize programs annotated with run-time checks. On the other hand, if the analysis and specialization system is multivariant (as in the case of `CiaoPP`) this is less of an issue.

In view of all the advantages and disadvantages discussed above, we currently use the transforming procedure definitions approach.

Transforming Single Predicate Assertions: We first consider the case where there is only one predicate assertion for a given predicate. We show schemes for transforming assertions into run-time checks for each type of

Assertion:	The definition of *Pred* is transformed into:
`:- calls` *Pred* `:` *Cond*.	*Pred* `:- rtcheck(`*Cond*`),` *Pred'*. *Pred'* `:-` ... .
`:- success` *Pred* `:` *Precond* `=>` *Postcond*.	*Pred* `:- checkc(`*Precond*`,F),` *Pred'*, `checkif(F,`*Postcond*`).` *Pred'* `:-` ... .
`:- comp` *Pred* `+` *Comp*.	*Pred* `:- check_comp(`*Comp*`(G),G,`*Pred'*`).` *Pred'* `:-` ... .
`:- comp` *Pred* `:` *Precond* `+` *Comp*.	*Pred* `:- checkc(`*Precond*`,F),` `checkif_comp(F,`*Comp*`(G),G,`*Pred'*`).` *Pred'* `:-` ... .

Figure 4.4: Translation schemes for different kinds of predicate assertions.

(kernel) predicate assertion, i.e., `calls`, `success`, or `comp`. Other, higher-level assertions (such as `pred` assertions) and all additional syntactic sugar (such as modes or "star notation") are translated by the compiler into the kernel assertions before applying the transformation. These schemes express what run-time library predicates are called and where such calls are placed.

Figure 4.4 shows the schemes, whereas the run-time library includes the following predicates predicates: `checkc`, `rtcheck`, `checkif`, `checkif_comp`, `check_comp` and `call_stack`, (which can be used for both the transforming calls and transforming procedure definitions approaches) and are described below. †

`checkc(`*C*`,`*F*`):`

checks condition *C* and sets *F* to true or false depending on whether it succeeds or not. Defined as:

```
(\+ C -> F = false ; F = true).
```

†The schemas for `entry`/`exit` assertions are the same as the corresponding to `calls`/`success` assertions, and thus are not shown in the Figure.

`rtcheck(`C`)`:

checks if condition C succeeds or not. If C fails, an exception is raised. This can be understood simply as `\+\+` C (so that bindings/-constraints produced by the condition succeeding are removed –an *entailment* check).

`checkif(`F`,`P`)`:

postcondition P is checked iff F is `true`. If P fails, an exception is raised. This can be defined as:

```
(F == true -> rtcheck(P) ; true).
```

`checkif_comp(`F`,`$Comp(G)$`,`G`,`$Pred'$`)`:

checks a computational property if F is *true*, for a given computational property $Comp(G)$, and a predicate *Pred'* to be checked. For example, if the property is `not_fails/1` and the predicate `qsort(A,B)`, then we call `checkif_comp(`F`,not_fails(`G`),`G`,qsort2(A,B))`. In turn, $Pred'$ is used to pass the direct call to the predicate (i.e., `qsort2(A,B)` in the example). If F is `false` then $Pred'$ is called, executing the procedure directly. If F is `true` then G is unified with $Pred'$ and $Comp(Pred')$ is called. This relies on the fact that `comp` properties are written assuming that the goal to be called is passed as an argument and that they take care of both running the procedure and checking whether the computational property holds. Again, if the (in this case, computational) property does not hold, an exception is raised. The predicate `checkif_comp/4` can be defined as:

```
checkif_comp(fail,_,_,Pred) :- call(Pred).
checkif_comp(true,CompCall,Pred,Pred) :- call(CompCall).
```

`check_comp(`$Comp(G)$`,`G`,`$Pred'$`)`:

a specialized version of `checkif_comp(true,` $Comp(G)$`,` G`,` $Pred'$`)`, where the first parameter is assumed to be true.

`call_stack`(C, L)**:**

adds the current source code locator L to the locator stack S allowing to show the call stack on run-time errors. This can be understood as:

```
intercept(C,rtc_error(S,T),send_signal(rtc_error([L|S],T))).
```

The previous library predicates are implemented in such a way that they perform the checks without modifying the program state, introducing side effects, errors, etc. In other words, if all run-time errors are intercepted, the semantics of the program must be preserved.

Combining Several Predicate Assertions: We now consider the case where there are several assertions for a given predicate. Translating several `calls` or `success` assertions is relatively straightforward: the corresponding `rtcheck/1` and `checkc/2` are placed before the call to *Pred'*, and any calls to `checkif/2` are gathered after it. In the case of `calls` assertions run-time check exceptions for the unsatisfied assertions are thrown only if *all* such checks fail.

Combining computational properties is somewhat more involved. First we consider the case of a single `comp` assertion with several properties, such as, e.g.:

```
:- comp qsort(A,B) : (list(A,int),var(B)) + (is_det,not_fails).
```

In this case the properties will simply be nested in the *Comp* field as follows: *prop1*`(`*prop2*`(` *... propN*`(`*Pred'*`)` *...* `))` (the *Pred'* field stays obviously the same). For example, for the assertion above the *Comp* field will be `not_fails(is_det(qsort_1(A,B)))`. If the `comp` property has a precondition, it will be checked only once and then either the *Comp* field or *Pred'* will be called.

The situation is more complex when several `comp` assertions have to be combined. Consider for example the following two `comp` assertions:

```
:- comp qsort(A,B) : (ground(A), var(B)) + is_det.
:- comp qsort(A,B) : (list(A,int), var(B)) + not_fails.
```

Assuming that `F1` and `F2` are the flags resulting from checking the conditions `ground(A), var(B)` and `list(A,int), var(B)` respectively, the composition of the two assertions above would be:

```
checkif_comp(F2,not_fails(G2),G2,
             checkif_comp(F1,is_det(G1),G1,qsort2(A,B))).
```

After all the transformations explained above have been made, an invocation of `call_stack/2` is instrumented in order to save the locator in the stack.

Finally, we give some intuitive ideas in order to relate Figures 4.3 and 4.4. The composition of all the transformations described in Figure 4.4 applied to assertions corresponding to external calls to predicate `p` in Figure 4.3 (e.g. `entry`/`exit` assertions), gives the new definition of `p`. The *Pred'* resulting from the last transformation in Figure 4.4 will be predicate `p1` in Figure 4.3. Also, the *Pred'* resulting from the last transformation in Figure 4.4 for `calls`, `success`, and `comp` assertions (which affect to internal calls) will be predicate `p2` in Figure 4.3. In particular, *comp-checks*(*call_stack(***p2***, locator)*) will be the composition of all the transformations corresponding to `comp` assertions.

Program-Point Assertions: Figure 4.5 shows how clauses are transformed in order to incorporate run-time checking of program-point assertions. This is a comparatively simpler task than transforming predicate-level assertions: the natural transformation is a similar one to the "transforming calls" approach, but with the advantage that only one program point needs to be transformed for each assertion. Also, only the `rtcheck/1` and `check_comp/1` primitives are required; and in the case of computational properties their definitions are called directly.

4.4 Defining Unit Tests

In order to define a unit test we have to express on one hand *what to execute* and on the other hand *what to check* (at run-time). A key characteristic of our approach is that we use the assertion language described in Section 4.2

Program-point assertion:	The clause is transformed into:
Pred :- ..., **check**(Cond), ...	Pred :- ..., rtcheck(Cond), ...
Pred :- ..., **check**(CompProp(Goal)), ...	Pred :- ..., check_comp(CompProp(Goal)), ...

Figure 4.5: Translation schemes for different kinds of program-point assertions.

for expressing what to check. This way, the same properties that can be expressed for static or run-time checking can also be checked in unit testing. However, we have added a minimal number of elements to the assertion language grammar for expressing *what to execute.* They appear underlined in Figure 4.2. In particular, we have added a new assertion schema:

`:- texec` *Pred* [`:` *Precond*] [`+`*Exec-Formula*]`.`

which states that we want to execute (as a test) a call to *Pred* with its variables instantiated to values that satisfy *Precond*. *Exec-Formula* is a conjunction of properties describing how to drive this execution. In our approach many of the properties usable in *Precond* (e.g., types) can be run as value generators for these variables, so that input data can be automatically generated for the unit tests (see the technique described in [30]). However, we have defined some specific properties, such as random value generators that follows a given distribution.
We now describe and illustrate with examples the new properties added to the assertion language for describing test cases.

Example 4.4.1. The assertion:

```
:- texec append(A, B, C) : (A=[1,2],B=[3],var(C)) + times(5).
```

expresses that a call to `append/3` with the first and second arguments bound to `[1,2]` and `[3]` respectively and the third one unbound should be executed five times. □

Example 4.4.2. We can define a unit test using the assertion in Example 4.4.1 together with the following two assertions expressing *what to check at run-time*:

```
:- check success append(A,B,C):(A=[1,2],B=[3],var(C))=>C=[1,2,3].
:- check comp    append(A,B,C):(A=[1,2],B=[3],var(C))+ not_fails.
```

The success assertion states that if a call to `append/3` with the first and second arguments bound to `[1,2]` and `[3]` respectively and the third one unbound terminates with success, then the third argument should be bound to `[1,2,3]`. The comp assertion says that such a call will not fail. □

Example 4.4.3. Testing Multiple Solutions: Assume now that we want to check all possible solutions to a call to `append`/3 with the first two arguments uninstantiated. We can write the following assertion for this purpose:

```
:- test append(A,B,C) : (var(A),var(B),C=[1,2,3])
        => member((A, B), [([],[1,2,3]),
                           ([1], [2,3]),
                           ([1,2], [3]),
                           ([1,2,3],[])]) + not_fails.
```

Note that the postcondition property is applied to all the solutions generated by the predicate being tested.

□

The advantage of the integrated framework that we propose is that the execution expressed by a `texec` assertion for unit testing can also be used for checking parts of other assertions that could not have been checked at compile-time and thus remain as run-time checks. This way, a single set of run-time checking machinery is used for both run-time checks and unit testing. In addition, static checking of assertions can safely avoid (possibly parts of) unit test execution.

In order to simplify the process of writing tests we introduce another predicate assertion schema, the `test` schema, which can be seen as syntactic sugar for a set of predicate assertions, and has the form:

`:- test` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-Exec-Formula*].

This assertion is interpreted as the combination of three assertions,[‡] one assertion expressing *what to execute*:

`:- texec` *Pred* [`:` *Precond*] [`+` *Exec-Formula*].

and two assertions expressing *what to check*:

`:- check success` *Pred* [`:` *Precond*] [`=>` *Postcond*].

`:- check comp` *Pred* [`:` *Precond*] [`+`*Comp-Formula*].

For example, the assertion:

```
:- test append(A,B,C) : (A=[1,2],B=[3],var(C)) => C=[1,2,3]
                      + (not_fails,times(5)).
```

is conceptually equivalent to the assertion in Example 4.4.1, together with the two assertions in Example 4.4.2.
Regarding the atomic formulas appearing in *Exec-Formula* (*Exec-prop* in the grammar) the following are examples of several predefined useful properties:

`try_sols(N):` Expresses an upper-bound `N` on the number of solutions to be checked. For example, the assertion:

```
:- texec append(A, B, C): (A=X, B=Y, C=Z) + try_sols(7).
```

expresses that the call to `append(X, Y, Z)` should be executed to get at most the first 7 solutions through backtracking.

`times(N):` Expresses that the execution should be repeated `N` times. This increases the chances of test failure, for intermittent failures. For example, while checking ISO prolog compliance, a test for the `retract`/1 predicate failed rarely, so that the test was modified adding the primitive `times`/1:

[‡]In fact, a completeness assertion –using "`<=`", see [56]– could also be generated.

```
:- test retract_test7(A) + times(50).
retract_test7(A) :- retract((foo(A) :- A, call(A))).
```

in order to repeat the test fifty times increasing the chance of failure.

`exception(Excep)`: Expresses that a test execution should throw the exception `Excep`. For example, consider the predicate `p/1` defined as follows:

```
p(a).
p(b) :- fail.
p(c) :- throw(error(c, 'error c')).
```

The following tests succeed:

```
:- test p(A) : (A = a) + not_fails.
:- test p(A) : (A = b) + fails.
:- test p(A) : (A = c) + exception(error(c,_)).
```

The first one states that the call `p(a)` should not fail, the second one that `p(b)` should fail, and the third one that `p(c)` should raise an exception. However, the following test reports an error, i.e., fails:

```
:- test p(A) : (A = c) + not_fails.
```

`user_output(String)`: Expresses that a predicate should write the string `String` into the current output stream. For example, the following test involving the library predicate `display/ 1` succeeds:

```
:- test display(A) : (A = hello) + user_output("hello").
```

However, the following tests report an error:

```
:- test display(A) : (A = hello) + user_output("bye").
:- test display(A) : (A = hello) + user_output("hello!").
```

Other properties are provided for example to express that a predicate should write the string `Str` into the current error stream (`user_error(Str)`), to express a time-out `T` for a test execution (`resource(ub, time, T)`), or to generate random input data with a given probability distribution (e.g., for floating point numbers, including special cases like *infinite*, *not-a-number* or *zero* with sign).

4.5 Generating User-friendly Messages

Whenever a run-time check fails, an exception is raised. An exception handler will then catch the exception and report the error. However, with the transformations presented so far little information can be provided to the user beyond the precondition or postcondition that is producing the violation, since this is the only parameter passed to most of the checking predicates. Reporting simply that some condition failed is less informative than saying where it did, to what assertion it corresponds, or what was the last call mode of the predicate that violated it. In the case of a `comp` assertion the actual call could also be printed.

In contrast, during compile-time checking, when an assertion is proved not to hold, both the assertion and the program point where the assertion was violated are reported, in a format designed so that the graphical program development environment can locate these points in the source code and highlight them automatically.

In order to also provide precise information when reporting violated assertions when performing run-time checks, we have added an extra argument to the checking predicates through which certain information is passed, such as the location of the corresponding assertion(s) and the calling program point in the source code. This information can then be passed to the exception handler when the exception occurs, which prints it in a format that is compatible with that used when reporting compile-time checking errors. Thus, run-time errors can also be easily traced back to the sources automatically by the development environment. The transformation instruments the transformed code to include the necessary information.

On the other hand, while having rich information available, when a run-time check fails it is crucial to being able to locate bugs in programs, there is a clear trade-off between the size of the program and the overhead introduced in it and the quality of the messages issued. Different levels of information may be appropriate for different contexts. For example, programs can be compiled with a setting that implies lower overhead and, if an exception is

raised, the program can be recompiled with a higher level of instrumentation and rerun until the exception is raised again, this time obtaining more precise information for location of the error in the sources. Also, in systems that are resource constrained, such as many pervasive and embedded systems, lower levels of instrumentation would be appropriate and perhaps even load and use of the pretty printer library can be avoided, since the error messages can be interpreted in a different host.

The current implementation of the run-time check transformations offers several optional levels of instrumentation. For brevity we report on two levels in our experiments, explained below:

Low: information is saved to report the actual assertion being violated and the property or properties that caused such violation.

High: in addition, predicates with assertions are further instrumented so that when a run-time check fails a call stack dump is also shown up to the exact program point where the violation occurs, showing for each predicate the literal in its body that caused such violation.[§]

To illustrate these levels, consider the following assertion and property definitions, in addition to a definition of `qsort/2` such as that of Figure 4.6:

```
:- success qsort(A,B) => (ground(B),sorted_num_list(B)).
:- prop sorted_num_list/1.
sorted_num_list([]).
sorted_num_list([X]):- num(X).
sorted_num_list([X,Y|Z]):- num(X),num(Y),X=<Y,
                           sorted_num_list([Y|Z]).
```

which ensures that `qsort/2` always returns a ground, *sorted* list. Assume also that the program has been written in a buggy way (to be discovered later). With *low* instrumentation level the output during execution would be similar to:

[§]This can also be done at a lower level, via engine primitives, but we are interested herein in measuring only the cost of source level transformations.

```
?- qsort([1,2],X).
{In /tmp/qsort.pl
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([1,2],[2,1]).
In *success*, unsatisfied property:
       sorted_num_list([2,1]).
ERROR: (lns 16-21) Check failed in qsort:qsort/2.}
```

Two errors are reported for a single run-time check failure: the first error shows the actual assertion being violated and the second marks the first clause of the predicate which violates the assertion. However, not enough information is provided to be able to determine the literal in which the predicate was called causing the violation. If we perform instead the transformation with the *high* instrumentation level the output is:

```
?- call_rtc(qsort([3,1,2],B)).
{In /tmp/qsort.pl
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([1,2],[2,1]).
In *success*, unsatisfied property:
       sorted_num_list([2,1]).
ERROR: (lns 16-21) Check failed in qsort:qsort/2.
ERROR: (lns 16-21) Check failed when invocation of
       qsort:qsort([3,1,2],_1)
       called qsort:qsort([1,2],_2) in its body.}
{In /tmp/qsort.pl
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([3,1,2],[3,2,1]).
In *success*, unsatisfied property:
       sorted_num_list([3,2,1]).
ERROR: (lns 16-21) Check failed in qsort:qsort/2.}
```

In this example we have used the `call_rtc`/1 meta-predicate, which intercepts the run-time error, shows the related message, and lets execution continue as if the program where not being checked. With this new output it is easier to locate the error. Looking at the call stack dump, we can see the

```
:- module(qsort, [qsort/2], [assertions, nativeprops]).

:- calls   qsort(A,B) : list(A,num).
:- success qsort(A,B) : list(A,num) => list(B,num).
:- comp    qsort(A,B) : (list(A,num), var(B)) + not_fails.

qsort([X|L],R):- partition(L,X,L1,L2), qsort(L2,R2), qsort(L1,R1),
                 append(R2,[X|R1],R).
qsort([],[]).

:- calls   partition(A,B,C,D) : (list(A), num(B)).
:- success partition(A,B,C,D) : (list(A), num(B)) => (list(C), list(D)).
:- comp    partition(A,B,C,D) : (list(A), num(B)) + (not_fails,is_det).

partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right):- E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):- partition(R,C,Left,Right1).
```

Figure 4.6: A quick-sort program with assertions.

list of predicates being checked up to the call of the buggy code. Note that the first part of the assertion is not violated, since `B` is ground. However, on success the output of `qsort/2` is a sorted list but in reverse order, which gives us a hint: inspecting the code, we realize that the arguments in the call to `append/3` are mistakenly swapped. The fixed version of `qsort/2` follows:

```
qsort([X|L],R):- partition(L,X,L1,L2),qsort(L2,R2),qsort(L1,R1),
                 append(R1,[X|R2],R).
qsort([],[]).
```

The call stack dump was implemented by reusing the exception handling mechanism which is native in `Ciao`. Each time an exception is cached in a predicate with run-time checks enabled, a locator is added to the exception. This way, a more informative message of the form "Failed when ... called ..." can be generated. However, such exception handling mechanism was implemented using meta-calls, assert and retracts, causing a negative impact in the benchmarks that use it.

4.6 Experimental Results

We now report on some experimental results from our implementation within the `Ciao/CiaoPP` system of the testing and run-time checking approach pro-

Qsort	Low						High					
Obj Size:	Inline			Library			Inline			Library		
7467 bytes	M	T	MT	M	T	MT	M	T	MT	M	T	MT
Entry	1.41	1.69	1.77	1.34	1.38	1.44	1.66	1.94	2.02	1.57	1.61	1.68
Exit	1.55	1.82	1.97	1.28	1.33	1.44	1.78	2.06	2.21	1.50	1.55	1.65
Comp*	1.67	1.89	1.93	5.46	5.49	5.54	2.05	2.28	2.31	5.64	5.68	5.73
E/E/C	2.32	2.67	2.88	5.88	5.95	6.11	2.88	3.23	3.44	6.25	6.31	6.48
Calls	1.42	1.64	1.75	1.32	1.33	1.43	1.62	1.84	1.95	1.50	1.51	1.61
Success	1.55	1.77	1.92	1.26	1.29	1.39	1.74	1.97	2.12	1.42	1.44	1.55
Comp	1.63	1.85	1.88	5.38	5.41	5.46	2.01	2.24	2.28	5.57	5.60	5.65
C/S/C	2.10	2.46	2.65	5.66	5.73	5.88	2.63	3.00	3.20	5.98	6.11	6.26

Table 4.1: Size increment of `qsort` with several configurations of run-time checks.

Qsort	Low						High					
Ex. Time:	Inline			Library			Inline			Library		
675 μs	M	T	MT	M	T	MT	M	T	MT	M	T	MT
Entry	1.00	1.86	1.87	1.05	1.89	1.90	1.01	1.89	1.87	1.03	1.91	1.91
Exit	1.02	2.73	2.73	1.03	2.76	2.78	1.02	2.74	2.75	1.03	2.79	2.80
Comp*	1.01	1.87	1.87	1.02	1.93	1.92	1.02	1.88	1.90	1.05	1.91	1.92
E/E/C	1.01	3.60	3.60	1.04	3.67	3.68	1.02	3.62	3.65	1.05	3.69	3.69
Calls	3.52	165	162	76	243	321	42	207	205	135	301	382
Success	5.62	329	333	164	515	667	42	380	383	229	595	746
Comp	6.39	166	167	106	272	343	82	254	254	264	447	512
C/S/C	9.77	352	353	194	578	761	91	450	453	379	776	948

Table 4.2: Slowdown of `qsort/2` with several configurations of run-time checks.

posed. Both have been integrated fully into the development environment allowing easy execution of tests and run-time checking of assertions present

in modules. The system is available in the latest `Ciao` betas (1.13.x) at `http://www.ciaohome.org`. The experiments measure both program size and time overhead due to run-time checks. We first used the `qsort` program in Figure 4.6, with an input list of size 600 to run several experiments for different settings:

- **Library or inlined run-time checks:** We have implemented the transformation first as described in the previous sections, where the `check` predicates are assumed to be in a library. The results are provided in the columns labeled **Library**. The ratios shown are with respect to the execution time of the program with no run-time checks. In addition, an alternative approach has been implemented in which the definitions of the run-time check library predicates are actually *inlined* in the calling program. Such inlining was implemented using a simple `Ciao` **package** [15] that transforms any call to the library in to an inlined version of such call. Whether to perform this inlining is a user option, so that it is possible to choose between both alternatives. This inlining often achieves better performance and, although intuitively one could expect that it increases the code size, there are cases in which the code is reduced because such inlining is, in fact, a restricted kind of partial evaluation that tries to solve as many unifications as possible at compilation time, and eventually, terms become smaller after such optimization.

- **Use of types or modes properties:** since checking complex types, such as in the `list(int)` check, which needs to traverse lists of integers over and over again,[¶] is more expensive than checking modes (which in our case is handled through a call to the `var/1` ISO Prolog builtin) we have separated these cases in the experiments. In columns labeled

[¶]This overhead can be significantly reduced via multiple specialization [58, 57]. However, that optimization has not been applied in this case in order to measure the overhead of fully checking the assertion.

T and M only types or modes are checked respectively, whereas in columns labeled MT both types and modes are checked.

- **Low or high instrumentation:** as defined in Section 4.5.
- **Using several kinds of assertions:** several combinations of different kinds of assertions have been tested (first column).

Tables 4.1 and 4.2 present the overhead, in size and time respectively, for the experiments expressed as a ratio w.r.t. the execution of the program with run-time checks disabled. Execution was on a MacBook Pro, Intel Core 2 Duo at 2.4Ghz, 2GB of RAM, Ubuntu Linux 8.10 and `Ciao` version 1.13. The columns in the tables present combinations of the configurations explained above. The rows show results for different kinds of assertions. For `comp` assertions we have that in **Comp*** the check is performed only at the entry point of the module, but not for the internal calls that occur inside.

The results show that the *high* level of instrumentation is quite expensive while the overhead implied by the *low* level is better, specially in the case of inlining. This confirms our expectations. The high overhead implied by the *high* level of instrumentation is due in part to the simplistic way in which this type of instrumentation is implemented for these experiments. Note also that the values of the `Lib.` column (library) are quite large when compared with the ones of the `Inl.` column (inline) because the inline transformation avoids metacalls.

Table 4.3 shows experimental results for larger programs, namely, the `Ciao`, `CiaoPP`, and LPdoc systems (including the libraries they use), all of which contain numerous assertions in their code. It shows the size (in kilobytes) of binary and object files using several instrumentation levels of run-time checks. The binary refers to the statically-linked executable of the main module of such systems which corresponds to the command-line executable. The object files include all the libraries used by such systems. Note that in all cases the sizes of the files depend on the number of assertions instrumented for run-time checking. Interestingly, the impact of run-time checks

App Name	Source Metrics				Compiled		Run-Time Checked (ratio)			
	Size		Assertions		Binary		Low		High	
	Lines		Modules		Object		Inl.	Lib.	Inl.	Lib.
`Ciao`	S	4340	A	3230	B	2965	1.22	1.20	1.26	1.23
	L	131392	M	634	O	16310	1.20	1.18	1.23	1.21
`CiaoPP`	S	4831	A	1199	B	13026	1.20	1.19	1.22	1.21
	L	152365	M	517	O	14562	1.19	1.18	1.21	1.20
LPdoc	S	438	A	153	B	1929	1.09	1.06	1.13	1.07
	L	12750	M	21	O	1167	1.12	1.07	1.14	1.08

Table 4.3: Size (in kilobytes) of binary and object files using several instrumentation levels of run-time checks, for large benchmarks.

on execution time in these much larger benchmarks is much smaller than for qsort. For example, the overhead introduced in the execution of LPdoc, which includes a good number of assertions in its source, is in practice below the measurement noise level.

Regarding unit tests, in order to facilitate the execution of tests, the unit testing framework has been integrated in the development environment allowing easy execution of tests present in modules. The execution of the tests is done as follows:

1. The user selects the module or the directory that contains the modules with tests to be executed.

2. The assertions are read and each time a test is found, a method is added to the main procedure of an auto generated program that invoques such method. The goal of such method is to call the predicate being tested in the way specified by the unit test commands.

3. The modules being tested are compiled with run-time checking enabled.

4. The main procedure that invoques the tests is called by the unit test driver in a separate process, to prevent undesirable side effects or fail-

ures if the program being checked aborts due to an unexpected error. This program writes a log file containing the results of the execution (such as, for example, exit or failure of the predicate, unhandled exceptions and so on), that is further analyzed by the unit test driver in order to take actions depending on the observed behavior.

5. If a test causes the failure of the main program, the control is returned to the driver, and the aborted test is recorded to be processed. After that, the driver (optionally) tries to execute the remaining tests. This process continues until all the tests are executed.

6. The generated log file is processed by the driver and, depending on the verbosity level, different information about the execution is presented, such as for example, the tests passed, failed, aborted and in each one the cause of such behavior. At this point, the run-time check exceptions saved in the log file are processed in order to show the related message.

We have added at the time of writing 220 unit tests to the `Ciao/CiaoPP` system (in addition to the other traditional system tests which did not use the unit test framework). These tests have been effective in detecting some errors introduced in those modules during later code changes. The execution time of such tests is approximately 90 seconds in the computer described before.

As we can see in Table 4.4, we also have applied the implemented framework to the verification of ISO Prolog compliance of `Ciao`. We have coded 976 unit tests for this purpose. These allowed the detection of a large number of previously unknown limitations and errors: 262 issues related to non-compliance with the standard, 90 related to missing predicates or functionality, and 39 related to bugs in the functionality. While a large number of these were repetitions of a few individual errors they have been nevertheless very useful. In fact, thanks to the collaboration of other `Ciao/CiaoPP` developers, only 84 (8.2%) of such tests fail at moment. These tests currently run in under 15 seconds. This time is much less than the other tests for `Ciao` because they are concentrated in only one file and the driver does not

Tests failed		
i	Incompatible format of syntax error exception.	10
i	Incompatible format of type error exception.	9
i	Incompatible format of permission error exception.	28
i	Incompatible format of domain error exception.	2
i	An error is expected, but ciao just fails.	138
i	Ciao throws an error different than the one specified in the standard.	15
i	The predicate in Ciao Fails, but in ISO, it should succeed.	22
i	The execution of a predicate should raise an error, but it succeeds.	19
m	Predicates with missing functionality.	24
i	`Ciao` adds more information to a predicate (module expansion).	6
i	More solutions than expected.	1
f	Stream manipulation related errors.	14
f	Unexpected abort of the test being executed.	14
i	Non-ascii characters (not ISO, but SICSTUS-EDDBALI-like behavior).	7
f	Aborted tests.	1
m	Tests changed because currently several errors can not be handled	7
m	Stream options unimplemented.	2
m	Alias for streams unimplemented.	32
m	Stream option eof_action unimplemented.	12
m	Stream option past_end_of_stream unimplemented.	2
m	Unimplemented options for close.	5
f	Character handling related errors.	2
i	Malformed body (negation of cut).	1
f	Current output related.	1
i	Predicate that succeeds.	1
m	Failed test because time_out(_) property is not implemented.	1
f	Tests with side effects.	7
i	Arity mismatch issues.	3
m	Not relevant tests in `Ciao`, due to unimplemented arithmetic behavior.	5
i	Incompatibilities.	262
m	Missing predicates or functionality.	90
f	Failures and errors.	39
	Total number of failed tests.	391
	Total number of executed tests.	976
	Percentage of passed tests.	60 %

Table 4.4: Summary of the first application of unit tests for ISO Prolog compliance.

need to scan all the source code. Note that in these experiments we are not doing any compile-time checking, which would in fact eliminate many of the unit tests.

4.7 Chapter Conclusions

We have described our design and implementation of a framework that unifies unit testing and run-time verification (as well as static verification and static debugging). A key contribution of our approach is that a unified assertion language is used for all of these tasks. This has allowed us to propose and implement unit testing via a minimal addition to the assertion language. We have proposed methods for compiling run-time checks for (parts of) assertions which cannot be verified at compile-time via program transformation. This transformation allows checking preconditions and postconditions, including conditional postconditions, properties at arbitrary program points, and certain computational properties.

We have also proposed a minimal addition to the assertion language which allows defining unit tests to be run in order to detect possible violations of the (partial) specifications expressed by the assertions.

We have implemented the framework within the `Ciao/CiaoPP` system and effectively applied it to the verification of ISO Prolog compliance and to the detection of different types of bugs in the `Ciao` system source code. Several experimental results have been presented to illustrate different trade-offs among program size, running time, or levels of verbosity of the messages shown to the user. The experimental results confirm our expectations regarding these trade-offs: run-time checks do not pose an excessive amount of overhead when low levels of instrumentation are introduced and the calls to library predicates are inlined, except with high levels of instrumentation (e.g., gathering information on the call stack). However, this is due to the simplistic way in which this type of instrumentation is implemented, which can be optimized using lower-level primitives. For example, it prevents the compiler from performing some classical optimizations like tail recursion.

The tests and run-time checks are proving quite useful in practice for detecting bugs. However, we have identified some improvements that we plan to perform as future work, as for example, to further extend the assertion language with more primitives such as `time_out(T)`, which can be used to

express that a test should finish in less than `T` milliseconds, `user_error(Str)` which expresses that a predicate should write the string `Str` into the current error stream, or to add more properties for generating random input data values with a given probability distribution.

We also plan to study how the multiple specialization present in `CiaoPP` can further reduce run-time overhead. Finally, we are also working on an improved and more compositional strategy to define computational properties.

Chapter 5

Conclusions and Future Work

We have developed a general framework for automatically inferring both upper- and lower-bounds on the usage that a logic program makes of resources in general. Such bounds are given as functions of input data sizes. Our approach gives support for platform-independent (or *user-defined/application-dependent*) resources, as well as platform-dependent resources.

The framework includes a global analysis which is parametric with respect to resources and type of approximation (lower- and upper-bounds). The user can define the parameters of the analysis for a particular resource by means of assertions.

We have applied the general framework to execution time estimation following two different approaches. The first approach performs the analysis based in the information available at source-code level, while the second one takes advantage of the information available at bytecode level and in the abstract machine. We have experimented with resource usage information supplied at source and bytecode levels. The experimental results were encouraging in both cases, obtaining execution time estimates with different levels of accuracy/efficiency useful for a wide range of applications.

Since not all resource-related properties can be verified statically, and that one of the most promising applications is the automatic verification of such properties, we have developed a framework that unifies static verification,

run-time checking and unit testing. In addition to those resource-related properties, we can process other properties like non-failure, determinism and state (or functional) properties like types of input/output arguments on calls or successes. The experimental evaluation of the framework is encouraging, in particular it has been effectively used for the verification of ISO Prolog compliance and in the detection of different types of bugs in the `Ciao` system source code.

A key contribution of this work is that we preserve the use of a unified assertion language for all tasks. Such language is used to define resources and resource-related properties that can be verified based on the results of the analysis and is powerful, general and extensible enough to express a large class of interesting properties.

All the developed methods and techniques, including the general resource usage analysis, its particularization to execution time estimation, and the unified framework for run-time checking, static verification and unit-testing have been implemented and integrated in the `Ciao`/`CiaoPP` system.

Applications of the work presented in this thesis include resource usage verification, performance debugging, certification of resource usage properties in mobile code, resource and granularity control in parallel/distributed computing and resource-oriented specialization. All progress in such applications represents, of course, an interesting source of future work.

Regarding run-time checking, an interesting source of improvement is the usage of multiple specialization in order to reduce the run-time overhead introduced by the instrumentation of the program when enabling run-time checks.

Although this thesis has been presented in the context of logic programs, we believe that almost all the techniques developed in it can easily be adapted to other programming languages. This adaptation, while certainly not trivial, to some extent would actually imply some simplification, since for example, backtracking does not need to be taken into account.

Bibliography

[1] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

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

[3] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resource verification. In *TPHOLs2004*, volume 3223 of *LNCS*, pages 34–49, Heidelberg, September 2004. Springer Verlag.

[4] R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.

[5] D. Basin and H. Ganzinger. Complexity Analysis based on Ordered Resolution. In *11th. IEEE Symposium on Logic in Computer Science*, 1996.

[6] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[7] I. Bate, G. Bernat, and P. Puschner. Java Virtual-Machine Support for Portable Worst-Case Execution-Time Analysis. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, Washington, DC, USA*, Apr. 2002.

[8] F. Belli and O. Jack. Implementation-based Analysis and Testing of Prolog Programs. In *ISSTA '93: Proc. of the ACM SIGSOFT Int'l. Symp. on Software Testing and Analysis*, pages 70–80, New York, NY, USA, 1993. ACM.

[9] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science*, 318(1-2), 2004.

[10] B. Brassel, M. Hanus, F. Huch, J. Silva, and G. Vidal. Run-Time Profiling of Functional Logic Programs. In *Proc. of the Int'l. Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 182–197. Springer LNCS 3573, 2005.

[11] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla-(Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, School of Computer Science, T.U. of Madrid (UPM), 2009. Available at `http://www.ciaohome.org`.

[12] 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 WS on Automated Debugging–AADEBUG*, pages 155–170. U. Linköping Press, May 1997.

[13] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[14] S. Buettcher. Warren's Abstract Machine - A Java Implementation. `http://www.stefan.buettcher.org/cs/wam/index.html`.

[15] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[16] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming (ESOP)*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.

[17] The CLIP Group. Program Assertions. The Ciao System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.

[18] S-J. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'05)*. ACM Press, 2005.

[19] S.J. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.

[20] S. K. Debray. Profiling Prolog Programs. *Software Practice and Experience*, 18(9):821–839, 1983.

[21] 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.

[22] 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.

[23] 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.

[24] S. Diehl, P. Hartel, and P. Sestoft. Abstract Machines for Programming Language Implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[25] Mireille Ducassé. Opium: An extendable trace analyzer for prolog. *J. Log. Program.*, 39(1-3):177–223, 1999.

[26] N. S. Eickelmann and D. J. Richardson. An Evaluation of Software Test Environment Architectures. In *ICSE '96: Proc. of the Int'l. Conf. on Software Engineering*, pages 353–364. IEEE Computer Society, 1996.

[27] Jochen Eisinger, Ilia Polian, Bernd Becker, Alexander Metzner, Stephan Thesing, and Reinhard Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proceedings of IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 15–20. IEEE Computer Society, April 2006.

[28] A. Ermedahl, J. Gustafsson, and B. Lisper. Experiences from Industrial WCET Analysis Case Studies. In Reinhard Wilhelm, editor, *Proc. Fifth International Workshop on Worst-Case Execution Time (WCET) Analysis*, Palma de Mallorca, July 2005.

[29] G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 75–88. ACM Press, 2002.

[30] M. Gómez-Zamalloa, E. Albert, and G. Puebla. On the Generation of Test Data for Prolog by Partial Evaluation. In *Workshop on Logic-based*

methods in Programming Environments (WLPE'08), pages 26–43, 2008. Report number: WLPE/2008/06.

[31] Bernd Grobauer. Cost recurrences for DML programs. In *International Conference on Functional Programming*, pages 253–264, 2001.

[32] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.

[33] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, 1999.

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

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

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

[37] Alston S. Householder. Unitary Triangularization of a Nonsymmetric Matrix. *Journal ACM*, 5(4):339–342, October 1958. DOI:10.1145/320941.320947.

[38] E. Yu-Shing Hu, A. J. Wellings, and G. Bernat. Deriving Java Virtual Machine Timing Models for Portable Worst-Case Execution Time Analysis. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, volume 2889 of *LNCS*, pages 411–424. Springer, October 2003.

[39] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Symposium on Principles of Programming Languages*, pages 331–342, 2002.

[40] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), 1988.

[41] P. López-García. *Non-failure Analysis and Granularity Control in Parallel Execution of Logic Programs*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 2000.

[42] P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[43] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21:715–734, 1996.

[44] David A. McAllester. On the complexity analysis of static analyses. In *Static Analysis Symposium*, pages 312–329, 1999.

[45] 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.

[46] E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th International Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.

[47] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Towards Execution Time Estimation for Logic Programs via Static Analysis and Profiling. In S. Mu noz and W. Vanhoof, editors, *16th Workshop on Logic Programming Environments*, pages 45–60. University of Namur, Institut d'Informatique, August 2006.

[48] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Using Combined Static Analysis and Profiling for Logic Program Execution Time Estimation. In *22nd International Conference on Logic Programming (ICLP'06)*, number 4079 in LNCS, pages 431–432. Springer-Verlag, August 2006.

[49] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages (PADL'07)*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.

[50] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[51] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd In-*

ternational Conference on Logic Programming (ICLP'07), volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.

[52] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Automatic complexity analysis. In *European Symposium on Programming*, pages 243–261, 2002.

[53] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *ILPS'97 WS on Tools and Environments for (C)LP*, October 1997. `ftp://clip.dia.fi.upm.es-/pub/papers/assert_lang_tr_disciplideliv.ps.gz`.

[54] G. Puebla, F. Bueno, and M. Hermenegildo. A Framework for Assertion-based Debugging in Constraint Logic Programming. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, Venezia, Italy, September 1999.

[55] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[56] 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.

[57] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

[58] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *JLP*, 41(2&3):279–316, November 1999.

[59] S. A. Jarvis R. G. Morgan. Profiling large-scale lazy functional programs. *Journal of Functional Programing*, 8(3):201–237, May 1998.

[60] G. Román-Díez and G. Puebla. Java Bytecode Timing Cost Models. Technical Report CLIP12/2007.0, Technical University of Madrid, School of Computer Science, UPM, December 2007.

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

[62] Patrick M. Sansom and Simon L. Peyton Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, March 1997.

[63] Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In *Perspectives Workshop: Design of Systems with Predictable Behaviour, 16.-19. November 2003*, volume 03471 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2004.

[64] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, September 2003.

[65] D. Wackerly, W. Mendenhall, and R. Scheaffer. *Mathematical Statistics With Applications 5th Edition*. P W S Publishers, 1995.

[66] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[67] J. Wielemaker. SWI Prolog Unit Tests. `http://www.swi-prolog.org/pldoc/package/plunit.html`.

[68] Reinhard Wilhelm. Timing Analysis and Timing Predictability. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium (FMCO)*, volume 3657 of *LNCS, Revised Lectures*, pages 317–323. Springer, 2004.

[69] L. Zhao, T. Gu, J. Qian, and G. Cai. Test Frame Updating in CPM Testing of Prolog Programs. *Software Quality Control*, 16(2):277–298, 2008.