Towards model-driven communications

Antonio Natali, Ambra Molesini

Abstract—In modern distributed software systems, the issue of communication among composing parts represents a critical point, but the idea of extending conventional programming languages with general purpose communication constructs seems difficult to realize. As a consequence, there is a (growing) gap between the abstraction level required by distributed applications and the concepts provided by platforms that enable communication.

This work intends to discuss how the Model Driven Software Development approach can be considered as a mature technology to generate in automatic way the schematic part of applications related to communication, by providing at the same time high level specialized languages useful in all the phases of software production. To achieve the goal, a stack of languages (meta-meta-models) has been introduced in order to describe – at different levels of abstraction – the collaborative behavior of generic entities in terms of communication actions related to a taxonomy of messages. Finally, the generation of platforms for communication is viewed as a form of specification of language semantics, that provides executable models of applications together with model-checking supports and effective runtime environments.

Keywords—Interactions, specific languages, meta-models, Model Driven Development.

I. INTRODUCTION

Today, many people might agree that in the production of software:

- applications should be the result of a mature software engineering (SE) process, to assure reproducibility and to allow the traceability from the code to design and requirements. The process itself should be represented with the purpose to make explicit knowledge – about the application (domain), design choices, etc. – that is often implicitly embedded in the code;

- after reading the code of a well structured software application one should recognize three macro-parts: a *general* part that can be shared among all the applications, a *specific* part that is peculiar to the application and a *schematic* part which is not general, but still reusable for different applications in the domain because it possesses the same systematics. This part should also be well recognizable, since based on specific design patterns [1] or pattern languages [2];

- the *architecture* is a key point for the software quality under at least three different point of views: *structure*, *behavior* and *interaction*.

Historically, a lot of attention has been focussed on the structural dimension, but in modern software systems the issue of interaction is going to become a critical point, even with reference to *communication*, that is the first, basic form of interaction. Many functional and non functional requirements

Alma Mater Studiorum - Università di Bologna, Italy. e-mail: {antonio.natali, ambra.molesini}@unibo.it

are directly related to communication; to capture relevant concepts. Service Oriented Architecture (SOA) [3], [4] and Web Services (WS) [4], [5] are proposing different approaches, e.g. by distinguishing between *orchestration* and *choreography* [6]. In fact, the communication style has important consequences on the overall architecture and on the single components of a software system: according to the communication patterns in which they are involved, application components can be modeled in very different ways, e.g. like processes, agents, actors, subjects, services, etc. The ubiquitous object oriented paradigm (oop) seems now more adequate to organize the schematic part rather that to express the conceptual space required by a modern software application.

In the concrete practise of SE, people tend to build distributed applications by exploiting advanced communication platforms such as Axis [7], Java Message Service (JMS) [8], Jade [9], etc. But an approach of this kind does not allow to overcome the growing gap that exists between the abstraction level required by the business logic and the concepts provided by platforms. Worst, the analyst and the designer have no reference language to reason about communication: even in the (improper) case in which the usage of a specific communication platform is assumed as a working hypothesis, it is quite uncommon that a platform exposes some formal model of the mechanisms it provides; therefore the analyst and the designer should resort to some sort of reverse engineering in order to find it. But it is quite improbable that people spend time for creating a workable model of a platform; therefore, the semantic gap is usually filled up by using oop to design and build subsystems and components that conceptually belong to the infrastructure rather than to the business level. In absence of a mature software development process, this job is repeated several times, each time in a different way.

In this scenario, the idea of extending conventional programming languages with general purpose communication constructs seems unwise and difficult to realize. But the Model Driven Software Development (MDSD) approach [10] could become a reference technology in this field, since model-to-model (M2M) mappers and model-to-code (M2C) transformers can continue to be part of the user-defined design, but with different scope and different life-time than business application code. In particular, the transformers could embed best design practices to build in automatic way the intrinsic systematic of the schematic part of an application.

Because a meta-model is a way to describe the abstract syntax of a language, a meta-model for communications becomes a way to introduce a language that can help in all the phases of the software development process: in the *analysis phase*, to express the interaction logic implied by the problem; in the *design phase* to define the overall architecture of a software system; in the *implementation phase* to improve

code readability and system modularity. This happens because each term used to specify a communication pattern can be associated with a precise semantics, that can help in finding incoherent or inconsistent system designs.

This work intends to follow a sort of *language oriented programming* [11] style, by introducing a meta-model as the abstract syntax of a language – called `contact` (Subsection III-A) – to describe the *communication* between generic entities called *subjects* in terms of actions related to the elements of a *taxonomy of message types.*

The discussion will be limited to very basic forms of communication, since `contact` intends to constitute just a starting point for a possible application of `MDSD` in this field. It can be viewed as a sort of domain-specific language whose semantics is in some sense *programmable* in order to meet specific application needs. The semantics of the language is given by a chain of `M2M/M2C` transformers whose end point is `Maude` [12] code, that allows us to promote model checking and model execution.

The main goal is to specify the semantics of communication actions in such a way that this specification can be used also as an effective operational support. To achieve this goal, a software factory has been defined around a stack of meta-languages actually composed by an intermediate language – called `medcl` (Subsection III-B) – and a low-level language—called `corecl` (Subsection III-C). Thus, the formal definition of the meaning of the `contact` language is performed in two steps: the first step consists in mapping a `contact` action into one `medcl` action; the second step maps each `medcl` action into a message structure and a sequence of `corecl` actions. The `M2M` mappers are used also as the first step of a production line that ends with a `M2C` transformer that builds a real implementation for a `OSGi` [13] `Java` environment.

So, the paper is structured as follows. Section II presents an overview of the communication issues, while Section III introduces the stack of meta-languages that have been defined and developed. Section IV presents the meaning of the `contact` language defined in terms of a chain of `M2M` mappings that transform a `contact` sentence into a sequence of low-level communication primitives for a shared space. Section V discusses how these mappings and `M2C` transformations can be used not only to specify the language semantics, but also to provide an effective implementation according to a layered architecture which can exploit different platforms and different supports for communications e.g. network protocols like `TCP`, `UDP`, `HTTP` etc. Conclusions follows in Section VI.

II. Overview

From the end-user point of view, the attention is focused on the high level, logical aspects of communication. As an example, let us consider, a very simple system made by two subjects – named `s1` and `s2` – in which:

- `s1` produces a document and asks subject `s2` to make comments on that document. Subject `s1` continues its local work (if any) without blocking; however it expects to receive some indication that its request has been accepted by `s2`;

- `s2` accepts the request of `s1` and performs the requested job. When this job is terminated, it sends a message to inform any interested observer on the result of the operation;
- after that `s1` becomes aware that `s2` has acquired its request, it is ready to receive the message emitted by `s2`, in order to perform some specific operation related to the result of the evaluation of the document.

Since this kind of specification is rather ambiguous, it should be rephrased in a more formal language. Using the `contact` language it could be possible to say that:

- `s1` *asks* to `s2` the *invitation* named `evalReport`.
- `s2` *accepts* the *invitation* `evalReport`; when the related job is terminated, it *emits* the *signal* named `evalReportDone`.
- after that `s2` has *acquired* the acknowledgment (`ack`) to the invitation `evalReport`, it can *sense* the *signal* named `evalReportDone`.

Verbs like "ask", "accept", "emit", "sense", are used to denote high level communication actions semantically related to entities such as "invitation" and "signal" that are specialized versions of the *Message* general concept. Thus, a basic vocabulary for message-based interaction is related to a set of message types as shown in TABLE I.

TABLE I
MESSAGES AND RELATIVE ACTION

Message Type	Sender	Receiver
Dispatch	forward	serve
Event	raise	perceive
Signal	emit	sense
Invitation	ask acquireAck	accept replyAck
Request	demand acquireResponse	grant replyResponse

The `contact` specification could be expressed in the following concrete syntactic form:

```
ApplicationSystem name=sys0 ;
// --- Messages
Invitation evalReport;
Signal evalReportDone;

// --- Subjects
Subject writer;
Subject expert;

// --- Actions
writer ask evalReport to expert;
writer sense evalReportDone;

expert accept evalReport;
expert emit evalReportDone;
```

The system named `sys0` is made by two concrete subjects named `writer` and `expert`, that communicate using an *invitation* and a *signal*. This specification can be viewed as a *constraint* on the collective behavior of the subjects, i.e. a specification of *what* they expect from each other, without

saying *how* their internal behavior is organized; in particular, the order of execution of the communication actions is not described here.

In the reminder of this section, the semantics issues related to different communication patterns are highlighted in Subsection II-A, while Subsection II-B introduces the idea of using MDSD techniques and tools to face semantic-related problems.

A. The issue of semantics

Each term used to specify a communication pattern must be associated with a precise semantics, that can help not only in finding incoherent system designs, but also in automating (relevant part of) the implementation of the underlying interaction support system.

Let us consider, as an example, the case of *invitation* (see TABLE I). An invitation is defined as a message that can be *asked* from a inviting-subject to $ND \geq 1$ receivers, with the expectation to receive back $0 \leq NA \leq ND$ *acknowledgment* (ack) messages. An invitation can be *accepted* by $NR \geq 1$ subjects, each *replying with* an ack to the requestor. Thus an ack is a message that can be sent (*replied to*) an inviting-subject with reference to a specific invitation, while *acquireAck* is the name given to the action of receiving the ack.

Again, this is a rather informal specification, that lives open several questions:

- if ND>1, how many ack does the sender expect?
- after an *ask* action, does a subject wait until some ack is acquired? or when exactly NA ack has been acquired?
- if a subject decides to wait for an ack, does it wait forever?
- a subject that accepts an invitation replies:
    - as soon as possible, i.e. when the invitation is acquired at the receiver site?
    - after that the invitation has been elaborated?
    - under explicit control of the application?
- what happens if the invitation cannot reach any receiver?

If an answer is given to all of these (and perhaps some other) questions, then the semantics of the idea of invitation is fixed. The problem is that two different applications can consider conceptually and pragmatically adequate two different answer sets. Thus, it is unwise to design and implement a communication support by freezing just one semantics; on the other hand, a support offering a set of API that implement all the possible semantics may be difficult to build and perhaps confusing to use. These considerations can justify the choice of many programming languages (and UML [16] too) to provide low-level communication mechanisms only, by leaving to the software designer the job to define a suited "communication protocol" for each specific situation posed by the application.

B. Model driven supports

An approach based on meta-models and MDSD can help to overcome the problems related to the definition of the semantics of high-level communication actions . In fact, the meta-model that defines the contact language provides the abstract syntax of a high level communication language, while

model-to-model (M2M) mappings and/or model-to-code (M2C) transformations can be used to define the semantics and the implementation of the language on different platforms.

Using M2M mappings or M2C transformations to obtain low-level code means to make the semantics "programmable", since this approach promotes fast (and controlled) refactoring of the schematic part of a software system. But it is also possible to maintain the semantics stable and to introduce new, more specialized, concepts. For example, *Invitation* could be conceived as another abstract class do to derive from it sub-classes like InvitationOneToOne and InvitationOneToMany.

In any case, the point is to find a way to specify the semantics of communication actions in such a way that this specification can be also be used as an effective operational support. To achieve this goal, a software factory has been defined around a stack of meta-languages.

III. THE STACK OF THE META-LANGUAGES

In order to define the meta-languages, the Xtext [14] – which is part of *openArchitectureWare / Eclipse Modeling Framework* (EMF [15]) – notation has been used.

Xtext can be used to describe both the concrete and the abstract syntax of a language. The Xtext framework supports the automatic generation of a parser, an Ecore [15] meta-model and a specific text editor for Eclipse IDE. Ecore is the EMF implementation of the Object Management Group (OMG) Meta Object Facility specification (MOF) [15].

At the top of the stack there is the high-level language contact while at the bottom of the stack there is a low-level language called corecl. The basic idea captured by corecl is that, in order to communicate, two entities must share a *medium* that provides support to transmit or acquire information, e.g. some shared memory space, a communication network, etc. Between the two there is an intermediate language called medcl that defines a set of abstract types of communication operations.

The overall situation is summarized in TABLE II.

TABLE II
(META)-META-LANGUAGES

Entity	Instance of	Meaning
User defined program	contact	Language used to specify communication at application level.
MedCl	medcl	Abstract high-level communication language.
Message	msgcl	Language used to define messages and logical channels.
LindaLikeCl	corecl	Abstract low level communication language.

The language called msgcl defines the structure of the message exchanged at intermediate level, while MedCl – instance of medcl – and LindaLikeCl – instance of corecl – are abstract languages whose meaning will be discussed in Section IV; since these two languages are meta-models, medcl and corecl can be classified as meta-meta-models. The next sub-sections introduce respectively the

`contact` language (Subsection III-A), the `medcl` language (Subsection III-B), the `corecl` language (Subsection III-C) and the `msgcl` language (Subsection III-D).

A. The high-level language `contact`

The aim of the `contact` language is to express two main aspects of the architecture of a software system: the structural dimension, i.e. the set of parts (subjects) that compose the system and the communication dimension, i.e. the way used by subject to exchange information. This subsection presents the communication-related sub-language that, as said in the overview, is based on an ontology of messages, as shown in Fig. 1.
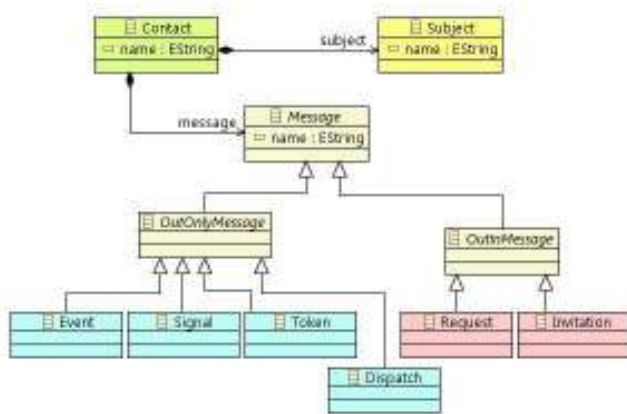

Fig. 1.   Message ontology.

Messages are classified in two main categories: the (abstract) class of *out-only messages* which represent messages that a subject sends without any expectation to receive information directly related to these messages and the (abstract) class of *out-in messages* which represents messages that a subject sends with the expectation to receive some information directly related to the message sent. As concrete forms of messages it is possible to distinguish between out-only messages such as *events*, *signals* or *dispatches*, and out-in messages such as *invitations* and *requests—token* will be discussed in Section VI.

The abstract syntax of communication actions is shown in Fig. 2 and Fig. 3; the informal description of their meaning is reported in TABLE III, while the formal definition of the meaning will be discussed in Section IV.

B. The intermediate language `medcl`

This language is a meta-meta-model (depicted in Fig. 4) that defines a set of abstract types of communication operations.

1) `OpToPereceiveInfo`: this is the class of the input (`OpIn`) operations that allows a subject (attribute `worker`) to acquire information emitted by any subject.

2) `OpToAcquireInfo`: this is the class of the `OpIn` operations that allows a subject (`worker`) to acquire information that subjects emit with reference to one or more specific receiver subjects.
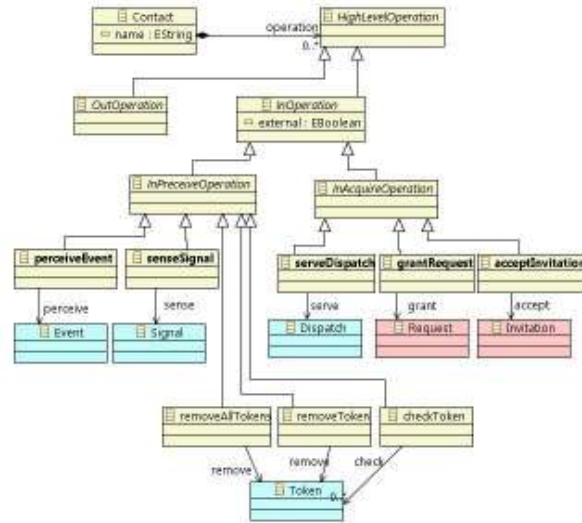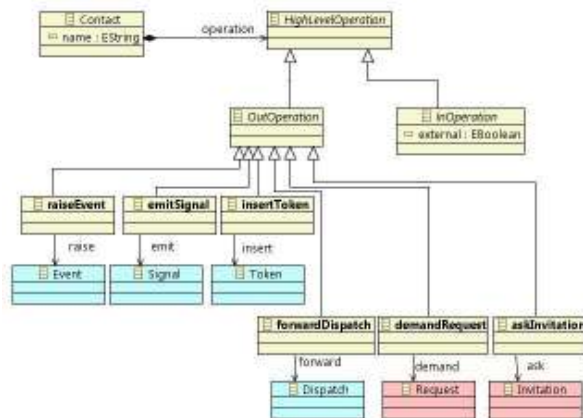

Fig. 2.   Contact input operations.

Fig. 3.   Contact output operations.

3) `OpToAcquireManyInfo`: this is the class of the `OpIn` operations that allows a subject (`worker`) to acquire - as an atomic action - information from several subjects.

4) `OpToEmitInfo`: this is the class of the output (`OpOut`) operations that allow a subject to transmit information without any notion of receiver and without any expectation to receive any reply information from the perceiver subject (if any).

5) `OpToSendToAReceiver`: this is the class of the `OpOut` operations that allows a subject (`sender`) to transmit information to a specific receiver (`dest`).

6) `OpToSendToMany`: this is the class of the `OpOut` operations that allows a subject to transmit information to a set of receivers.

The abstract class `IntermediateOperation` includes properties common to all the operation types:

- `name`: this attribute represents the name of the operation;

TABLE III

INFORMAL DEFINITIONS

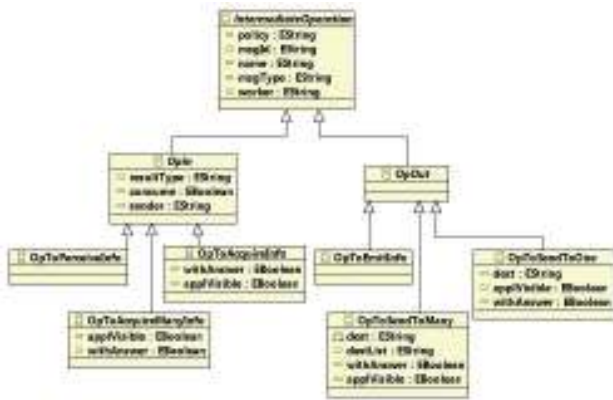| Message Type	Meaning
Event (OutOnlyMessage)	a message that a subject can *raise* to change the state of the environment (world); an event can be *perceived* by N≥0 - time uncoupled - subjects each acting in a way independent from the others
Signal (OutOnlyMessage)	a message that a subject can *emit* with the expectation that it can be broadcasted by some transmission medium; a signal can be *sensed* by N≥0- interested, time uncoupled - subjects each acting in a way independent from the others.
Dispatch (OutOnlyMessage)	a message that a subject can *forward* to a specific receiver-subject, with the expectation that it will *serve* it.
Invitation (OutInMessage)	a message that a subject can *ask* to ND≥1 receivers, with the expectation to *acquire* NR≥0 acknowledgments (ack); a receiver will *accept* an invitation and will *reply* with an ack to the sender.
Request (OutInMessage)	a message that a subject can *demand* to ND≥1 receivers, with the expectation to *acquire* NR≥0 responses; a receiver will *grant* a request and will *answer* to the sender with a *response*.

Fig. 4.   The `medcl` meta-model

- `msgType`: this attribute represents the category of message handled by the operation;
- `msgId`: this attribute represents the identifier denoting a specific message.

The meaning of the other attributes is the following. If the attribute `consume` of an `OpIn` operation is `true` information is removed from the transmission medium; otherwise, information is only logically consumed, i.e. the `worker` reads it, just once. If the attribute `sender` of an `OpIn` operation is `not null`, then the receiver will handle messages sent from that specific source only. If the `withAnswer` attribute of an `OpIn` operation is `true`, then the `worker` must send back some information to the transmitter. If the `withAnswer` attribute of an `OpOut` operation is `true`, then the `worker` must acquire some information from the receiver. If the `applVisible` attribute is `true`, the operation implies that the received answer must be explicitly handled by the application, i.e. by user-defined code.

C. The basic language `corecl`

The `corecl` language is a meta-meta-model that introduces a set of low-level concepts as represented in Fig. 5:

1) The medium used to transmit or acquire information is a *shared space*.
2) Information transmission is obtained by writing data (messages) in the shared space (operation class `CoreWrite`) while information acquisition is obtained by removing data from the shared space, either logically (`CoreRead`) or physically (`CoreConsume`, `CoreConsumeMany`). When a data is logically consumed, it is no more available for the subject that acquires it, but remains available for the other subjects.
3) Communication primitives based on shared memory usually provide a support to store unsatisfied requests to acquire/perceive data and to inform a subject when some required data becomes available; without this support, subjects are obliged to perform *busy form of waiting* in order to acquire information. The availability of such a support is expressed through the boolean attribute `resumeSupport`.

The choice to make reference to a shared space as the basic medium for communication is motivated by the idea that communication between two or more subjects is logically possible because the subjects share a common world. Moreover a shared space basically provides *many-to-many* communication for free, but does not exclude to support also *point-to-point* forms of interaction. In any case, the shared space is our reference point for expressing the operational semantics of communication actions; other kinds of medium can be used for real implementation, as it will discussed in Section V.
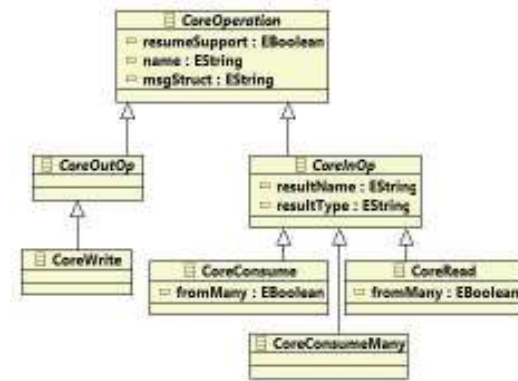

Fig. 5.   The `corecl` meta-model

D. The message language `msgcl`

The data to be transmitted and acquired (*messages*) must own a proper structure in order to allow a wide range of communication patterns. The definition of the `msgcl` meta-model reported in Fig. 6 states that, besides the message content, a message should also contain data useful:

1) to specify a logical *point-to-point* connection between subjects, since a shared-space basically provides a *many-to-many* communication model. This kind of data is represented by the `channelId` attribute (see Section IV).

2) to determine the subject that wants to transmit or to acquire information. This kind of data is represented by the `workerName` attribute.

3) to determine the kind of message to be transmitted or acquired. This kind of data is represented by the `msgId` attribute.

4) to distinguish between two messages in which the values of the attributes defined so far are identical. This kind of data is represented by the `msgNum` attribute.
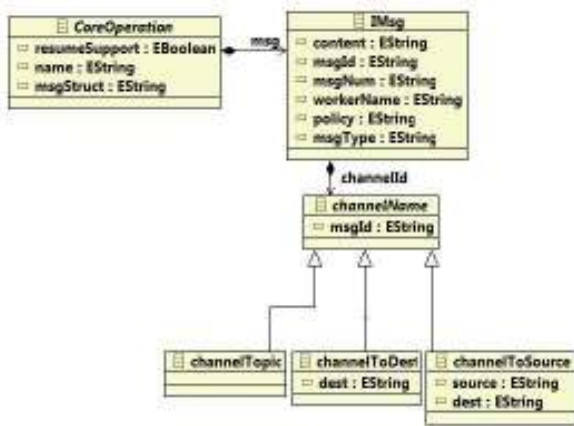

Fig. 6. The `msgcl` meta-model

IV. SEMANTICS VIA M2M AND M2C GENERATION

The formal definition of the meaning of the `contact` language is performed in two steps, as shown in Fig. 7.

The first step consists in mapping a `contact` action into one `medcl` action; the second step maps each `medcl` action into a `msgcl` structure and a sequence of `corecl` actions. The `M2MContactToMed` transformer is a model-to-model mapper that translates a specification written in `contact` into an instance of the `medcl` meta-language. The `M2MMedToCore` transformer is a model-to-model mapper that translates a specification written in `medcl` into an instance of the `corecl` meta-language. The ultimate output of the transformation chain is a model that specifies the behavior of each `contact` operation as a sequence of `corecl` operations. This model can be finally translated by a `M2C` generator in the implementation code with reference to some target platform, e.g. `Java` (see Subsection V-A).

The model transformation performed by each `M2M` mapper implicity defines an abstract language – as already summarized in TABLE II – which can be taken as a reference for an implementation based on layers, as it will discuss in Section V. Once defined, the `M2M` transformers on the meta-meta-models can be used to obtain the semantics of the high-level communication operations written in `contact`, as shown in Fig. 8.

To give an example, let us consider the `contact` specification given in the Section II. The output of the `M2M-ContactToMed` mapping is an `Ecore` model whose standard representation is an `XMI` [17] file; to make such a model more readable, a `M2C` transformer can be defined so to produce some pseudo-code like:

```
(OpToSendToOne)
 writer ask msgId=evalReport
 msgType=Invitation
 to expert withAnswer=true

(OpToPerceiveInfo)
 writer sense
 msgId=evalReportDone
 msgType=Signal
 consume=false

(OpToAcquireInfo)
 expert accept msgId=evalReport
 msgType=Invitation
 consume=true withAnswer=true
```

The `M2M-ContactToMed` transformer has mapped each `contact` operation into one `medcl` action with the same name. Thus each user program (a in Fig. 8), which is an instance of the `contact` meta-language, becomes (is transformed into) a model (b in Fig. 8) instance of the `medcl` meta-language. This model implicitly defines the abstract syntax of a language, whose explicit definition is given by the language `MedCl` of TABLE II. If this model (b) is given in input to the `M2M-MedToCore` transformer, then a new model (c in Fig. 8) instance of the `corecl` meta-language is obtained; it can be described by the following pseudo-code:

```
//Invitation
writer ask Invitation evalReport to expert
------------------------------------------
out(expert_evalReport(
        writer,evalReport,"content",1))
IMsg obtainedMsg =
in(writer_expert_evalReport(
        expert, evalReport, "ack" ,1))
------------------------------------------

expert accept Invitation evalReport
------------------------------------------
IMsg replyMsg = in( expert_evalReport(
        ANY, evalReport,M,N) )
out(replyMsg.msgEmitter()_expert_evalReport(
    expert,evalReport,ack,replyMsg.msgNumStr()) )
------------------------------------------

//Signal
writer sense Signal evalReportDone
------------------------------------------
IMsg sensedMsg = rd(evalReportDone(
        ANY, evalReportDone,M, _))
------------------------------------------
```
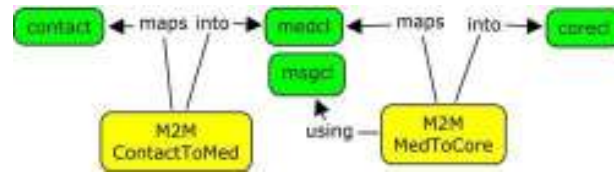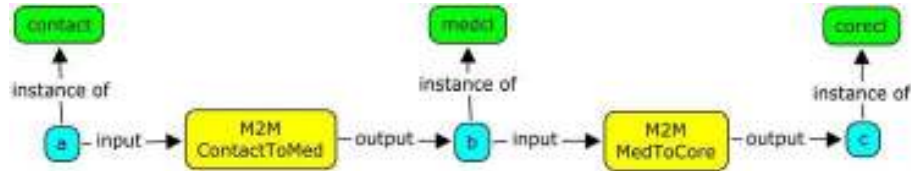
Fig. 7. Mapping between the languages

Fig. 8. Code generation through `M2M` mapping

TABLE IV
INFORMAL MEANING

corecl primitive	Description
out(String M) (CoreWrite)	The worker subject informs the shared space IS that it intends to execute a `CoreWrite` of a message M. IS stores the message into the `message-sequence` and looks into its local `request-queue` (see primitive `in`). If there is some subject `ws` waiting for a message that can be *unified* with M, then IS informs `ws` that it can retry to acquire the message.
IMsg in(String Q) (CoreConsume)	The worker subject informs the shared space IS that it intends to execute a `CoreConsume` of a message. IS looks into the `message-sequence`: if it finds a message M that can *unify* with Q, IS removes M from the set and gives M to the worker subject. Otherwise, IS stores the request into the `request-queue`.
IMsg rd(int n,String Q) (CoreRead)	The worker subject informs the shared space IS that it intends to execute a `CoreRead` of a message. IS looks into the `message-sequence`: if it finds a message M whose sequence number is higher than n that can *unify* with Q, IS gives M to the worker subject. Otherwise, IS stores the request into the `request-queue`.

```
expert emit Signal evalReportDone
-----------------------------------------
out(evalReportDone(
       expert,evalReportDone,"content",_))
-----------------------------------------
```

This model implicitly defines the abstract syntax of a language, whose explicit definition is given by the language `LindaLikeCl` of TABLE II. In fact, the operations `out`, `in`, `rd` represent low-level actions similar to those of `Linda` [18]; their informal meaning is summarized in TABLE IV.

The formal semantics of `LindaLikeCl` primitives is given by a `M2C` transformer from `corecl` to `Maude`, while a `M2C` transformer from `corecl` to Java provides a workable implementation of the primitives. Section V will return on this point. The reminder of this section is dedicated to discuss

the concept of `channel` (Subsection IV-A), by starting from the pseudo-code above. In particular Subsection IV-B and Subsection IV-C discuss in some detail the case of invitations and signals, while Subsection IV-D give details related to the other operations. Finally Subsection IV-E explains how to obtain supports for checking and executing application models.

A. Channels and low-level messages

The messages exchanged through the shared space are strings built either in a static way by a `M2M`/`M2C` transformer or in a dynamic way by some operator. The string is written by using the syntax of Prolog and includes the attributes defined by the language `msgcl`:

`channelId(workerName,msgId,content,msgNum)`

Messages are expressed in Prolog syntax not only for the sake of simplicity and clarity: also our real, Java-based implementation of the shared space is based on a Prolog engine in order to exploit unification for accessing message content (see Section V). From a logical point of view, information related to point-to-point messages (dispatches, invitations, requests) flow along specific *channels* related to a particular type of application message (`medcl::msgType`) and to some particular subject (the receiver, the sender or both). If two subjects declare to send a message with `medcl::msgId`=m to a same subject of name `medcl::worker`=r, conceptually they make reference to the same logical channel with `msgcl::channelId`=r_m. Other channel identifiers can be introduced as required. The channels represent the possible communication opportunities that exist between the subjects. Since the content of a message includes the name of the sender, a receiving subject can dynamically acquire the opportunity to communicate with a subject of which it has no static knowledge.

B. Invitations

In the case of one-to-one invitation, it is necessary to capture also the idea of a reply channel from the receiver to the sender.

The approach is to define the `msgcl::channelId` of the reply channel by combining the (unique) name of the sender with the (unique) name of the receiver and with the invitation identifier `medcl::msgId`. These rules are reflected in the structure of the `ask` and `accept` operations of the example above. The `ask` operation is translated into a sequence of two primitives: an `out` of the invitation `evalReport` on the channel of name `expert_evalReport`, and then an `in` on the channel of name `writer_expert_evalReport`;

Also the `accept` operation is translated into a sequence of two primitives: an `in` from `ANY` sender of invitation `evalReport` on the channel of name `expert_evalReport` and then an `out` operation of the `ack` reply on the channel of name `writer_expert_evalReport`. The prefix `writer` of the reply channel is the name of the sender which is included in the message sent by the `ask`. This information is given by the `msgEmitter()` operation executed on the result (`replyMsg`) given by the `in` at the receiver site.

An important element in the semantics of invitation is the fourth argument of the message (`msgNum`) that represents a message (unique) identifier at subject level. In fact, the `msgNum` associated to the message by the `out` performed by an `ask` is used by the sender to complete the specification of the structure of the reply message to be obtained. At the receiver site, the `msgNum` included in the received message, given by the `msgNumStr()` operator, is used to mark the reply message itself.

This semantics does not fix the local behavior of the sender: if the sender performs the `in` immediately after the `out`, then it blocks until the `reply` is obtained. But the sender could also continue its work by executing other operations in between, including another invitation to the same subject: the message number will specify the reply that the sender wants to obtain at a given point of its computation.

C. Signals

Signals are classified as `OutOnlyMessages`. Interaction based on a signal does not require that the sender has knowledge of the receivers nor it implies any feedback that the signal has been sensed by some subject. Since there is no need of point-to-point channels, as `msgcl::channelId` the name of the signal is used. The `emit` operation is mapped into an `out` while the `sense` operation is mapped into a `rd`.

The operation of sensing a signal is a `medcl::OpToPereceiveInfo` operation. Thus, it has the following meaning: when a subject receives from the interaction space the information that a signal is available, the signal is only logically removed form the interaction space but it physically remains available for other subjects.

D. Other operations

Events are similar to signals. Requests can be handled in a very similar way to one-to-one invitations: however, differently from an `ack`, which can be sent by the infrastructure, a *response* must be sent by the application code, while the sender

is now interested to handle at application level the content of the response.

One-to-many invitations (or requests) present a more complex semantics. When the same invitation (or request) is sent to many ($N \geq 1$) subjects, the sender can acquire $NA \geq 0$ `ack`; one possible semantics is that the *ask* operation is considered terminated when *one* of the `ack` has been obtained. Let us suppose that a subject `s1` asks invitation `evalReport` to two subjects `s2` and `s3`:

```
s1 ask evalReport to s2,s3;
s2 accept evalReport;
s3 accept evalReport;
```

The result of the `M2M-ContactToMed` mapping is:

```
s1 askInvitation
    evalReport to [s2,s3] withAnswer=true
```

Now the result of the `M2M-MedToCore` mapping could be:

```
out( s2_evalReport(
    s1,evalReport, "content", 2 )

out( s3_evalReport(
    s2,evalReport, "content", 2)

IMsg replyMsg = in( s1_ack_evalReport(
            ANY,evalReport,ack(1),2 ))
```

The sender subject `s1` now expects to obtain an `ack(1)` message from some not well identified subject (`ANY`): this message means that all the expected `ack` (just one, in this case) has been received.

This pattern is necessary since the `resumeSupport` attribute of the shared space has been set to `true`. If we generate as many `in` as the receivers, and the number of expected `ack` is less then the number of the receivers, then the sender could wait forever for an `ack` that will never be sent. In a real working system, the subject that generates the `ack(1)` message can be another subject that observes the message traffic on the shared space or the shared space itself.

E. Executable models

The model produced by `M2M-MedToCore` (see Fig. 7) can be given in input to a `M2C-CoreToTarget` transformer to obtain code for a specific run-time platform. For the specification of language semantics, our target platform is `Maude`, a reflective language and system supporting both equational and rewriting logic specification and programming.

Rewriting logic is a logic of concurrent change that can naturally deal with state and with concurrent computations. Distributed systems can be modeled in `Maude` as multisets of entities, loosely coupled by some suitable communication mechanism. An important case is object-based distributed systems in which the entities are *Maude objects*, each with a unique identity, and the communication mechanism is based on *messages* that, in our terminology, have the *dispatch* semantics.

The usage of `Maude` as target language allows us to achieve three main goals:
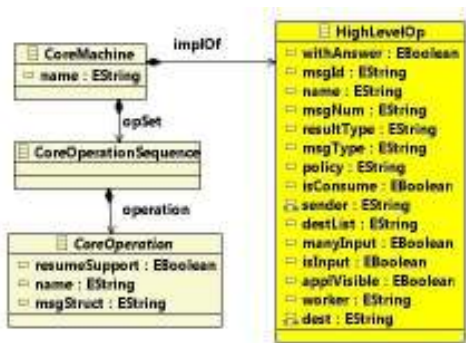
Fig. 9. `corecl` extension

- to give a precise semantics to each low-level construct;
- to provide a support for model checking;
- to provide a support for model execution.

These are very important aspects that it is not possible to further discuss here. The interested reader can see [12].

V. FROM SEMANTICS TO ARCHITECTURE

The `M2C` generator at the end of the transformation chain of Fig. 8 is designed so to avoid the "macro-expansion" of a high-level operation into a sequence of target instructions, since this approach would lead to code difficult to read and understand. Rather, the generator builds code with reference to a run-time support which is a `Maude` implementation of the basic communication medium. This support is in its turn built by a `M2C` transformer that takes in input the `LindaLikeCl` introduced in TABLE II, which is defined (in `XText`) as follows:

```
CoreMachine LindaLike
coreConsumeOp in IMsg withResume;
coreReadOp rd answerRd IMsg; //no resume
coreReadOp rdw answerRd IMsg  withResume;
coreWriteOp out withResume;
coreConsumeManyOp inMany
   answerInMany "Vector<IMsg>" withResume;
coreReadOp selectOneFromMany
   answerRd IMsg fromMany withResume;
```

To achieve the goal, the `corecl` model has been extended – as showed in Fig. 9 – to include links to the high level operation implemented by a sequence of `corecl::CoreOperation`.

The `M2C` transformer can then "compile" each `contact` operation into a sequence of calls to the *primitives* of a low-level machine layer exposing a conventional `API`. The `M2C` transformer is designed so to produce modular code, by following design criteria that are in a large extent independent of the target language. Since our goal is to keep concrete implementation aligned with the specification of semantics, these criteria are applied not only to the generation of `Maude` code, but also to the generation of implementation code for specific operative platforms like `JVM`, `CLR`, `.NET`, `J2EE`, etc. Currently, our reference platform for real implementation is

the `OSGi` platform Equinox [19] over a `JVM`; thus the system provides also a `M2C-CoreToJava` generator that invokes the primitives of a basic communication medium that implements the `API` defined by the `ILindaLike` interface below, using our Java interpreter of Prolog, called `tuProlog` [20].

```
public interface ILindaLike {
IMsg in(String Q) throws Exception;
IMsg rd(int n,String Q) throws Exception;
IMsg rdw(int n,String Q) throws Exception;
void out(String M) throws Exception;
Vector<IMsg> inMany(
   Vector<String> tokens) throws Exception;
IMsg selectOneFromMany(
   int n,Vector<IMsg> Q) throws Exception;
}
```

The same strategy is adopted to define a high-level support layer for `contact`, as described in Subsection V-A. In the reminder of this section, Subsection V-B faces the problem of the interaction between a subject and the underlying communication support, while Subsection V-C tackles the problem of managing user-defined answers; finally, Subsection V-D explains how to move from shared space to other communication supports, with particular reference to network protocols.

A. Layered systems

Translating each `contact` operation into a sequence of low-level primitives like `out`, `in`, `rd` is not enough to produce readable code: our goal is to generate application code by making explicit reference to the available set of high-level communication operations. To achieve this goal, the `MedCl` language of TABLE II has been given as input to a `M2C` generator (called `MedclPlatform`) that builds the run-time support for `contact`. The architecture of the software system is now naturally structured according to the *layers* pattern [2], in which it is possible to distinguish as many layers as the meta-languages. The application code can assume the form that follows.

```
// --- Subject writer ----
IAcquireOneReply answer = support.ask(
   "writer", "evalReport", M,"expert")
...
if( answer.replyAvailable() ){
IMsg reply=answer.acquireReply();
...
IMsg sensedMsg = support.sense(
   "writer", "evalReportDone")
}

// --- Subject expert ---
IMsg m = support.accept(
   "expert", "evalReport")
...
 support.emit(
   "expert", "evalReportDone", "content")
```

B. *Subject behavior*

Besides generating calls to low-level communication primitives, the `MedclPlatform` generator must take important decisions about the nature of the *subjects* (that could be conceived as objects, processes, agents, etc.) and on the way adopted by a subject to interact with the communication primitives themselves. If the target environment is a platform like `Java`, then the shared space can be structured as a conventional object and the interaction between a subject and the medium can be based on conventional procedure calls. When the target environment is `Maude`, the communication medium can be represented by a `Maude` object, but the interaction with the medium must be based on `Maude` *messages* (*dispaches* in our terminology); thus the medium is conceptually a *process*.

Another important aspect regards the issue of blocking or non-blocking operations. High-level operations related to *out-in messages* are translated into a sequence of `LindaLike` primitives that usually ends with an `in` related to some reply/answer from the receiver. In imperative run-time environments, this usually means that the sender subject could be blocked by a primitive that represents the transmission of a message. But the decision to wait or not for a reply/answer should be explicitly taken by the subject and not by the platform; thus, the `MedclPlatform` generator implements the operations of class `medcl::OpOut` having the attribute `withAnswer` set to `true`, so they return an object that can be used by a sender to acquire the reply when it needs it. One interface for this kind of objects is:

```
public interface IAcquireManyReply {
boolean numOfReplyExpected();
boolean numOfReplyReceived();
IMsg acquireReply(int n) throws Exception;
}
```

The operation `numOfReplyExpected` returns the number of the reply expected by an operation `withAnswer`, while the `numOfReplyReceived` operation gives the number of the reply received so far. The operation `acquireReply` blocks the caller until the `n-th` reply-message is available,

C. *Application level answers*

Input `medcl::OpIn` operations having attributes `withAnswer` and `applVisible` both set to `true`, must delegate to user-defined code the production of an answer to be sent to the caller. To achieve this goal, these operations return an object that implements the following interface:

```
public interface IMessageAndContext {
  void replyToCaller(String m)
                throws Exception;
  IMsg getReceivedMessage();
}
```

The operation `getReceivedMessage` returns the input message, while the `replyToCaller` operation embeds all the implementation details related to the communication from the current (server) subject and the specific subject that sent that particular input message.

D. *From shared space to protocols*

The idea of using a shared space as basic logical reference for communications has been adopted having in mind minimal conditions to make communication possible: the presence of a (stateful) shared world surrounding a set of subjects.

With reference to the real world, the concept of shared space recalls the idea of (basic) wireless communication, where the transfer of information occurs "in the space" without the use of electrical conductors or "wire". However, when "real code" is written, subject communication is usually realized on top of network protocols like `UDP`, `TCP`, `HTTP` and many software designers could believe more convenient to structure a software system around the libraries implementing these protocols.

But many software designers could also recognize the advantage of making the application code independent of a specific communication technology. In this case the layered architecture allows us to exploit network protocols without changing the high-level description discussed so far. To achieve this goal the `contact` language has been extended, so that each `contact::OpToAcquireInfo` can optionally define some concrete communication support. For example, our `expert` could declare to accept invitations by using `TCP`:

```
expert accept evalReport support=TCP
  [ host="localhost" port=1860 ] ;
```

Moreover, the system configuration part is divided into a set of execution *contexts*. The subjects associated to each context can be loaded and run on different physical nodes of a network.

The functionalities of the generated schematics are extended by putting more "intelligence" in it. In particular each context always owns a local shared space, that is used to hide the usage of protocol-specific supports when a message must be exchanged on the network among different contexts.

If some subject within a context `C1` presents a `contact::InOperation` specification including the declaration of a network protocol, a *receiver subject* is created in `C1` to acquire information according to that protocol. In the case of connection-based protocol like `TCP`, the *receiver* waits for a connection and creates a *RequestHandler* subject for each connection accepted. The *RequestHandler* is the `TCP`-equivalent of the channel that defines the logic connection between the source and the destination. This internal subject interacts with the destination subject defined by the application by substituting itself to real caller; this goal is achieved by simply modifying the emitter *name* attribute of the received `msgcl` message and forwarding the new message into the local shared space. In this way the behavior (and the code) of a receiver subject is made independent of any protocol used to acquire information from the external world.

If some subject within a context `C1` presents a `contact::OutOperation` to a receiver that does not belong to context `C1`, a *ClientRequest subject* is created. The task of this subject is to transmit information according to the protocol specified by the pair receiver-message.

The architecture of the platform takes the typical form described by a `Broker` [2] pattern, with the advantage that the

design choices are now reified into the structure of a generator that allows us to automate the creation of the (non trivial) schematic part of our distributed application.

VI. CONCLUSIONS

The growing semantic gap between the abstraction level required by modern distributed applications and the concepts provided by general purpose supports for communication is actually filled up by software developers, which are often called to design and build (again and again) components that belong to an infrastructure rather than to the business level.

The `MDSD` approach could become a reference technology in this field, since `M2M` and `M2C` transformers can continue to be part of the user-defined design, but with different scope and different life-time than business-related application code. In particular, the transformers could embed best design practices, including pattern languages, to build in automatic way cumbersome and repetitive parts of the application.

The meta-model introduced in this work defines a language (`contact`) that allows the description of very basic forms of communication. This is intended to be just a starting point for a possible application of a language oriented programming style, so to handle high-level communication actions as a sort of domain-specific language whose semantics is in some sense "programmable" according to specific application requirements. This language is put at the top of a stack that is currently composed of other two languages; as a consequence, it is possible to generate the schematic part of an application according to a layered logical architecture: the layer related to the intermediate language (`medcl`) allows keeps the application code independent of the medium, while the layer related to the low level language (`corecl`) keeps the communication medium independent of the implementation platform, by facilitating the mapping of `contact` on media of different type, e.g. a communication network rather than a shared space.

Although its simplicity, `contact` can be used to express in an explicit, systematic and open-ended way concepts that are actually hidden into evolute communication supports like `Jade` [9] or `JMS`. Together with `msgcl` and `corecl`, it can be extended to capture more evolute forms of communication, like for example those proposed by `FIPA` [21] or to help in abstracting service-oriented features provided by configurable communication networks currently described using *Network Description Languages* (`NDL`) based on `RDF` [22]. The slogan coined by James Hendler for the Semantic Web can be adopted also here: *a little semantics goes a long way*. For example the message ontology discussed in this work has been easily extended so to introduce the concept of `Token` (see Fig. 1, Fig. 2 and Fig. 3): a `Token` is new `contact::OutOnlyMessage` related to very basic communication actions called *insertToken, removeToken checkToken* and *removeAllTokens*. The peculiarity of *removeAllTokens* is that a set of tokens can be consumed – within a specific execution context – as an *atomic action* expressed as an instance of `medcl::OpToAcquireManyInfo` and implemented by using a `corecl::CoreConusmeMany` operation. In this

way it is possible to exploit Petri-Nets [23] not only as a formalism to model a software system, but also as a means to concretely build it.

The approach adopted in this work puts in the foreground the subjects, and leaves in the background the concept of communication channel. In fact, communication `channels` are deduced from a `contact` specification, and used as a logical device to drive the generation of the infrastructure with reference to different target communication supports. This approach seems adequate to describe systems in which subjects observe interactions that are visible and use the observations to determine their state with respect to the others. More complex forms of choreography, e.g. situations in which subjects have shared knowledge and explicit notion of a shared state, will probably require extensions similar to those proposed in `WS-CDL` [24] to ensure that state is known globally, or at least between subject roles, when needed.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.
[2] F. Buschmann and D. C. Henney, Kevlin Schmidt, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing.* John Wiley & Sons, 2007, vol. 4.
[3] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, 2005.
[4] D. K. Barry, *Web Services and Service-Oriented Architectures: The Savvy Manager's Guide.* Morgan Kaufmann Publishers, 2003.
[5] W3C, "Web services architecture home page," http://www.w3.org/TR/ws-arch/.
[6] C. Peltz, "Web services orchestration and choreography," *Computer*, vol. 36, no. 10, pp. 46–52, Oct. 2003.
[7] Axis, "Axis home page," http://ws.apache.org/axis/.
[8] Sun, "JMS home page," http://java.sun.com/products/jms/.
[9] JADE, "Home page," http://sharon.cselt.it/projects/jade/, 2000.
[10] T. Stahl and M. Volter, *Model-Driven Software Development.* Jhon Wiley & Sons, Ltd, 2005.
[11] M. P. Ward, "Language oriented programming," http://www.cse.dmu.ac.uk/ mward/martin/papers/middle-out-t.pdf.
[12] Maude, "Maude home page," http://maude.cs.uiuc.edu/.
[13] OSGi, "OSGi home page," http://www.osgi.org/Main/HomePage.
[14] Xtext, "Xtext home page," http://www.eclipse.org/Xtext/.
[15] EMF, "EMF home page," http://www.eclipse.org/modeling/emf/.
[16] UML, "Home page," http://www.uml.org/.
[17] XMI, "XMI home page," http://www.omg.org/technology/documents/formal/xmi.htm.
[18] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan. 1985.
[19] Equinox, "Equinox home page," http://www.eclipse.org/equinox/.
[20] "tuProlog at SourceForge," http://tuprolog.sourceforge.net.
[21] FIPA ACL, "Fipa aclhome page," http://www.fipa.org/repository/aclspecs.html.
[22] RDF, "RDF home page," http://www.w3.org/RDF/.
[23] J. Peterson, *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.
[24] WS-CDL, "WS-CDL home page," http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

Antonio Natali Antonio Natali is – since 1990 – full Professor in the Department of Electronic, Informatic and Systems (DEIS) of the Alma Mater-University of Bologna. He is active in the filed of computer engineering since October 1974, teaching courses on Programming Languages, Foundations of Informatics and Software Engineering. He is co-author of more than one hundred articles, papers and three books and has participated to several European, national and regional projects. In August 2008 Antonio Natali received the IBM Faculty Award (https://www-304.ibm.com/jct09002c/university/scholars/facultyawards/) a worldwide competitive program, to foster collaboration and promote courseware and curriculum innovation. The research activity of Antonio Natali is centered on software systems and software engineering, with particular reference to software architectures, programming paradigms and tools.

Ambra Molesini Ambra Molesini received her laurea magistrale degree in computer science engineering in October 2004 and her PhD in computer science engineering in 2008, both from the Alma Mater Studiorum - University of Bologna. During the final examination, the committee also proposed the award of "Doctor Europaeus". Currently, she has a research grant at the DEIS in the context of the project "Methodologies and Processes for the engineering of complex software systems".

She is currently researching on several topics in the Software Engineering and Agent-Oriented Software Engineering (AOSE) fields and particularly on: design methodologies, multi-agent systems (MAS) infrastructures, methodologies' and infrastuctures' metamodels, flexible approaches for the composition of software development processes, software architectures and architectural styles for MAS, interaction engineering, Model Driven Software Development.

She has written over 15 articles on agent-oriented systems, software engineering, software architectures and architectural styles published in international conferences and workshops. She has organised a special track at 24th Annual ACM Symposium on Applied Computing (AOMS@SAC 2009) and at 25th Annual ACM Symposium on Applied Computing (AOMIP@SAC 2010); she has been and currently is also a member of the Program Committees of different conferences and workshops and Co-chairs of the IEEE FIPA DPDF Working group.