Sistemi Distribuiti

TuCSoN communication language: abstract tuples

Michele Braccini 0000709784
Alessandro Fantini 0000732678

Contents

Introduction

This is the final project report for the Distributed Systems course hold at University of Bologna. The aim of the projects assigned in the context of this course is "learn what it means to model/design/implement/test a middleware", so, in order not to start from scratch, we will make use of TuCSoN as a middleware.[1]

This project is part of the "TuCSoN as it will be", that is a redesign of specific and well-delimited TuCSoN pieces of software based on its formal model as described in TuCSoN papers.

We will focus on the TuCSoN communication language[2], in particular we will give an abstract model of the tuples with the goal of maintaining the widest compatibility and transparency.

[1] TuCSoN is a tuple-based coordination model and infrastructure.

[2] Communication language: the syntax used by coordinated entities to express the information exchanged among themselves.

Chapter 1

Abstract tuples

In this chapter we'll try to give an abstract definition of what a tuple is by making use of the literature we found about it.

1.1 Linda's tuples

Linda is a *coordination language* and it's based on the so-called *generative communication paradigm*: if two processes wish to exchange some data, then the sender generates a new data object (referred to as a **tuple**) and places it in some shared dataspace (known as a **tuple space**) from which the receiver can retrieve it. [4] So the abstract computation environment called "tuple space" is the basis of Linda's model of communication. [2]

Tuples are actually sequences of typed fields. They are retrieved from the "tuple space" by means of associative *pattern matching*.

At this point, we can summarize the features of Linda's tuples. A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort:

- with a record-like structure,
- with no need of field names,
- as an easy aggregation of knowledge,
- containing all information concerning a given item.

Tuple structure is based on:

- arity,
- type,
- position,
- information content.

The **template**, or **anti-tuple**, is a specification of set/classes of tuples and it is used in the *matching mechanism*, with the aim of defining the belongingness of a tuple to a set.

Tuple examples:

$$(\text{“}student\text{”}, \text{“}Michele\text{”}, \text{“}Braccini\text{”}, 24).$$
$$(\text{“}student\text{”}, \text{“}Alessandro\text{”}, \text{“}Fantini\text{”}, 24).$$

Template example:

$$(\text{“}student\text{”}, ?x, ?y, 24).$$

Note that Linda tuples and templates are **unnamed**.

1.2 Logic tuples

Logic tuples come from the attempt to add the Linda primitives and shared dataspace into Prolog. The typical mapping is to represent tuples by *terms*, and replace pattern matching with unification. [1] In Prolog, a tuple is modeled as a compound term, which is composed of an atom called a *functor* and a number of *arguments*, which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero.
tuProlog tuple examples:

$$student(alessandro, fantini, 24).$$
$$student(michele, braccini, 24).$$

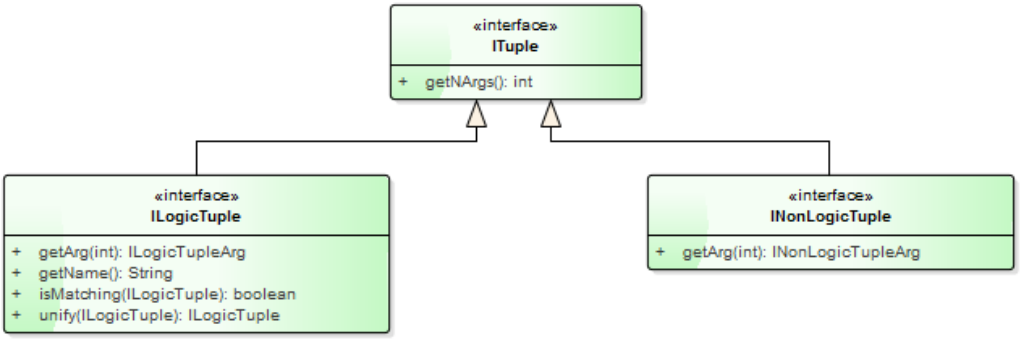
Chapter 2

TuCSoN as it will be

In this chapter, with reference to the previously presented literature about tuples and with the goal of maintaining the widest compatibility and transparency in mind, we will model, through UML class diagrams, the various kinds of tuples that TuCSoN will have.

2.1 Domain model

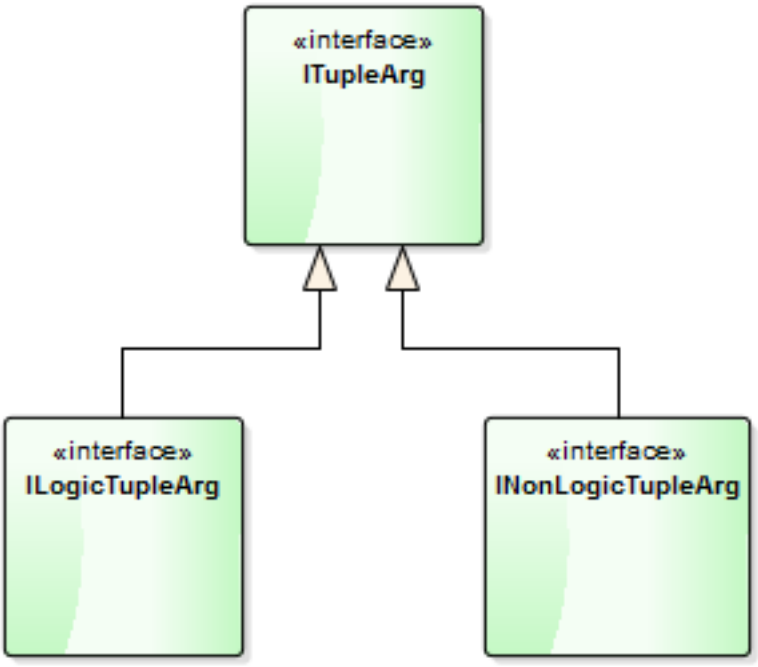
The current version of TuCSoN supports two different kinds of tuples: logic (*alice.logictuple*) and non logic (*alice.tuples.javatuples*). In order to maintain the widest compatibility and transparency, operations over tuples and templates that are common to all kind of tuples have been put in an upper-level interface, *ITuple*. Then we added *ILogicTuple* and *INonLogicTuple* interfaces as specializations of *ITuple*.



The *getNArgs* method returns the tuple arity while *getArg* returns an argument, given its index. The *getName* method returns the functor name (for logic tuples only).

2.2 Tuple arguments

Given the tuple model shown in the previous section, we defined a similar model for the tuple arguments. All tuple arguments are *ITupleArg*s. There are two main kinds of arguments: *ILogicTupleArg*s (for *ILogicTuple*s) and *INonLogicTupleArg*s (for *INonLogicTuple*s).

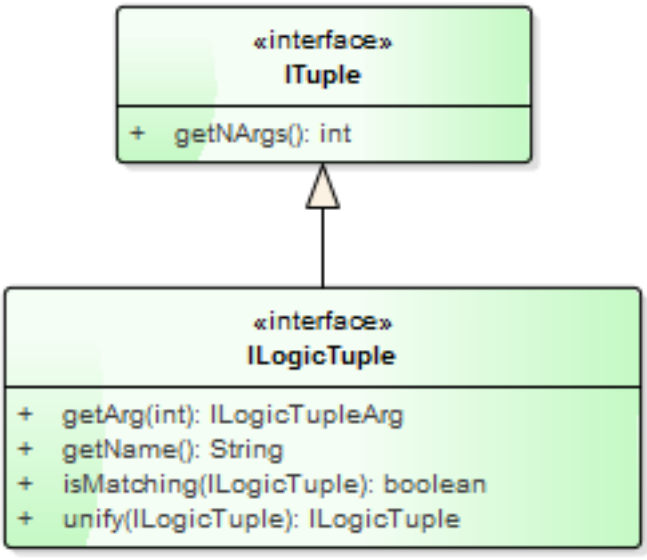


In the following, we'll extend this hierarchy introducing, for any kind of tuple, a different interface for:

- argument representing a *tuple value*,
- argument representing a *tuple variable*.

2.3 Logic tuples

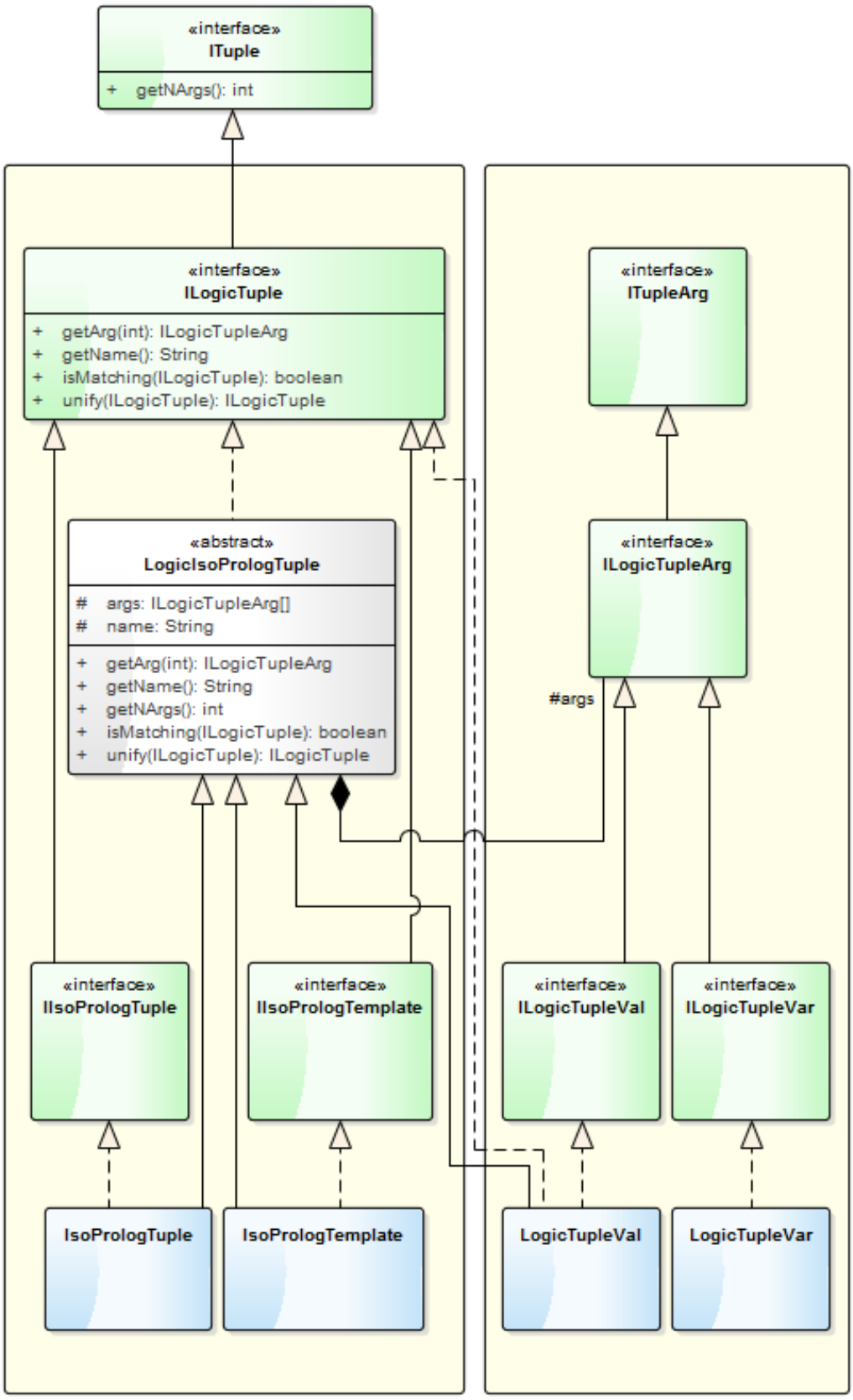
The current version of TuCSoN contains an implementation of logic tuples, which are modeled in package *alice.logictuple*. *ILogicTuple* interface has been designed to contain logic tuples operation declarations that are specific to this variant of tuples.



The semantics of the *isMatching* operation corresponds to the *match* method, already present in the *alice.tuplecentre.api.TupleTemplate* interface, while *unify* is the same as *propagate*.

2.3.1 Logic tuples and ISO Prolog standard

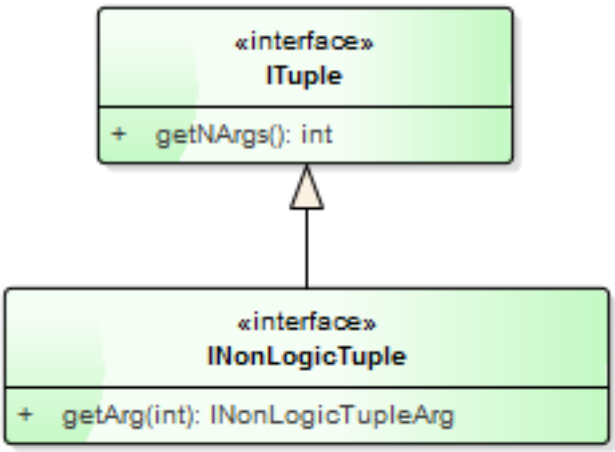
Because code that strictly conforms to the *ISO Prolog* core language is portable across ISO-compliant implementations and because *SWI-Prolog* and *GNU Prolog* are both fully compliant with *ISO Prolog* standard, we decided to implement an ISO-compliant version of logic tuples. We also wanted tuple and templates to be *ILogicTuple*s.


«interface»
ITuple
+ getNArgs(): int
«interface»
ILogicTuple
+ getArg(int): ILogicTupleArg
+ getName(): String
+ isMatching(ILogicTuple): boolean
+ unify(ILogicTuple): ILogicTuple
«interface»
ITupleArg
«abstract»
LogicIsoPrologTuple
args: ILogicTupleArg[]
name: String
+ getArg(int): ILogicTupleArg
+ getName(): String
+ getNArgs(): int
+ isMatching(ILogicTuple): boolean
+ unify(ILogicTuple): ILogicTuple
«interface»
ILogicTupleArg
#args
«interface»
IIsoPrologTuple
«interface»
IIsoPrologTemplate
«interface»
ILogicTupleVal
«interface»
ILogicTupleVar
IsoPrologTuple
IsoPrologTemplate
LogicTupleVal
LogicTupleVar

Note that in this way, different implementations of logic tuples are still possible by creating abstract classes that implement *ILogicTuple*.
Remark: in this diagram we have introduced two interfaces representing a specific value (*ILogicTupleVal*) and a variable (*ILogicTupleVar*) for logic tuples. *LogicTupleVal* and *LogicTupleVar* are concrete implementations of these interfaces. Moreover, we can notice that the *LogicISOPrologTuple* abstract class is composed by *ILogicTupleArg*s and the *LogicTupleVal* class implements the *ILogicTuple* interface. In this way, we can realize a sort of **nesting** of logic tuples within other logic tuples.

2.4 Non logic tuples

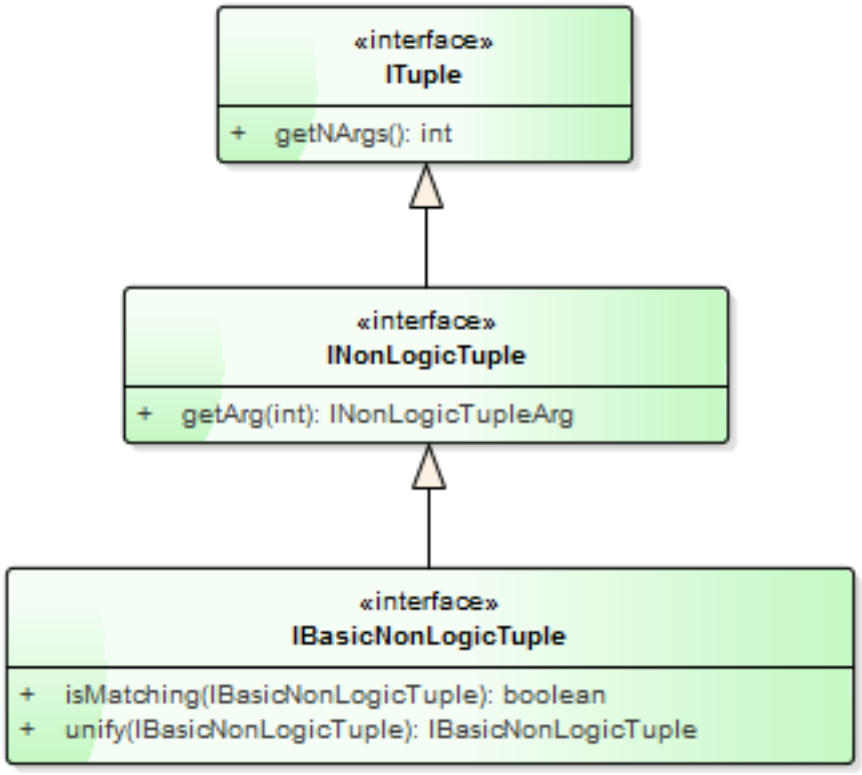
Non logic tuples have been designed around the original definition of tuple by Gelernter. [2]



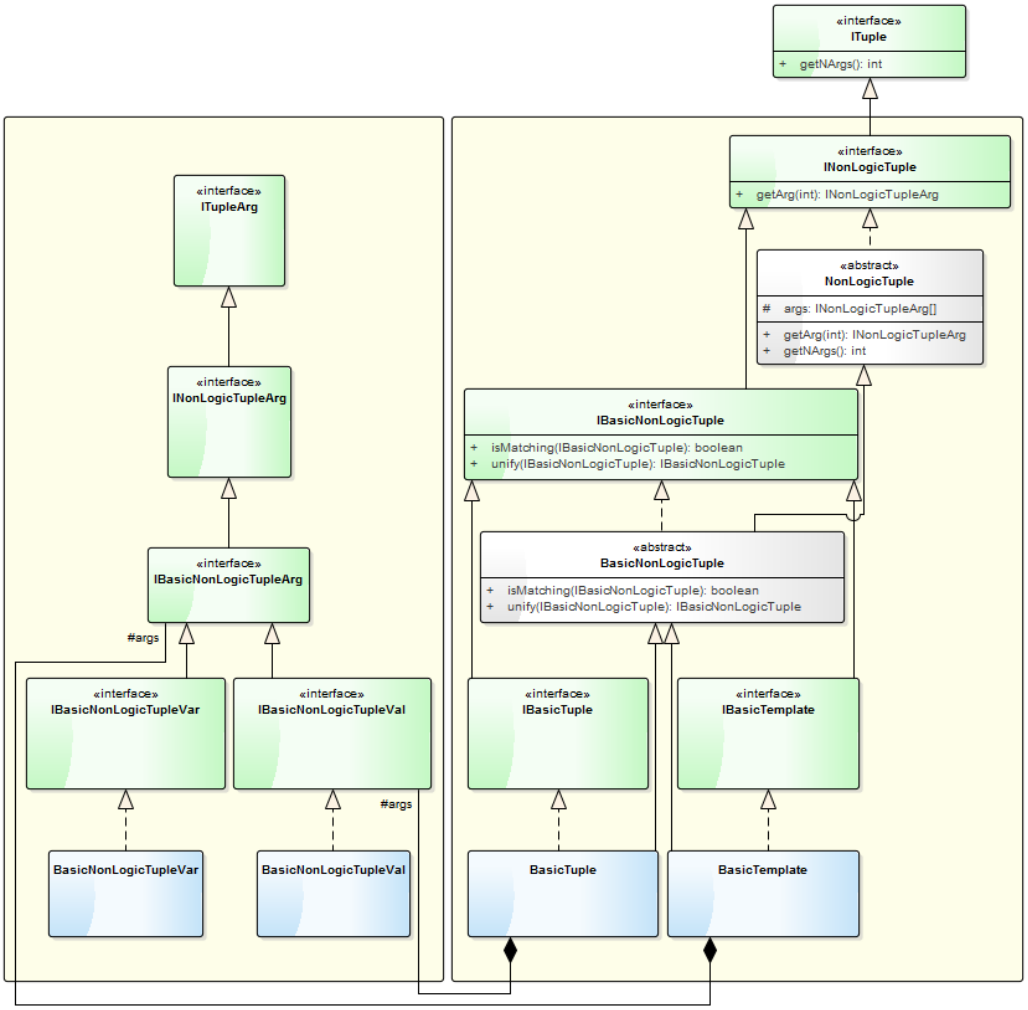
To be more precise, there are actually two categories of non logic tuples. **Basic** tuples are directly related to Gelernter's definition whereas **advanced** ones are an extension of basic tuples with the addition of **nesting** support. This is why *INonLogicTuple* interface doesn't contain *isMatching* and *unify* method declarations: they will be defined in *IBasicNonLogicTuple* and *IAdvancedNonLogicTuple*.

2.4.1 Basic tuples

Following on from what has been said above, *IBasicNonLogicTuple* defines *isMatching* and *unify* methods for basic non logic tuples.



The latter interface is then specialized with *IBasicTuple* and *IBasicTemplate* interfaces because we want tuples and templates to be kept apart. *NonLogicTuple* and *BasicNonLogicTuple* are abstract classes that implement the operations over tuples and templates that are defined in the interfaces mentioned above. Actually, *IBasicTuple* and *IBasicTemplate* interfaces are empty but in future revisions of the scheme they can be used to add operations over tuples that are not supported by templates and vice versa. *BasicTuple* and *BasicTemplate* implement their relative interfaces.

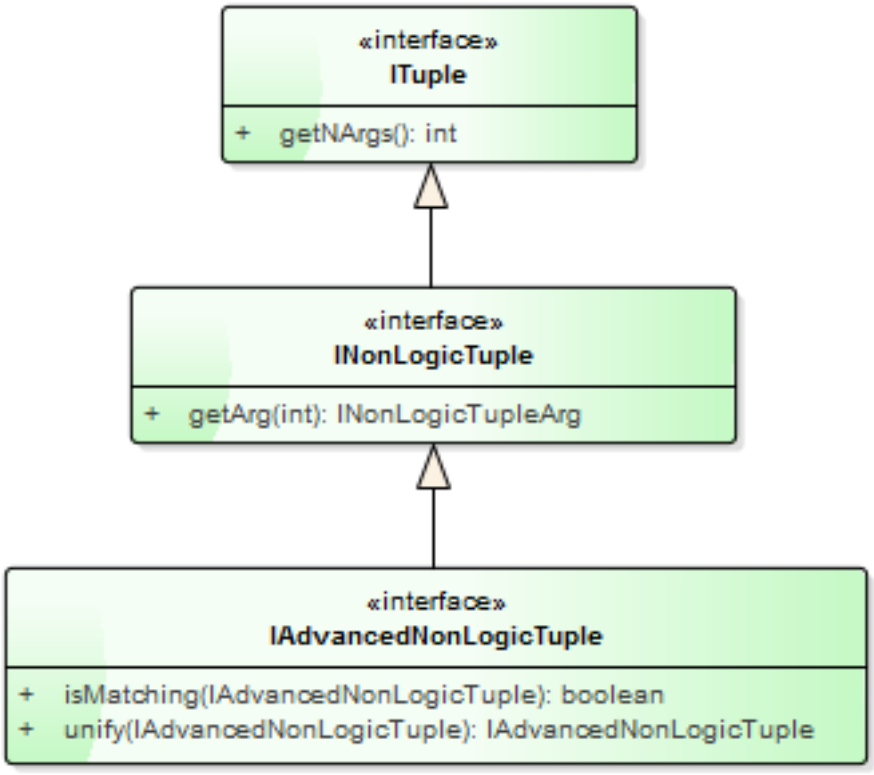


Remark: note that while *BasicTemplate*s are composed by *IBasicNonLogicTupleArg*s (i.e. *IBasicNonLogicTupleVar*s and *IBasicNonLogicTupleVal*s), *BasicTuple*s can only be composed by *IBasicNonLogicTupleVal*s. This was done because a template must have at least an argument of variable type.

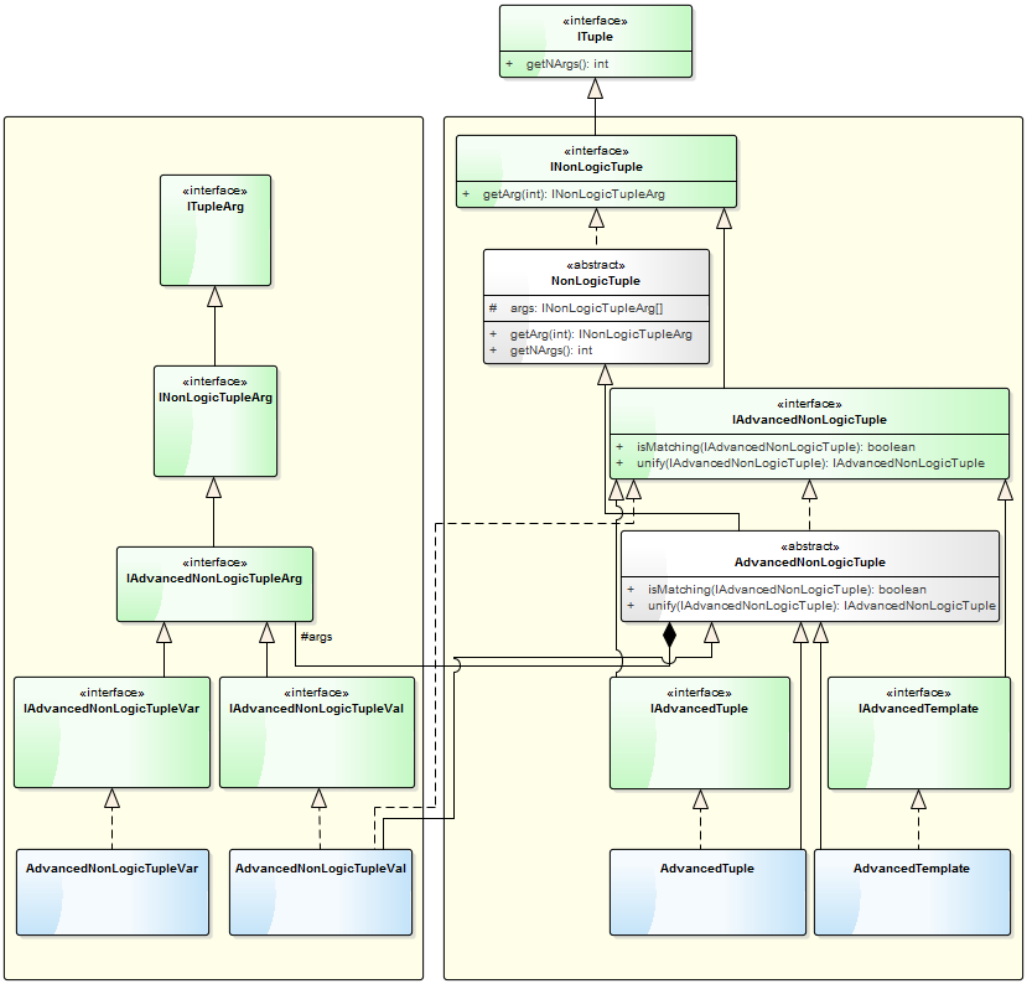
The hierarchy of arguments is similar to that of the logic tuples: we have a value (*IBasicNonLogicTupleVal*) and a variable (*IBasicNonLogicTupleVar*) of *basic* type.

2.4.2 Advanced tuples

Advanced tuples are similar to basic tuples with the addition of **nesting** support. These kinds of tuples are not present in the current version of TuCSoN. *IAdvancedNonLogicTuple* defines *isMatching* and *unify* methods for advanced non logic tuples.


«interface»
ITuple
+ getNArgs(): int
«interface»
INonLogicTuple
+ getArg(int): INonLogicTupleArg
«interface»
IAdvancedNonLogicTuple
+ isMatching(IAdvancedNonLogicTuple): boolean
+ unify(IAdvancedNonLogicTuple): IAdvancedNonLogicTuple

As can be seen in the following diagram, nesting support was added by making *IAdvancedNonLogicTuple*s be valid *AdvancedNonLogicTupleVal*s.

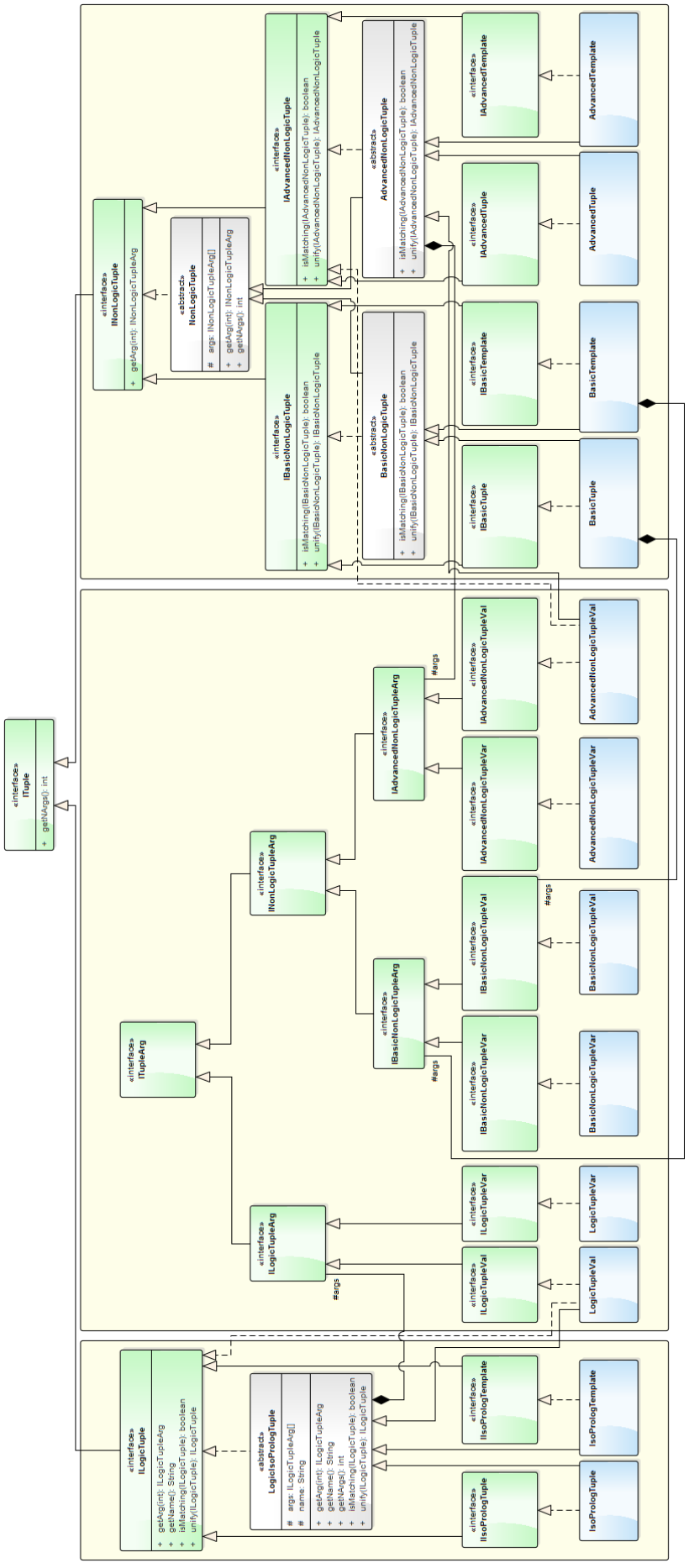


Remark: in this diagram, as in the case of logic tuples, we have added the composition relation. In fact, an *AdvancedLogicTuple* is composed by *IAdvancedNonLogicTupleArgs*. Note that, the *AdvancedNonLogicTupleVal* class implements the *IAdvancedNonLogicTuple* interface, and this is what makes **nesting** possible. We also have introduced two interfaces that represent a specific value (*IAdvancedTupleVal*) and a variable (*IAdvancedTupleVar*).

Chapter 3

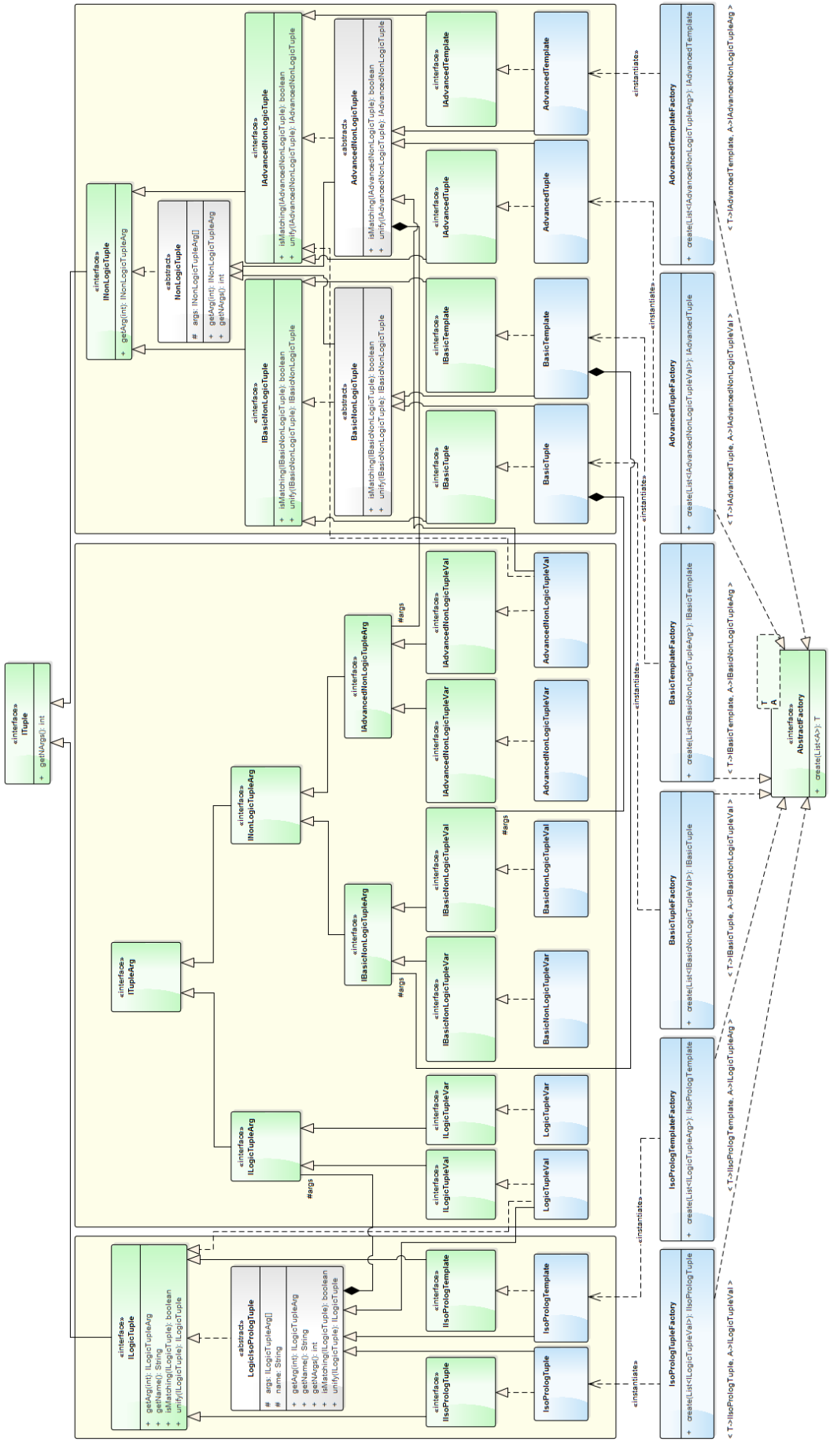
Domain model overview

This chapter summarizes all the concepts and features related to the tuple domain by means of an UML diagram: that provides an overall overview of the work done.


«interface»
ITuple
+ getNArgs(): int
«interface»
ITuplePrologArg
«interface»
ITuplePrologVal
«interface»
ITuplePrologVar
«interface»
ILogicTuple
+ getArg(int): ILogicTupleArg
+ getName(): String
+ isMatching(ILogicTuple): boolean
+ unify(ILogicTuple): ILogicTuple
«abstract»
LogicIsoPrologTuple
args: ILogicTupleArg[]
name: String
+ getArg(int): ILogicTupleArg
+ getName(): String
+ getNArgs(): int
+ isMatching(ILogicTuple): boolean
+ unify(ILogicTuple): ILogicTuple
«interface»
IIsoPrologTuple
«interface»
IIsoPrologTemplate
IsoPrologTuple
IsoPrologTemplate
«interface»
ILogicTupleArg
«interface»
ILogicTupleVal
«interface»
ILogicTupleVar
LogicTupleVal
LogicTupleVar
«interface»
ITupleArg
«interface»
INonLogicTupleArg
«interface»
IBasicNonLogicTupleArg
«interface»
IBasicNonLogicTupleVal
«interface»
IBasicNonLogicTupleVar
BasicNonLogicTupleVal
BasicNonLogicTupleVar
«interface»
IAdvancedNonLogicTupleArg
«interface»
IAdvancedNonLogicTupleVal
«interface»
IAdvancedNonLogicTupleVar
AdvancedNonLogicTupleVal
AdvancedNonLogicTupleVar
#args
«interface»
INonLogicTuple
+ getArg(int): INonLogicTupleArg
«abstract»
NonLogicTuple
args: INonLogicTupleArg[]
+ getArg(int): INonLogicTupleArg
+ getNArgs(): int
«interface»
IBasicNonLogicTuple
+ isMatching(IBasicNonLogicTuple): boolean
+ unify(IBasicNonLogicTuple): IBasicNonLogicTuple
«abstract»
BasicNonLogicTuple
+ isMatching(IBasicNonLogicTuple): boolean
+ unify(IBasicNonLogicTuple): IBasicNonLogicTuple
«interface»
IAdvancedNonLogicTuple
+ isMatching(IAdvancedNonLogicTuple): boolean
+ unify(IAdvancedNonLogicTuple): IAdvancedNonLogicTuple
«abstract»
AdvancedNonLogicTuple
+ isMatching(IAdvancedNonLogicTuple): boolean
+ unify(IAdvancedNonLogicTuple): IAdvancedNonLogicTuple
«interface»
IBasicTemplate
«interface»
IBasicTuple
«interface»
IAdvancedTemplate
«interface»
IAdvancedTuple
BasicTemplate
BasicTuple
AdvancedTemplate
AdvancedTuple

As a final iteration, we have introduced a series of concrete *factories* implementing the *AbstractFactory* interface, with the task of building an instance of a tuple or a template of a given family (ISOProlog, Basic and Advanced). With the use of factories, the creation of a tuple or a template is coupled with its right list of arguments, so we have avoided any risk of type mismatch.

In the following page, the complete UML diagram with the addition of the factories.


«interface»
ITuple
+ getNArgs(): int
«interface»
ITuplArg
«interface»
ILogicTuple
+ getArg(int): ILogicTupleArg
+ getName(): String
+ isMatching(ILogicTuple): boolean
+ unify(ILogicTuple): ILogicTuple
«abstract»
LogicIsoPrologTuple
args: ILogicTupleArg[]
name: String
+ getArg(int): ILogicTupleArg
+ getName(): String
+ getNArgs(): int
+ isMatching(ILogicTuple): boolean
+ unify(ILogicTuple): ILogicTuple
«interface»
IIsoPrologTuple
«interface»
IIsoPrologTemplate
«interface»
ILogicTupleArg
«interface»
ILogicTupleVal
«interface»
ILogicTupleVar
«interface»
INonLogicTuple
+ getArg(int): INonLogicTupleArg
«abstract»
NonLogicTuple
args: INonLogicTupleArg[]
+ getArg(int): INonLogicTupleArg
+ getNArgs(): int
«interface»
IBasicNonLogicTuple
+ isMatching(IBasicNonLogicTuple): boolean
+ unify(IBasicNonLogicTuple): IBasicNonLogicTuple
«abstract»
BasicNonLogicTuple
+ isMatching(IBasicNonLogicTuple): boolean
+ unify(IBasicNonLogicTuple): IBasicNonLogicTuple
«interface»
IBasicTuple
«interface»
IBasicTemplate
«interface»
IBasicNonLogicTupleArg
«interface»
IBasicNonLogicTupleVal
«interface»
IBasicNonLogicTupleVar
«interface»
IAdvancedNonLogicTuple
+ isMatching(IAdvancedNonLogicTuple): boolean
+ unify(IAdvancedNonLogicTuple): IAdvancedNonLogicTuple
«abstract»
AdvancedNonLogicTuple
+ isMatching(IAdvancedNonLogicTuple): boolean
+ unify(IAdvancedNonLogicTuple): IAdvancedNonLogicTuple
«interface»
IAdvancedTuple
«interface»
IAdvancedTemplate
«interface»
IAdvancedNonLogicTupleArg
«interface»
IAdvancedNonLogicTupleVal
«interface»
IAdvancedNonLogicTupleVar
IsoPrologTuple
IsoPrologTemplate
LogicTupleVal
LogicTupleVar
BasicTuple
BasicTemplate
BasicNonLogicTupleVal
BasicNonLogicTupleVar
AdvancedTuple
AdvancedTemplate
AdvancedNonLogicTupleVal
AdvancedNonLogicTupleVar
IsoPrologTupleFactory
+ create(List<ILogicTupleVal>): IIsoPrologTuple
IsoPrologTemplateFactory
+ create(List<ILogicTupleArg>): IIsoPrologTemplate
BasicTupleFactory
+ create(List<IBasicNonLogicTupleVal>): IBasicTuple
BasicTemplateFactory
+ create(List<IBasicNonLogicTupleArg>): IBasicTemplate
AdvancedTupleFactory
+ create(List<IAdvancedNonLogicTupleVal>): IAdvancedTuple
AdvancedTemplateFactory
+ create(List<IAdvancedNonLogicTupleArg>): IAdvancedTemplate
«interface»
AbstractFactory<A>: T
+ create(List<A>): T
T
A
«instantiate»
#args
< T->IIsoPrologTuple, A->ILogicTupleVal >
< T->IIsoPrologTemplate, A->ILogicTupleArg >
< T->IBasicTuple, A->IBasicNonLogicTupleVal >
< T->IBasicTemplate, A->IBasicNonLogicTupleArg >
< T->IAdvancedTuple, A->IAdvancedNonLogicTupleVal >
< T->IAdvancedTemplate, A->IAdvancedNonLogicTupleArg >

Conclusion

This project gave us the possibility to face the issues related to modeling, firstly, and then to the design of a main concept like a tuple, in a middleware based on a tuple centre as coordination media, like TuCSoN is.

A tuple can seem a simple concept at a first glance, but as you can see from the previously presented diagrams, its modeling is definitely not so trivial. Indeed, several hours of work, discussions and meetings with the responsible of TuCSoN were needed in order to formalize all the features of every kind of tuples.

We are proud of this work because we have obtained an abstract model of tuple satisfying all the desired requirements.

Bibliography

[1] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing, Vol. 12, No. 3, May, pp.251-284*, 1994.

[2] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[3] Andrea Omicini. On the semantics of tuple-based coordination models. pages 175–182, 28 February – 2 March 1999. Special Track on Coordination Models, Languages and Applications.

[4] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. 46:329–400, 1998.