ENTRA

318337

Whole-Systems ENergy TRAnsparency

Common Assertion Language

Deliverable number:	D2.1
Work package:	Energy Modelling Through the System Layers (WP2)
Delivery date:	1 October 2013 (12 months)
Actual date:	13 November 2013
Nature:	Report
Dissemination level:	PU
Lead beneficiary:	University of Bristol
Partners contributed:	Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited

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

Short description:
Software developers can benefit from the ability to express non functional specifications and information such as resource consumption in their programs. The ENTRA common assertion language seeks to address this need. We introduce an internal assertion language used by resource analysis frameworks that is not tied to particular software development languages. This can be used to express complex specifications including energy consumption functions which depend on data properties, such as data size, other environmental properties such as clock frequency and voltage and inter-module contracts. We also introduce a front end to express parts of these specifications in the source code. We do this by extending an existing embedded software language, which allows the end user to express these in their own source code. We describe these two parts of the Common Assertion Language, together with a mechanism that propagates information from the front end, through the various system layers and all the way to the resource analysis framework.

Contents

Date	Partner	Comment
	IMDEA	Created document template.
10.04.2013	Bristol	Introduced some requirements and related work.
12.05.2013	Bristol	Document fleshed out following plenary meeting in Madrid.
20.05.2013	IMDEA	More related work introduced. Ciao Common Assertion Language introduced.
20.06.2013	Bristol	Document reorganised
30.07.2013	Bristol	First draft finalised and passed onto other partners.
16.09.2013	Bristol	Draft updated following Roskilde's suggestions.
10.09.2013	XMOS	Contributed the description of the language front end.
23.09.2013	Bristol	Document updated following consensus at plenary meeting in Roskilde.
17.10.2013	IMDEA	Section 4 updated following consensus at plenary meeting in Roskilde.
12.11.2013	ROSKILDE	Added scenarios and modified requirements.

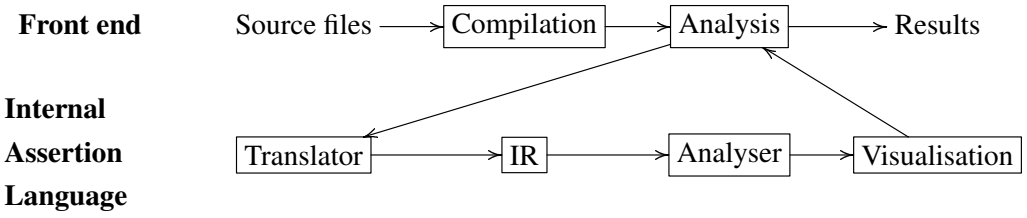

Figure 1: ENTRA toolchain, and the relationship of the front end to the internal assertion language.

1 Introduction

The Common Assertion Language plays an essential role in the ENTRA project, as it is one of the main elements that makes energy transparency possible. It allows the expression of resource consumption such as energy, together with other useful information at different levels of abstraction, and to propagate such information up and down the levels. It is the language in which the user communicates with the energy aware tools developed in the project, but also the language that allows interoperability among different analysis, verification and optimisation tools in a seamless and unified way.

The notion of the *Common Assertion Language* needs to be clarified at an early stage. In the ENTRA project, this assertion language is made from a front end and an internal assertion language for the resource analysis framework. In fact the Common Assertion Language is pervasive across the source code language and the intermediate representations used by the resource analysis framework (D3.1). The role of the front end and the internal assertion language can be seen in Figure 1. Here we can see a block diagram of the ENTRA toolchain. At the top, we have the programmer's toolchain, including the compiler and user-facing analysers. Here, we expect that the user will only interact with the front end of the Common Assertion Language. The internal assertion language, on the bottom, allows the expression of assertions in the intermediate representation language of the resource analysis framework, and is an integral part of this framework. For the internal assertion language, we first focus on the definition of such elements conceptually, ignoring the syntax as much as possible. Since this language is internal to the resource analysis framework, the embedded software developer does not express properties directly in this. We define the assertion language in Section 4, where we adopt a domain specific language based initially on Ciao, instantiated and extended to support the requirements of ENTRA. Moreover, we want to produce a language that can be easily instantiated to deal with different programming languages and systems. Hence, the assertion language is not language- or platform-specific. The end users of the front end and the internal assertion language have different goals. The person using the internal assertion language is the actual resource analysis toolchain developer. The front end is however used by the embedded systems developer. We therefore expect that the expressiveness in the internal assertion language to be more than that of the front end. For instance, instruction level energy models are expressed using the internal assertion language rather than using the front end.

Prior to defining the elements of our deliverable we have also performed an extensive study of languages and language extensions used to express functional and non-functional properties. We adopt useful features and identify necessary extensions to the state-of-the-art (see Section 3). Since the internal assertion language will be used to "glue" the various components of the resource analysis framework (D3.1) together, a starting point of this language is defined as early as possible and is one of the first deliverables of the project. For this purpose, we make use of the Ciao assertion language since it meets some of the requirements and can be extended to meet others. In this document we will not fully specify this language, and the syntax we use is for presentation purposes.

Another part of the Common Assertion Language is the front end, which is user-facing. Referring back to Figure 1, the front end is used to annotate the existing code with useful hints and information. This information is subsequently propagated to the underlying layers in the programming toolchain such as intermediate compiler representations (LLVM IR) or assembly code. This part of the language is defined in Section 5, where we also discuss any changes or extensions to the source code language. In our case, this happens to be the XC [Wat09] language, a language designed for programming multicore systems. Hence, we can say that the front end language exposes an interface to a subset of the internal assertion language of the resource analysis framework.

2 Requirements

In order to design the various parts of the Common Assertion Language, we have first performed a study of requirements. Since we have two parts of the common assertion language, we divide the requirements into two subsections. The following subsection describes the requirements of the internal assertion language, used in the resource analysis framework. Section 2.2, on the other hand, describes the requirements from an embedded software developer point of view, including the required extensions to the source language and the mapping of information expressed in this language through the respective toolchains. Therefore, Sections 2.1 and 2.2 correspond to Sections 4 and 5 respectively.

2.1 Internal Assertion Language requirements

Ciao provides a generic and extensible assertion language, which can be extended and instantiated with energy specific features. The internal assertion language "glues" together the various parts of the toolchain, such as the energy models, program metadata, LLVM IR (Low Level Virtual Machine Intermediate Representation language) to ISA (Instruction Set Architecture) mappings, and so on. In this section we assume that "the user" refers to the user/developer of the resource analysis framework. We have identified the following requirements for the internal assertion language:

- Contains a base language of relations and functions over standard domains including arithmetic and boolean values, and program datatypes such as arrays and strings. Value ranges can be given probability distributions and accuracy specifications.
- Allows the expression of properties of computation states of the program, such as constraints on the values of input data and invariants at specific program points. Such functional properties are often vital in order to infer non-functional resource-related information.
- Contains special constructs for the expression of non-functional resource-related specifications (including energy, power and timing specifications) at various levels of abstraction, such as source language or any intermediate representation. Such specifications should be expressible as functions on various parameters of the input data and environment, including for example input data size, range, clock frequency, voltage, number of running threads, and so on.
- Allow the description of models of underlying hardware (environment). The hardware is abstracted as a set of relevant properties (e.g., clock frequency, voltage, contents of registers, etc.), that we call environment properties. A set of concrete values for such properties is what we call a "hardware state" or environment. The change in the environment by the execution of instructions can also be expressed using the assertion language.
- Allows assertions to be marked as "trusted", meaning that such assertions are assumed to be true by the verification and analysis tools. Such trusted assertions include energy models at different abstraction levels, and energy consumption assertions about parts of a system which

are not developed yet or for which the analysis infers inaccurate information, so that it can be propagated by the analysis tools to estimate their impact in the energy consumption of the whole system. Adding trusted information also helps with the scalability of the resource analysis.

- Allows assertions to be automatically verified or inferred by the analysis tools, and therefore marked as `checked` or `true`/`false` respectively.

- Allows information relating internal or intermediate program representations to source code program points, thus permitting analysis results to be relayed as automatic program documentation generation or visualisation of resource usage.

- Is flexible and extensible to express new properties and resources and not tied to a particular programming language or hardware platform.

In the rest of this section, we give more detailed information for the requirements.

2.1.1 Elements of the assertions

Semantically, assertions should express properties of programs and should talk about preconditions, postconditions, whole executions, program points, etc. At the core of an assertion is a base language of relations and functions for expressing arithmetic and boolean properties. On top of this, assertions should provide linguistic constructs, which include the following elements:

- *Status*: Describes whether the assertion is provided by the programmer and is to be checked (i.e., as a specification) by the resource analysis framework, i.e., `check`. The resource analysis framework generates another assertion stating whether the provided assertion is correct (`checked`) or `false`. Assertions should also specify information to be trusted, i.e., `trust`.

- *Scope*: Specifies the scope of application of the assertion, e.g., whether the assertion refers to a function, to a ISA instruction, to a piece of code marked with delimiters, etc.

- *Precondition*: Specifies the conditions under which the assertion is applicable. The *Precondition* should involve program state properties as well as environment properties.

- *Postcondition*: Specifies the conditions that hold after the successful execution of the element described in the *Scope*, provided the *Precondition* holds. This field involves the same peoperties as the *Precondition*.

- *Computational properties*: Describes the properties of the whole computation of a *Scope* that meets a *Precondition*. This field involves non-functional global properties, such as energy consumption, execution time, or resource usage in general.

2.1.2 Energy models and annotations

As part of Work Package 2, we will deliver energy models for assembly level instructions and also higher levels including intermediate representations such as LLVM IR and source language such as XC. The assertion language should not be tied to a specific platform and should be flexible and expressive enough to describe energy models at all levels, using a common syntax and toolchain. This would include but not limited to upper and lower bounds for ISA instructions (possibly for different platforms), LLVM IR instructions or XC programming language terms. Another useful metric to include with the energy values is the variance or confidence level of the energy values. This should come in useful especially for energy models that are extrapolated from others.

Section 4.2.4 includes examples of how the energy model should be expressed at various levels, and how a function can be annotated with energy information.

2.2 Front end requirements

It is possible to extend C and XC such that extra information is added to the source code of a program. This information must not affect the semantics of the executing program. Rather, this would be used by the programming tools to check certain properties such as obeying non-functional requirements.

2.2.1 Program analysis

Although we expect the number of manual annotations to be minimal, a number of hints can be given in advance by the programmer by annotating the source code. These allow the programmer to express a limit for *iteration bounds* for loops and recursion. This is important in cases were such a limit is not declared or cannot be inferred. Otherwise a number of measurements such as worst case execution time and energy consumption become unbounded. The likelihood of taking a particular branch over another can also be expressed, and this helps the analysis to produce tighter results. Another important thing that needs to be expressed is probability distributions of integer variables and arrival time of events.

2.2.2 Generic language requirements

In ENTRA, we want to augment the source language such that trusted information can be added. We would also want to check that certain information holds. In order to do so, we need two different assertion modes, `check` and `trust`, which correspond to the same modes in the Internal Assertion Language. We do not however require all the modes supported by the Internal Assertion Language. As in the Internal Assertion Language, expressions support relations and functions for expressing arithmetic and boolean properties.

The front end language should be expressive enough to make assertions about functional and non-functional properties. Therefore, assertions can be made about the values stored inside program

variables, input values, etc., but also about energy consumption and timing. For the latter, we need specific linguistic constructs, such as counters, which are explained in the next section.

2.2.3 Energy related extensions

We propose extensions to the source language (XC), that cater for the following requirements:

Power: The embedded programs we shall be dealing with in ENTRA are typically non-terminating and therefore the most relevant property we can specify is the target power dissipated by the micro-controller. The power budget therefore applies for the whole program and is specified in watts.

Accuracy: Throughout the ENTRA project, we might also investigate extensions to the type system such that the accuracy of operations [SDF$^+$11] can be specified. This would entail adding information to the existing types. For example, a type in a variable declaration statement could be annotated, indicating that the accuracy of the value stored in the variable is unimportant.

Bounds: For the purposes of verification, bounds can be given to individual variables and therefore these need to be annotated with this information.

Counters: Through the use of the front end, it is required to specify assertions about power consumption and timing in various parts of the system. The addition of global energy and time counters to the language makes it possible to use these values within our assertions. These constructs are erased at runtime.

2.2.4 Information flow through the toolchains

The information conveyed through the front end language has to be retained and mapped from source code to intermediate compiler representations and ISA level. This information also has to be translated to the assertion language for the resource analysis framework. In order to accomplish this, we need to establish which elements in a language can be propagated to the language in the intermediate compiler representation. We also want the source language extension to be least invasive as possible. Therefore, any annotations and extensions to the source language need to be separate from the language and have their own namespace.

3 Background

In this section we review related work, which has influenced the design decisions of the Common Assertion Language. We start by describing design by contract [Mey92]. This work has influenced other languages such as Ciao [HBC+12]. We also investigate languages to describe and verify realtime and concurrent systems. This is especially relevant since XC is used for these applications. Even though the deliverable here is about software, it is also important to investigate languages used to describe hardware properties, for example power modes. Finally we summarise how these languages and extensions are integrated within other systems.

3.1 Design by Contract

Design by contract is the addition of formal specifications describing preconditions, postconditions and invariants of blocks of executable code (collectively known as contracts) to improve both safety and documentation. This idea was initially implemented in the Eiffel language by Bertrand Meyer [Mey92] and later adopted in many other languages.

Traditionally, contracts have focused on functional properties of the execution, specially properties of the data that is given as input or returned as output. When moving towards more expressive languages, contracts have been enhanced for expressing specifications about the functions themselves (what we call higher-order contracts) and about non-functional properties such as execution time or energy consumed.

3.1.1 Eiffel-style Contracts

In this section we describe the kind of contracts found in imperative and object-oriented languages with no support for higher-order functions.

Eiffel is an object-oriented language with automatic management of memory through garbage collection. The syntax resembles that of Pascal. It offers support for multiple inheritance and method overriding in child classes. Error handling is based on the exception mechanism.

In Eiffel, the main unit of organisation is the *class*. Each class may have several *features* (corresponding to both fields and methods in other object-oriented languages). For each feature, a level of visibility can be specified, expressing whether other classes can access it or not. For example, the following Eiffel code declares two classes: A class named ANIMAL with a field named “colour” and a method named “speak,” and another class named “dog” inheriting from it, that overrides the method “speak.”

```
class ANIMAL
feature
  color : INTEGER
  speak is do
      print("??%N")
    end
end
```

```
class DOG
inherit ANIMAL
    redefine speak end
feature
  speak is do
      print("Woof!")
    end
end
```

Each contract block is made of one of more assertions, which have the form `name: boolean condition`. The boolean condition may include function calls, what makes possible to abstract common patterns in contract checking for libraries.

The fact that any function can be used in assertions implies that:

- Libraries of reusable contract definitions can be created.
- Functions that encode universal and existential quantification for specific data structures can be used but in a limited way, as Eiffel does not allows passing functions as parameters).
- Static analysis gets much more difficult, as there is no limitation on the properties being defined.

For features, blocks defining preconditions and postconditions can be added. The following example defines a feature that computes the square root of a real number n, to which a contract is added with the precondition that n should be non-negative, and a postcondition expressing a tolerated margin of error in the computation of the square root of n:

```
feature
  square_root (n : REAL) : REAL is
    require:
      non_negative: n >= 0
    do
      -- implementation omitted
    ensure
      result_is_ok: abs(n - result*result) <= 0.0001
end
```

Apart from feature assertions, we can include class-wide and loop invariants and conditions in any other point of the program using a `check` block.

Spec# [RLM08] is an extension of the C# language with support for contracts. The essence of its assertion framework is very similar to that found in Eiffel, as both are object-oriented languages. However, Spec# has some unique features that are interesting to discuss.

Sometimes invariants or conditions may refer to private fields. This is discouraged (and in some cases even banned) in Spec#. Instead, the developer should use *model fields*. Those fields do not exist in the implementation itself, but allow to model information that can be useful for stating the assertions. The declaration of model fields has this form:

```
model Type Field {
  satisfies Assertion;
}
```

It can be used later in any assertion as any other field (but cannot be used in the implementation). One extra feature of Spec# is the addition of `forall` and `exists` for assertions. They usually allow to include not efficient specifications of methods. For example, the postcondition for searching an element into a specific data structure could take the form:

```
bool has(K item)
  ensures result = exists{ K i in this.values; item == i };
{ /* implementation */ }
```

A method in Spec# can specify exactly which members of a receiving object are going to be modified during the execution of a method. This is done using a `modifies` specifier after the other contracts. An example of its use can be found in a Spec# Tutorial:

```
public void Swap(int[] a, int i, int j)
  requires 0 <= i && i < a.Length;
  requires 0 <= j && j < a.Length;
  modifies a[i], a[j];
  ensures a[i] == old(a[j]) && a[j] == old(a[i]);
{ ... }
```

If the method is known to have no side-effects (no modifications but neither printing on the screen, saving to disk...), it can be marked using the `[Pure]` attribute (attributes are to C# as annotations to Java). Searching an element in an array is an example of a pure method:

```
[Pure]
public int Find(int[] a, int element)
  ensures -1 <= result && result < a.Length;
{ ... }
```

As with the other contracts, the static verifier will try to prove that the method is indeed pure. Finer-grained levels of non-purity can be specified with other attributes.

Usually classes are not completely closed, but use information and code for other classes, aggregating them. So it may well be that contract of a class refer to members of its component classes. To represent this fact, in the declaration of a component in the aggregate class, a `[Rep]` attribute must be added. For example:

```
public class Car
{
  String name;
  [Rep] Motor motor;
  ...
}
```

In this way, when any operation changes the `motor` being part of a `Car`, contracts found in the `Car` class referring to it must also be checked.

Apart from basic ownership, Spec# also supports the notion of *peers*, which are sets of objects with a common parent. For example, all elements in a linked list can be thought as members of the list object. This creates a stronger notion of consistency of objects called *peer consistency*.

3.1.2 Higher-order Contracts

Racket (previously known as PLT Scheme) is among the best-known functional programming languages. This language is interesting to consider because it provides a way to specify assertions over functions that will be passed as parameters, something not found in previously discussed assertion frameworks.

In this section the new-style contract syntax for Racket, introduced in version 5.2, is used. Essentially, the contract declarations take the form:

```
(provide
  (contract-out [some_definition its_contract]
                [other_definition another_contract]
                ...
  ))
```

In Racket, functions are also values, so there is no need to have special syntax for them. Moreover, any boolean predicate can be used as a condition over the value. This is an example of a contract declaring that the "amount" variable should always hold a positive value:

```
(provide (contract-out [amount positive?]))
```

For functions, the contracts can be written as `(->` *pre post*`)` or `(`*pre* `.->.` *post*`)` . This can be combined to add conditions to higher-order functions (those which take other functions as parameters). A contract for a function that takes as parameter any function which only returns positive numbers, and builds negative numbers out of it is an example of this syntax:

```
(provide (contract-out [ fn ((any .->. positive?)
                                    .->. negative?) ] ))
```

Finally, structures can also be annotated with contracts. For example:

```
(provide (contract-out [ struct position (
                          (x number?) (y number?) ) ] ))
```

A larger example taken from the Racket Guide, showing some features mentioned above:

```
(define id? symbol?)
(define id-equal? eq?)
(define-struct basic-customer (id name address) #:mutable)

(provide
 (contract-out
  [id?                    (-> any/c boolean?)]
  [id-equal?              (-> id? id? boolean?)]
  [struct basic-customer ((id id?)
                          (name string?)
                          (address string?))]))
```

Contracts can have any kind of boolean function as argument, so larger contracts can be built by combining these functions (for example, here `id?` is given an implementation and a contract, and later used in another contract). Contracts are also specified separately from the definitions of functions and data structures.

3.1.3 ANSI/ISO C Specification Language

Eiffel style contracts reappear under different guises in different programming languages. For instance, the Java Modelling Language (JML) [LBR99] is a specification language for Java programs. It uses Hoare style pre- and postconditions and invariants, and follows the design by contract paradigm. Specifications are written as annotations, inside comments. Similarly, the ANSI/ISO C Specification Language [BFM$^+$] follows the same pattern.

Since annotations are written inside comments, standard compilers can be used to parse the original source files. An example of the ANSI/ISO C Specification Language can be seen in the following code:

```
/*@ requires \valid(p);
  @ assigns *p;
  @ ensures *p == \old(*p) + 1;
  @*/
void incrstar (int *p);
```

In this case, the precondition requires that `p` be a valid pointer. The annotation also says that `*p` is rewritten and that `*p` is incremented exactly by 1. We note that as opposed to JML, in the ANSI/ISO C Specification language, in most functions we need to add pre-conditions to make sure that the input arguments are valid pointers. ACSL is used by Frama-C [CKK+12], a set of interoperable program analysers for C programs.

3.1.4 Assessment

There are several conclusions from the analysis of contract languages:

- Many languages have only a fixed set of properties that can be expressed, which is good for analysis, while many others allow any boolean property, which is very expressive but also very hard to analyse. There is indeed a trade off between extensibility, something desired in the assertion language, and analysability of the assertions;

- Contracts or assertions are applied to different elements in each language: classes, methods, loops... These elements are usually the basic blocks of the programming language that hosts the contracts. To target interoperability and wide applicability of our assertion language, the design must not be dependent of particular code structures in a particular paradigm of languages;

- The relation between functional properties and non-functional properties for a specific hardware platform must be figured out to gather a good set of basic assertions. For example, bigger data sizes usually imply greater energy consumption. Along with the set of properties that directly express energy, time and precision, any other that may affect those must be part of the language.

3.2 Realtime and concurrent systems

One of the special requirements of the assertion language is that it should tie together high-level languages in which algorithms and applications are written with low-level and even hardware operational information, which is ultimately responsible for energy and time consumption. For that reason we look at existing assertion languages used for hardware design and assertion languages used in writing systems software.

3.2.1 UML for Real Time systems (MARTE)

UML is a graphical modelling language for object-oriented development. The UML Profile for Modelling and Analysis of Real-Time and Embedded Systems (MARTE) [Gro11] allows modelling of real-time systems.

MARTE is a good source of inspiration because it has graphical ways to talk about non-functional concerns. Specifically, it adds to the UML language:

- A set of annotations that can be added to sequence diagrams, specifying its time constraints,

- A way to encode non-functional properties into diagram elements. For this matter it introduces the Value Specification Language (VSL), allowing to give textual representation of complex properties.
- A set of classes representing resources, which can be used to model complete systems.

MARTE distinguishes between two kinds of non-functional properties:

- *Quantitative* properties are those specified by a number together with a unit of measurement (in MARTE there is support for different units, along with conversions between them).
- *Qualitative* properties are more complex, defining for example a probability distribution or a pattern of clock ticks.

The NFP package in MARTE predefines several of the units and types of properties in a way that can later be used for in-depth analysis (for example, when given a duration, it is given with its mean value and its possible error). Some of these predefined types are energy, transmission rate or clock frequency. The same package also defines the most used probability distributions, such as Gaussian or Poisson.

Special care is taken for modelling time, which is defined in a separate `Time` package. There are three different related concepts introduced in that package:

- *Clocks*, which can be physical or logical;
- *Usage*, which allows to model events, repetition, constraints...;
- *Structure*, which defines the basic blocks for time modelling.

Graphical time specification can be very tricky, so using VSL expressions is quite useful in this case.

Properties can be attached to UML models to specify their non-functional properties. They are interesting in combination with sequence diagrams, because they allow to model precisely the real-time constraints of a system.

Finally, MARTE has the notions of *resources* and *components*, which can be used to model hardware and software systems. However, the scope of them is outside of the assertion of properties in existing languages.

3.2.2 Promela

Promela is a language for the specification of interactive concurrent systems. These consist of a finite number of separate components, which are independent from each another, and interact through message passing over channels.

Generally, the correct functioning of concurrent systems depends on the timely coordination of their interacting components. In Promela [Hol91] assertions can be specified to indicate system states that should never happen (*never claims*). These can also be generated using linear temporal logic

(LTL) formulae, which describe correct (or incorrect) behaviour of the system. the formalism for temporal reasoning deals only with the qualitative aspect of time, for example, the order of certain system events. An example of such a specification is "system state 1 never happens after state 2 and before state 3". Promela specifications can be simulated or verified using a model checker (Spin [Hol97]). Executable code can also be generated from these specifications.

3.2.3 Bound-T Assertion Language

Bound-T [HLS10] is a software tool that uses static analysis to compute upper bounds on the execution time (WCET) and stack usage of embedded programs and is developed by Tidorum Ltd. It has a very expressive high-level assertion language which allows users to express facts about program in different contexts. Here the context can be global (about whole programs) or local (subprogram, loops, calls, program points, and so on). The facts that can be expressed range from a variable's value through the number of loop iterations, invariants, and number of executions of a call to time consumption of a subprogram or call. Assertions are specified in a different file than the source file thus allowing separation of concerns, however linking this assertion file to the source file can be tricky and complicated. As the language is expressive enough the user has to be well aware that asserting facts may introduce errors, yielding erroneous analysis results as a consequence.

Some examples of assertions include:

- `variable numSensors` $\leq$ `15`. It is the global assertion about the value of the variable `numSensors`, which is less or equal to 15.

- `subprogram Initialize variable numSensors > 0; end Initialize`. This says on entry of subprogram `Initialize` the value of variable `numSensors` is greater than 0.

- `subprogram ScanData call to Check invariant numData ; end call; end ScanData`. This says in a subprogram `ScanData` , calling a subprogram `Check` the values of `numData` does not alter, thus is an invariant in this context.

3.2.4 XMOS Timing Analyser (XTA)

XC is a C-like language developed by XMOS and targeting hardware solutions. XC comes with a Timing Analyser [XMO12] which provides analysis of time requisites. The assertions language is not as general as others, but it models timing requirements.

The analysis of XC code timing is based on the notion of *endpoints*, which are programs points that can later be specified as starting or finishing paths of computation. Endpoints are defined in source code with an assertion resembling

```
# pragma xta endpoint " name "
```

Each pair of endpoints gives raise to one or more routes, which are analysed. Then, the worst execution time of all routes is chosen as an upper bound for time spent.

Additionally, there are two additional kinds of endpoints:

- Function points, which define all possible paths between entries and exits to a function.
- Loop points, i.e., endpoints at the start and end of a path and excluding everything outside the loop.

Sometimes, the tool may need extra information which cannot be automatically inferred:

- Exclusions: this allows to discard results for a exceptional route that arises only on error cases, but does not reflect the usual behaviour of the system,
- Loop iterations: sometimes the tool cannot infer an upper bound on the number of iterations, so the code must be annotated explicitly. This can be made for all executions of the loop or only for executions in a particular route.

Once such information has been given, one run of the tool generates the entire set of paths that the program may follow, which are shown in a diagram resembling a flow chart. Then, the best route along the chart (the one with less execution time) is shown in green, and the worst one is shown in red. The developer can click on a different route and see the information related to that path of execution.

Apart from the models of execution for their hardware, the XMOS Timing Analyser allows the programmer to specify the running times of single instructions, functions of paths between two endpoints, which can enrich the analysis and allows to get better bounds for execution times. Finally, as extra parameters to analysis, the number of cores, threads and clock frequency can be specified, both in some routes or globally. The tool, however, does not handle recursion, and cannot analyse parallel programs.

3.2.5 Assessment

The study of these languages helps the design of the assertion language in several ways:

- The embedded system community is very interested in the timing analysis of components. In case that we decide that the common assertion language should include some kind of temporal properties, we have to decide how to model them;
- There is a good support from RTL design tools for checking PSL and MARTE assertions both statically and by simulation. The assertion language could interoperate with these tools, which give valuable input for higher-level energy and time usage;
- MARTE has whole packages devoted to non-functional properties and time. This can help when modelling properties. MARTE is an standard from MG, so looking at it will help with further standardisation.

3.3 Specification of properties for hardware design

3.3.1 Property Specification Language (PSL)

Several languages have been created to support assertions on hardware specification. The main language for this task is the Property Specification Language (PSL) [PSL05], which was standardised in 2005 and augments Verilog [Pal03] and System Verilog with assertions.

These are specified as "layers": boolean, temporal and verification. The temporal layer allows us to specify certain properties similar to the way we would using temporal logics. For example, an expression could say "in the following ten clock rises". The verification layer tells what to do in the checker. Possible values are `assert` (check it), `assume` or `restrict` (useful to guide the checker). The boolean and temporal part of an assertion define a property, which can be given a name, for example:

```
property mutex(boolean clk, a, b) = always (!(a & b)) @(posedge clk);
```

declares that a and b are mutually exclusive on the positive edge of the clock, and once the property is defined we can assert it:

```
assert mutex(clk, write_en, read_en);
```

PSL temporal assertions can be very expressive, allowing us to express sophisticated patterns of temporal conditions. PSL assertions can be checked using a model checker or by simulation.

3.3.2 Unified Power Format

Unified Power Format (UPF) [SM07] is a standard for specifying power intent in power optimisation of electronic design automation. It is a language to expression the design intent of the hardware power management. This include the power domains and their voltage levels and the different power states of the system. UPF is therefore used to augment a hardware specification to define the power architecture for a given implementation. UPF is used in conjunction with the hardware description as input to the implementation and verification tools to verify correct power down/up sequences and detect power architecture structural errors.

UPF makes use of a Power State Table (PST), which describes all the possible power states of a design. This has a hierarchical structure, and the transitions between power states are described in a table. Not all possible transitions are valid transitions. A feature similar to a PST could potentially be used to augment the Common Assertion Language. This would allow, for example, the system to transition from one power state to another, and this transition would be triggered by the software.

3.4 Language integration and syntax

The concrete syntax of the languages described here can be represented in various ways, or integrated into other host languages. In this section, we categorise the different methods used to represent the assertion languages.

Comments: Assertions are inserted as comments referring to or adjacent to an intended code segment or term. This option gives the largest amount of freedom for extending and changing the assertion language. However, it has one main disadvantage: a possible implementation would find difficultly in finding the segment or term to which a comment may apply.

External language: In the case of UML, for example, where OCL is used as an assertion language, properties of the code are defined in a separate file. This has the potential problem of keeping synchronised the code and the assertions, whenever the former changes.

Native: In some languages, assertions are part of its specification (as in the case of Eiffel), and therefore the language implementation understands assertions and can perform analysis and transformations based on this. While this option allows the tightest integration, it has a drawback in terms of extensibility, because the language implementation has to be changed significantly.

Extending the language: To work around this drawback, sometimes a language is extended into a larger one, and a transformation is developed such that code can be transferred back to the "parent" compiler. This is the case of Spec#, which augments C# with specifications. The main advantage is that the implementers of the new language can reuse an already developed compiler, while still keeping a "native" experience.

Using language extensibility: Some languages already allow the programmers to add extensions to the language. For example: Java annotations, Lisp macros or Prolog directives. Assertion languages for an extensibile host language typically use this feature in their implementation.

Now that we have classified the integration strategy of some of these languages, we have a better understanding about the best way to represent both the front end and Internal Assertion Language. We are going to base the Internal Assertion Language on the Ciao assertion language, which is extensible in nature and thus allows us to encode all of the required properties. For the front end language, we are planning to extend the XC source language with annotations and pragmas.

4 Internal Assertion Language

The Internal Assertion Language (IAL from now on) is the language used by the analysis tools to communicate and express non-functional information. As we show in deliverable D3.1, there is a large number of different programs, scripts and processes involved in the analysis of a piece of code. This highlights the importance of a common language that allow them to communicate freely and to encode all possible information that may be needed in different stages of the analysis or in later stages of optimization and visualization.

As mentioned before, the IAL is based on the assertion language found in Ciao [HBC+12]. The reason underlying this decision is that Ciao is a multiparadigm programming language (supporting functional, logic and constraint programming at the moment of writing), so it meets our need in the assertion language for supporting several styles of programming, since the IAL could be generated from any kind of programming language (in particular, in the ENTRA project we are using XC, an imperative and concurrent language).

Also, the assertion language in Ciao supports many of the constructs and properties that we defined as requirement for the IAL. Furthermore, in some cases we perform a translation of a program to CLP (based on Horn clauses) form, which can be directly managed by the Ciao runtime, and thus minimizing the time spent in translations in the process.

It is important to remember that in this document we will not define any specific syntax for the IAL, but rather a conceptual model, but as a syntax is ultimately needed in the concrete implementation, we have studied two ways in which we can represent the assertions:

- Using Prolog directives: this is the path taken by the Ciao language (which embodies logic programming). In this case each assertion is represented as a term. The main advantage of this approach is that directives can be written near the syntactic elements they affect, making unnecessary in many cases to clarify the scope of an assertion.

  An example of an assertion represented as a term is:

  ```
  :- trust add(X,Y,Z) + resource(avg, energy, 15.0)
  ```

- Using JSON (JavaScript Object Notation) or XML: many languages support data interchange in any of those formats. In this case each assertion is a JSON object or an XML node, with each of the attributes being part of the information to convey.

  The previous example could be represented using JSON as:

  ```
  { assertion: {
      status:  'trust',
      scope: { function: 'add' },
      properties: {
  ```

```
    computational: [
      { resource: { approx: 'avg', id: 'energy', value: 15.0 } }
    ]
  } } }
```

The previous examples are a possible path towards a concrete representation of an assertion using these syntactical constructs. At this stage the IAL is defined at a conceptual level.

4.1 Assertions

Assertions are the constructs that will be used throughout the analysis to express the properties of a computation in both functional and non-functional terms. In this section, we shall focus on the overall structure of an assertion, and the properties that can be expressed will be the topic of next section. Each of the assertions will consist of five parts: status, scope, list of preconditions, list of postconditions and the list of computational properties.

The first of the items, *status*, tells the analyzer what to do with the assertion that will be read. The main statuses to be used, and which are found both in the IAL and the front end, are:

- `check`, which asks the analyzer to verify that the properties indicated in the assertion hold in the code that this assertion talks about;
- `trust`, which asks the analyzer to take the information of the assertion as true without further checks, and to include it in the knowledge base of facts needed throughout the analysis. One example of information to be trusted if the energy model for a specific platform, which is provided from an external source.

In addition to these, the IAL includes the following status that are used to indicate information about the analysis results:

- `true` and `checked` are used to specify results that have been proven to be true. For a subsequent analysis, it should be taken as `trust` with subtle differences in meaning: `true` is used to specify analysis results produced by the analyzer which were not explicitly asked for checking (for example, results about auxiliary procedures) and `checked` is used when the verified assertion comes from a `check` assertion given by the user;
- `false` is used to identify assertions which have been proven to be false.

Notice that after an analyzer performs its work, it may be the case that some assertions still have a `check` status, because that analyzer was not able to prove or verify the assertion.

The second part of an assertion is its *scope*, that is, the program element to which the assertion refers to or should be applied. Examples of scopes may be an entire function or procedure, a module, a specific program point, a loop or conditional block, a piece of code between two delimiters and so on.

The scopes that can be expressed in the IAL have a strong dependence on the analyzers that are involved in the whole pipeline. For example, if the assertion language is used by an analyzer which processes CLP programs, the basic scope will be the predicate and the clause. It is important to notice that some extra translation work may be needed in the case that the same kind of scopes are not available in different analyzers.

With each identified scope we can associate three lists: preconditions, postconditions and computational properties. Conceptually, they are just listings of instances of the properties which will be introduced in the next section. Albeit similar, the semantics of each list is quite different:

- *Preconditions* specify the conditions under which the assertion is applicable. The precondition should involve program state properties as well as environment properties.

  For example, an assertion about the energy consumption of some predicate may only be applicable for numbers up to a defined value. The assertion will have a precondition with a constraint over the value to express this limited applicability.

- *Postconditions* specify the conditions that hold after the successful execution of the element described in the scope, provided the Precondition holds. Involve the same properties as the precondition. It should be noted that the postcondition only refers to a successful execution of its scope (the exact definition of "successful" depends on the context, in CLP for example it refers to finding at least one answer).

- *Computational properties* describe the properties of the whole computation of a scope that meets a precondition, similarly to postconditions. But in constrast to them, computational properties involve non-functional global properties, such as energy consumption, execution time, or resource usage in general, that do not refer to a specific state but rather to a path of execution.

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

```
:- trust pred append(A,B,C): (list(A),list(B),var(C)) =>
      (list(A),list(B),list(C)) + resource(ub, energy,length(A)+100).
```

states that for any call to predicate `append/3` with the first and second arguments bound to lists and the third one unbound, if the call succeeds, then the third argument is also bound to a list. It also states that an upper bound on the energy consumed by the execution of any of such calls is $length(A) + 100$ mJ, a function on the length of list A.

4.2 Properties Supported

The IAL has to be equipped with a rich set of properties to be used in assertions in order to achieve all the functionality stated in the requirements. So far, we have defined a set of new useful properties, and we foresee the definition of more new properties during the project. Some of the desired properties were already present in the Ciao assertion language on which the IAL is based.

We have classified the properties in four groups. Properties in the two first two groups are intended to be used in pre- and post- conditions. There are *program state* properties that refer to functional properties of the execution over some data, and *environment* properties that refer to the state of the hardware and the surrounding execution environment. An example of program state property is the type of a variable. The power consumption of a function may depend on whether the function takes a list or a number as an argument. An example of *environment* property is the frequency of the processor, which affects the energy consumed.

Then, we have properties which characterise the on-going effect of computation, such as memory or energy consumption: these are referred as *computational properties*. Finally, we discuss the possible ways in which this effect can be aggregated, giving rise to *modes of execution*, such as sequential or parallel.

In order to be formal, properties will be described as terms using a grammar-like formalism. Each possible choice for a rule will be denoted as `Rule -> format`. The shorter notation `A*` will describe a list of terms given by the rule `A`. Base types `Number` (describing an integer or floating-point value), `String` (describing a quoted list of characters) and `Id` (describing an unquoted string of characters) are taken as part of the language. Comments will be marked with `%`.

The arguments of some of the properties are numbers (as for example, the frequency). Depending on the kind of analysis, the end user may want to express a strict value for a number, a range which bounds it, or a probability distribution on the values. For that reason, the IAL does not enforce any of these choices (although in the actual implementation these choices will be made).

```
NumberArg -> Expr
NumberArg -> (Expr, Expr)
NumberArg -> dist(Id, Expr*)
               % The first argument is the type of distribution
               % For example: uniform, bernoulli, normal
```

where `Expr` denotes mathematical expressions including between numbers and variables.

As we have already stated, this notation is not enforced, but just a way to view the elements conceptually, and a concrete syntax based on JSON or XML could be used instead.

4.2.1 Program State Properties

In the program state we usually need to refer to variables. In the rules for properties, `Var` denotes the name of one of the variables appearing in the intended scope or `result`, which identifies the value returned by the function.

Types and modes of execution: These properties refer to the shape of the data that will take part in the computation. Types are a well-known formalism for this task, and we expect to have different flavours of types depending on the programming language hosting the assertion language. For example, object-oriented languages usually have a static type system with inheritance, functional languages tend to

have type systems related to the one defined by Hindley and Milner, whereas in logic languages, we have formalisms like regular types or tree grammars that describe sets of terms.

```
Type -> Id(Var)
        % The Id corresponds to a type taken from a set
        % defined by the underlying language
Type -> Id(Var, InnerType*)

InnerType -> Id
InnerType -> Id(InnerType*)

Mode -> +Var
Mode -> -Var
Mode -> ?Var
```

Types can also provide interesting information for guiding the analysis. For example, when analysing an assembly program, type declarations from the high-level language that produced that assembly can be attached to registers to know whether it holds an integer, a floating-point number, or a pointer to a larger structure in memory.

Data sizes: In order to estimate the resources used by a program, we first need to infer bounds on the loop iterations or recursive steps in a function or predicate. This usually depends on the sizes of the data structures traversed, like the length of a list or the dimensions of an array. For this reason, it is important to infer size relationships between sizes of the involved structures at different program points.

```
Size -> size(Var, NumberArg*)
Size -> size(Var, SizedType)
```

Accuracy: Another property, about the data in the computation, is accuracy. This is important because in some architectures speed and energy gains can be achieved by losing some accuracy in the computed results. The IAL will include properties for absolute and relative error, and using its commodities, these errors can be related to the mode in which the machine is running or the data sizes that are used.

```
Accuracy -> accuracy(Var, NumberArg)
```

Although some concepts of accuracy can be modelled by probabilistic data sizes, we have chosen to model it independently, because of its particular relationship with energy consumption.

Aliasing and sharing of variables: The fact that data structures have shared parts in memory, affects the analysis in at least two ways. The first is the consumption of resources, since shared variables may use less memory and less computational power than independent variables (because we only need an

operation in memory to update two structures sharing a variable). The second one is the possibility of adding parallelism. For instance, to be able to split a sequential task into several parallel tasks, the optimiser needs to ensure that there will not be any conflicting access to memory which can be proven for example by ensuring that the variables used in each thread are not aliased or that they point to data structures that do not share any variable.

```
Sharing    -> mshare(SharingElt*)
SharingElt -> [ Var* ]
```

There is already some work by some this project's partners to analyse and detect this requirement. In particular, the XC compiler by XMOS can detect when a variable will be shared between two threads, forbidding compilation in that case. CiaoPP also includes a variable sharing analysis. Both analysis could be extended to deal with fine-grained information.

4.2.2 Environment Properties

The environment properties refer to the computational architecture and hardware state that may affect the non-functional properties of the code being analysed.

Frequency, voltage, temperature: In general, any physical or logical property of the processor affects the non-functional behaviour. The IAL should include the most common properties in this field (but these can be extended by defining a new property). Processor frequency and voltage are related to both speed and dynamic power consumption, so they are essential to energy transparency. Temperature affects static power consumption, so including it as a parameter would give us a more precise outcome of the analysis.

```
Frequency   -> frequency(NumberArg)
Voltage     -> voltage(NumberArg)
Temperature -> temp(NumberArg)
```

Communication structure: Another source of delays and energy consumption are the buses in which information is transmitted. So in order to obtain accurate analysis results, information about how communication is done, distances to each node, whether communication is parallel or sequential is needed. The exact amount of information will depend on the architecture where the analysis and optimisation procedures are applied.

Caches: As the communication structure mentioned above, the structure and use of the cache affects resource consumption in a great deal: retrieving some data from a near cache level is both faster and less energy-consuming than doing so from main memory. The IAL includes properties describing the different cache levels, along with information on the distribution of cache hits and misses, which can be used for probabilistic reasoning.

Input and output on ports: In many embedded systems, input and outputs from ports to peripherals or other nodes in the network are as important as the data taken or returned by functions. Somehow

we need to describe the data in these ports as we do for regular arguments. The proposed solution is to use the same kind of assertions that we do for the latter, but tagged with the name of the port.

```
PortIO ->  in(Port, DataProp)
PortIO -> out(Port, DataProp)

DataProp -> Type
DataProp -> Size
DataProp -> Accuracy
```

In these rules, a `Port` is syntactically similar to a variable, but coming from a predefined set dependent on the architecture.

4.2.3 Computational Properties

Resources: The main category of non-functional properties, that are interesting in achieving energy transparency, corresponds to counters for a numerical properties that change while a piece of code is executed. This leads to the notion of resource captured by such counters. The resources in the IAL are very close to those found in Ciao. In particular, to describe a resource, we have to provide the following information:

1. A name that uniquely identifies the resource throughout the analysis and optimisation phases. For example, the resource `energy`;

2. How does each elementary operation in the program modifies the resource.

   These leads to *models*, which express how the execution of some basic operations in a hardware platform affect the resource consumption. For example, a model can express that the consumption of `energy` by the execution of a basic `add` assembly instruction is 10 mJ;

3. How to aggregate the resource consumption of a piece of code from the information of its constituent parts. For example, when executing functions f and g in parallel, the `time` consumed is roughly the maximum of the time of each function, whereas the `energy` consumed depend on whether the processor runs at a lower speed, whether the work is divided between cores, etc.

   ```
   Resource -> resource(Approx, Id, NumberArg)
   ```

We have made a preliminary list of resources that the system must tackle:

- *Energy resources*: Energy, peak power, average power and capacitance fall into this category. They are highly dependent on frequency, voltage and temperature of the processor;

- *Time resources*: such as clock time, number of steps or number of executed instructions;

- *Data resources*: refer to information stored or transmitted by the system. This includes used memory or bytes sent or received from a port.

Failure and determinism: In many cases, the analysis also needs to know whether some code will be executed "to the very end" or there is a chance that it ends in other way. This encompasses three main ideas:

- In some languages, functions can throw exceptions. Knowing which exceptions can be thrown can help us giving accurate resource estimations;
- In logic languages, predicates can either succeed or fail. Non-failure analysis, as implemented in CiaoPP, is instrumental to obtain non-trivial lower bounds for the consumption of some resources;
- Also in logic programming, it is important to know which set of clauses can be executed at a given time, because it restricts the amount of work needed, for example, to obtain all solutions.

```
FailDet -> throws(Id*)
FailDet -> not_fails
FailDet -> possibly_fails
FailDet -> is_det
```

Probability of branches: Failure or determinism are very sharp distinctions in many cases. Instead, a probability can be assigned to each branch of execution.

```
Probability -> probability(Num)
```

Modes of execution: As mentioned in the section on resources, those can be aggregated in different ways depending on how a piece of code is mapped into execution in the actual hardware. Furthermore, some of the optimisations that will be proposed are based on changing this mode of execution, usually between sequential and independent parallel execution. The initial set of modes we would like to consider in the IAL include sequential, independent parallelism and communicating parallelism (as can be found in the XC language that we will target).

4.2.4 Examples

In this section we will see how the IAL and the properties stated before are enough for defining some of the information about the hardware we want to model, for annotating functions with energy information, and other functionality required. As in the rules, we are going to use Prolog directives and terms as the concrete syntax.

Energy models: In the most basic case, a model for the energy consumption of a processor assigns a constant energy to each assembly instruction executed. So, given a set of basic instructions, the energy model for a simple processor would look like:

```
:- trust add(X,Y,Z) + resource(avg, energy, 154).
:- trust sub(X,Y,Z) + resource(avg, energy, 176).
:- trust mul(X,Y,Z) + resource(avg, energy, 330).
...
```

Note that the first argument (`avg`) of the resource property expresses that the given energy consumption for the add instruction is an average value.

In a finer version, we could include the frequency as a parameter, and distinguish different types of operands. For example, an `add` instruction could take either two registers, or one register and one immediate value (we model these differences using types), giving different energy results:

```
:- trust add(T,R,S)
         : register(R), register(S), frequency(F)
         + resource(avg, energy, 74*F).
:- trust add(T,R,S)
         : register(R), immediate(S), frequency(F)
         + resource(avg, energy, 71*F).
```

Note that in this case, the energy consumption is given as a function on the frequency F.

Dynamic Voltage and Frequency Scaling: The fact that we can express environment conditions before and after the execution of a piece of code is very useful for modelling functions that change the state of the processor.

Assume an instruction `chgfrq` that has just an operand that is the new frequency in which the processor must run. This operand is a number, so its size is barely its value. The time needed for making the switch depends on the difference between the old and new frequency:

```
:- trust chgfrq(F)
         : ( size(F,FNew), frequency(FOld) )
        => frequency(FNew)
         + resource(ub, time, 11*abs(FNew - FOld)).
```

Annotating functions with energy consumption: Assume that we want to annotate a function (e.g., a factorial function in a library) with trusted energy consumption information. The following is an example of an assertion that could be used for it:

```
:- trust factorial(N)
         : ( int(N), frequency(F) )
         + resource(ub, energy, 70 * N * F).
```

which express an upper bound (ub) for the energy consumption of a call to the factorial function, given as a function on the input integer N and the clock frequency F.

5 Front end

The front end of the Common Assertion Language consists of syntactic extensions to the XC language, so that some of the properties described in the assertion language are also expressible using the front end language. We also describe how the information expressed in the front end can be extracted and propagated through the system layers. We intentionally do not fully specify the front end language here since the exact features will depend on experimentation with real case studies.

5.1 Expressions and constraints

In XC, the basic constraints are composed of arithmetic and boolean expressions about program objects. The linguistic elements of the front end language will likely consist of:

1. Simple XC boolean and arithmetic expressions.
2. Additional energy/power specific functions and predicates.
3. Additional probability density functions.

Expressions are simple arithmetic and probabilistic constraints on program variables *e.g.*:

```
x < 10
x = normal_distribution(...)
```

In addition, specific predicates and functions can specify resource usage properties. For example, a `max_energy` function that represents the maximum amount of energy a function will consume when run parametric to input arguments can be represented as:

```
max_energy(f(x)) < N + M * x
```

Often, functions run indefinitely so there will be also be a need for power analysis functions *e.g*:

```
max_power(f) < N
```

Another useful addition, which also helps with the translation of the front end language to the Internal Assertion Language, is the use of symbolic global counters. For instance, we can add two counters for time and energy called `time` and `energy` respectively:

```
en1 == energy   // if true, there exists a variable en1,
...             // containing the energy consumed up to this point
en1 + 100mW > energy
```

These global counters can only be used in boolean expressions such that their actual values do not have to actually be evaluated. For instance, the following is disallowed:

```
time == 100
```

Whatever the exact nature of the constraint expressions, these need to be embedded into the XC program. The following section describes a method to achieve this.

5.1.1 Global properties

At the global scope of an XC program, the user can specify global level properties to either trust or check a property (`trust` and `check` having the same meaning as in Section 4):

```
#pragma entra check expr;
#pragma entra trust expr;
```

The expressions within the trust/check predicates may reference global variables and functions declared in the XC source.

The properties can have an optional name *e.g.*:

```
#pragma entra [assertion_1] check expr;
```

This allows the user to refer to the property in other tools. It is expected that most power and energy checks will be expressed in terms of global functions. For example:

```
void f();
#pragma entra [max_power_of_f] check (max_power(f) < 2mw);
```

5.1.2 Local properties

At local scope the user can make local properties. These can be trust or check properties but also require a temporal modality (one of `now`, `always`):

```
[[entra::trust now expr]];
[[entra::trust always expr]];
```

Here, the expressions can reference both global and local variables in scope.

The `now` modality says that that expression is true at that point in time e.g.:

```
x = f();
[[entra::trust now (x < 10)]];
```

The `always` modality says that that expression is true at all points in program execution. This can be used to express, amongst other things, invariants on variables in the program:

```
int x=0;
[[entra::trust always (x < 10)]];
```

5.1.3 Program point labels

Program points in the execution of the program can be given labels:

```
[[entra::label name]];
```

These labels represent the set of states that the program may reach at that program point.

5.1.4 Control flow properties

Control flow properties can only be used when attached to a particular loop construct and have no modality. The following property lets the user state the number of expected iterations of a loop:

```
[[entra::trust max_loop_iterations(N)]]
while (e) {
   ...
}
```

The `cond_probability` lets the user specify the probability of a conditional expression being true:

```
[[entra::trust cond_probability(0.4))]]
if (e) {
  ...
}
```

The `ignore_if_true` and `ignore_if_false` let the user specify that a branch should be ignored with respect to any analysis.

```
[[entra::ignore_if_true]]
if (e) {
  ...
}
```

These can also be suffixed with `_wrt`, which is interpreted to mean that a branch is ignored when checking a particular property (using the named):

```
[[entra::ignore_if_true_wrt(max_power_f)]]
if (e) {
  ...
}
```

5.2 Retaining information in LLVM IR and ISA representations

The assertion pragmas and attributes can be maintained through to the LLVM IR and ISA levels by translating the assertions into inline assembly at an early stage of compilation (during parsing). For example, the pragma:

```
#pragma entra [max_power_of_f] check (max_power(f) < 2mw);
```

would translate to an XC inline `asm` statement as follows:

```
asm("#entra check [max_power_of_f] check (max_power(f) < 2mw)");
```

This assembly comment will be maintained through compiler optimisations to the ISA level. To allow accurate analysis, any global functions referenced in assertions are not inlined by the compiler. This allows any analysis framework to know all uses of the function (at the expense of some possible compiler optimisations).

Local assertions translate to *volatile* inline asm to maintain their position in control flow relative to other code. The front end of the compiler also has to keep track of any local variables referred to in the expression, and make them inputs to the inline assembly. For example, the following code:

```
int x=0;
[[entra::trust always (x < 10)]];
```

would be translated to:

```
int x;
asm volatile ("#entra trust always ([x,%0] < 10)]]"::
              "r"(x):"memory")
```

This way, the value of `x` is always available in the LLVM IR and the final ISA.

Declaring inline assembler statements in this way makes the assertions easier to track during compiler and optimisation but can prevent certain optimisations from happening. We also need to keep in mind that single locations in source code can map to multiple locations in assembler code - `asm` statements are therefore not immune to program transformations such as loop unrolling and inlining.

After full compilation and optimisation, the inline assembly comments can then be translated into constraints for an analysis tools (e.g. CiaoPP).

Control flow assertions also translate to inline assembly by passing the value of the condition to the assembly as a parameter. For example, the following condition:

```
[[entra::trust cond_probability(0.4))]]
if (e) {
  ...
}
```

would translate to:

```
int tmp = e;
asm volatile("#entra trust cond_probability([e,%0], 0.4))"::
             "r"(tmp):"memory")
if (e) {
  ...
}
```

The one exception to this method of translation is the `max_loop_iterations` assertion. This uses a different method of translation that utilises already existing mechanics in the XC compiler backend to annotate the back-edge jump of a loop with the information. This feature already exists as part of the XMOS timing analyser (see Section 3.2.4).

5.3 Linking front end to internal assertion language

As part of the translation from a source language to an internal representation that will be analyzed (which is explained in deliverable D3.1), the assertions written in the front end language have to be converted in the corresponding ones in the internal assertion language.

The status and properties remain almost unchanged in the translations. The scope is the part that needs more work, because the kind of syntactical constructs may differ between the source and intermediate languages. For example, when converting an XC program into a CLP representation, the assertions must be moved from program points into Horn clauses.

As an example of the aforementioned translation, consider the following program written in XC with assertions:

```
#pragma entra check (energy(factorial(n)) < 200*n);

int factorial(int n) {
  [[entra::trust now (n >= 0)]]
  [[entra::trust cond_probability(0.4]]
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

The translation to CLP with assertions in the IAL using term syntax would take this form:

```
:- check pred factorial(N,R)
        : int(N), var(R), size(N,(0,'Infinity'))
        + resource(ub, energy, 200*N).

:- trust clause factorial/2/1 + probability(0.4).
factorial(N,R) :- N is 0, R is 1.
:- trust clause factorial/2/2 + probability(0.6).
factorial(N,R) :- N1 is N - 1, factorial(N1,R1), R is N * R1.
```

There are several things to notice:

- The `check` assertion in the initial XC program has been converted to a `check` assertion in the CLP program. The latter assertion contains some extra preconditions that are only implicit in

the XC code: the types of variables, and the assertions that apply right at the beginning of the code of the `factorial` function;

- Since different paths of execution are represented as Horn clauses, the syntax for the probability now has as scope a clause (for example, `factorial/2/1` is the first clause of the `factorial/2` predicate).

6 Scenarios for the Use of the Common Assertion Language

We have compiled three different scenarios, which demonstrate how the common assertion language is utilised in various parts of the system. We demonstrate how the assertions are preserved between source, intermediate compiler representations and machine representations and how these are translated to internal assertion language representations and vice versa.

6.1 Scenario 1. Checking and verifying an energy budget

In this scenario the program to be analysed is a function that will be called repeatedly in a system powered by a battery. The battery can supply 100,000 units of energy (the units are not specified here) and the developer wants the battery to supply power for at least 1,000 calls to the function. The function can be called with an input parameter that affects the energy consumed by a call to the function.

Therefore the developer sets a budget of 100 units of energy, and aims to explore what range of input values will be handled by the function without exceeding that budget. The aim in this case is not to simply check that the energy budget holds unconditionally. Therefore, the system is expected to automatically infer a range of input values for which the budget will hold.

For the purpose of the example we assume that the function is *factorial* (called *fact* below) which has one integer argument.

1. An assertion is added to the XC source of the *fact* function stating that the energy consumed is at most 100 units. An example of how this might appear as a source language annotation (front end assertion language) is as follows.

   ```
   #pragma entra check (energy(fact(n)) < 100);
   ```

   Notice that the keyword `check` means that the analysis tools should try to verify that the assertion holds in the program. Also note that `n` is a free variable in the expression, which means that the resource analysis framework will try to infer its type and its possible ranges in order to try to meet the conditions expressed in the assertion.

2. The source code pragma is mapped from source code to intermediate compiler representations and ISA level. A solution that does not involve deep changes to the XC compiler is to simply copy all `entra` pragmas at a module level into the LLVM IR. LLVM has a facility to add inline assembler expressions to a module. These assembler expressions are then copied to the ISA representation. Since comments are also valid assembler statements, the pragma can be translated and added to the LLVM IR module:

   ```
   module asm ".loc 0 3 2"
   module asm ";#entra check (energy(fact(n)) < 100)"
   ```

The rest of the standard compiler toolchain (`llc`) copies this line of inline assembler to the prelude of the assembler file when compiling to ISA instructions. Note that information is retained on the relation of compiled assertions to the position of original assertions. This is preserved using the DWARF standardised debugging format, and preserves line and column numbers, so that analysis results can be reported back to source level.

3. The ISA code including assertions is translated to the internal representation (IR) which is the input to the analysis tools. The energy assertion, when translated into the internal assertion language (IAL) of the IR, has the following form (we omit positional assertions for presentation purposes):

```
:- check pred fact_internal(A, B): (int(A), var(B))
             + resource(energy, 0, 100).
```

   Here `fact_internal` is assumed to be the internal name for the predicate representing the *fact* function in the source program.

4. The IR of the ISA program is combined with assertions representing the ISA energy model. In this simple scenario each instruction is assigned an energy value, represented as a `trust` assertion. For instance the `ADD` instruction (integer unsigned add) is given an upper-bound energy consumption by the following assertion.

```
:- trust pred add(X,Y,Z) + cost(ub, energy, 1215439).
```

5. The resource analysis of the IR infers both upper and lower bounds on the resource "energy". The analysis results are written in the IAL as:

```
:- true pred fact_internal(A,B)
      : (int(A), var(B) )
     => (int(A), int(B), rsize(A, num(LA,UA)),
         rsize(B, num('Factorial'(LA),'Factorial'(UA)))
         )
      + resource(energy, 21 * A + 16, 21 * A + 16).
```

   This means that the analysis in this case infers that both the lower and the upper bound resource usage function are the same: 21 * A + 16, a linear function on the value of the input argument to the factorial program. In general the upper and lower bound functions are different.

   Note: The analysis also infers upper and lower bounds for the result of the *fact* function. These bounds are inferred to be `UA!` and `LA!` respectively, represented in the assertion as `'Factorial'(LA)` and `'Factorial'(UA)`, where `UA` and `LA` are the respective upper and lower bounds for the input argument. However this information is not used in checking the energy assertion.

6. Then, a comparison of the analysis results with the "check" assertion (the specification) is performed. As a result, the following assertions in IAL are produced:

```
:- checked pred fact_internal(A, B):
     (int(A), intervals(int(A), [i(0,4)]), var(B))
     + resource(energy, 0, 100).

:- false pred fact_internal(A, B):
     (int(A), intervals(int(A), [i(5,inf)]), var(B))
     + resource(energy, 0, 100).
```

   The first assertion means that the specification `energy < 100` is true if the input argument of the factorial program `A` is in the interval `[0, 4]`. The second one means that the specification is false if `A > 4`.

7. These results are translated back to a representation compatible with the source language. A new assertion is derived from the original assertion, which indicates that it has been checked, and holds subject to a constraint.

```
#pragma entra checked
  (n >= 0 && n =< 4 ==> energy(fact(n)) < 100);
```

6.2 Scenario 2: Checking a power budget

In this scenario the programmer wishes to ensure that the average power during execution does not exceed a fixed limit. (Note that checking peak power is a different scenario and would have a different resource name).

Therefore the programmer sets a fixed limit of say, 100 power units (again, the units are not important). As before, assume that this program is the *factorial* function.

1. An assertion is added to the XC source of the function stating that the average power dissipated is at most 100 units. An example of how this might appear as a source language annotation is as follows.

```
#pragma entra check (avg_power(fact(n)) < 100);
```

   The keyword `check` means that the analysis tools should try to verify the assertion.

2. As in the previous scenario, `xcc` preserves this assertion in the compilation to LLVM/XS1 code.

3. The ISA code including assertions is translated to the internal representation (IR) which is the input to the analysis tools. The average-power assertion, when translated into the internal assertion language (IAL) of the IR, has the following form.

```
:- check pred fact_internal(A, B):
     (int(A), var(B)) + resource(avg_power, 0, 100).
```

   where `fact_internal(A, B)` is a logic predicate (`B` is an output argument that represents the returned value of the XC *fact* function). The property `resource(R, L, U)` represents that `L` and `U` are lower and upper bounds respectively on the usage of resource `R` by any call to `fact_internal(A, B)` that meets the precondition `int(A), var(B)` (the comma represents a conjunction).

   The assertion has the status `check`, which expresses that we want the system to attempt to verify it.

4. Assuming that there exists an ISA-level energy and timing model, the resource analysis engine must contain the strategy that in order to infer the resource `avg_power` it has to infer the resources `energy` and `time` and derive `avg_power` from them,

5. The resource analysis infers both upper and lower bounds on the resource `avg_power`. The analysis results are written in the IAL as:

```
:- true pred fact_internal(A,B)
    : ( int(A), var(B) )
       => ( int(A), int(B),
         rsize(A, num(LA,UA)),
         rsize(B, num('Factorial'(LA),'Factorial'(UA))) )
          + resource(avg_power, 10, 90).
```

   which means that the lower and upper bounds inferred by the analyser are 10 and 90 respectively. Note the status `true` of this assertion. The variables `UA` and `LA` and the associated constraint have the same meaning as in Scenario 1.

6. Then, a comparison of the analysis results with the `check` assertion (the specification) is performed. As a result, the following assertion in IAL is produced:

```
:- checked pred fact_internal(A, B):
     (int(A), var(B)) + resource(avg_power, 0, 100).
```

   which means that the specification has been proved (status `checked`).

7. We can have other outcomes of the verification process depending on the analysis results. For example assume now that the lower and upper bounds inferred by the analysis were 150 and 250 respectively:

```
:- true pred fact_internal(A,B)
    : ( int(A), var(B) )
       => ( int(A), int(B),
         rsize(A, num(LA,UA)),
         rsize(B, num('Factorial'(LA),'Factorial'(UA))) )
          + resource(avg_power, 150, 250).
```

   In this case, the specification would be false:

```
:- false pred fact_internal(A, B):
     (int(A), var(B)) + resource(avg_power, 0, 100).
```

8. Finally, assume that the lower and upper bounds inferred by the analysis were 10 and 150 respectively:

```
:- true pred fact_internal(A,B)
    : ( int(A), var(B) )
       => ( int(A), int(B),
         rsize(A, num(LA,UA)),
         rsize(B, num('Factorial'(LA),'Factorial'(UA))) )
         + resource(avg_power, 10, 150).
```

   In this case, the system wouldn't be able to prove whether the specification is true or false, which implies that the assertion status remains as `check`:

```
:- check pred fact_internal(A, B):
     (int(A), var(B)) + resource(avg_power, 0, 100).
```

9. As in Scenario 1, the result of analysis is reported back to the user by setting the `checked`, `false` or `check` status in the XC source assertion, corresponding to the sample analysis results above.

6.3 Scenario 3. Using trusted assertions for unknown code

In this scenario the code of a function called in the main function is not available or is not implemented yet. The developer wants to know the impact on energy consumption of such a function in the main function.

1. Assume the following `sqr` program that calls the function `sqr_aux`, which is not implemented yet:

```
int sqr_aux(int n);

int sqr(int n)
{
  if (n == 0)
    return 0;
  return sqr_aux(n) + sqr(n-1);
}
```

2. The developer assigns an estimated energy consumption value to the function `sqr_aux`, and tells the system to trust it in order to see its energy impact on the whole system with an assertion:

```
#pragma entra trust (energy(sqr_aux(n)) < 100);
```

3. As in the previous scenarios, `xcc` preserves this assertion in the compilation to LLVM/XS1 code, which gets translated into the internal assertion language as:

```
:- trust pred sqr_aux_internal(X,Y) + cost(ub, energy, 100).
```

   which means that the analyser will trust un upper bounds on the energy consumption for the `sqr_aux` function to be equal to 100, and will infer the energy consumption function `200 * X + 140` for the main function `sqr`, where `X` is its input argument. This is expressed using the assertion:

```
:- true pred sqr_internal(X, Y):
        (int(X), var(Y)) =>
          (int(X), int(Y))
             + cost(ub, energy, 200 * X + 140).
```

4. Now, assume that we provide a parameterised expression for the energy use of `sqr_aux`:

```
#pragma entra trust (energy(sqr_aux(n)) < 151 * n + 141);
```

   where `n` is the input argument to the `sqr_aux` function in XC. The translation in this case results in:

```
:- trust pred sqr_aux_internal(X,Y)
           + cost(ub, energy, 151 * X + 141).
```

5. The analysis will trust the energy information for `sqr_aux` and will infer energy consumption for the `sqr` program by taking into account this trusted information:

```
:- true pred sqr_internal(X, Y):
    (int(X), var(Y))
       => (int(X), int(Y))
          + cost(ub, energy, 103 * exp(X, 2) +
                        205.8 * X + 188.32).
```

7 Summary

The Common Assertion Language will be used throughout the project to express specifications that are used by the tools developed in ENTRA. These specifications are expressed in the source language, propaged through the system layers and are also expressed in the internal assertion language. We have come up with a description for the front end that is generic enough to be integrated into different source languages. The Ciao assertion language is used as a starting point for the internal assertion language, and will be extended and modified as needed. This language will be heavily used in the resource analysis framework (D3.1), and plays an essential role in the ENTRA project. At the same time, the specification of the Internal Assertion Language is designed such that it is generic enough to be applied to other resource analysis frameworks.

Thanks to the Common Assertion Language definition, all partners have achieved consensus on how the tools developed under the ENTRA project will interface to each other. In terms of future work, we intend to modify and implement the languages based on the kinds of analysis and the sorts of optimisations we will be performing in future work packages.

References

[BFM+] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification language, version 1.4, 2009. *URL http://frama-c. cea. fr/acsl. html*.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[Gro11] Object Management Group. UML profile for MARTE: Modeling and analysis of real-time embedded systems, 2011.

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

[HLS10] Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Bound-T assertion language. http://bound-t.com/, Feb. 2010.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Springer International Series in Engineering and Computer Science*, pages 175–188. Springer US, 1999.

[Mey92] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Pal03] Samir Palnitkar. *Verilog HDL: A guide to digital design and synthesis, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.

[PSL05] IEEE Standard for Property Specification Language (PSL). Technical report, 2005.

[RLM08] K. Rustan, M. Leino, and Peter Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In Peter Müller, editor, *LASER Summer School*, volume 6029 of *Lecture Notes in Computer Science*, pages 91–139. Springer, 2008.

[SDF+11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: approximate data types for safe and general low-power computation. In *Proc. of PLDI'11*. ACM Press, 2011.

[SM07] Phillip Stanley-Marbell. What is IEEE P1801 (unified power format)? *SIGDA Newsl.*, 37(19):1:1–1:1, October 2007.

[Wat09] D. Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.

[XMO12] XMOS. XMOS timing analyzer manual, 2012.