1 The Project proposal

1(a) State-of-the-art and objectives

With the advent of web technologies and the proliferation of programmable and inter-connectable devices, we are faced today with a powerful and heterogeneous computing environment. This environment is inherently parallel and distributed and, unlike previous computing environments, it heavily relies on communication. It therefore calls for a new programming paradigm which is sometimes called communication-centered. Moreover, since computation takes place concurrently in all kinds of different devices, controlled by parties which possibly do not trust each other, security properties such as the confidentiality and integrity of data become of crucial importance. The issue is then to develop models, as well as programming abstractions and methodologies, to be able to exploit the rich potential of this new computing environment, while making sure that we can harness its complexity and resolve its security vulnerabilities. To this end, calculi and languages for communication-centred programming must be security-minded from their very conception, and make use of specifications not only for data structures, but also for communication interfaces and for security properties. In addition, the need of programming and maintaining *eternal*, highly decentralised systems (e.g., those considered by the FP7-FET initiative *Forever Yours*) is emphasising the requirement of reliable and dependable applications. However, the forms of verification, mainly static, we are used to tend not to be enough anymore. Therefore, there is a strong demand for designing forms of runtime verification.

If we adopt the perspective of the programmer, this means being able to write computational units able to adapt to different contexts, implying the need of flexible specifications depending, for example, on how much a unit is trusted by the environment executing it, or on the resources offered to it. From the system perspective, this requires writing software coordinating and orchestrating units which are not guaranteed to fully follow the protocols they claim to respect.

These new needs can be observed at every level of the design and implementation of the computational units. In particular, at the level of software development, modern applications tend to be multi-threaded, sometimes distributed, concurrent and interactive. Application components are discovered and attached dynamically at runtime and they are often asked to cooperate with components whose behaviours can vary over time, can be stochastic or can be just unexpected.

Given this amount of evidence, it is easy to speculate that the trend towards fragmented, distributed, heterogeneous, dynamic systems will increase at a steady pace. Unfortunately, the complexity in programming these systems is doomed to increase similarly, and it is clear that currently available methodologies, techniques and tools for the development of correct, dependable, resilient software are largely inadequate. They are not able to cope with situations where not all the elements of a picture are under control, for instance when the code of some computational units is not known. Moreover, they are not suited to make it possible the development of software that adapts its specifications to the new requirements of a dynamic context, whose rules can change over the time, possibly in a stochastic way. In a word, they do not allow programmers to write the certified, self-adapting and autonomic code that the market will require soon.

Type theories need to be rethought in order to cope with these new requirements. While the development of systems operating in stochastic environments, or operating according to real-time constraints, has been supported by formal methods such as model checking, there is little work on type theories for such systems that can guide our research on typing computational units with time dependent and/or stochastic behaviours.

The PI has worked essentially on type theories and in the present project she will lead a multi-disciplinary team, collecting competencies from programming language theory system security, program correctness and stochastic model checking: this is essential to obtain the synergy needed for developing the next generation of formal methods based on type theories for the specification, design, implementation, verification, deployment, evolution, orchestration and reuse of the computational units of the future.

Types as the SALT of programming The notion of *type* in logic was introduced by Bertrand Russell

[71] as a way to eliminate the paradoxes (showing the inconsistency) of Gottlob Frege's naive set theory. The basic feature of the notion of type, that is shared, although in quite different ways, by all type theories, is, on the one hand, that of restricting the terms of an underlying calculus in order to avoid meaningless terms and, on the other hand, that of representing the behaviour of the underlying computational entity at a higher level of abstraction. During the last century, types have become standard tools both in logic, especially in proof theory (see [41] and [48]), and in programming languages [66].

Intersection types can be considered as a primitive form of behavioural types for their ability of classifying untyped terms according to their functional behaviour ([25]). As the work by the PI has proved, intersection types are a powerful tool to connect term syntax and their denotational semantics. In [26] it is investigated for the first time how filter models can be seen as a concrete presentation of a large class of models including Scott D_∞ models, where types play the role of compact points of domain theoretic λ-models, plainly corresponding to finite approximations to the denotation of terms that they type. Filter models have paved the way to a number of studies by the PI and others concerning the relation among syntax and semantics of the type free λ-calculus: [38] on F-semantics; [27] on λ-theories; [37] on the approximation theorem; [78] on λ-trees; [39] on infinitary terms; [3] on filter models as D_∞ models in the light of Abramsky's work [1]; [36] on preorders over intersection types; [4] on easy terms and related issues; [34] on a characterization of sets of terms with similar computational behaviours. Some recent work has also shown that intersection types can be used to define logical models also in the case of object calculi [30, 79] and of Parigot's $\lambda\mu$-calculus [77], which includes (a form of) continuations.

In a similar vein, also union types have been considered by the PI, both in the context of pure λ-calculus [8] and in that of its parallel and nondeterministic extensions [33]. The interest in having union types is better understood by looking at them as a form of abstract interpretation. This is how they have been employed by Buneman and Pierce in [18] which, together with Reynolds' usage of intersection types in [70] and Kfoury's and Well's work on expansion variables [21], is an example of how theoretical results about intersection and union types can be used in practice.

The real utility of intersection and union types for programming has been proved by Giuseppe Castagna and Véronique Benzaken through the development of the $\mathbb{C}$Duce project. $\mathbb{C}$Duce is a modern XML-oriented functional language. Distinctive features of $\mathbb{C}$Duce are a powerful pattern matching, first class functions, overloaded functions, a very rich type system (with arrow, sequence, pair, record, intersection, union, difference type constructs), precise type inference for patterns and error localisation, and a natural interpretation of types as sets of values, according to the semantic subtyping approach [40, 24]. It is enriched also with some important implementation aspects: in particular, a dispatch algorithm that demonstrates how static type information can be used to obtain very efficient compilation schemas.

The key difference between standard types (including intersection and union types) and *behavioural types* we will briefly survey in the following is highlighted by process execution. In fact, the reduction of processes preserves standard types, while behavioural types reduce together with the processes increasing the information they provide about the process continuation. In other words, if a process P reduces to a process Q, then a standard type of P is also a standard type of Q, while a behavioural type of P reduces to a behavioural type of Q. This dynamism of behavioural types allows a finer-grained analysis of computations, in particular they have been used for abstracting communication-centered computations.

The papers introducing behavioural types are [49, 76]. More precisely, [49, 76] propose *session types*, that is, types which control communications between parallel threads by supporting lightweight descriptions of protocols. The possibility for protocol participants to *delegate* their works to other participants has been added in [50] by enhancing the expressivity of session types.

A renewed interest in behavioural types springs from the proposal of [52], where CCS-like processes are used to type π-calculus processes, allowing us to capture precisely interleaving and characteristic properties like deadlock and liveness [55, 56]. Session types in particular have been made more expressive by enriching them with many constructs and they have been developed for various languages and calculi; for an overview see [32]. While session types describe protocols from the single viewpoints of the

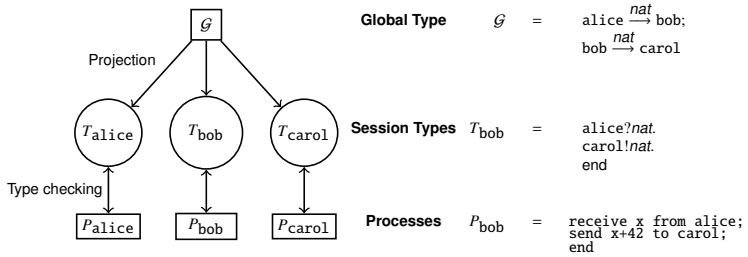

Figure 1: Global types and multi-party sessions in a nutshell.

participants, *global types* give the choreography of the whole protocols. Global types were introduced in [51] for multi-party sessions and in [31] for sessions with roles. [23] presents a new, streamlined language of global types whose features and restrictions are semantically justified. The relation between global types, session types and processes is illustrated by Figure 1. The idea is to specify behaviours globally and locally, and that these two levels are linked together by a projection mapping. The processes then must be type checked against session types, ensuring in this way that they comply with the whole protocol description given by the global types.

Weigh the SALT! Quantitative interactions inspire Type Systems The modelling and analysis of complex systems is increasingly reliant on the interplay between qualitative and quantitative factors. For example, in distributed algorithms the symmetry that can arise between computational units can be broken by the use of randomisation of some of the units' choices; in the analysis of biological systems, many phenomena are naturally modelled as being stochastic; in embedded systems, the efficient use of power can mean the difference between success and failure of a product in a competitive marketplace; and in safety-critical systems, the timeliness of the reactions of the system can mean the difference between life and death of the system user. The interplay of qualitative and quantitative factors requires an adequate support both from the field of formal modelling and analysis techniques and from the field of programming languages.

A number of formalisms have been proposed for modelling stochastic interactions between components, some of them based on process algebras [47, 53, 67, 46] and others on rewrite systems [57, 64]. In both cases most of the effort has gone in the definition of suitable semantics, in showing the expressive power by codifying significant examples, and in providing verification tools. Type systems, enforcing invariants on computations, can be used to help in managing the complexity of systems, by providing suitable abstractions not only on the application domain, but also on the behaviours of the components. To this extent, considering the CLS formalism [10] to specify systems' compartmentalization, in [6, 9, 35, 13] types characterise the elements and/or the computational rules that may or may not be contained in different compartments. Type-checking, and the definition of a typed operational semantics ensure that, during the evolution of the system, compartments preserve their properties. In [14] types are used to support a more flexible stochastic semantics for CLS in which strategies of reductions different from the law of mass action may be implemented. This is done by defining a typed semantics that takes into account the type of the context of the reduction to compute the probability of the execution of reductions.

The modelling and analysis of complex stochastic systems is supported by a number of tools. We first mention the model-checking tools PRISM [58] and MRMC [54], which have been applied to a large number of case studies, from network protocols to biological systems. Furthermore there is a wide range of tools developed for simulating the quantitative evolution of specific language models. In particular, the TSCLS simulation tool incorporates type abstractions to deal with stochastic evolution of biological systems.

Numerous formalisms have been introduced for the specification of real-time systems. In particular, timed automata [5] is a widely-accepted automata-based formalism for modelling timed systems. Timed automata have also been used as the basis of compositional design methods [29, 28]. The model-checking

tool UPPAAL [11] is widely-used for the verification of correctness properties of timed-automata models of real-time systems, and has been extended to obtain the tool ECDAR for the compositional development of timed systems [28]. We also note that each of PRISM, MRMC and UPPAAL have been extended to include quantitative information on costs or rewards that can be accumulated during system execution (such as accumulated energy usage). Finally, timed process algebras have been introduced in [68, 45, 82, 7], and formalisms which encompass timed and stochastic issues, combining aspects of both automata-based formalisms and process algebra, have also been presented [17].

1(b) Methodology

The core objective is the development of a new generation of typed languages for the description of computational units interacting with each other and of environments hosting the interactions. To this aim, we draw inspiration from the frameworks recently developed for the verification of session-based protocols, described in Section 1(a). The present approach is based fundamentally on four assumptions that hinder its application to the scenarios we aim to address:

- the topology of a system is fixed or severely constrained, the number of participants (or of roles) and the nature and sequence of interactions are pre-determined;
- a system is analysed in isolation with respect to the outside world, *all* participants in a session have been type-checked against the corresponding session types and therefore behave accordingly, *all* participants join the interaction at some pre-determined time (typically, when the session is opened and at fixed configurations);
- the type verification is done at compile time and its result is on-off: either the interaction is acceptable or not. In most cases, an acceptable communication could be achieved, even if less appealing, by restricting partner behaviours;
- unexpected, stochastic, or time-dependent behaviours are not considered.

Despite these limitations, we claim that the architecture in Figure 1 contains, in a primitive form, all the ingredients we need for the modelling, design and verification of new generations of interacting systems. The realization of our goals will be guided by upgrading our conceptual metaphors as follows:

Session $\Rightarrow$ *Interaction environment* *Global type* $\Rightarrow$ *Norms* *Session type* $\Rightarrow$ *Behavioural type*

An *interaction environment* is a delimited scope defined by a set of *norms* where *components/applications* can communicate. Unlike a session, the environment is open in that it accepts other computational units, provided that their (declared) behavioural type complies with the norms of the environment.The behavioural type attached to a computational unit describes a *contract* that can be used for checking whether the component can play a role within the environment, for negotiating an agreement between the component's objectives, the ones of the other components in the environment and the environment's norms, for monitoring the observable behaviour of the component, and for blaming the component if its observable behaviour mismatches its type. Apart from the fact that new components may join the environment, the behavioural type of components already in the environment may change (reflecting changes in the components). Moreover, also the norms may change. Generally, it would be more convenient to split the verification into two parts: a compilation step, to be executed at deployment time, where the abstract behaviour is determined, and a matching step to be executed at run time, for every pair of partners willing to communicate.

In the following sections, we present three scenarios which pose new challenges from different points of view. First we consider the scenario where flexibility is required but still the computational units are considered trusted, so that the type they declare is truthful. In the second scenario this constraint is relaxed and computational units can fail to behave as declared. While the first two scenarios have in common that they consider qualitative aspects only, in the third scenario we consider quantitative dynamics, that is, situations where types can change along time and the interactions can be stochastic. We structure the following sections according to contributions: design, semantics, algorithms, tools.

Scenario A: New types for trusted participants. *Imagine an online shop where buyers and sellers can interact following the shop's regulations. Before committing to a transaction, the participants negotiate*

the conditions. The data exchanged can be sensitive: typically only certified agencies can receive credit card numbers. Moreover, the shop guarantees that sensitive data will not be kept and, if an agreement is not reached, then the negotiation data will be destroyed.

DESIGN The flexibility of protocols calls for a relaxation of behavioural types to partial orders between interactions/events and their enrichment with (possibly modal) logics, along the lines of what is currently happening with global and local assertions [15].

Notions of polymorphic and dependent types will be investigated for accommodating more flexible forms of dynamic negotiation. The interaction among participants will distinguish between three phases: negotiate, commit and execute. The participants negotiate their behaviours, and, if an agreement is reached, they commit and start an execution which is guaranteed to respect the interaction scheme agreed upon. A suitable representation of negotiation constraints is by logic programs, thus reducing the solution of constraints (and consequently the establishment of a contract) to the execution of a logic program. A negotiation can fail but, if and when a solution is found (commit), under certain conditions a coordinated computation can start, which is guaranteed to have the properties agreed upon in the negotiation phase [19].

As systems become more distributed, controlling access becomes a major concern, in particular, when sensitive private data are memorised on devices where external applications are stored. Properties such as confidentiality and integrity of data become all the more crucial in an open, unreliable environment, where communicating participants may not trust each other, even if the participants have been type checked against the corresponding behavioural types. Along the lines of [20], we need to design types which guarantee secure information flow and access control according to different policies. Both data and participants will have security levels and we will also consider a form of declassification for data [73], as required by most practical applications. We will envision types ensuring the property of *noninterference* [44], that is, that there is no flow of information from objects of a given level to objects of lower or incomparable level [80, 75, 72].

SEMANTICS A fundamental concept of every type theory is that of *subtyping*, which captures the formal relationship between types that are deemed "compatible" with respect to a set of properties of interest. In our setting, subtyping is a key notion that underpins several purposes, including the discovery of computational units exhibiting a certain behaviour, the safe replacement of compatible computational units, and the detection of safe upgrade/refactoring operations over computational units.

An emerging approach to the semantics of types which is particularly appealing in our context is semantic subtyping. Following this approach, refinement/subtyping relations are either defined set-theoretically on some model of types [40, 24] or as the relations that preserve some properties of interest [22, 65] (typical examples of such properties are communication safety, deadlock freedom, and liveness). Because of their intimate connection with process algebras, the semantics of behavioural types can be given in terms of (possibly refined) behavioural equivalences, in particular trace equivalence, weak and strong bisimulation, and testing equivalences. In this respect, the semantic approach to defining subtyping is particularly interesting since some properties (in particular, liveness properties) are difficult to axiomatise (for example, at present there exists no complete axiomatisation of the well-known, liveness-preserving, should-testing pre-congruence [69]).

ALGORITHMS The notion of type adequacy is usually intended as a static one, i.e., verifiable at compile time. One main goal of type analysis is to exclude the presence of runtime data mismatch errors and to make sure that well-typed components are accepted within the environments the components are targeted to.

In our setting, it is fundamental to be able to reason about the compatibility between types at runtime, for verifying whether a component declaring a certain type is acceptable in an environment, possibly imposing some (type driven) restrictions on its behaviour. Partly, this notion of compatibility coincides with the one of subtyping. However, we also expect that the acceptance of a computational unit into an environment may involve a *negotiation* in which the unit tries to reach an agreement with the other units on

its allowed behaviour. The agreement takes into account the mutual dependencies and objectives of the computational units regulated by the norms that characterise the environment. Polymorphic type instantiation and type unification can be seen as basic techniques for reaching "structured" forms of negotiation, but we envision more flexible notions of compatibility, possibly based on constraint-solving techniques, that may determine a dynamic adaptation of the behaviours of the components involved.

Tools We will develop prototypical Domain Specific Languages (DSLs) for describing interactions in terms of norms and behavioural types, together with tools for:

- verifying properties of interactions;
- comparing different interactions;
- checking whether a computational unit written in a, suitably annotated, fragment of a mainstream programming language satisfies an interaction.

The DSLs will be supported also by a modern IDE (Integrated Development Environment), for instance Eclipse, which is one of the de-facto standards. This support is crucial to increase productivity and to facilitate test-driven development. These mechanisms are already very important (if not essential) to be productive and to write clean and tested code in standard programming scenarios; in the context we are considering, where the final application environment is the Internet and where testing becomes harder, the support of an IDE is necessary. In this respect, we can employ the experience we gained for the implementation of DSLs relying on type systems with IDE support [12, 74].

Scenario B: Self-adapting types for untrusted participants. *In the online shop the behaviours of participants can be different from what they agreed upon and can violate the shop's regulations. Changes of these regulations are also allowed. This obliges buyers and sellers to be able to adapt themselves to unexpected situations. Mechanisms for trust, reputation and sanctions are useful to increase the confidence of the customers in the shop.*

Design A challenge is to introduce types that express not only the ideal behaviour expected from the participants in an interaction, but also sub-ideal states, which express what should be the behaviour after a violation. This requires the matching distinctions which are in opposition to duties, like the cancelling or the shadowing of obligations with mechanisms from programming languages like exceptions. The main difference is that exceptions just aim at recovering the control flow moving to the first execution point where the exceptions can be handled. The fact is that they are part of the program, foreseen from the very beginning; instead, in this scenario, events can be unexpected.

Other aspects to be considered in transferring the normative system metaphor to types are the conditional character of norms, which specify what to do depending on the situation, the distinction being the one between obligations and permissions.

We plan to introduce as specifications a new form of self-adaptive types. When accepted to work within an environment, a computational unit carries its original specification, which will be updated along the application's life, according to its observable interactions with other applications and with the customers' applications. In this way, not only it would be possible to trace faults or malicious actions, in order to expel the application, but also to collect statistics to measure the application's performance and reputation.

One possible solution for trust representation is to allow the types to contain partially specified parts, like "holes'' which can be filled in function of the behaviours and of the trust and reputations of the participants involved in the current interaction. The participants are diffident since everybody can lie and their type must change when they observe unexpected behaviours. Taking inspiration from the blame calculi [81, 2] we will enforce a policy of sanctions and a management of reputations.

Semantics While in the first scenario types are discarded after compilation time, here types become a fundamental part of the operational semantics, as in typed operational semantics [43]. The semantics of types becomes dependent on the execution environment; therefore some contextual modal type theory (see [63]), in which modalities represent the requirements on the context and/or the changes that could fix possible mismatches, should be developed.

ALGORITHMS For this second scenario we need new algorithms of type checking and type inference which can deal with partially-specified code. The main novelty of such algorithms is that the obtained types may contain holes which can be only filled at run time.

In order to make types self-adapting, we need algorithms which modify types dynamically in response to unexpected behaviours of participants or when the norms of the environment change. For this goal also, types with holes are a key tool. The difference of this type adaptation with respect to the (standard) replacement of type variables in generic types is that the context surrounding the hole influence the type filling the hole, producing a dynamic variant of it.

Trust and reputation, too, require algorithms to manage them, so as to deal with responsibility assignments and sanctions. We plan to design various algorithms for treating both objective and subjective forms of reputation and responsibility. The sanctions can be either decided by a single authority or by a democratic agreement between the participants, or by some intermediate mechanism, and all these possibilities require suitable algorithms to realise them.

TOOLS We will enhance the DSLs and tools (in particular, their integration in an IDE like Eclipse) outlined for Scenario A in order to support:

- the programming of applications with adaptive types in a test-driven fashion;
- vulnerability checkings.

Note that the second point is crucial in the scenario we are considering: we may have to adapt computational units and their types, and we must make sure that their behaviour does not change too drastically with respect to their original shape.

Scenario C: Types expressing non-functional/quantitative properties *The negotiation between buyers and sellers of the online shop is based on the amount of individual satisfaction for the different alternatives offered and must maximise the general satisfaction. If a merchandise does not arrive to the buyer within a certain amount of time (which can be dependent on some probability distribution), she can require her payment back, according to different policies depending on the situation. Moreover, her degree of satisfaction can depend upon the rapidity of the delivery.*

DESIGN We note that, in many cases, it is natural to model probabilistic processes (hence their associated behavioural types) using a formalism which contains both probabilistic and nondeterministic choices, as in the probabilistic process algebra of [53]. Distributed computational units can use randomised algorithms to improve the performance of their tasks (for example in determining consensus): single computational units can be modelled as probabilistic processes or automata, while the whole environment can be modelled as the parallel composition of the participating units. In this scenario, probabilistic choices (choices of the individual units) and nondeterministic choices (which unit will make the next choice) are interleaved. Schedulers are used to resolve the nondeterministic choices. We plan to develop a type system for schedulers in order to characterise the properties of the different schedulers involved in the choice resolution.

Furthermore, we will address the issue of types for real-time systems. This problem has already been identified as important in [60], given that the numerous difficulties in model-based development of real-time systems (including the presence of timing issues both in the software and in the platform, and scalability issues) has made the impact of formal methods on their development relatively limited so far. For this topic, previous work on interfaces for real-time components (such as [29, 42]) will be used as a starting point.

SEMANTICS As in the case of non-quantitative systems, the formal development of probabilistic, stochastic and timed systems can make use of behavioural equivalences or preorders. In these cases, the definitions of such behavioural relations take into account quantitative information: for example, equivalent processes according to probabilistic bisimulation [59] have the same probability of exhibiting the same observable behaviour; instead, equivalent processes according to timed bisimulation [82] exhibit the same observable behaviour, with corresponding observable events occurring at the same exact time for both processes. Similarly, behavioural preorders such as simulation or testing preorders have also been extended to admit quantitative information. We plan to explore such relations in the context of our notion of quantitative types,

to provide a notion of semantic subtyping.

ALGORITHMS The quantitative aspects of the devised types requires a radically new approach for the design of type-checking and type-inference algorithms. In particular different types can be assigned to the same process with different probabilities.

The agreement algorithms, too, need to be redesigned in order to take into account the various degree of preference for the possible alternatives, when both degree and alternatives can vary over the time in a probabilistic way. Therefore, we plan to use a suitable variant of probabilistic logic programming [61]. Another dimension to take into account is when the participants to an interaction have different weights and the agreement must maximise their satisfaction in a parametric way in relation to these weights.

The stochasticity adds a further challenge to the design of the type-checking, type-inference and agreement algorithms. We will take inspiration from stochastic logic programming [62] to deal properly with agreements.

TOOLS The key contribution that will be made is with regard to tools for the analysis of quantitative systems, including those for determining behavioural preorders or other relations that we use as a basis of our notion of semantic subtyping. In this case, we expect that existing tools for timed and probabilistic preorders, as reported in [28] and [16], will be useful, either to highlight strengths and weaknesses of existing relations, or to provide a back-end to new tools.

Workplan The project is articulated along the three scenarios, with the four steps for each scenario providing also the focus of the research: first, the design and the semantics of types, then the algorithms and the tools based on the algorithms. These steps will be done cyclically, since the development of algorithms and tools can suggest improvements to the design of the types, which necessarily will imply new semantics. The first two scenarios, trusted and untrusted, are more related to each other, with the former being in part a precondition to the latter, in particular, with respect to the definition of flexible protocols. Therefore, the activities for the two scenarios will be synchronised carefully. The activity for the third scenario will start after one year from the beginning of the project, when the types for the first two scenarios will be mature enough, since the third scenario essentially adds quantitative aspects to the second one. Because the project is interdisciplinary, merging competences from model checking with competences from type theory, at the beginning activities will be planned to make all participants acquainted with the state of the art of the areas they have to work on. Indeed, this interdisciplinary approach to types is also one of the most innovative elements of the project: this requires, however, an analysis of the project's advancement. The team will organise monthly meetings (physical or virtual) to monitor the progress and, in case of problems, elicit a revised plan and possibly reassign tasks to participants. Given the 2 year-length of the project, it is important to follow developments in other settings and emerging new challenges due to technological evolution. Thus, further opportunities of importing newly-developed methodologies from related areas will be pursued.

Quality assurance will be managed via:

1. Dissemination through publications in peer-reviewed journals and conferences to get feedback from the scientific community.
2. Joint seminars organised between the diverse areas making the project interdisciplinary.
3. Courses at the level of PhD students to divulge the interdisciplinary approach.

References

[1] S. Abramsky. Domain theory in logical form. *Ann. Pure Appl. Logic*, 51(1-2):1–77, 1991.

[2] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *POPL'11*, pages 201–214, 2011.

[3] F. Alessi, M. Dezani-Ciancaglini, and F. Honsell. Inverse limit models as filter models. In *HOR '04*, pages 3–25. RWTH Aachen, 2004.

[4] F. Alessi, M. Dezani-Ciancaglini, and S. Lusin. Intersection types and domain operators. *Theor. Comp. Sci.*, 316(1-3):25–47, 2004.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[6] B. Aman, M. Dezani-Ciancaglini, and A. Troina. Type disciplines for analysing biologically relevant properties. In *MeCBIC'08*, volume 227 of *ENTCS*, pages 97–111, 2009.

[7] J. C. M. Baeten and C. A. Middelburg. *Handbook of Process Algebra*, chapter Process Algebra with Timing: Real Time and Discrete Time, pages 627–684. Elsevier, 2001.

[8] F. Barbanera, M. Dezani-Ciancaglini, and U. de' Liguoro. Intersection and union types: Syntax and semantics. *Inf. Comp.*, 119:202–230, 1995.

[9] R. Barbuti, M. Dezani-Ciancaglini, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. A formalism for the description of protein interaction. *Fund. Inf.*, 103:1–29, 2010.

[10] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. A calculus of looping sequences for modelling microbiological systems. *Fund. Inf.*, 72(1–3):21–35, 2006.

[11] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *QEST'06*, pages 125–126. IEEE, 2006.

[12] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. TraitRecordJ: A programming language with traits and records. *Sci. Comp. Prog.*, 2011.

[13] L. Bioglio, M. Dezani-Ciancaglini, P. Giannini, and A. Troina. A calculus of looping sequences with local rules. In *DCM'11*, 2011. To appear.

[14] L. Bioglio, M. Dezani-Ciancaglini, P. Giannini, and A. Troina. Typed stochastic semantics for the calculus of looping sequences. *Theoretical Computer Science*, pages 165 –180, 2012.

[15] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 162–176, 2010.

[16] J. Bogdoll, H. Hermanns, and L. Zhang. Flowsim simulation benchmarking platform. In *QEST'09*, pages 211–212. IEEE, 2009.

[17] H. C. Bohnenkamp, P. R. D'Argenio, H. Hermanns, and J.-P. Katoen. MODEST: A compositional modeling formalism for hard and softly timed systems. *IEEE Trans. Soft. Eng.*, 32(2):812–830, 2006.

[18] P. Buneman and B. Pierce. Union types for semistructured data. In *RISSDP*, volume 1949 of *LNCS*, pages 184–207, 2000.

[19] M. G. Buscemi, M. Coppo, M. Dezani-Ciancaglini, and U. Montanari. Constraints for service contracts. In *TGC'11*, LNCS, 2012. To appear.

[20] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk. Session types for access and information flow control. In *CONCUR'10*, volume 6269 of *LNCS*, pages 237–252, 2010.

[21] S. Carlier, J. Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *ESOP'04*, LNCS 2986, pages 294–309, 2004.

[22] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundations of session types. In *PPDP'09*, pages 219–230. ACM, 2009.

[23] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multi-party sessions. *Logical Methods in Computer Science*, 8:1–45, 2012.

[24] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP'11*, pages 94–106, 2011.

[25] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.

[26] M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structures and filter λ-models. In *Proc. Log. Coll. '82*, pages 241–262. North-Holland, 1984.

[27] M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi. Type theories, normal forms and D_∞-λ-models. *Inf. Comp.*, 72(2):85–116, 1987.

[28] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC'10*, pages 91–100. ACM Press, 2010.

[29] L. de Alfaro, T. Henzinger, and M. Stoelinga. Timed interfaces. In *EMSOFT'02*, volume 2491 of *LNCS*,

pages 108–122, 2002.
[30] U. de' Liguoro. Characterizing convergent terms in object calculi via intersection types. In *TLCA'01*, number 2044 in LNCS, pages 315–328, 2001.
[31] P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL'11*, pages 435–446, 2011.
[32] M. Dezani-Ciancaglini and U. de' Liguoro. Sessions and session types: an overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28, 2010.
[33] M. Dezani-Ciancaglini, U. de' Liguoro, and A. Piperno. A filter model for concurrent λ-calculus. *SIAM J. Comp.*, 27(5):1376–1419, 1998.
[34] M. Dezani-Ciancaglini, S. Ghilezan, and S. Likavec. Behavioural inverse limit models. *Theor. Comp. Sci.*, 316(1-3):49–74, 2004.
[35] M. Dezani-Ciancaglini, P. Giannini, and A. Troina. A type system for required/excluded elements in CLS. In *DCM'09*, volume 9 of *EPTCS*, pages 38–48, 2009.
[36] M. Dezani-Ciancaglini, F. Honsell, and F. Alessi. A complete characterization of complete intersection-type preorders. *ACM Trans. Comp. Log.*, 4(1):120–147, 2003.
[37] M. Dezani-Ciancaglini, F. Honsell, and Y. Motohama. Approximation theorems for intersection type systems. *J. Log. Comp.*, 11(3):395–417, 2001.
[38] M. Dezani-Ciancaglini and I. Margaria. A characterisation of F-complete type assignments. *Theor. Comp. Sci.*, 45(2):121–157, 1986.
[39] M. Dezani-Ciancaglini, P. Severi, and F.-J. de Vries. Infinitary λ-calculus and discrimination of Berarducci trees. *Theor. Comp. Sci.*, 298(2):275–302, 2003.
[40] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–64, 2008.
[41] R. O. Gandy. The simple theory of types. In *Logic Coll. '76*, pages 173–181. North-Holland Co., 1977.
[42] M. Geilen, S. Tripakis, and M. Wiggers. The earlier the better: a theory of timed actor interfaces. In *HSCC'11*, pages 23–32. ACM Press, 2011.
[43] H. Goguen. Typed operational semantics. In *TLCA'95*, volume 902 of *LNCS*, pages 186–200, 1995.
[44] J. A. Goguen and J. Meseguer. Security policies and security models. In *SP'82*, pages 11–20, 1982.
[45] M. Hennessy and T. Regan. A process algebra for timed systems. *Inf. Comp.*, 117(2):221–239, 1995.
[46] H. Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*. Springer, 2002.
[47] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
[48] J. R. Hindley. *Basic Simple Type Theory*. Cambridge University Press, England, 1997.
[49] K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523, 1993.
[50] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, 1998.
[51] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM Press, 2008.
[52] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comp. Sci.*, 311(1-3), 2004.
[53] B. Jonsson, K. G. Larsen, and W. Yi. *Handbook of Process Algebra*, chapter Probabilistic Extensions of Process Algebras, pages 685–710. Elsevier, 2001.
[54] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Perf. Eval.*, 68(2):90–104, 2011.
[55] N. Kobayashi. A type system for lock-free processes. *Inf. Comp.*, 177:122–159, 2002.
[56] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4–5), 2005.
[57] J. Krivine, R. Milner, and A. Troina. Stochastic bigraphs. In *MFPS'08*, volume 218 of *ENTCS*, pages 73–96, 2008.
[58] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV'11*, volume 6806 of *LNCS*, pages 585–591, 2011.
[59] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inf. Comp.*, 94(1):1–28, 1991.

[60] E. A. Lee. Computing needs time. *Comm. ACM*, 52(5):70–79, 2009.
[61] T. Lukasiewicz. Probabilistic logic programming under inheritance with overriding. In *UAI'01*, pages 329–336. Morgan Kaufmann, 2001.
[62] S. Muggleton. Stochastic logic programs. In *New Generation Computing*. Academic Press, 1996.
[63] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM TOCL*, 9(3), 2008.
[64] N. Oury and G. D. Plotkin. Multi-level modelling via stochastic multi-level multiset rewriting. *Math. Struc. Comp. Sci.*, 2012. To appear.
[65] L. Padovani. Fair subtyping for multi-party session types. In *COORDINATION'11*, volume 6721 of *LNCS*, 2011.
[66] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.
[67] C. Priami, A. Regev, W. Silverman, and E. Shapiro. Application of stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Proc. Let.*, 80(1):25–31, 2001.
[68] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *ICALP'86*, volume 226 of *LNCS*, pages 314–323, 1986.
[69] A. Rensink and W. Vogler. Fair testing. *Inf. Comp.*, 205(2):125–198, 2007.
[70] J. C. Reynolds. Design of the programming language Forsythe. Technical report, CMU CS, 1996.
[71] B. Russell. Mathematical logic as based on the theory of types. *Am. J. of Math.*, 30:222–262, 1908.
[72] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Comm.*, 21(1):5–19, 2003.
[73] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW'05*, pages 255–269. IEEE, 2005.
[74] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC'10*, volume 6287 of *LNCS*, pages 77–91, 2010.
[75] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL'98*, pages 355–364. ACM Press, 1998.
[76] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.
[77] S. van Bakel, F. Barbanera, and U. de' Liguoro. A filter model for $\lambda\mu$. In *TLCA'11*, number 6690 in ARCoSS/LNCS, pages 213–228, 2011.
[78] S. van Bakel, F. Barbanera, M. Dezani-Ciancaglini, and F.-J. de Vries. Intersection types for λ-trees. *Theor. Comp. Sci.*, 272(1-2):3–40, 2002.
[79] S. van Bakel and U. de' Liguoro. Logical equivalence for subtyping and recursive types. *Th. Comp. Sys.*, 42(3):306–348, 2008.
[80] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comp. Sec.*, 4(2,3):167–187, 1996.
[81] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ESOP'09*, volume 5502 of *LNCS*, pages 1–16, 2009.
[82] W. Yi. Real-time behaviour of asynchronous agents. In *CONCUR'90*, volume 458 of *LNCS*, pages 502–520, 1990.