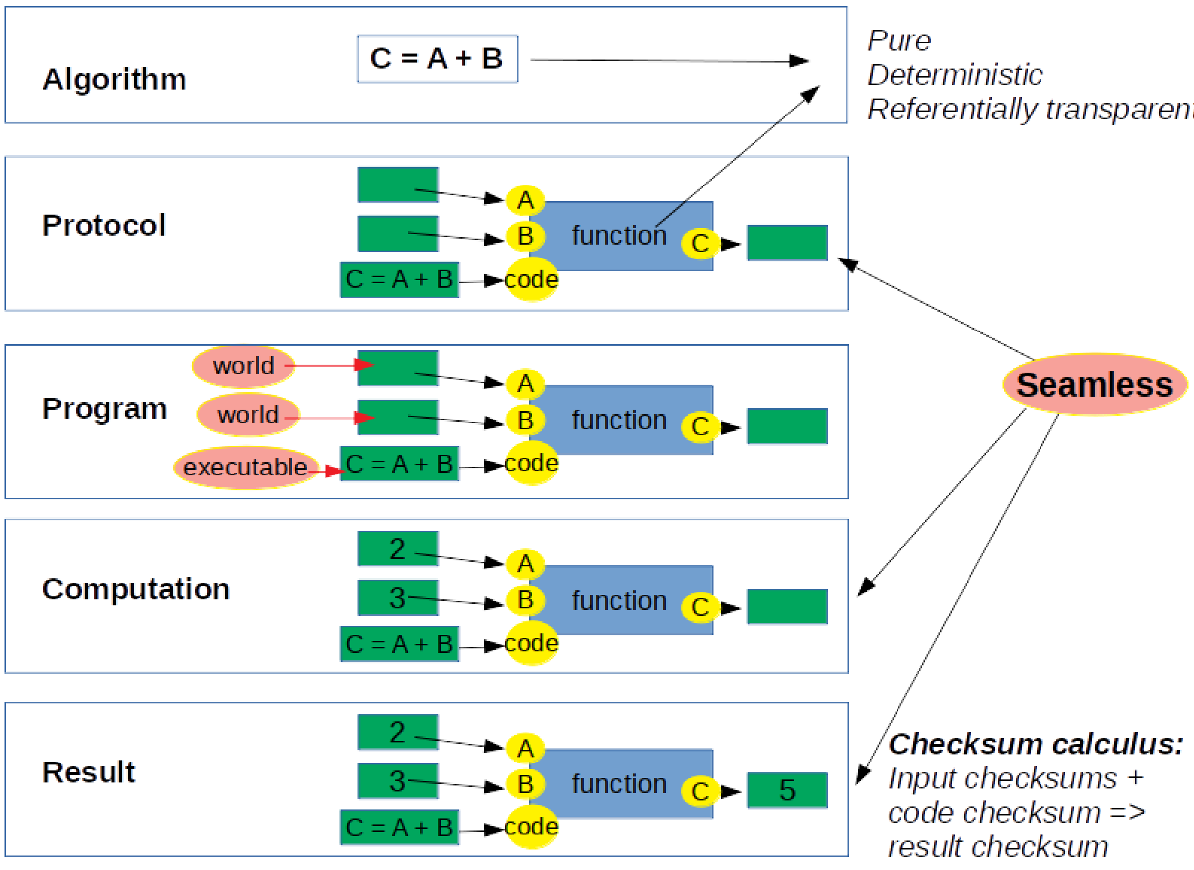
# Theory and implementation of strong interoperability in scientific computing

Sjoerd J. de Vries, April 2019.



## A. Scientific protocols and computations

A **computation** is a function that operates on a defined set of inputs to produce a result. A proper **scientific computation** operates on empirical facts to produce a **deterministic** result. This means that the inputs are constant and the function must be **pure** (free of side effects). A proper scientific computation is therefore **referentially transparent**: it can be replaced with its result. It is also **composable**: a directed acyclic graph (DAG) of computations is itself a computation. A **protocol** is a (graph of) computation(s) where inputs must be provided by the user.
Pure functional programming (FP) languages such as Haskell can be used to implement proper scientific *computations*. Unfortunately, scientific *protocols* must be incarnated in a **program**, which is the protocol plus instructions on how to obtain its inputs. In Haskell, these are implemented as “awkward” (ref 1) *IO monads*, i.e. as **interactions** with the outside world. Execution of the program involves impure operations through the IO monads, interleaved with pure computations on the rest of the program (ref 1). In other words, even in Haskell, proper scientific protocols are not possible. In any case, pure FP languages are deeply impopular (to the point of obscurity) among scientists, who prefer imperative languages such as Python, C and Fortran.
In addition to pure FP languages, there exist a large number of scientific **workflow systems** that address the problem of scientific computation (refs: Taverna, Galaxy, CWL, OpenAlea, SnakeMake, Reflow). By replacing FP language constructs with domain-specific syntax (or even visual programming) these systems are much easier to use. Some of them (Reflow) explicitly aim for referential transparency, whereas others are just pragmatic wrappers around shell commands. This second category shows some overlap with software build systems such as Make or even Excel (see ref: “Build systems a la carte” for an excellent review on these), which are occasionally used to perform scientific computations.
While superficially similar, data flow systems (Naiad, NoFlo), reactive systems (Rx libraries) and Functional Reactive Programming tackle a very different problem, namely computations on graphs that connect dynamic data streams, instead of constant inputs.

None of these systems address the problem that interaction with the outside world is fundamentally impure. In other words, proper scientific protocols are currently impossible. Since in a scientific context, inputs are constant, the problem can be somewhat mitigated by interacting only with static resources (files and URLs), but this requires that these resources are **findable** (well-named) and **accessible** (well-connected). This has serious implications for (weak) scientific *interoperability*, as decribed below.

**B. Scientific interoperability**
**Interoperability** can be defined as establishing a **similarity relation** between two protocols (or computations) P and Q.
This similarity relation can be:
1. P and Q are identical. In that case, their results must be identical, regardless of execution environment (**reproducibility**).
2. P is a subset of Q. In that case, the result of P must be usable as an input to the rest of Q (**reusability**).
3. P is a modification of Q, i.e. some, but not all, elements of Q are re-used by P. In that case, only the modified elements need to be re-implemented and re-evaluated (**incrementality**).
4. P and Q are both subsets of a protocol PQ, i.e. they exchange inputs and outputs within the context of PQ (**composition**).

Where interoperability fails, P and Q must be considered as separate protocols, requiring independent implementations. When they are in fact related, this means “re-inventing the wheel”. In a social scientific context, this is disastrous, for two reasons:
- Protocols produce scientific insights. For efficient scientific progress, insights obtained using P must be generalizable to similar protocols Q, and within composite protocols PQ (“standing on the shoulders of giants”).
- Protocols are used to make predictions, and research groups constantly attempt to improve the predictive power of their protocols. For efficient scientific progress, improvements to P must be generalizable to PQ (Unix philosophy: doing one thing well).

**Weak interoperability** can be defined as the establishment of interoperability through *metadata*. Metadata describe the format, origin and properties of scientific data, and of the protocols that produced them (provenance). They are traditionally defined in the Materials & Methods section of a scientific article. Weak interoperability implies *weak reproducibility,* which means that a sufficiently expert human is able to understand the metadata, reconstitute the protocol, and obtain results that are (within a certain precision) identical. From this, interoperability arises naturally by the ability of such a human expert to design similar protocols.
Metadata include names, identifiers, version numbers, and time stamps, of which the systematization greatly facilitates interoperability. A well-known traditional example is the systematic Latin naming of biological organisms established by Linnaeus. In more recent times, the **FAIR (Findable, Accessible, Interoperable, Reusable) data** initiative (ref) constitutes a comprehensive set of principles, systematization guidelines and metrics to establish weak interoperability. A heavy emphasis is on computer-assisted metadata: the establishment of machine-readable, community-standard formats that capture semantics (meaning and intent), that are findable under unique and persistent content identifiers, and are accessible via standardized communication protocols. By following these principles, data retrieval and analysis by both humans and machines becomes greatly facilitated. However, as they are not necessary to obtain immediate results, these metadata put a burden on researchers who must provide them. In other words, while the fruits of interoperability (a more open and collaborative mode of scientific research) are enjoyed by the community as a whole, the price of interoperability is paid by the individual researcher. As long as there are no strong social incentives to pay this price (fame and recognition for providing elaborate metadata, or making them a condition for funding and publication), interoperability essentially depends on altruism and idealism.

Is it possible to reduce the burden of interoperability? In experimental science and engineering, protocols interact with the natural world in an imprecise manner (measurement errors, imperfect replication of conditions). This means that they are not deterministic: weak interoperability (including weak reproducibility) is the best what is possible. Also, as they typically involve expensive equipment, there is already an economical incentive for the maintenance of protocol metadata (albeit not necessarily for their publication). For experimental science, promoting FAIR data principles appears to be the best possible strategy to achieve interoperability.

In contrast, in computational science, proper scientific protocols and computations are deterministic. This makes it in principle possible to achieve **strong interoperability**: interoperability that arises from defining a computation so precisely that automatic, machine-level recognition of similarity relations becomes possible. This requires a model of scientific computation that is **universal** and **abstract**, i.e. that can represent any scientific computation without being tied to concrete details about programming syntax or about interaction with the outside world. In such a model, metadata still play a role, but only in a social context, to make a computation more understandable to humans (annotation, documentation). This is in contrast to weak interoperability, where metadata (findable and accessible identifiers) are a requirement to *define* a computation.

## C. Checksum calculus: a universal model for scientific computation

Here, **checksum calculus** is proposed as a model for scientific computation. It defines a single operator, **transform**, that takes as input an arbitrary (but pre-defined) number of data inputs and a single source code input, all of which are constants. The transform operator then produces a single result in a deterministic manner. All inputs and results are defined by their value, rather than by file names, URLs, or other identifiers, and each value is described by its checksum (computed using the SHA3-256 algorithm). Values can be composite (nested lists and/or trees), and their checksums may be represented as Merkle trees. Both values and computations can be cached using checksums: only when values are actually needed, they are retrieved.
Checksum calculus is a **universal** computation model. Note that “universal” is distinct from “Turing complete”: there are many pure, Turing complete computation models where computations are composed from operations on very few different values (for some models, such as untyped lambda calculus, without any values at all). In contrast, checksum calculus is universal *and* Turing complete because each transformer can choose from an infinite set of values that encode the tape (input data) and state register (source code) of a Universal Turing Machine. This infinite set is represented by the set of checksums, which is infinite in terms of practical similarity relations (the chance that two non-identical values collide into the same checksum value is vanishingly small). In addition, checksum calculus is universal because it does not prescribe a particular mode of how to interpret the source code, as long as such a mode has been defined. In other words, source code inputs are **polyglot** (can be written in any programming language), as long as the code is deterministic and has no hidden dependencies. While the current implementation (Seamless) supports only a few programming languages, supporting a new language has well-defined requirements that need to be implemented only once per language.
Finally, the format for data is rather flexible and can accommodate JSON, binary structs, and any nested mixture thereof. Constraints on the data can be expressed using JSON Schema (ref) augmented with invariants (validators) expressed in any programming language, which means that arbitrary **dependent types** are supported.

## D. Seamless: a framework for strong interoperability based on checksum calculus

I have implemented checksum calculus in a programming framework, **Seamless** (ref), with the aim of achieving strong interoperability. Seamless works with graphs (DAGs) of checksum calculus transform operators (**transformers**), operating on stateful **cells**. The Seamless library, written in Python, consists of a high-level API to build graphs, a mid-level library that translates a graph into a runtime representation, which is then evaluated by a low-level execution engine. All three parts are independent and replaceable pieces of software: graphs could be constructed using an unrelated API, Seamless-constructed graphs could be executed using an unrelated execution engine, or both.
The essential element is the checksum calculus graph format (which will be formally standardized in the future). In Seamless, these graphs are used as a unified representation of *scientific protocols* (graphs where one or more input cells are undefined), *scientific computations* (graphs with all input cells but not all output cells defined) and *scientific results* (graphs where all cells have been defined) (Figure 1). Graphs contain no values but only checksums, and are therefore small in size, but nevertheless well-defined (as there are no files, URLs, or other identifiers that cannot be computed from the cell values).
Notably absent is the representation of a **program**, i.e. a protocol plus a mechanism to convert it into a computation. This omission is deliberate, following the “programs considered harmful” conclusion from section A. Instead, while running, Seamless allows a protocol to be curried with checksums, eventually creating a computation. Such currying events are considered as **acts-of-authority**, arising spontaneously from any source. Concrete implemented sources-of-authority are the IPython/Jupyter REPL, the mounting of

cells onto the file system, cell modification through a REST interface, and network communication from other Seamless instances. In addition to checksum definition (and re-definition) events, the first and the last source also support the spontaneous re-definition of parts of the graph.

Another source-of-authority can be the graph itself. In addition to transformers, graphs may contain **macros** (operators that produce a connectable subgraph rather than a computation result) and **reactors** (operators that hold state and can produce multiple outputs). In addition to data cells, macros take as input a single graph-generating code cell, whereas reactors take three code cells (start, update and stop). Reactors can be used as optimized transformers (that respond more efficiently to cell changes), but they can also be set up as a source-of-authority for a targeted cell (which is distinct from having the cell as an output; all outputs must be deterministically computed from the inputs, whereas acts-of-authority arise spontaneously). Combined with the fact that untranslated graphs are themselves just JSON data (and can be the value of a cell), it is possible to implement a wide variety of syntactic constructs and embedded data editors inside Seamless. Changes to cell values are never modeled explicitly over time: a modification of a cell is always based on observational equivalence with “the cell having had that value since the beginning”, i.e. “history doesn’t matter” (similar to a spreadsheet). In combination with incremental re-evaluation in reaction to any modification (of data, code or graph), this makes programming with Seamless a distinctly interactive experience.

Strong interoperability is primarily achieved through distributed caching and deployment. This is possible because 1) each computation is independent, and 2) protocols and computations are well-defined before they are executable: values are only needed at the moment of execution. Until then, graph descriptions are small and can be easily communicated.
Formal reasoning with computations is therefore possible at an early stage. One example is a caching server that stores the results of computations. Because of checksum calculus, a computation solves a theorem of the form: input checksums + code checksum => result checksum, a description that is both universal and small to store. Any computation for which its solved theorem is found will not have to be executed. It is conceivable that all time-consuming scientific computations ever performed could be stored on a single machine, but load balancing and redundancy are of course desirable.
A server for the reverse form, result checksum <= input checksums + code checksum, is essentially a universal provenance server. Through repeated requests, a computation could be traced all the way back through its original empirical data and parameters. To obtain a human-understandable context, this should be combined with requests to a checksum => metadata server.
For the caching of values, a different strategy is followed. Note that Seamless has no concept of programs that contain explicit instructions (code and/or identifiers) to obtain values. Instead, Seamless delegates obtaining values to any data source available: Redis checksum-to-value caches, lists of resource identifiers (filenames, cloud identifiers and/or URLs), or torrents. Seamless queries all registered data sources until a hit has been obtained. Value requests can also be forwarded to networked Seamless instances that accept them: this allows Seamless to be set up as a (reverse) proxy server for value caching.
Requests to perform computations can be delegated in the same way, to Seamless instances that execute them locally or that forward them further. In this way, a computation graph can be automatically split into individual transforms, and forwarded to a downstream pool of Seamless instances configured as simple execution slaves. However, the fact that the computations are already well-defined allows more sophisticated distribution/deployment policies. This involves the detection of particular subgraphs or computations by specialized execution slaves. For example, a Seamless instance can be configured to accept only computations that operate on a particular input checksum, bringing the computation to where the data is. This decision can be made long before the data is physically present. In addition, a remote web server for program X can be emulated through a specialized Seamless instance, specifying the restriction that the code checksum can only be that of X. Users can submit well-defined computations to the remote web server even if they do not possess the source code of X, only its checksum. Since they are well-defined and have provenance, scientific results obtained in this manner can be automatically validated using the computer power of a trusted re-computation agency (e.g. a scientific journal) that does have the source code of X.
Note that computation requests are always small. If the computation has been computed already, the response is also small (a single result checksum). Potentially voluminous checksum-to-value queries only happen if a) the computation needs to be executed, and the executing Seamless instance needs the input values (which may be obtained from local cache or from the Seamless instance that made the request), or b) the value of the result is actually needed (which may not be the case for an intermediate result).

## E. Seamless versus existing systems

To the best of my knowledge, Seamless is the first formulation and implementation of a universal computation model that leads to strong interoperability. Several of its concepts have been recognized before, though. Universal computation systems have been created before, including the use of checksums to uniquely identify a computation (ref. CIEL). In addition, checksums are widely used for incremental computation in build systems and workflow systems (ref Bazel, Shake), and sometimes, for a unified description of computation and result (ref. Nectar/DryadLINQ).
However, even the CIEL system is not a truly universal system, since it requires the computation to be defined using the CIEL API instead of a universal abstract DAG format. Conversely, a system like Musketeer (ref) that aims to decouple APIs and execution engines using a DAG, is still based on domain-specific primitives and does not provide a universal system for pure user-defined functions in an arbitrary language.
More importantly, in all of these systems, checksums are used only as ephemeral values within the execution engine, and not to achieve purity (i.e. liberating the computation's dependence on files and URLs). In other words, they are exploited for their convenience, not to establish a model of strong interoperability.

The same can be said of interactive programming in the context of notebooks, Smalltalk environments and similar efforts to create an interactive programming experience. In Seamless, interactivity arises naturally from the underlying computational model, as a beneficial side effect of **strong incrementality**, which means that re-submitting a modified graph becomes automatically cheap. Non-pure systems can achieve such interactivity only (if at all) by continuous polling of their external resource dependencies.

Strong interoperability implies **strong reproducibility**, namely that byte-identical results are obtained when a computation is repeated. In itself, strong reproducibility of a computation is already achievable through virtualization technology such as Docker, placing the entire computation in a self-contained black box that does not communicate with the outside world. This is however strongly discouraged by the Docker community (ref Docker best practices), for good reason, since every form of composition and reusability becomes impossible. Instead, Docker containers are recommended to be ephemeral/stateless (ref Decoupled Docker), with well-defined communication between containers (i.e. not unlike Seamless graphs), but this does no longer give strong reproducibility.

Domain-specific systems for Big Data / cloud computing (e.g. Spark, Hadoop) are well-optimized in terms of execution and deployment. In principle, Seamless graphs could benefit from this. Being a theory, checksum calculus provides an *abstract* model of computation. It is not required that the computation is actually performed using the code that corresponds to the code checksum: only that the result is the same. This means that more efficient execution (and improved interoperability) can be obtained by converting a Seamless graph to run on an optimized domain-specific system.
For Big Data / cloud computing, systems consist of a front-end API coupled to a back-end executor. In between, many of them use a DAG internally, and there are systems such as Musketeer that interconvert between Big Data front-ends/back-ends through a common DAG format of primitive functions that are general-purpose, yet domain-specific for Big Data. The conversion of a Seamless DAG to e.g. a Spark DAG would involve deployment of all input data as (temporary) resources in the cloud, and the detection of Seamless code checksums or subgraphs that correspond to Spark primitives, such as map() and filter(). The same applies to machine learning (Tensorflow) and scientific workflow systems (Common Workflow Language ) that use a general-purpose, domain-specific DAG.
In principle, conversion to Seamless is also possible. Since Seamless has a universal DAG model where functions can also be DAG-generating macros, a domain-specific DAG can be converted into Seamless as long as the DAG semantics correspond to the scientific model (constant inputs, pure functions) and an implementation as Seamless function/subgraph/macro is available for each primitive. Conversion to Seamless makes the DAG pure, well-defined and delocalized as any references to files and URLs are replaced by checksums.

## F. Seamless, type theory and the burden of interoperability

As discussed in section B, weak interoperability imposes a burden on the computational researcher, which a strong interoperability solution such as Seamless can eliminate. The question is how much of a burden still remains. While metadata and systematic knowledge systems are no longer necessary, some discipline is still required to code functions that are pure (even though they use temporary state internally), but I believe that pure code is good practice in computational science under any circumstances. Some scientists prefer to code in an object-oriented style, which is a bit problematic for cells, but adding syntax support for user-defined methods to Seamless cells turned out to be not so difficult.
More serious is the burden of novelty that Seamless requires: building a graph instead of a program, not using files as the primary storage medium, delocalized data and computation, and, on the short term: bugs and cryptic error messages. Still, referential transparency and automatic interoperability come with many advantages, including interactive programming, caching, incremental computing, parallel execution, distributed deployment, polyglot programming, reproducibility, provenance, and reasoning about datasets larger than physical memory (aka "Big Data"). I have good hope that this will provide a "carrot" that will encourage researchers to bear the remaining burden.
One concern that I have not mentioned much is the requirement of a precise semantic model to execute source code. At one side, this can be remedied by making code inputs provide *all* the code (i.e. a list of source code texts that Seamless will compile, including compilation flags), on the other side by running commands in checksum-labeled Docker containers. Even so, there are plenty of cases where semantic ambiguity remains. If a code cell is written in C++, which is the language version? What is the compiler? Which system libraries are being used? A reference implementation could be chosen (i.e. "python" may stand for CPython 3.6.7, compiled on 64-bit Linux), and may be tied to a certain timestamp (i.e. for graphs labeled "June 2020", "python" may default to PyPy 3.7, instead). In other words, a weak interoperability solution. I believe that this will be not so much of a practical problem, since most languages and libraries are forwards compatible in the large majority of cases (and if not, a large majority of the remaining cases will simply result in an error, which is infinitely preferable over a wrong result!).
A second overlooked concern is the role of types in interoperability. Dependent types are immensely useful in a user interface context, since they generate highly domain-specific error messages on the web form (for example, that the user-selected molecular scoring function is incompatible with more than one iteration of the user-selected energy minimization algorithm, unless more than two molecules are being modelled; such type constraints are heavily used in my HADDOCK and ATTRACT web interfaces). For interoperability, it would be welcome if type constraints could benefit from some form of automated inference, to guarantee that a certain protocol will always (or never) maintain its type constraints, without running the computation.
For both concerns, I feel that much promise is to be expected from progress in type theory and type inference. For a long time, there has been the concept of **proof-carrying code** (ref Necula 1998). This involves a program that can inspect binary code to generate verification conditions that must hold to maintain the specified type constraints. An external theorem prover can then be used to find a proof for these conditions, and such a proof can be easily verified.
Seamless is in principle well suited to this. Compared to a monolithic executable, Seamless computations are modular graphs of transformations that are executed essentially independent from each other. There is a strict distinction between ephemeral state (temporary variables) and persistent state (the inputs and result of computation), and because of their aforementioned practical value for user interfaces, dependent types are expected to be abundant. Combined with the advent of languages like Idris (purely functional languages with dependent types), it may become possible to prove formally that specified type constraints hold (or to infer even tighter type constraints), and perhaps, that a particular change in code version has no effect. This may lead to even stronger forms of interoperability.

**References:**

**(ref 1): https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/mark.pdf?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Fsimonpj%2Fpapers%2Fmarktoberdorf%2Fmark.pdf**

**Musketeer: http://ionelgog.org/data/papers/2015-eurosys-musketeer.pdf**
**CIEL: http://anil.recoil.org/papers/2011-nsdi-ciel.pdf**
**Docker best practices: https://docs.docker.com/develop/develop-images/dockerfile_best-practices**
**Necula 1998: http://www.cs.berkeley.edu/~necula/Papers/pcc_lncs98.ps**