Towards S$^+$Net: compositional extra-functional specification for large systems

Raphael POSS [a], Merijn VERSTRAATEN [a], and Alex SHAFARENKO [b]
[a] *University of Amsterdam, The Netherlands*
[b] *University of Hertfordshire, United Kingdom*

Abstract System coordination is recently facing new challenges: at the application level, the Cloud and Big Data era has pushed functional specifications to a new level of complexity, where responsibility is spread across multiple independent organizations. At the platform level, parallelism at multiple scales, heterogeneous performance and functional units, thermal and energy budgets and more frequent faults have also brought global-scale deployment and scheduling to a new level of complexity beyond the understanding of the most seasoned system experts. In this article, we propose to equip *compositional specification*, such as traditionally advertised in functional languages, with *extra-functional semantics* appropriate for coordination, as a promising approach to harness the staggering complexity of large systems. This approach establishes a bridge between the functional programming and systems community: from the functional world, using the power of compositional semantics for inductive reasoning and programming, and from the systems world, resource awareness and in particular budget constraints. We apply our approach to the coordination language S-Net, yielding its next incarnation, S$^+$Net. Its extra-functional combinators include feedback loops from the execution environment, composable granularity parameterization, composable energy, latency, throughput and storage budgets constraints, composable isolation specifications, and composable scoping of activities and state onto hardware resources. We also illustrate their impact on relevant industrial applications.

Keywords. system coordination, parallel computing, green computing, energy-aware operating systems

Introduction

Engineering large applications to run efficiently on a variety of computing systems presents major challenges, in particular outside the domain of trivially scalable or regular numerical applications. Today's small-scale homogeneous multi-core processors already challenge conventional software engineering tools and techniques; novel approaches are urgently necessary to make software run efficiently, utilizing productively and simultaneously massive numbers of heterogeneous resources on chip, and networks thereof in datacenters.

The traditional challenge of coordination is to bring together functional components provided by different governance bodies into a single application. To overcome this challenge, coordination requires both mechanisms to translate an application specification into run-time "coordination glue" that actually connects the components together, and proper software interfaces between components to enable data exchanges. This challenge is well-understood and has been thoroughly investigated already. The most successful coordination framework to-date is probably the variety of Unix-like platforms, where individual program-component-processes are connected together in arbitrary ways via a common file system, pipes and network sockets. Within this framework, multitudes of scripting and integration languages now exist to group components into larger applications.

Since the turn of the century, coordination is facing a new challenge: the increasing pressure of limited thermal/energy budgets and the increasing complexity of parallel hardware. Until then, the main spectrum of extra-functional behavior that was visible to coordination was the static trade-off between specialization (faster code, slower to compile) and generality (slower code, faster to compile). Performance could be defined by plugging the speed of uniprocessors into the algorithmic complexity equations of programs; with well-balanced systems [2] throughput was merely a factor of how many processes were sharing the processor simultaneously. Whether a program was compute-bound or network-bound was a static co-property of an algorithm implementation and the platform on which it was run. In other words, extra-functional behavior of programs was *visible* to coordination (in the observable behavior of programs) but there was not much to *manage* at run-time. This has now changed.

The first factor is the growing interest to manage energy consumption, by balancing cost against extra-functional requirements like latency or throughput. Thanks to frequency scaling, it is possible to choose to run a processor slower and save energy, at a larger latency budget. With multiple cores, throughput can be preserved while saving energy and increasing latency thanks to the quadratic factor between power consumption and frequency. Newer processor integrate fine-grained power modes whereby a portion of a chip can be turned off and back on in a short delay for yet more dramatic energy savings, at the expense of additional jitter. The decision of which budget parameters to adapt at run-time is essentially extra-functional, under the realm of coordination where the high-level requirements of entire applications are known.

The second factor is the renewed popularity of parallel hardware, which introduces resource heterogeneity in platforms that must then be managed. The presence of hardware accelerators, from integrated SIMD units in processor cores to many-core GPUs, introduces a new trade-off between throughput and set-up costs and contention, since sharing these resources between components is still problematic. Multi-core chips and Multi-Processor Systems-on-Chip may both propose few large cores optimized for sequential performance at higher energy demands and multiple, low-power smaller cores.

Next to this hardware heterogeneity, there exists also extra-functional heterogeneity in the abstract resources created by the virtualization of parallel hardware in operating systems. Shared data structures in memory were perceived as a necessary source of contention, until the advent of software-transactional memory

(STM) revealed there exists a trade-off between speculation (higher throughput overall, less jitter but potentially higher latency under contention) and locking (lower throughput, more jitter but predictable latency under contention). The exploitation of multi-core parallelism or memory parallelism by software is also a factor of how much the concurrency revealed in programs is exploited at run-time, and how well the application communication patterns maps to the hardware's topology.

While the large diversity of these choices may seem overwhelming, we highlight that most of them can be expressed as a trade-off between latency, throughput, jitter, static implementation costs and run-time costs (energy, real estate on chip). We have chosen these aspects after observing that the large diversity of applications deployed in data centers today are controlled by their operators by balancing the energy and real estate budgets against their development costs, income and latency/throughput/jitter requirements.

Yet there are only few coordination systems in use today that are able to parameterize high-level application specifications by such extra-functional requirements. We have found that Google's App Engine and Amazon's Elastic Compute Cloud (EC2), for example, enable developers to control the trade-off between budget and latency/throughput by automatically duplicating or eliminating server instances or request caches. EC2 also enables control of the trade-off between reliability and cost. However, neither of these services, nor most programming languages in use in production, enable control over the trade-offs related to hardware heterogeneity and the various strategies to exploit on-chip parallelism in software. Meanwhile, the hope that there could exist general methods to automatically transform a program to its best form for a given hardware model and extra-functional objective, born in the 1960's with the advent of functional languages, has been largely curtailed by theoretical results that show such transformations are essentially limited [10]. Therefore, human intervention will be required for the foreseeable future, requiring a simple intuition of the relationship between specification and observed behavior of programs in complex environments.

The challenge, it seems to us, is that both computer engineering and theoretical computing science[1] still struggle to extend the relationship between extra-functional human expectations and application specifications. Since the traditional sequential computer model, where extra-functional behavior was primarily defined by adapting algorithmic complexity to processing rates in hardware (steps or network packets by second), few steps have been made in specification systems to account for both the reality of hardware diversity, resource budgets, and the availability of multiple methods to manage platforms, at run-time.

We thus propose to advance the discussion by introducing compositional, inductive extra-functional semantics for coordination systems. Our key observation is that while compositionality and induction is mostly used to define functional semantics in traditional programming languages, i.e. how components are linked

[1] To the exception of embedded system design, where the relative simplicity of applications has enabled the development of powerful models over the last twenty years, for example synchronous dataflow (SDF) networks or Kahn process networks (KPN), which now have well-understood extra-functional trade-offs. However, the models developed for embedded systems do not yet account for the diversity or generality of applications found at a larger scale.

together to compute valid output data from the systems' input, compositionality and induction can also be applied to extra-functional behavior such as resource budgets and performance constraints.

To demonstrate this, we proceed as follows.

We first delineate in section 1 our assumption, namely the desirability of component-based design and coordination. We highlight the need to distinguish between component specifications and inductively defined run-time instances, and we show in particular in section 2 how inductive coordination can be used for both functional and extra-functional purposes. Our interest in component instances and inductive coordination really stems from the general need for software that adapts automatically and dynamically to resource availability in the environment, which we explain in section 3. In this context, we then formulate in section 4 two general requirements on the design of coordination languages. We also highlight that while extra-functional specifications are often expressed "by contract" by designers (e.g. "guaranteed maximum latency"), contract predicates on behavior are not always automatically implementable as they depend on dynamic resources availability.

With this understanding in mind, we are able to propose in section 5 a *compositional coordination tool box* in the form of compositing operators with orthogonal functional and extra-functional semantics. We present our tool box within the context of S$^+$NET, a coordination language for streaming networks, but we also highlight it should be reusable in other coordination technologies as well. We subsequently illustrate its impact on industrial applications in section 6.

We then review briefly in section 7 the design rationale of our contribution, with comments about its (future) applicability. An overview of our argument and contributions concludes in section 8.

1. Context: component-based design and coordination

We place our contribution in the context of applications and systems designed using *software components* and assemblies thereof.

The word "component" is both versatile and usually well-understood. For the present article, we reuse the definition from [7], itself extended from [1]: components are defined by their *interface*, which specifies how they can be used in applications, and one or more *implementations* which define their actual behavior. The two general principles of *component-based design* are then phrased as follows. The first is *interface-based integration*: when a designer uses a component for an application, he agrees to only assume what is guaranteed from the interface, so that another implementation can be substituted if needed without changing the rest of the application. The second is *reusability*: once a component is implemented, a designer can reuse the component in multiple applications without changing the component itself.

Component-based design is embedded in different programming paradigms using different abstractions. For example, in object-oriented languages, classes define components: the set of methods defines the component interface, and the set of attributes and method implementations define the component implementation.

Abstraction	How interfaces are defined	How implementations are defined
Classes (OOP)	Method interface	Method code and attributes
Functions (FP)	Function signature	Function code
Unix commands	Manual page (list of command-line arguments and program description)	Executable file
Network service	Protocol	Service implementation
Hardware	Signalling specification	Logic design

Table 1. How components are defined in different paradigms

In functional languages, individual functions can be seen as components: the function signature (list of argument and return types) define its interface, whereas the function definition ("right-hand side") defines its implementation. The rest of this article assumes that componentization is available without assuming a specific programming model.

1.1. Coordination environments and languages

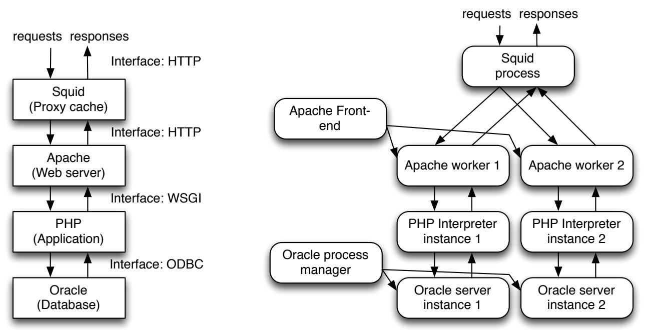
Beyond the basic definitions of components, component-based design relies on *compositionality*: defining new aggregate or *composite* components built out of sub-components. To achieve this, an application designer works in a *coordination environment* which provides both facilities to specify composites, i.e. a *coordination language*, and to run these composite specifications, i.e. a *coordinating run-time system.*

The distinguishing feature of coordination languages is that a programmer can define composites using components defined "outside" of the language. As explained in [7], a programming language can be used for coordination if it offers a *foreign interface* for components defined and provided only after the coordinating program has been written. This is possible in C/C++ with the `extern` keyword or `dlopen` API, in Java with `native` method specifications, in Unix shell scripts by adding new commands via the file system, etc.

1.2. Specifications and instances

For this article, we need a further distinction which is less commonly found in related work: the difference between *component specification* and *component instance.*

To illustrate this distinction, we can consider the perspective of a software engineer tasked with designing a web CRM, who decides to realize the work by combining a proxy cache, a web server, PHP and a database server. From this engineer's perspective, the "advertised" structure of the application is likely to conform to fig. 1a, which highlights the logical relationship between the 4 components the engineer has reused. In contrast, the system administrator who observes the application at run-time may instead see the situation described in fig. 1b. Here, contrary to the "abstract" specification in fig. 1a, the Squid proxy process does not communicate with the Apache server directly; instead it communicates with two worker instances spawned by the Apache server. Each worker instance has in turn spawned its own PHP process to process its incoming requests. On



(a) Specification of a web CRM.

(b) A particular instance of the web CRM.

Figure 1. Specification and instance of a web CRM.

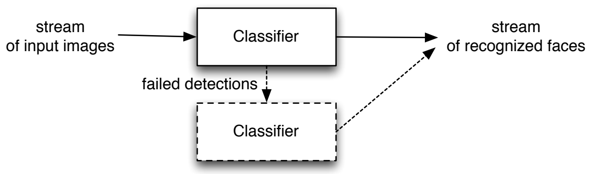
Conceptual domain	Word for blueprint	Word for real-world reification
Object-oriented programming	“class”	“object”
Functional programming	“function”	“activation record”
Operating systems	“program”	“process”
Software builds	“source code”	“object code”
Instruction execution	“executable code”	“instruction stream”
Computer architecture	“design”	“implementation”
Simulation	“model”	“simulator”
Parsers	“grammar”	“parse tree”
Component-based design	“**specification**”	“**instance**”

Table 2. Vocabulary for models and instances

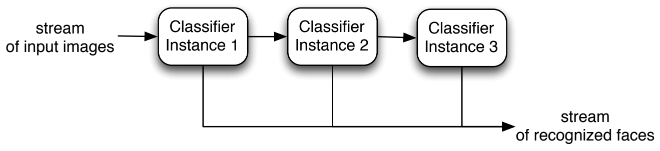
the database side, a duplication has also occurred: each time a PHP process requests an ODBC connection, Oracle creates a new server process specific to that connection. In this run-time scenario, 9 components are involved instead of 4.

In the rest of this discussion we name *component specification* the result of the design work by the programmer, and *component instance* the real-world representation of a specification at run-time. As the previous example shows, each instance is indirectly “caused by” one specification, but a single specification may “cause” multiple instances. Again, the distinction between specification and instance is found in many shapes across computing domains; related terms are given in table 2.

The reason why this distinction is often not needed or used is that most systems traditionally have a one-to-one mapping between specifications and instances. In the example above, in the early age of the Internet the specification would be reified using exactly one Squid process, one Apache process, one PHP interpreter and one Oracle process. Both the application programmer and the system administrator could then use the same words “the Apache server” to designate either the specification or the instance, using context to disambiguate meaning.



(a) Specification for a cascade of classifiers.



(b) Instance of a cascade of classifiers.

Figure 2. Example application for real-time face detection

2. Component instance multiplicity

We can identify two motivations to replicate a single specification into multiple instances at run-time, i.e. introduce *instance multiplicity* in applications.

The first one is *parallel replication*, already illustrated by fig. 1. This form of replication is used mostly for speedup, load balancing, fault tolerance and isolation (security). The result is a set of component instances that are functionally equivalent and work side-by-side.

The other is *inductive replication*, which occurs when a component's behavior is defined by using *recursively* multiple component instances, either copies of itself or instances of different specifications, but where *the number of instances is dependent on the actual input* at run-time. An example can be found in the real-time face detection system described in [9]. In this application, face detection is implemented by a cascade of classifiers, where each classifier receives a stream of images as input. The first classifier instance searches for faces with an inaccurate but fast model. For each input image, if a face is recognized the instance succeeds and the face information is emitted on the application's output. Otherwise, *a new classifier instance is chained in series with the first*, implementing a new, more accurate but slower model. The second instance only receives images for which the first instance has failed, but may cause further inductive instantiation of more refined classifiers. In this example, the number of instances and the depth of the algorithm changes dynamically, depending on the difficulty of recognizing faces in the stream of input images.

Independently from the motivation, the result of multiplicity can be quantified using *instance arity*, that is, the actual number of instances at some point in time. In the general case, arity may be dynamically variable.

3. Component multiplicity in adaptive software

Adaptive software is software whose realization at run-time is parameterized by environment characteristics that are only fully known during execution. In component-based design, adaptivity is found at two levels.

At the level of individual primitive components, a component's interface can specify a number of "tuning knobs" in the form of *behavior parameters.* The "quality factor" of the JPEG compression algorithm is an example, which both influences the quality of the result and the performance of the algorithm for a given input image.

At the level of coordination languages, the operators that combine components together may be parameterized as well. For example, a "split" combinator in a process network language, which takes two processes and load balances the data received on a common input across them, may be parameterized by priorities for each branch of the split.

Of special interest to us is *parameterization of instance multiplicity* when a single component is to be replicated at run-time.

3.1. Need for arity parameterization

Regardless of the motivation, whenever multiplicity is desired it must be specified somehow. The "simple," almost uninteresting specification mechanism for replication is to instantiate *in extenso* by naming each instance, when the language allows it. For example, the Unix shell script:

```
distccd --daemon -p 3632
distccd --daemon -p 3633
distccd --daemon -p 3634
distccd --daemon -p 3635
```

defines an application which uses the component `distccd` (a distributed build server [3]) with arity 4. A reason to do so could be that `distccd` is not multi-threaded and multiple instances thus provide some form of load balancing on a multi-core machine. In practice, however, most languages allow iterative or inductive specifications, for example:

```
for ((i=3632; i<=3635; ++i)); do
   distccd --daemon -p $i
done
```

Which obtains the same effect at run-time. However, here the programmer takes responsibility for choosing the arity. Were the application migrated later to a new machine with 8 cores, maintenance would be required to adapt the application to instantiate 8 times instead. In most cases, this situation is not desirable and we should be instead looking for solutions where *the programmer delegates the choice of arity to the coordination environment, at run-time.*

The solution commonly found is a cooperation between the coordination environment and the programmer. On the one side, the environment defines parameters that characterize the platform, with commonly agreed names but whose values are only known at run-time. On the other side, the programmer uses these variables in the coordination logic. In the example above, if the programmer wants to keep the arity equal to the number of cores in every case, the following logic can be used:

```
LASTCPU=‘awk ’/processor/{n=$3}END{print n}’ /proc/cpuinfo‘
for ((i=3632; i<=3632+$LASTCPU; ++i)); do
   distccd --daemon -p $i
done
```

This example exploits the run-time environment characteristics listed, at run-time, in the file `/proc/cpuinfo` (Linux).

This specific example highlights two fundamental aspects of programming for coordination. The first is that standard mechanisms agreed upon by convention should be provided by the environment to document the run-time platform. Were the information not available, it would not be possible to write adaptive software.

The second aspect is more subtle but yet of tremendous and growing importance. In the example above, the specific solution that chooses to equate the instance arity to the number of cores obscures the fundamental requirement of the application: that the `distcc` instances exploit the *available resources* to balance load and increase throughput. If this is the only application running on the platform, then “number of cores” accurately models “available resources”; however, the situation is not so clear if *multiple applications share the platform side-by-side.* It is also not so clear if thermal constraints or transient faults make some cores occasionally unavailable.

As the example illustrates, adaptive software that reacts to variable resource availability requires a language where the programmer can “tell” the coordination system to add or remove instances dynamically depending on external factors and optimization goals. For this specific example, unfortunately, the common Unix shell does not provide such facilities. We revisit this example later in section 6.

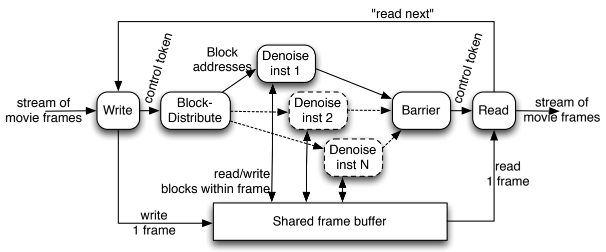
3.2. Need for topology parameterization

Adaptivity in the context of coordination also requires to parameterize the topology between component instances, i.e. how instances are connected together and to the rest of the application at run-time.

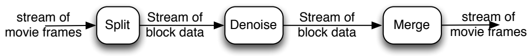
To illustrate this, consider the example of real-time movie denoising: a video input (e.g. a camera) must be connected to a filter (e.g. a Gaussian convolution) that smoothes out each frame before the stream can be further exploited. In this application, the main component is the denoising filter, and both the input and output is real-time movie data, i.e. a stream of image frames.

Now, consider how implementation differs *depending on the platform available.* When using a multi-core computer with a shared memory, the preferred implementation may be similar to the one depicted in fig. 3a. Here, each frame in turn is stored in a shared memory buffer, and each core runs a separate instance of the denoising filter on a sub-region of the shared buffer identified by address, e.g. via coordinates. As above, the arity of parallel replication is a (possibly indirect) function of the hardware parallelism available, not the input images. Meanwhile, the replication of the denoising filter is accompanied by the *introduction of helper components* to read each frame in the buffer, to generate a set of regions to work on, and to barrier synchronize the work for each frame.

In contrast, consider an implementation on FPGA instead. There, the denoising filter can be made much faster per pixel, but there is typically not enough



(a) Shared memory implementation of movie denoising.



(b) FPGA implementation of movie denoising.

Figure 3. Example application where topology is dependent on hardware resources available.

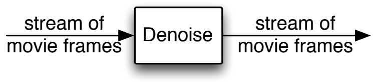


Figure 4. Functional specification of movie denoising.

memory on the FPGA chip to hold an entire input frame at once. This suggests the implementation depicted in fig. 3b instead, where the image is separated into sub-block and the block data itself, not the coordinates, is streamed through a fixed number of instances of the denoising filter.

One could possibly argue that the two scenarios fig. 3 could be seen as two distinct application specifications, to be selected by a human operator depending on the resources available. However, this view is not satisfactory in larger environments where heterogeneous resources are available and the choice could be made automatically depending on extra-functional parameters, such as desired throughput or transient unavailability of a specialized hardware (which could be occupied by another application for some time).

To automate this dimension of coordination, we propose to state that the *functional specification* of this application is as simple as fig. 4, and that the choice to implement as either fig. 3a or fig. 3b could be done via a parameterized mechanism at run-time. We call this *instance topology parameterization.*

4. Engineering work flow and requirements of extra-functional coordination

As the examples from the last section suggest, component coordination is not merely a matter of ensuring that composites are functionally correct, i.e. that components are connected in the “right way” and the overall data input-output relationship satisfies the computational definition of the application. Next to functional composition, the role of coordination is to ensure an application satisfies *extra-functional requirements* on the overall behavior of the application in its run-time environment.

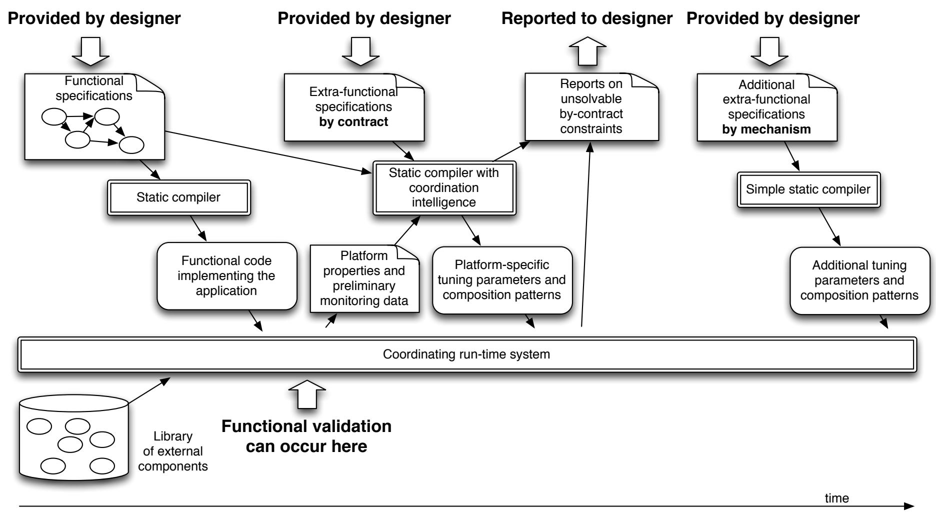

Provided by designer
Provided by designer
Reported to designer
Provided by designer
Functional specifications
Extra-functional specifications **by contract**
Reports on unsolvable by-contract constraints
Additional extra-functional specifications **by mechanism**
Static compiler
Static compiler with coordination intelligence
Simple static compiler
Functional code implementing the application
Platform properties and preliminary monitoring data
Platform-specific tuning parameters and composition patterns
Additional tuning parameters and composition patterns
Coordinating run-time system
Library of external components
Functional validation can occur here
time

Figure 5. Design workflow for extra-functional coordination.

In the same way that computations can be defined either in a rather "functional" or "imperative" way, extra-functional coordination can be defined either *by contract* or *by mechanism*. An example extra-functional contract for the denoising application from fig. 4 may "run the `denoise` function so that its overall frame throughput must be at least 30 per second." An example mechanism that achieves the same may be "run the `denoise` function on a 8-core, 2.4GHz processor with 2GB of RAM and a 100MB/s bidirectional I/O link to the video equipment, using 128x128 frame blocking."

From the perspective of the final user of an application, extra-functional contracts are more desirable because they typically directly correspond to business requirements. However, from an implementation perspective they require *coordination intelligence* that translates them to the actual mechanisms. Unfortunately, in contrast to functional programs for computations which can always be translated to machine code (Church-Turing thesis), extra-functional by-contract specifications may be *unsolvable* for a given coordination technology. Whenever a particular contract is unsolvable (or not yet supported), the software engineer must then resort to a mechanical specification instead.

The corresponding work flow from the designer's perspective is illustrated in fig. 5. In an initial phase, the functional behavior of the application is specified and validated, using some library of external components. Concurrently or soon afterwards, the extra-functional requirements are first expressed by contract, and the coordination technology attempts to translate the contracts to actual coordination mechanisms. This process is dependent on the contracts, the functional specification, but also information about the execution platform such as which resources are available. All contracts that can be successfully converted to mechanisms yield tuning parameters or optimizations that can be directly deployed. Otherwise, the designer is informed and can choose instead to specify additional coordination mechanisms, or change the functional design and start anew.

This work flow is independent from which particular coordination technology is used, and reveal two fundamental constraints on the usability of the coordination language.

The first is relatively straightforward. Suppose the application fig. 4 was first specified functionally, and then (afterwards) extra-functionally constrained to a minimum frame rate of 30fps. Suppose that later in time a new extra-functional requirement comes in for 60fps instead. It is important that the coordination technology allows the designer to update the extra-functional constraint without altering the functional semantics, otherwise functional validation must be carried out again. This pushes for *orthogonal functional and extra-functional semantics in coordination languages.*

The other is more subtle yet no less important. Suppose for example that the designer of the application in fig. 4, unsatisfied by the translation of the performance contract on frame throughput, has manually entered an additional by-mechanism specification to use the instance topology from fig. 3a. If the coordination technology provides two different specification environments or languages for the functional and extra-functional semantics, the designer would then have first produced a specification like fig. 4, then *separately* one like fig. 3a. Now, suppose that a new functional requirement then comes in, that denoising should be followed by color balance correction. Because the environments are separate, now two specifications instead of one must be modified. If the application is already deployed, chances are the designer will modify only the latter, and forfeit the opportunity to further specify extra-functional requirements by contract on top of the "pure" functional specification, which has thenf diverged. This scenario reveals that the coordination technology should not be separated in different languages and that *both functional and extra-functional specifications should be integrated* to ensure they stay consistent over time.

This conjunction motivates our proposal for *compositional extra-functional coordination*, where extra-functional specifications are interleaved with functional aspects in application definitions. In our vision, a designer should be able to update either the functional part or the extra-functional part of a specification and see the other part adapt automatically to the change. Moreover, the language should allow an automated process to automatically elide all extra-functional operators of a combined specification and reveal the functional core so it can be validated separately.

5. Compositional extra-functional coordination

In the previous sections we have introduced separately:

- the purpose and form of component instance multiplicity, and the need for both arity and topology parameterization (section 3), and
- the engineering work flow for adaptive software using coordination and the need for orthogonal, yet integrated, functional and extra-functional semantics for coordination languages (section 4).

When a coordination technology answers both these requirements together, a new problem arises: *how to express extra-functional requirements meaningfully over components with instance multiplicity?*

This is where we position the main contribution of this article. Given the aim to support both specifications with multiplicity and the separation of functional and extra-functional semantics, we investigated *which operators to place in the extra-functional tool box which preserves compositionality and expresses meaningful constraints over components with multiplicity.*

We have carried out this study in the context of S$^+$Net [6], a coordination language for streaming networks. In the following sections, we show examples of our proposed extra-functional toolbox, by applying some of the key features of S$^+$Net to the examples already described earlier and a couple of industrial applications. Note however that we aim our proposal to be more general; we expect it to be applicable to other coordination technologies as well, as long as they exhibit functional compositionality in specifications and map composite specifications to composite mechanisms in their run-time systems.

5.1. Functional coordination with S$^+$Net*: an overview*

The primitive component in S$^+$Net is a *stream processor*: an entity which processes a stream of input events over time and produces a stream of output events. This encompasses both computational functions, which are called anew on each successive input, and more complex stateful processes. S$^+$Net is principally a coordination language: it is designed mainly to integrate external components developed in other languages via its *box* construct. However it is not purely a coordination language: simple coordination-related computation or synchronization functions can also be implemented directly in S$^+$Net via its "transducers" construct.

The basic abstraction on top of stream processors is the *streaming network*: the connection of one or more stream processors with I/O endpoints to the "real world." To group stream processors into networks, S$^+$Net provides a functionally complete set of compositional *network combinators*, sufficient to group stream processors in arbitrary computational patterns.

From the functional perspective, all primitive networks and composites thereof are abstracted as if they had a single input stream and a single output stream (SISO). However, all streams are typed, and may carry multiple types. In an implementation, each logical stream may thus be instantiated as multiple communication channels that connect type-wise matching process endpoints. This means the SISO abstraction does not imply full serialization of I/O between stream processors and concurrency of communication can be exploited on parallel hardware, for fully MIMO run-time behavior. At the outer level, stream endpoints are connected to the environment's I/O endpoints, typically network sockets.

The functional combinators of S$^+$Net are:

- *sequential composition* of two or more sub-networks, i.e. pipelines;
- *selection* between two or more sub-networks depending on the actual type of input messages (routing);
- *replicated composition* which inductively replicates the inner sub-network at run-time depending on a guard condition.

Because the branches of a selection and the exit paths from a replicated composition are concurrent, their respective outputs may be emitted out of order

Network	Notation
External primitive component A	`box` A
Transducer with function S	`[|`S`|]`
Ordered replication composition of N with guard G	N`*`G
Unordered replication composition of N with guard G	N`!*`G
Sequential composition of N and M	N`..`M
Selection between N and M	N`|`M
Restoration of input order around N	`?`N`#`

Table 3. Functional specification constructs in S$^+$Net.
(Constructs listed in order of operator precedence: `box` binds more tightly than `|`.)

relative to their common input. An additional unary *stream reordering* combinator can be used to force the output order of a sub-network to match the input order when so desired. There are further two forms of replicated composition:

- in *ordered replicated composition*, the instances form an ordered list: the order in which the replicas are traversed by input events is fixed and new replicas are introduced at the end of the list;
- in *unordered replicated composition*, the instances form an unordered set: each input event may traverse the set of replicas in a different order, although they are guaranteed to go through all replicas.

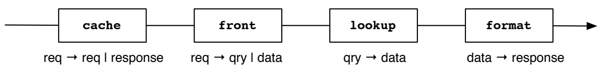
Ordered replicated composition is appropriate for inductive chaining of sub-computations, for example as required by the face recognition algorithm mentioned in section 2. It can be used also to support cycles in the process graph, by translating the cycle to an ever-expanding replication: automatic garbage collection ensures finite resource usage. Unordered replicated composition is useful to implement "worker pool" concurrent patterns, where an input is split into sub-events and all sub-events must be processed, although order preservation is not required. The reader is referred to [6, Sect. 2.4.4 & 5.4] for details about this operator.

The syntax for functional specifications is given in table 3. Parentheses can be used for grouping. Types are inferred automatically from inner networks outwards. Messages/events in S$^+$Net are called *records* and can carry arbitrary sets of key-value pairs.

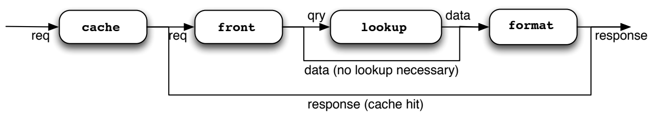
5.2. Example functional specification of a web application

Consider the example of the web CRM mentioned in section 1.2. The specification from fig. 1 was architectural, describing the components from a system administrator's perspective. From a functional perspective, the application can also be described as a pipeline of the following components: a cache, taking as input requests from the network and producing either responses (cache hits) or the requests themselves (misses); a handler that parses the request and determines what to do; a database lookup; and a formatting step that gathers data and places them in web page templates. The output of the pipeline is the response data back to the network interface. A possible functional specification of this application in S$^+$Net goes as follows:

```
net {
```



(a) Functional specification of a web application pipeline.



(b) Component instances and flow bypasses at run-time.

Figure 6. Example specification of a web application with S$^+$Net.

```
   box cache : req -> req | response;
   box front : req -> qry | data;
   box lookup : qry -> data;
   box format : data -> response;
} connect cache..front..lookup..format
```

This declares four externally defined `box` components, declares their type signature, and connects them in a pipeline using the `..` operator. The equivalent visual representation is given in fig. 6b. Because the output of the `cache` component has two possible types `req` and `response` and the next component `front` only takes `req` as input, *flow inheritance* is used to automatically create communication bypasses at run-time, as revealed in fig. 6b.

5.3. Example extra-functional coordination in S$^+$Net*: load balancing*

The specification so far only yields one instance of each component at run-time. S$^+$Net's coordination logic cannot force the introduction of parallelism because the streams connecting the components are ordered and concurrency would break this order. A working but inelegant way to achieve the scenario illustrated in fig. 1b is to *modify* the functional specification as follows:

```
net {
   box cache : req -> req | response;
   box balance : req -> req1 | req2;
   box front1 : req1 -> qry | data;
   box front2 : req2 -> qry | data;
   box lookup : qry -> data;
   box format : data -> response;
} connect cache..balance..
  (front1..lookup..format | front2..lookup..format)
```

With this specification, the component `front` is explicitly aliased two times and its input type is given two different names, to create two branches implementing the same function but with different types. A "load balancing" component which simply annotates requests with a different type name is then placed in front, with a parallel selection. This solution works but would require manual editing of the aliasing to change the number of pipeline branches. A more elegant separation of concern is obtained using the following specification instead:

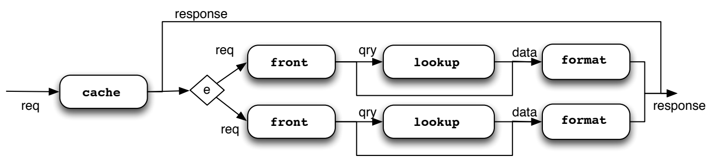


Figure 7. Run-time situation with replication selection and two inner instances.

```
net {
   box cache : req -> req | response;
   box front : req -> qry | data;
   box lookup : qry -> data;
   box format : data -> response;
} connect cache..[|->{<p=1>},{<p=2>}|]..(front..lookup..format)!<p>e
```

Here the component specification and the functional part of the pipeline are left unchanged. Two constructs are added:

- the phrase "`[|-><p=1>,<p=2>|]`" specifies an *initialization transducer* which emits two records at application start-up, with a single key `p` and successive values 1 and 2.
- the phrase "`(...)!<p>e`" uses S$^+$Net's *replication selection*. This unary combinator automatically instantiate the sub-network to which it applies when it receives records containing the key specified, here `<p>`. It distributes further incoming records among the existing instances according to a *selection policy*, here "`e`" which means even distribution across available branches.

When instantiated at run-time, the initialization transducer is run once, then the coordination logic recognizes it has become inactive and automatically elides it from the run-time environment. The resulting component instance network is illustrated in fig. 7.

This specification as it now stands is appropriate when running on platforms with at least two physical processors. However, if the number of processors is lower than the number of pipeline branches, an undesirable behavior can arise: the S$^+$Net run-time system is allowed to schedule the components using cooperative scheduling, and a blocking request on one branch would then also block the other branch.

To avoid this issue requires clearly stating what is the *intent* of replication: that the processing of each branch *progresses independently* from the other. S$^+$Net provides a general primitive combinator for this: *extra-functional isolation.* Given a behavior trait, for example progress independence noted "`f`" (fairness), the network described by "`N//f`" ensures that all instances of `N` at run-time will be scheduled fairly with respect to each other. Other isolation traits are also available, including bandwidth and storage independence.

Here however the instances are created by the `!` operator; the specification "`N//f !<p>e`" would not serve our purpose, because then the entire network `N//f` would be replicated and not the inner operand of `//f`. To solve this, we must use another S$^+$Net combinator: labeling, noted "`N'label`". This associates an

Operator name	Description	Notation
Internal isolation	Isolate those sub-network instances of N labeled X from each other respective to extra-functional trait t	$N/X/t$
External isolation	Isolate those sub-network instances of N labeled X from each other *and from the enclosing network instance* respective to extra-functional trait t	$N/X/{+}t$
Budget contract	Cap the extra-functional budget r available to those sub-network instances of N labeled X to either an absolute value or a proportion of the budget available to the enclosing network instance	$N/X{:}r$

Table 4. S$^+$Net operators that specify extra-functional coordination *by contract.*

arbitrary text label with a sub-network and allows another operator to refer to this label, for example:

```
... connect cache..[|->{<p=1>},{<p=2>}|]..
    (front..lookup..format)'serve !<p>e  /serve/f
```

This specification associates the entire sub-network "`front..lookup..format`" with label "`serve`", then places it within a replication selection. The operator "`/serve/f`" is then applied to the entire construct, and specifies that "all inner instances named `serve` are scheduled independently from each other," achieving the desired purpose.fr

Finally, the example can be both generalized and simplified greatly by using one extra feature of S$^+$Net: automatic parallel replication of sub-networks where stream ordering between input and output is known to not matter. This is specified by replication selection without tag, e.g. "`N!e`", or even "`N!`" because `e` happens to be the default policy for `!`. Using this feature, the specification can be reduced to:

```
... connect cache..(front..lookup..format)'s ! /s/f
```

With this, the number of instances of the inner network labeled `s` scales to the number of available parallel resources at run-time, automatically, and each branch is guaranteed fair scheduling.

5.4. Extra-functional coordination by mechanism or by contract

The example so far has illustrated two families of extra-functional combinators in S$^+$Net: replication selection belongs to the group of *mechanism* operators, whereas isolation belong to *contract* operators.

We propose three by-contract specification operators, listed in table 4. During instantiation and at run-time, contract operators are automatically mapped to mechanisms to the maximum extent possible. Unsolvable mappings are reported as errors during initialization.

Extra-functional isolation, already introduced in the previous section, provides behavior and resource usage independence between component instances, across any of the traits listed in table 5. At run-time, the contracts are mapped to different resource scheduling mechanisms. There are two variants of isolation:

Trait	Description	Mechanism
`f`	Relative progress independence (fairness): the progress of each instance not starved on input or blocked on output is guaranteed independently from other instances.	Preemptive scheduling
`b`	Relative bandwidth independence: the internal bandwidth of processors and channels onto which each instance is mapped is reserved and free of contention from other instances.	Real-time scheduling priorities and QoS over virtual channels
`s`	Relative storage independence: the storage allocated by the component instances and network management is sourced from separate storage pools.	Storage partitioning
`p`	Relative power supply independence: the power demands of each replica are satisfied independently from the power usage of other instances.	Physical partitioning

Table 5. Valid traits for the internal and external isolation combinators.

Budget	Description	Always solvable? (mechanism)
`Mc(`x`)`	Maximum memory storage	no (exceptions)
`Ma(`x`)/ma(`x`)`	Maximum (resp. minimum) component arity	yes (dynamic management)
`Mti(`x`)/Mto(`x`)`	Maximum input (resp. output) throughput	yes (throttling)
`mti(`x`)/mto(`x`)`	Minimum input (resp. output) throughput	no (exceptions)
`Mfl(`x`)/Mll(`x`)`	Maximum first (resp. last) latency	no (exceptions)
`mfl(`x`)/mll(`x`)`	Minimum first (resp. last) latency	yes (forced delays)

Table 6. Valid specifications for the budget contract operator.

internal isolation isolates named instances only from each other, and *external isolation* isolates named instances from each other and also from the enclosing composite network.

The last by-contract operator is the *budget contract*, which establishes a run-time limitation on a behavior metric, including at least those metrics listed in table 6. At run-time, some budget contracts map to enforcing mechanisms and are thus always solvable (e.g. throttling for maximum throughput), while others depend on the availability of physical resources (e.g. minimum throughput) and may thus fail with an exception.

When the by-contract operators are inappropriate or more fine-grained control is required, we propose the five by-mechanism operators listed in table 7.

Replication selection was already described in the previous section.

Exception handling provide a way to adapt dynamically to run-time budget contract violations, such as specifying an alternate behavior. An example use of exception handling is given in the next section.

The environment awareness construct enables a specification to use run-time parameters for further coordination. It is not really an operator but rather specifies a new primitive component with no computational effect.

The projection operator is used to control the way records flow through the network, and thus allows to trade latency for space usage. For example, when instantiating a sequential composition $A\,..\,B$ over two parallel processors, it allows to choose between either a dynamic pipeline with one instance of A on one processor and one instance of B on the other (more latency, less space), or two

Operator name	Description	Notation
Replication selection	Explicit replication of N using instantiation key t and distribution policy p	N`!<`t`>`p
Replication selection	Automatic replication of N to available parallel resources using distribution policy p	N`!`p
Exception handling	Report using key a any exception of type E caused in those sub-network instances of N labeled X	N`$`X`(`a`=`E`)`
Environment awareness	Read environment variable E using key v into any input records of type x	`[<`x`>.`v`=`E`]`
Process projection	Force execution strategy s for those sub-network instances of N labeled X	N`/`X`!`s
Hardware mapping	Assign those sub-network instances of N labeled X to a subset of physical resources used by the enclosing network instance labeled Y using the placement strategy s	N`/`X`@`Y`:s`

Table 7. S$^+$NET operators that specify extra-functional coordination *by mechanism.*

instances of the function $B \circ A$, one on each processor (less latency, more space). The reader is referred to [4, Part 2] and [6, Sect. 4.8] for more details.

Finally, the hardware mapping operator provides fine-grained control over the mapping of components over hardware resources. It is provided for completeness but its use is complex and is thus expected to be used only as a back-end mechanism when defining libraries of higher-level abstractions.

This extra-functional tool box was carefully designed so that all operators are orthogonal from each other and can be combined freely. The reader is referred to [6] for a detailed specification and discussion of all extra-functional constructs.

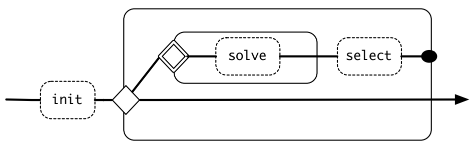
6. Additional use cases and applications

Consider how the example from section 3.1, which requires scaling the number of `distcc` instances to the number of cores, can be re-expressed in S$^+$NET simply using:

```
([.port=3632+InstanceIndex]..distcc)!
```

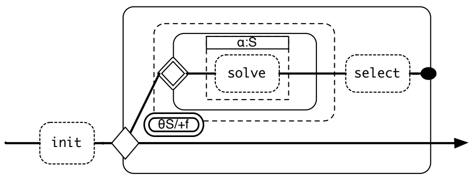
The replication selection `!` dynamically scales the number of instances of the sub-network to which it is applied, to the number of cores available on the platform. Each instance then inspects its own dynamic run-time index and uses it to compute a port number parameter to `distcc`. The number of instances can also vary dynamically depending on resource availability. This specification is both simpler and largely more powerful than what can be obtained in most glue languages for systems programming.

As we have seen so far, all the relatively simple examples from section 1 are trivially addressed using our proposed extra-functional tool box. To further demonstrate its applicability, we also consider in the following sub-sections two production-grade industrial applications explored during the EU ADVANCE project.



“`init .. (solve! .. select)*`”

Figure 8. Example ant colony optimization.



“`init .. ((solve’S)! /S/+f .. select)*`”

Figure 9. Optimized ACO using the isolation combinator.

6.1. Process starvation in ant colony optimizations

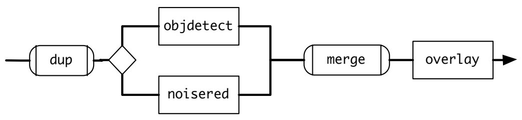
In [5], the authors observe a shortcoming of S-Net when running an application that uses ant colony optimization (ACO) to solve the single machine total weighted tardiness problem (SMTWTP) found in industrial logistics. The overall application uses bulk-synchronous parallelism, where each iteration computes sub-parts of the problem and a reduction step selects the best intermediate solution. The corresponding (simplified) S$^+$Net specification is given in fig. 8.

The issue was found around the inner network using replication selection enclosing `solve`. This network processes n concurrent data sub-streams, where n is the number of simultaneous ants in the ACO parameters. The expectation was that since the input sub-streams are concurrent, speedup should be observed using parallel resources. However, S-Net does not specify how replication selection is mapped onto parallel hardware, and thus does not guarantee speedups. Indeed, the authors investigated and determined that in some cases, the management task that distributes work to the `solve` replicas also shares a processor with one of the `solve` replicas. As soon as this `solve` replica receives work, it starts to compute, starves the management task from processing time, and consequently starves the following `solve` replicas that have not yet received work.

The issue appeared because S-Net does not provide the ability to specify that the management task must be separately scheduled from the solvers. With S$^+$Net, this can be trivially expressed using the labeling and isolation combinators as explained in section 5.3, yielding the solution illustrated in fig. 9.

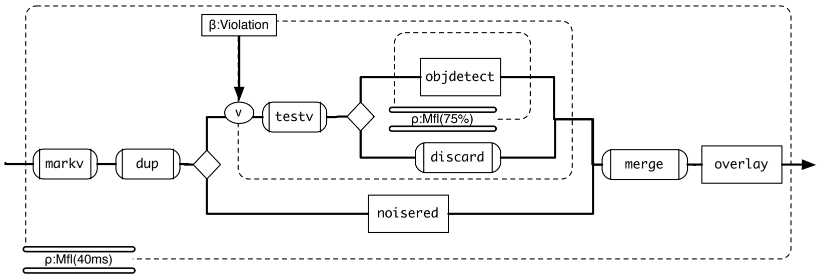
6.2. Medical imaging with soft realtime constraints

Another application is the coordination of a piece of medical equipment used in surgery: a camera captures a video of the operating site at the patient, then a system analyses the image to recognize the organ to operate, and overlays a cursor or metadata with the video in the image presented to the surgeon. Simultaneously with object recognition, a simple linear filter reduces noise on the captured image.



"`dup .. (noisered | objdetect) .. merge .. overlay`"

Figure 10. Example medical imaging application.



"P`/:Mfl(40ms)`",
with P = "`markv..dup..(`T` | noisered)..merge..overlay`",
with T = "`(testv..(objdetect/:Mfl(75%) | discard))$(v=Violation)`"

Figure 11. Medical imaging application with extra-functional annotations.

Functionally, the processing pipeline is fed with video frames; at the head of the pipeline, frames are duplicated to be processed concurrently by object recognition and noise reduction; and finally the result of object recognition is overlaid with the result of noise reduction to obtain a resulting video. We can thus specify the application with the functional network in fig. 10, where `dup` is a filter that duplicates video frames, `noisered` and `objdetect` the two processing boxes, `merge` a synchronizer and `overlay` to fuse intermediate results into resulting frames.

Next to this basic functional specification, the application has two additional extra-functional requirements: the frame latency through the entire pipeline should not exceed 40ms, so that the surgeon keeps an up-to-date image of the operating site at all times; and the object detection algorithm may take an unpredictable amount of time to complete, which may cause its results to be outdated relative to the video, in which case the overlay should be discarded.

The extra-functional specification is transparent with S$^+$NET; we give its notation in fig. 11:

- the entire network is enclosed in a budget combinator ("`/:Mfl(40ms)`" or ρ in the diagram) to cap the overall latency;
- the `markv` filter equips all frames with a v field used subsequently by the exception handling construct ("`$(v=Violation)`" or β in the diagram);
- the inner `objdetect` is annotated with another budget combinator to restrict to 75% of the outer Mfl budget, i.e. 30ms latency;
- if the 30ms latency is violated, an exception is raised, caught by the exception handler construct, and the `testv` filter changes the type of the input frame to be routed to `discard` instead.

As is, this specification instructs the S$^+$NET environment to make a best effort at guaranteeing the latency stays below 40ms; however this may violated at run-time if insufficient resources are provisioned or the coordination fails to find a suitable schedule. In this case the entire application would fail. It is possible to enclose the entire network into another scaffold of "`markv..(testv..(`P`/:mfl(40ms)|discard))$(v=Violation)`", to instead simply drop the input frames upon latency violations.

Finally, the combined specification degrades automatically to the original functional specification when the extra-functional operators are removed, despite the semi-functional introduction of the components `markv`, `testv` and `discard`. We demonstrate as follows. The budget combinator around `objdetect` can be removed without effect on its inner operand. Once removed, the budget violation exception cannot be raised any more, and the exception handling mechanism is known to never report the exception to `testv`, which can then be automatically elided. When `testv` is elided, the `discard` branch is determined via typing to become inactive and can be elided as well. Meanwhile, once `testv` is elided, there is no component left using `markv`'s output, which can then be elided as well. Finally, the outer budget combinator can be removed without effect. The result is identical to the purely functional specification in fig. 10.

7. Discussion and future work

True to the principles established in [8], S$^+$NET separates two forms of distributed application design: component and system, with the former focused on the mathematics of data processing and the latter exclusively on its logistics. S$^+$NET, like its predecessor S-NET, is a glue that is effective in combining the two without one contaminating the other.

The author of [8] had concluded with two open issues. To enable self-adaptation and self-reconfiguration, which are strong requirements for large distributed applications, a "disciplined form of feedback" was required. Strong with industrial experience, we have embraced the realization that a solution to this issue necessarily involves a mix of human tuning with automated optimization based on run-time environment feedback; our proposed environmental awareness has answered this need.

The other identified issue was how to deal with the dual nature of coordination: either the transparent view where component specifications are reified directly to instances at run-time; or a competing view in which coordination is treated as a high-level computation modulo non-determinism, where pipelines can be reifed by functional composition, at the cost of a non-trivial behavior intuition. We have recognized that this duality is in fact an extra-functional trade-off between locality of computation vs. locality of communication, and we propose to embrace this duality by letting the coordination designer choose which approach to take for any sub-network in a specification.

From an implementation perspective, S$^+$NET is an ongoing project at the University of Amsterdam, the University of Hertfordshire and their research partners. Two implementation directions are investigated: one using a custom technology stack, primarily oriented towards research in language design; and one as a library of reusable coordination tools in existing concurrency-aware languages, e.g. Google's Go or Haskell, for deployment in existing applications. In this latter view, we do not market our proposal as a specialized language on its own, but rather a *compositional coordination tool box* reusable in a variety of environments.

From here, we envision that this coordination tool box will be used in two scenarios. As a programming language, its extra-functional constructs enable application designers to establish contracts between extra-functional constraints, such as energy budgets or latency deadlines, and the execution environment. By accounting semantically for unexpected failures or unsolvable constraints, our coordination tool box also provides a mechanism by which a specification can adapt to an environment where extra-functional contracts cannot be honored.

Meanwhile, compositional coordination can also be used as a modeling language. When considering any other parallel programming system, it is common to see the implementers make concrete run-time choices relative to the mapping and scheduling of source-level concurrency, while not documenting these choices or let them vary unpredictably across platforms. By providing a consistent and orthogonal vocabulary of concepts beyond the mere functional description of the input-output relationship, our tool box allows a scientist to observe any concrete parallel application, then describe and reason about the extra-functional decision mechanisms of its execution environment.

8. Conclusion

This article has presented a compositional coordination tool box with separate constructs for functional and extra-functional specifications. The functional part is complete and can serve to define arbitrary computational patterns. The extra-functional part contains both "by-contract" specification constructs, which are high level but may be sometimes unsolvable, and "by-mechanism" constructs relevant when by-contract constructs are insufficient or for performance tuning.

Our tool box is reusable in any coordination technology which aims to support component-based design, inductive specifications of component instance multiplicity, and compositional extra-functional constraints over inductive specifications. According to our analysis, these traits are becoming increasingly relevant in large deployments due to the complexity of working with resource heterogeneity.

We are constructing this tool box in the context of S$^+$Net, a coordination programming environment originating from the previous work S-Net. Like S-Net, S$^+$Net is primarily based on concepts from stream processing networks, although we show via examples that its coordination facilities are applicable to client-server web applications and industrial-grade computational optimizations.

Acknowledgements

This research was partly supported by the European Union under grant number FP7-248828 ADVANCE[2].

[2]`http://www.project-advance.eu/`

References

[1] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992. ISSN 1049-331X. doi:10.1145/136586.136587.

[2] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proc. 16th International Conference on Data Engineering.*, pages 3–10. IEEE, 2000. doi:10.1109/ICDE.2000.839382.

[3] Jes Hall. Distributed computing with distcc. *Linux J.*, 2007(163), November 2007. ISSN 1075-3583.

[4] Philip Kaj Ferdinand Hölzenspies. *On run-time exploitation of concurrency.* PhD thesis, University of Twente, Enschede, the Netherlands, April 2010. Available from: http://doc.utwente.nl/70959/, doi:10.3990/1.9789036530217.

[5] Kenneth MacKenzie, Philip Kaj Ferdinand Hölzenspies, Kevin Hammond, Raimund Kirner, Nguyen Vu Tien Nga, Rene te Boekhorst, Clemens Grelck, Raphael Poss, and Merijn Verstraaten. Statistical performance analysis of an ant-colony optimisation application in S-Net. In Clemens Grelck, Kevin Hammond, and Sven-Bodo Scholz, editors, *Proc. 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures*, 2013. Available from: http://www.project-advance.eu/wp-content/uploads/2012/07/proceedings.pdf.

[6] Raphael Poss, Merijn Verstraaten, Frank Penczek, Clemens Grelck, Raimund Kirner, and Alex Shafarenko. S+Net: extending functional coordination with extra-functional semantics. Technical Report arXiv:1306.2743v1 [cs.PL], University of Amsterdam and University of Hertfordshire, June 2013. Available from: http://arxiv.org/abs/1306.2743.

[7] Raphael 'kena' Poss. The essence of component-based design and coordination. Technical Report arXiv:1306.3375v1 [cs.SE], University of Amsterdam, June 2013. Available from: http://arxiv.org/abs/1306.3375.

[8] Alex Shafarenko. Non-deterministic coordination with S-Net. In Wolfgang Gentzsch, Lucio Grandinetti, and Gerhard Joubert, editors, *High Speed and Large Scale Scientific Computing*, number 18 in Advances in Parallel Computing. IOS Press, 2009. ISBN 978-1-60750-073-5. doi:10.3233/978-1-60750-073-5-74.

[9] Paul Viola and Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004. ISSN 0920-5691. doi:10.1023/B:VISI.0000013087.49260.fb.

[10] Jeroen Voeten. On the fundamental limitations of transformational design. *ACM Trans. Des. Autom. Electron. Syst.*, 6(4):533–552, October 2001. ISSN 1084-4309. doi:10.1145/502175.502181.