Control-Driven Data Flow

Antoniu Pop, Albert Cohen

Control-Driven Data Flow

Antoniu Pop, Albert Cohen

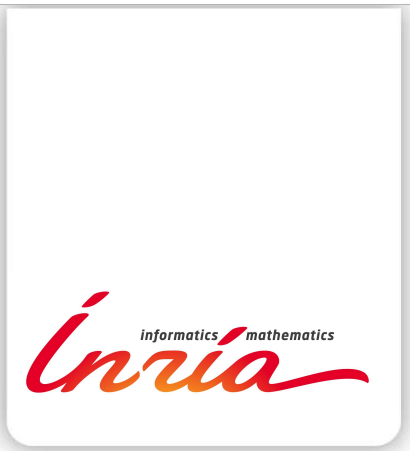
RESEARCH REPORT

N° 8015

July 2012

Project-Team PARKAS

ISSN 0249-6399 ISRN INRIA/RR--8015--FR+ENG



Control-Driven Data Flow

Antoniu Pop, Albert Cohen

Project-Team PARKAS

Research Report n° 8015 — July 2012 — 36 pages

Abstract: This paper presents CDDF, a model of computation underpinning the formal semantics of a number of parallel programming languages. CDDF integrates control flow elements for the dynamic construction of task graphs, and data flow elements to express dependent computations and to decouple these using unbounded streams (Kahn process networks). It is a common ground to define the formal semantics of imperative programming languages with dynamic task creation, as well as data-flow or concurrent functional languages, as a special case of more general dependent task languages with channels or streams. We prove essential properties for languages fitting this model of computation, including deadlock-freedom, functional and deadlock determinism, and serializability. We also compare the model's hypotheses with Cilk's strictness and the Kahn principle.

Key-words: Model of computation, operational semantics, programming languages, data-flow, stream computing, parallel programming.

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Control-Driven Data Flow

Résumé : Ce papier présente CDDF, un modèle de calcul qui permet de donner une sémantique formelle a un certain nombre de langages de programmation parallèle. Il intègre des éléments de contrôle de flot pour la construction dynamique de graphes de tâches, ainsi que des éléments de flot de données pour exprimer les dépendances des calculs et découpler ceux-ci à l'aide de streams non bornés à l'instar des réseaux de processus de Kahn. Le modèle CDDF fournit une base commune pour définir la sémantique formelle des langages impératifs à création dynamique de tâches, ainsi que des langages fonctionnels concurrents ou à flots de données, en tant que cas particulier de langages à tâches dépendantes avec cannaux de communication ou streams. Nous prouvons des propriétés essentielles sur les langages basés sur ce modèle, dont l'absence d'interbloquages, le déterminisme fonctionnel et d'interbloquage, ainsi que la sérialisabilité. Nous comparons les hypothèses utilisées dans CDDF à la condition d'exécution stricte de Cilk ainsi qu'au principe de Kahn.

Mots-clés : Modèle de calcul, sémantique opérationnelle, langages de programmation, flot de données, calcul par streams, programmation parallèle.

1 Introduction and Related Work

The principal motivation for research into data-flow models comes from the incapacity of the von Neumann architecture to exploit massive amounts of fine-grain parallelism. The early data-flow architectures [10, 9, 29] avoid the von Neumann bottlenecks by only relying on local memory and replacing the global program counter by a purely data-driven execution model, executing instructions as soon as their operands become available. Programmer productivity is another important motivation: debugging concurrent applications with low level threads is a daunting task, mostly because of the non-deterministic nature of races and deadlock-related errors. Data-flow languages closely follow the hardware model: the execution is explicitly driven by data dependences rather than control flow [17]. Data-flow languages offer functional and parallel composition of parallel programs preserving functional determinism. Recent data-flow architectures, execution models, and languages rely on the same principles, albeit at a coarser grain, executing sequences of instructions, or data-flow threads, instead of single instructions.

Among the most notable data-flow languages, Lucid [3] relies on the `next` keyword within loops to achieve a similar effect to advancing in a stream of data, by consuming or producing in a channel, or, in the synchronous languages domain, to the advancement of clocks on signals. Sisal [12] explicitly introduces the notion of stream, which is naturally very close to lists. If stream processing systems are understood as the parallel implementation of stream transformers, which is the functional interpretation of a process network mapping a set of input streams to a set of output streams, then any functional language can be used for stream programming. This corresponds to the lazy interpretation of functional languages; see [7] for a Haskell [16] implementation of Lucid Synchrone [6]. First-class streams of data improve expressiveness for a variety of communication and concurrency patterns such as broadcast, delays, and sliding windows. This was observed by data-flow computing pioneers, who designed I-structures as unbounded streams of futures to alleviate some of the overheads of a pure data-flow execution model [2]

Kahn process networks (KPN) [18] capture the essence of stream computing. Processes communicate through unbounded FIFO channels, write operations are non-blocking, and read operations wait until sufficient data is available. Assuming processes are deterministic, a key property of this model is that the network as a whole is also functionally deterministic by composition. In his survey [27] of stream processing, Stephens classifies stream processing systems based on three criteria: synchrony, determinism and the type of communication channel. Fundamentally, stream-based models of computation all share the same structure, which can generally be represented as a graph, where computing nodes are connected through streaming edges. However, cyclic networks can lead to deadlocks or unbounded growth of in-flight data, which has spurred the development of a restricted form of Kahn Process Networks, Cyclo-Static Data-Flow (CSDF) [20, 5]. While processes in KPNs execute asynchronously and can produce or consume variable amounts of data, CSDF processes have a statically defined behaviour. With rates of production and consumption known at compile time, it is possible to statically decide whether the execution is free of deadlocks and to statically schedule the execution. It can also guarantee the absence of resource deadlocks when executing on bounded memory, a realistic restriction. StreamIt [28] is a recent instantiation of the synchronous data-flow model, building on the strong static restrictions of the underlying model to enable aggressive compiler optimizations, it achieves excellent performance and performance portability across a variety of targets [13] for a restricted set of benchmarks that properly map on this model. Data-flow synchronous languages such Lustre [14] have been widely adopted in the certified design flow of reactive control applications. They offer determinism, deadlock-freedom, bounded reaction time and memory. Unlike CSDF, they are not restricted to periodic activations and communications.

Processes respond instantly and communicate through signals, also used to define a notion of time and causality. Signals differ from streams in that they are sampled rather than consumed.

Independently, parallel programming languages have seen the emergence of a rich set of constructs to express inter-task dependences. StarSs [22] is a pragma-based language to program distributed-memory and heterogeneous architectures; it supports both data-flow and control-flow programming styles. SMPSs is one of the StarSs incarnations for shared-memory targets [21]. X10 [8] and Habañero Java [26] feature dynamic clocks and phasers, generalizing barrier and point-to-point synchronization. Unlike most streaming languages, these parallel languages also support dynamic task creation. Recently, we proposed a stream-computing extension of OpenMP called OpenStream with an unprecedented combination of expressiveness, dynamicity, and compilation to an efficient model of execution [25]. OpenStream supports dynamic task creation, variable and unbounded sets of producers/consumers, separate compilation, and first-class streams. Up to now, these highly expressive languages lacked a formal semantics and proofs of their important properties.

In this paper, our objective is to provide a formal framework to analyze the operational semantics of parallel data-flow languages and show that determinism can be achieved without sacrificing expressiveness. We introduce a new model of computation, CDDF, that captures the semantics of the OpenStream programming model and, by extension, that of any less expressive programming paradigms. CDDF relies on Kahn-like unbounded streams for communication, although CDDF streams support variable and unbounded sets of producers and consumers: a CDDF task graph is actually a dynamic hypergraph. This is achieved by enforcing a deterministic interleaving of accesses in streams, modeled as infinite indexed arrays, thus eliminating the non-determinism issues resulting from the ordering consensus of concurrent accesses to a FIFO channel. This order of attribution of indexes in streams is called the *schedule* of access indexes, a central element of CDDF built by the *control program*. The notion of control program is necessary to allow the dynamic construction of task graphs, with arbitrarily predicated task creation and dynamically selected stream connections.

Our model relies on a trace-based operational semantics where we abstract many of the characteristics of tasks, focusing simply on individual activations, which are represented by the set of stream accesses they perform. This is the minimal information required to characterize their dependences and reason about the scheduling requirements. This low-level abstraction permits a high level of precision on the conditions required to provide the different guarantees proved with this model, generally leading to not only sufficient, but also necessary conditions. More specifically, the CDDF model allows us to prove that, under certain conditions, programming languages implementing this model are guaranteed to be: (1) deterministic, both functionally and for the state of the program when a deadlock occurs; (2) free of spurious deadlocks introduced by the operational semantics itself, limiting the cause of deadlocks to algorithmic errors in the source program; and (3) serializable. Furthermore, all of these proofs are made despite the additional difficulty that dependences are not only enforced by classical Bernstein conditions [4], but also by a Kahn-inspired prefix-closure of stream indexes, therefore ensuring that runtime stream synchronization can be efficiently implemented on conventional multiprocessors [24].

The paper is organized as follows. Section 2 presents the CDDF model, defining its new concepts and notations. Section 3 introduces a new concept, stream causality, which is a less restrictive form of the Cilk [11] strictness condition, and proves that stream causality is a sufficient condition to avoid deadlocks. Section 4 presents a complementary and more relaxed form of causality, task causality, which is similar to process continuity in Kahn networks, and shows that no spurious deadlocks can occur in task-causal programs. Independently, we also show that only a limited, functional form of deadlocks can occur in programs where the hypergraph contains no strongly connected components. Section 5 proves that all CDDF programs benefit from

functional and deadlock determinism. Section 6 shows that stream communication provides a very strong form of consistency, strict consistency, and proves necessary and sufficient conditions for two forms of serializability. Section 7 summarizes the properties of CDDF programs and the conditions required to prove that such properties hold. Finally, Section 8 concludes this paper. Throughout the paper, longer proofs are detailed in the appendix.

2 Definitions and Notations

Informally, a programming paradigm implementing the CDDF model of computation is built around three key concepts: a *control program*, *streams* and *tasks*. The control program is, for example, the main program in OpenMP or Cilk and is responsible for spawning tasks, with an explicit specification of task dependences, at the spawning point, in the form of data produced or consumed on streams. CDDF tasks are a simple form of communicating coroutines: like data-flow threads, they run to completion (no synchronization once a task starts executing) and communicate through point-to-point data copying, using streams exclusively. The latter restriction is meant to avoid considerations on the memory model: shared-memory communications and in-place updates are not disallowed, but their semantics is not covered by CDDF and must be described separately for languages providing such options. Furthermore, we rely on the following simplifying assumptions to focus the presentation on the core concepts and results: (1) the memory space is unbounded, which allows us to model stream accesses in (dynamic) single assignment; and (2) the control program is a deterministic sequential process. Lifting these assumptions is out of the scope of the paper [23].

2.1 CDDF Program Structure

Our model focuses on the communication patterns in streams and on the scheduling constraints resulting from data dependences. We define them as mere identifiers for now.

Definition 1 (Stream)**.** A stream is a symbol $s \in \mathcal{S}$, where $\mathcal{S}$ is an infinite set of symbols.

Streams are more expressive than FIFO queues: they are indexed in the set of natural numbers. Thanks to the unbounded memory assumption, they can be seen as infinite arrays: an injective function from $\mathcal{S} \times \mathbb{N}$ maps streams and indexes to (possibly typed) memory locations, with no intra- or inter-stream aliasing.

Definition 2 (Stream access)**.** We define the set $\mathcal{X}$ of stream accesses, where we distinguish between read (R) and write (W) accesses:

$$\mathcal{X} = \{R, W\} \times \mathcal{S} \times \mathbb{N}$$

Contrary to existing streaming models, we do not give much importance to the notion of a repetitive filter or process, corresponding to the iterative creation of tasks applying the same work function to a given set of input and output streams. As our model is inherently dynamic, it tends to be less focused on regular communication patterns. Instead, our abstraction only keeps the notion of *task activation*, in the form of one execution of a work function. Task activations are the fundamental and atomic unit of work.[1] As work functions can be called with different input and output streams, they cannot constitute or fully identify filters/processes.

We further abstract the notion of task activation to only consider the stream accesses necessary for the task activation's execution, irrespectively of the work function applied.

[1] In this context, atomicity refers to task activations being run to completion: once a task starts executing, it does not need to wait for external events.

Definition 3 (Task activation)**.** A task activation a is defined by the set of stream accesses it uses to read and write data in streams:

$$a \subset \mathcal{X}$$

The set $\mathcal{A}$ of task activations is the powerset of $\mathcal{X}$:

$$\mathcal{A} = \mathcal{P}(\mathcal{X})$$

This view is entirely focused on the streams and stream indexes that are read and written during the execution of the task activation. This represents the minimal information to characterize the data dependences between task activations.

Definition 4 (Input and output streams)**.** We say that a stream $s \in \mathcal{S}$ is an input (resp. output) stream to a task activation $a \in \mathcal{A}$ and we write $s \in \mathcal{I}(a)$ (resp. $s \in \mathcal{O}(a)$) if the task activation a contains a read (resp. write) access to stream s:

$$\mathcal{I}(a) = \{s \in \mathcal{S} \mid \exists i \in \mathbb{N}, (R, s, i) \in a\}$$

$$\mathcal{O}(a) = \{s \in \mathcal{S} \mid \exists i \in \mathbb{N}, (W, s, i) \in a\}$$

The core of our model is the *control program*, which dynamically and deterministically attributes indexes to data in each stream, based on the dynamic control flow of the control program. We refer to this attribution as the *schedule* of access indexes in a stream. As we are not interested in the semantics of the underlying programming language, we only require the execution of the program to be non-blocking aside from synchronization barriers. We model the control program by its execution trace where we only keep two types of operations: task activation points and barriers. All other operations are omitted from the trace.

Activation points can be thought of as task spawn calls and their evaluation results in the creation of new task activations. The barrier's informal semantics is to stall the control program until the completion of all task activations it produced up to the barrier.

Definition 5 (Activation point)**.** An activation point π is a statement of the control program that generates a task activation. We do not care about the precise work associated with this task activation, but only about the effect on its input and output streams. We thus abstract an activation point as a finite set of descriptors of its stream accesses:

$$\pi \subset \left\{(u, s, b, h) \in \{R, W\} \times \mathcal{S} \to \mathbb{N} \times \mathbb{N}\right\}$$

where each descriptor is a function from (u, s) to (b, h), u determines if the task activation will consume from or produce to stream s, b is the burst, or number of data items produced or consumed, and h is the horizon, which is the amount of data accessed in the stream. The horizon is greater than or equal to the burst; when strictly greater it allows read accesses ahead into the stream, which is also called a *peek* operation in related work.

The set of valid activation points for any control program is defined as:

$$\Pi = \Big\{\pi \in \mathcal{P}\big(\{R, W\} \times \mathcal{S} \to \mathbb{N}^2\big) \mid (R, s, b, h) \in \pi \Rightarrow b \leqslant h \\ \wedge\ (W, s, b, h) \in \pi \Rightarrow b = h\Big\}$$

The restrictions on valid activation points are meant to prevent read descriptors from discarding data, in the case where the burst would be greater than the horizon, and write descriptors from both publishing uninitialized stream locations or writing ahead, which would break the single-assignment property on stream accesses. Without loss of expressiveness, we also disallow multiple read (resp. write) accesses to the same stream at a given activation point.

The semantics of the control program is operationally defined as a transition system. We use partial execution traces to label the program states. These states are ordered by prefix inclusion on traces, relying on the sequential hypothesis on the control program.

Definition 6 (Control program trace)**.** The execution trace of the control program is a possibly infinite word on $\Pi \cup \{barrier\}$. The set $\mathcal{K}$ of partial execution traces recognized by the control program's transition system is defined as:

$$\mathcal{K} = (\pi + barrier)^*$$

Using the notations above, we can define the state of a CDDF program as its (partial) execution trace and two sets of task activations, one for already executed activations and the other for outstanding task activations.

Definition 7 (CDDF program state)**.** We define the set Σ of possible program states:

$$\Sigma = \mathcal{K} \times \left(\mathcal{P}(\mathcal{A})\right)^2$$

A state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o) \in \Sigma$ of the CDDF program can be identified by the (partial) trace k_e of the control program, and the sets $\mathcal{A}_e$ of executed task activations and $\mathcal{A}_o$ of outstanding task activations.

Definition 8 (Control program execution)**.** The abstract execution of the control program is modeled by two functions: (1) an oracle function $\Omega : \mathcal{K} \to \Pi \cup \{barrier\} \cup \{\top\}$—where $\top$ marks the end of the control program—abstracts the sequence operator and provides the next operation to be executed by the control program; and (2) the activation point evaluation function $\xi : \mathcal{K} \times \Pi \to \mathcal{A}$ defining how an activation point is evaluated to generate a task activation:

$$\xi(k, \pi) = \left\{ (u, s, i) \in \mathcal{X} \mid \exists (u, s, b, h) \in \pi \ \wedge \ i \in \left[\alpha, \alpha + h\right[\right\}$$

where $\alpha = \sum_{\substack{\pi' \in k \\ (u,s,b',h') \in \pi'}} b'$.

For the new task activation generated by the control program at a given activation point, ξ determines the location of data produced or consumed on each stream. The activation point is identified by its (partial) trace k at and activation point π. It relies on the past activation points that were executed by the control program, summing the bursts b' of all activations with the same operation type u on the same stream s. As the burst is the amount of data produced or consumed by a task activation, it determines the shift inside the stream required for the activation. The sum α of all such shifts along the execution of the control program determines the indexes of the stream accesses of the new task activation.

Our model is centered around the data schedule computed by the control program. To ensure that our results are general, not limited to any given class of applications, we make no assumption on the future of the control program. In the absence of any information on the oracle function Ω, which we assume deterministic, but *not determinate*, we abstract the union of all *possible futures* of the control program, and therefore of the possible future stream accesses. For technical reasons that will become clear in the remainder, we choose to represent this abstraction as a special task activation, the *continuation activation*. Its role is to capture and aggregate all of the stream accesses that can still be scheduled by any possible continuation of the control program. In a given state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, the past of the control program is defined by the partial trace k_e. For this reason, the continuation activation is constructed as the result of a function $\mathcal{C} : \mathcal{K} \to \mathcal{P}(\mathcal{X})$ that maps a partial trace $k \in \mathcal{K}$ into a set of stream accesses in $\mathcal{X}$. This approach ensures that there are no hidden assumptions on the control program, and therefore guarantees the generality of our proofs.

Definition 9 (Continuation activation)**.** In all CDDF program states $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, the continuation activation $\mathcal{C}(k_e)$ is implicitly added to $\mathcal{A}_o$. This continuation is defined by:

$$\mathcal{C}(k_e) = \left\{ (u,s,i) \in \mathcal{X} \quad | \quad i \geqslant \sum_{\substack{\pi \in k_e \\ (u,s,b,h) \in \pi}} b \right\}$$

The initial state of a CDDF program is therefore:

$$\sigma_{init} = (\epsilon, \varnothing, \{\mathcal{C}(\epsilon)\}) \qquad \text{where } \mathcal{C}(\epsilon) = \mathcal{X}$$

By construction of the ξ function, this continuation activation contains all stream accesses, and in particular the write accesses, that cannot occur before the control program makes progress, because the control program has not yet scheduled these stream accesses in any task activations. When the control program reaches a barrier, it models all write stream accesses that can only happen after the barrier passes. If for example a write stream access (W, s, i) belongs to the continuation activation and there is an outstanding activation containing (R, s, i), then unless the control program can make further progress, there is an unsatisfiable flow dependence and hence a deadlock.

Proposition 10. *The state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ of a CDDF program satisfies:*

$$\bigcup_{a \in \mathcal{A}_e \cup \mathcal{A}_o} a = \mathcal{X}$$

This state invariant holds by construction of the continuation activation, which attributes to the continuation, on every stream, each stream access not yet scheduled in a task activation generated by the execution of the control program. There is also a (safe) overlap for read accesses as those in $\mathcal{C}(k_e)$ start at the sum of bursts on each stream, which can be lower than the highest horizon. This property can also be rewritten as:

$$\forall (u,s,i) \in \mathcal{X}, \exists a \in \mathcal{A}_e \cup \mathcal{A}_o,\ (u,s,i) \in a$$

2.2 Ordering Constraints on Task Activation Execution

One of the key insights of the CDDF model is the separation of constraints between the control program, which executes a deterministic sequence of activation points, and the execution of task activations. We elaborate, in Section 3.1, on the issue of the order relation induced by Ω on activation points, and therefore by precedence in the control program trace, while in this section we discuss the ordering constraints that need to be enforced on the execution of task activations.

As we see below, the only fundamental order requirement for task activations execution is induced by the enforcement of flow dependences between activations. Due to efficiency concerns in the implementation of stream synchronization we choose to over-constrain the execution of task activations to enable a provably efficient synchronization scheme [23], synchronizing over closed prefixes of stream indexes: indexes lower than a given threshold become read-only once all producers have moved past that point. In our view, this is a mandatory aspect, barring hardware support like full-empty bits, for efficient synchronization of stream communication. This means that our synchronization scheme over-approximates the real scheduling constraint requirements, which brings us to one of the principal issues tackled in this paper: to which extent does this over-approximation induce new programming errors, which we define as spurious deadlocks below, and what conditions must be satisfied by a CDDF program to avoid such errors.

To model these constraints, we define the stream prefix order relation and the derived scheduling constraints enforced on the execution of task activations.

$$\text{(GEN)}\ \frac{\pi := \Omega(k_e) \qquad \pi \in \Pi}{(k_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (k_e.\pi, \mathcal{A}_e, \mathcal{A}'_o)}$$
$$\text{with} \quad \mathcal{A}'_o = \mathcal{A}_o \cup \{\xi(k_e, \pi), \mathcal{C}(\mathcal{K}_e.\pi)\} \setminus \{\mathcal{C}(k_e)\}$$

$$\text{(BAR)}\ \frac{\mathcal{A}_o = \{\mathcal{C}(k_e)\} \qquad \Omega(k_e) = \mathit{barrier}}{(k_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (k_e.\mathit{barrier}, \mathcal{A}_e, \mathcal{A}_o)}$$

$$\text{(TERM)}\ \frac{\mathcal{A}_o = \{\mathcal{C}(k_e)\} \qquad \Omega(k_e) = \top}{(k_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (k_e, \mathcal{A}_e, \mathcal{A}_o)}$$

$$\text{(EXEC)}\ \frac{\mathcal{A}_o = \{a\} \cup \mathcal{A}'_o \qquad \mathcal{A}_e \ltimes a}{(k_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (k_e, \mathcal{A}_e \cup \{a\}, \mathcal{A}'_o)}$$

Figure 1: CDDF execution rules.

Definition 11 (Stream prefix order)**.** We define a binary relation $<$ in $\mathcal{P}(\mathcal{A}^2)$ on task activations $(a, a') \in \mathcal{A}^2$ as:

$$a < a' \quad \triangleq \quad \exists (s,i) \in \mathcal{S} \times \mathbb{N}, \exists j \leqslant i,\ (W, s, j) \in a \ \wedge\ (R, s, i) \in a'$$

Which means that a' reads data in a stream s while a writes in stream s somewhere in the prefix of the read accesses of a'. Note that this relation is not transitive as any relations $a < a'$ and $a' < a''$ can arise from different streams.

We derive the relation $\ltimes \in \mathcal{P}(\mathcal{A}) \times \mathcal{A}$ to model our scheduling constraint for the execution of task activations; for an activation $a \in \mathcal{A}$ and a set $A \subset \mathcal{A}$:

$$A \ltimes a \quad \triangleq \quad \forall (R, s, i) \in a, \forall j \leq i, \exists a' \in A,\ (W, s, j) \in a'$$

This relation, pronounced "a is ready in A", expresses the fact that for a given program state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, if an outstanding task activation $a \in \mathcal{A}_o$ is in a relation $\mathcal{A}_e \ltimes a$, then all of its ordering constraints are satisfied by the already executed activations in $\mathcal{A}_e$ if and only if there is no task activation $a'' \in \mathcal{A}_o$ such that $a'' < a$ (recalling the role of $\mathcal{C}(k_e) \in \mathcal{A}_o$ as an aggregator of possible futures). As there can only be one single write access operation per stream index, and as Proposition 10 ensures that no write access escapes $\mathcal{A}_e \cup \mathcal{A}_o$, this relation can be defined more concisely as:

$$\begin{aligned} \forall a \in \mathcal{A}_e \cup \mathcal{A}_o,\ \mathcal{A}_e \ltimes a \ &\triangleq\ \forall a' \in \mathcal{A}_e \cup \mathcal{A}_o, a' < a \ \Rightarrow\ a' \in \mathcal{A}_e \\ &\triangleq\ \forall a' \in \mathcal{A}_o,\ a \leqslant a' \end{aligned}$$

The second version of the definition, relies on the fact that $\mathcal{A}_e$ and $\mathcal{A}_o$ are disjoint, which holds by construction of any valid program state, as we show below.

We present on Figure 1, the operational semantics of the execution of CDDF programs, and an overview of the CDDF execution model on Figure 2. We detail the semantics of the execution rules below.

(GEN) The activation generation rule states that the control program can execute an activation point as soon as it is reached by its oracle function, Ω. The result of its evaluation by ξ is added to $\mathcal{A}_o$ and the activation point is appended to the existing program trace k_e. The old continuation activation, $\mathcal{C}(k_e)$ replaced in $\mathcal{A}_o$ by a new one, $\mathcal{C}(k_e.\pi)$. By definition, $\mathcal{C}(k_e.\pi) \subset \mathcal{C}(k_e)$, which models the restriction on the possible futures of the program.

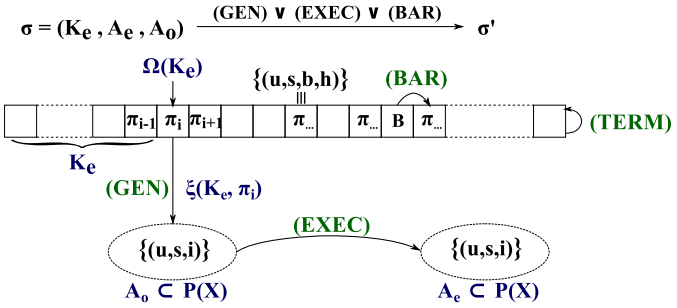


Figure 2: Overview of CDDF execution model.

(BAR) The barrier rule states that the control program only passes a barrier once all outstanding activations are executed, with the natural exception of the continuation activation. The barrier is also appended to the control program trace once cleared.

(TERM) The termination rule marks the end of the program as soon as (1) the control program finishes, which is marked by the $\top$ operation, and (2) no outstanding activations remain. It has similar semantics to the barrier rule, but does not modify the control program's trace. Termination of the CDDF program occurs immediately at the first application of the termination rule. This rule is used as a guard for program termination as it allows an infinite number of transitions, that do not modify the program state, once the program reaches termination.

(EXEC) The execution rule states that an outstanding activation a can only be executed once all the activations that need to be scheduled before it, in the stream prefix order, are executed, which as we have discussed is modeled by $\mathcal{A}_e \ltimes a$. We will see in Lemma 17 that the continuation activation $\mathcal{C}(k_e)$ cannot be executed: $\forall \sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o), \neg(\mathcal{A}_e \ltimes \mathcal{C}(k_e))$.

2.3 Program Progress and Deadlock

Based on the execution rules above, we define a simple artificial measure of program progress that adds the length of the control program trace to the number of executed activations:

$$|(k_e, \mathcal{A}_e, \mathcal{A}_o)| = |k_e| + |\mathcal{A}_e|$$

Note that all rules on Figure 1, except `(TERM)`, are strictly monotonically increasing the state of the program w.r.t. this measure.

Definition 12 (Program progress)**.** A CDDF program makes progress from state σ if any execution rule can be applied. The resulting state σ' satisfies $|\sigma| < |\sigma'|$ or the program has terminated.

This measure and the definition of program progress could be construed as flawed if we do not accept schedules where an infinite sequence of `(GEN)` occurs. Intuitively there is no point in generating an infinity of activations if they never get executed, and a bounded memory model would yield a resource deadlock. However, under the current unbounded memory assumption for the CDDF model, we admit such schedules as correct. The resulting definition of deadlock corresponds to the impossibility of progress from a given state.

Definition 13 (Program deadlock)**.** A CDDF program is in a deadlock state σ, noted $D(\sigma)$, if no execution rule can make a transition.

From this definition and that of the execution rules, we deduce the following property on the state of a deadlocked CDDF program.

Lemma 14 (Deadlock state)**.** *A CDDF program is in a deadlock state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, $D(\sigma)$ if and only if:*

$$\big(\Omega(k_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(k_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o, \neg \mathcal{A}_e \ltimes a\big)$$

2.4 Deadlock Characterization

In order to define the different types of deadlocks that can occur in our model, we first need to introduce a flow dependence relation on task activations. We will use Bernstein's definition of data dependences [4] and adjust it to task activations. As stream accesses are in single assignment and all correct reads come, by definition, after the unique write operation, only read-after-write dependences are possible.

Definition 15 (Task activation dependence relation)**.** We define the data flow dependence relation between task activations $\delta \in \mathcal{P}(\mathcal{A}^2)$ using the common definition of flow dependences, $\forall (a, a') \in A^2$:

$$a \ \delta \ a' \quad \triangleq \quad \exists (s,i) \in \mathcal{S} \times \mathbb{N}, \ (W, s, i) \in a \ \wedge \ (R, s, i) \in a'$$

As with the stream prefix order relation, we derive a relation $\Delta \in \mathcal{P}(\mathcal{A}) \times \mathcal{A}$ modeling the fundamental scheduling constraint corresponding to a set of task activations A satisfying all flow dependences for the execution of a task activation a:

$$A \, \Delta \, a \quad \triangleq \quad \forall (R, s, i) \in a, \exists a' \in A, \ (W, s, i) \in a'$$

This relation, pronounced "A satisfies the dependences of a", means that write access operations belonging to the task activations in A cover all the read operations in a, so all flow dependences of a are satisfied by the execution of the task activations in A.

From this definition, it is easy to see that the stream prefix order enforced in the CDDF model is more restrictive than necessary. We do not detail the proof of the following proposition, as it directly results from Definitions 11 and 15.

Proposition 16. *Flow dependences are subsumed by the stream prefix order relation.*

$$\begin{aligned} &\forall a, a' \in \mathcal{A} : \qquad a \, \delta \, a' \quad \Rightarrow \quad a < a' \\ &\forall a \in \mathcal{A}, \forall A \in \mathcal{P}(\mathcal{A}) : \qquad A \ltimes a \quad \Rightarrow \quad A \, \Delta \, a \end{aligned}$$

This proposition also implies that enforcing the stream prefix order is sufficient to enforce flow dependences in CDDF programs. We also provide the following lemma which guarantees that the continuation activation can never be a candidate for execution, irrespectively of the dependence relation used.

Lemma 17. *For a CDDF program in state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, the dependences of the continuation activation, $\mathcal{C}(k_e)$, are never satisfied by $\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}$.*

Proof. The proof of this lemma hinges on two properties: (1) $\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}$ is a finite set of task activations, each containing a finite number of write operations by definition of ξ and Π; and (2) $\mathcal{C}(k_e)$ contains an infinite number of read stream accesses. All flow dependences associated with read accesses of $\mathcal{C}(k_e)$ cannot be satisfied by a finite number of writes. □

The earlier blanket definition of deadlocks can be further refined, based on the source of the deadlock, in three categories: functional deadlocks, insufficiency deadlocks and spurious deadlocks. We will respectively note $FD(\sigma)$, $ID(\sigma)$ and $SD(\sigma)$ when a functional, insufficiency or spurious deadlock occurs in state σ. We informally define these deadlock types as follows.

Functional deadlocks occur when all outstanding activations have unsatisfied flow dependences and the control program cannot make progress. This type of deadlocks corresponds to an algorithmic error, in situations where no schedule preserving flow dependences can exist, for example because of an unsatisfiable flow dependence cycle.

Insufficiency deadlocks occur when the control program cannot make progress, because of a synchronization point or because of its termination, and insufficient data has been scheduled to be produced to meet the requirements of consumers in some stream. The control program cannot generate any more task activations that could produce the missing data and there is at least one outstanding task activation that cannot execute. In our model, this happens if an outstanding activation depends on $\mathcal{C}(k_e)$ and the control program cannot make progress.

Spurious deadlocks are all remaining deadlocks, which are due to the over-synchronization scheme we enforce rather than a more fundamental issue. While insufficiency and functional deadlocks are a program correctness issue, arising from algorithmic errors, which cannot be resolved and are considered to fall under the programmer's responsibility, the presence of spurious deadlocks underscores inconsistencies in our model, induced by the over-synchronization of flow dependences.

A fourth type of deadlocks, resource deadlocks, stem from the limitations of the amount of memory available for stream buffers or outstanding activations, so they are naturally absent from this unbounded memory abstraction.

2.4.1 Functional deadlocks

The definition of functional deadlocks is similar to that of CDDF deadlocks. They occur when no progress can be made in a program where only flow dependences (Definition 15) are enforced instead of our over-approximated stream prefix order (Definition 11).

Definition 18 (Functional deadlock)**.** A CDDF program is in a functional deadlock in state σ, and we note $FD(\sigma)$, if no execution rule can apply in that state, replacing rule (EXEC) with:

$$(\text{EXEC}_\Delta)\ \frac{\mathcal{A}_o = \{a\} \cup \mathcal{A}'_o \qquad \mathcal{A}_e \,\Delta\, a}{(k_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (k_e, \mathcal{A}_e \cup \{a\}, \mathcal{A}'_o)}$$

We can prove the following property of functional deadlock states, using the same reasoning as that of Lemma 14.

Lemma 19 (Functional deadlock state)**.** *A CDDF program is in a functional deadlock state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, noted $FD(\sigma)$, iff:*

$$\big(\Omega(k_e) \notin \Pi\big) \ \wedge\ \big(\mathcal{A}_o \neq \{\mathcal{C}(k_e)\}\big) \ \wedge\ \big(\forall a \in \mathcal{A}_o, \neg \mathcal{A}_e \,\Delta\, a\big)$$

2.4.2 Insufficiency deadlocks

Insufficiency deadlocks represent deadlocks where the control program stopped too early and did not generate the task activations necessary for the completion of the existing outstanding task activations, which is why their definition relies on the continuation activation. Note, however, that we primarily use the flow dependence relation rather than the stream prefix order, and that

there is no equivalence between the corresponding deadlock states. We will get an equivalence once we relax the definition of deadlock states.

Definition 20 (Insufficiency deadlocks)**.** A CDDF program is in an insufficiency deadlock in state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, and we note $ID(\sigma)$, when the control program cannot make progress and all outstanding task activations depend, transitively, on the continuation activation:

$$\big(\Omega(k_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(k_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o,\ \mathcal{C}(k_e) \ \ \delta^+ \ \ a\big)$$

We similarly define insufficiency deadlocks on the stream prefix order (Definition 11), and we note $ID_<(\sigma)$:

$$\big(\Omega(k_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(k_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o,\ \mathcal{C}(k_e) <^+ a\big)$$

Lemma 21. *All insufficiency deadlocks are functional deadlocks. A CDDF program in a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ verifies:*

$$ID(\sigma) \quad \Rightarrow \quad FD(\sigma)$$

2.4.3 Spurious deadlocks

We finally define spurious deadlocks as non-functional deadlocks.

Definition 22 (Spurious deadlock)**.** A spurious deadlock is a non-functional deadlock. For a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, we write:

$$SD(\sigma) \quad \triangleq \quad D(\sigma) \ \wedge \ \neg FD(\sigma)$$

This is the only type of deadlock that results from our over-approximation of flow dependences (Definition 11) and not from program semantics. Our primary objective is to ensure that we can avoid such deadlocks in CDDF programs.

2.4.4 Weak deadlock states

In many cases, we are able to show that the program will necessarily deadlock in the future of a given state, without knowing precisely when it will deadlock or whether the current state is already a deadlock state. A state from which all execution schedules lead to a deadlock will be called a weak deadlock state. The purpose here is to switch from universal quantifiers in the conditions of deadlock states to existential quantifiers in the conditions of weak deadlocks.

Definition 23 (Weak deadlock state)**.** The state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ of a CDDF program is a weak deadlock state, for any type of deadlock, if there is a state $\sigma' = (k'_e, \mathcal{A}'_e, \mathcal{A}'_o)$ such that:

$$D(\sigma') \quad \wedge \quad |\sigma| \leqslant |\sigma'| \quad \wedge \quad k_e = k'_e$$

We respectively note $WD(\sigma)$, $WFD(\sigma)$, $WID(\sigma)$ and $WSD(\sigma)$ when the state σ satisfies the weak deadlock condition respectively for a general deadlock, a functional deadlock, an insufficiency deadlock or a spurious deadlock.

This definition is entirely directed at simplifying the definition of deadlock states. As we require $D(\sigma') \ \wedge \ k_e = k'_e$, we ensure that the control program cannot make further progress and therefore cannot generate new task activations:

$$\Omega(k'_e) = \Omega(k_e) \in \{barrier, \top\}$$

This condition is very important as it precludes a form of live-lock where only a part of the program is in a deadlock, the remainder progressing possibly indefinitely if the program is non-terminating.

It also preserves the continuation activation, as it depends only on the control program trace. As only the (EXEC) rule can apply we have:

$$\mathcal{A}'_o \subseteq \mathcal{A}_o$$

This ensures that $|\sigma'| - |\sigma|$ is finite, so the maximum number of transitions required to reach a deadlock state is finite.

2.4.5 Weak insufficiency deadlock state

This weaker definition of deadlock states allows to reason in terms of local deadlock conditions rather than global conditions, on all outstanding task activations. For example, a program is in an insufficiency deadlock state only if all outstanding activations depend on the continuation activation, but we show that it is sufficient to know that a single outstanding activation depends on the continuation $\mathcal{C}(k_e)$, once the control program reaches a barrier, to conclude to a weak insufficiency deadlock state. For instance, these weaker conditions allow proving the equivalence of weak insufficiency deadlock states when occurring due to the enforcement of flow dependences or of the stream prefix order.

Lemma 24 (Weak insufficiency deadlock state)**.** *A CDDF program state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, verifies*

$$\begin{aligned} WID(\sigma) \Leftrightarrow \Omega(k_e) \notin \Pi \ \wedge \ \left(\exists a \in \mathcal{A}_o,\ a \neq \mathcal{C}(k_e) \wedge \mathcal{C}(k_e) \ \ \delta \ \ a\right) \\ \Leftrightarrow \Omega(k_e) \notin \Pi \ \wedge \ \left(\exists a \in \mathcal{A}_o,\ a \neq \mathcal{C}(k_e) \wedge \mathcal{C}(k_e) < a\right) \end{aligned}$$

Note that this equivalence of weak insufficiency deadlock states is possible only because we do not require a global condition on the set of outstanding activations. Indeed, the real insufficiency deadlock states are possibly different when enforcing either of the constraints, but the cause is the same: these deadlocks represent states where insufficient data can be produced on some streams, which are filled incrementally and therefore any data missing in the prefix of a stream index leads to data missing in the remainder.

We can show that a state σ satisfies:

$$ID_<(\sigma) \quad \Rightarrow \quad WID(\sigma) \quad \Leftarrow \quad ID(\sigma)$$

This is sufficient to say that any insufficiency deadlock stems from an unsatisfiable flow dependence. Even though our over-synchronization scheme may cause the program to deadlock earlier, we do not introduce additional deadlocks.

2.4.6 Weak functional deadlock state

The characterizations of weak functional deadlock states and of general CDDF weak deadlock states are very similar, and their proofs identical, so we present them at the same time. Note that this definition is very general, thanks to the continuation activation modeling the future schedule of data in streams, and encompasses all types of deadlocks.

Lemma 25 (Weak (functional) deadlock state)**.** *All deadlocks result from the presence of a cycle between task activations based on the order relation corresponding to the deadlock type.*

For a CDDF program in state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, we have:

$$\begin{aligned} WD(\sigma) &\Leftrightarrow \Omega(k_e) \notin \Pi \ \wedge \ \mathcal{A}_o \neq \{\mathcal{C}(k_e)\} \\ &\quad \wedge \exists a \in \mathcal{A}_o, a \neq \mathcal{C}(k_e), \ a <^+ a \ \vee \ \mathcal{C}(k_e) <^+ a \\ WFD(\sigma) &\Leftrightarrow \Omega(k_e) \notin \Pi \ \wedge \ \mathcal{A}_o \neq \{\mathcal{C}(k_e)\} \\ &\quad \wedge \exists a \in \mathcal{A}_o, a \neq \mathcal{C}(k_e), \ a \, \delta^+ \, a \ \vee \ \mathcal{C}(k_e) \ \delta^+ \ a \end{aligned}$$

2.4.7 Hierarchy of weak deadlock states

Based on Proposition 16 and the Lemmas 24 and 25, we can provide a hierarchy of weak deadlock states. It is important to stress the fact that any given state can satisfy the weak deadlock condition for multiple types of deadlocks, possibly in a completely independent manner. If such is the case, the stronger deadlock property is naturally taken into account.

Proposition 26 (Weak deadlock state hierarchy)**.** *Any state σ of a CDDF program satisfies:*

$$WID(\sigma) \quad \Rightarrow \quad WFD(\sigma) \quad \Rightarrow \quad WD(\sigma)$$

And the definition of spurious deadlocks is preserved:

$$WSD(\sigma) \quad \Leftrightarrow \quad WD(\sigma) \ \wedge \ \neg WFD(\sigma)$$

In the remainder of this paper, we analyze the properties of the CDDF model, we provide conditions under which our model does not introduce spurious deadlocks and we prove the serializability and the determinism of CDDF programs.

3 Stream Causality

This section starts by introducing the notion of stream causality and a characterization of stream causal programs based on the existence of stream causal schedules. In a second step, we prove that CDDF programs that are stream causal, in each state where the control program waits for a barrier, are free of all forms of deadlocks.

3.1 Characterization of Stream Causality

As our model is asynchronous, we cannot use a global notion of time as a basis for the definition of causality. Instead, we can use streams as a set of independent local clocks. Each stream can be considered to define its own time based on access indexes, each task activation representing a synchronization point between the clocks of the streams it writes to and, to a lesser extent, those it reads from. As write accesses to streams are exclusive in our single assignment model, we can define the *Stream Clock* (SC) of an activation as its ordered set of write accesses. More importantly, this allows us to define a precedence relation between task activations producing data in the same stream. This relation is a subset of the total order on activations induced by the control program order.

Definition 27 (Stream clock precedence relation)**.** We define a reflexive and antisymmetric binary relation $\preccurlyeq_{\text{SC}} \in \mathcal{P}(\mathcal{A}^2)$ as:

$$\begin{aligned} &\forall (a, a') \in A^2, \quad a \preccurlyeq_{\text{SC}} a' \quad \triangleq \\ &\exists s \in \mathcal{S}, \exists i, j \in \mathbb{N}, \ j \leqslant i \ \wedge \ (W, s, j) \in a \ \wedge \ (W, s, i) \in a' \end{aligned}$$

In any state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, we extend this relation to order *sink* task activations[2] with $\mathcal{C}(k_e)$. This ensures that the continuation activation is an upper bound:

$$\forall a \in \mathcal{A}_e \cup \mathcal{A}_o : \qquad a \preccurlyeq_{\text{SC}} \mathcal{C}(k_e)$$

This relation is antisymmetric by definition of the activation point evaluation function ξ (Definition 8), as it enforces a consistent schedule of write access indexes in all streams. The underlying order relation is the precedence of activation points in the control program trace: if task activations a, a' are the result of the evaluation of activation points π, π', then the order of occurrence of these activation points in the control program trace determines the direction of the precedence relation $\preccurlyeq_{\text{SC}}$:

$$a \preccurlyeq_{\text{SC}} a' \quad \Rightarrow \quad \pi = \pi' \ \vee \ \pi \to \pi'$$

Flow dependence chains between task activations can naturally be assimilated to causality chains, and we use them to this effect in our definition of causality, but they are not sufficient in our model. Indeed they do not allow to relate a given read access (R, s, i) to write accesses $(W, s, j)_{j<i}$ occuring in the prefix of (W, s, i) on stream s. We define stream causality in CDDF programs as the absence of time reversal inside causality chains.

Definition 28 (Stream causality)**.** A CDDF stream $s \in \mathcal{S}$ is causal in a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ of a program if the stream's local clock progresses forward along all causality chains:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o, \neg(a = a' = \mathcal{C}(k_e)), s \in \mathcal{O}(a) \cap \mathcal{O}(a'), \\ \neg\Big(a \preccurlyeq_{\text{SC}} a' \ \wedge \ a' \, \delta^+ \, a\Big)$$

The restriction $\neg(a = a' = \mathcal{C}(k_e))$ means that we do not check the causality of the barrier itself; this precaution is necessary as $\mathcal{C}(k_e) \preccurlyeq_{\text{SC}} \mathcal{C}(k_e)$ and $\mathcal{C}(k_e) \, \delta \, \mathcal{C}(k_e)$ are always true. This validates our intuition that the continuation activation is inherently non-causal. However, as long as the continuation activation is only found at the end of causality chains, it does not preclude stream causality.

A CDDF program is stream causal in a state σ if each stream in the program is causal. We note:

$$SC(\sigma) \quad \triangleq \quad \forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o, \ \neg(a = a' = \mathcal{C}(k_e)), \\ \neg\Big(a \preccurlyeq_{\text{SC}} a' \ \wedge \ a' \, \delta^+ \, a\Big)$$

Note that stream causality is entirely determined by the schedule of data in streams and not by the execution schedule of task activations. For this reason, the causal nature of a CDDF program state only depends on the control program's trace in that state. This can be seen in Definition 28, where no difference is made between executed and outstanding activations. As a result, stream causality is well defined for a given program, since the control program itself is deterministic.

A second important remark is that our choice of modeling the behaviour of barriers through the continuation activation is entirely consistent with this definition of stream causality. The stream clock of this activation is greater than that of any task activation generated by the control program, which properly models the possible continuation of the program after the barrier passes. The restriction, in Definition 28, that we do not check the causality of the barrier itself, means

[2] Activations that contain no write accesses and would therefore not be in any relation w.r.t. stream clocks.

that the current state of the program does not allow us to deduce whether its future will or will not be causal.

Stream causality violations represent conflicts between flow dependences and stream clocks. To model causal program schedules, we define the minimal constraints for scheduling functions that respect both forward time in each stream and flow dependences.

Definition 29 (Stream causal schedule)**.** A scheduling function $\theta : A \to \mathbb{N}$ enforces a stream causal schedule iff:

$$\forall (a, a') \in A : \quad \begin{cases} a \; \delta^+ \; a' & \Rightarrow \quad \theta(a) < \theta(a') \\ a \preccurlyeq_{\text{SC}} a' & \Rightarrow \quad \theta(a) \leqslant \theta(a') \end{cases}$$

We use the existence of causal schedules, irrespectively of their capacity to enforce the stream prefix order, to characterize stream causality in CDDF programs. According to Definition 28, the schedule of the execution of task activations is only constrained by flow dependences, and the forward progress of time on stream clocks. In general, we only need to know whether a causal schedule exists, but we do not enforce it on the execution of task activations.

Proposition 30 (Causal CDDF program)**.** *A CDDF program is causal in a state* $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ *if and only if the program admits at least one causal schedule in that state.*

3.2 Deadlock-Freedom in Stream Causal CDDF Programs

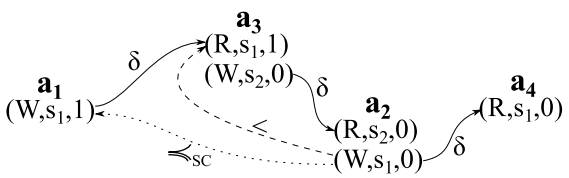
Without any restriction, spurious deadlocks may occur in CDDF programs due to the over-approximation, and therefore over-synchronization, of data dependences. As the model allows absolute freedom in the communication patterns between task activations, we can build a schedule that leads to a state where a CDDF program is in a deadlock, but not in a functional deadlock. This situation occurs, for instance, in the following example.

Example 31 (Spurious deadlock in a CDDF program)**.** *Consider a program state* $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ *such that:*

$$\mathcal{A}_e = \{a_1\} \qquad \textit{where} \qquad a_1 = \{(W, s_1, 1)\}$$

$$\mathcal{A}_o = \{a_2, a_3, a_4\} \qquad \textit{where} \qquad \begin{cases} a_2 = \{(W, s_1, 0), (R, s_2, 0\} \\ a_3 = \{(R, s_1, 1), (W, s_2, 0)\} \\ a_4 = \{(R, s_1, 0)\} \end{cases}$$

The figure below shows the true dependences δ *as well as the additional constraints* $<$ *on stream prefixes. It also marks, for the discussion, the stream clock relation* $\preccurlyeq_{\text{SC}}$ *between activations* a_2 *and* a_1*.*



The following flow dependences are present in this state:

$$a_1 \; \delta \; a_3 \qquad a_3 \; \delta \; a_2 \qquad a_2 \; \delta \; a_4$$

As flow dependences allow the simple schedule (a_3, a_2, a_4) *to complete the execution, considering that* a_1 *has already been executed, there is no functional deadlock. However, the addition*

of the stream prefix constraint leads to a cycle between activations a_2 and a_3, and therefore to a deadlock when the control program reaches a barrier:

$$a_1 < a_3 \qquad a_3 < a_2 \qquad a_2 < a_4 \qquad \textit{and} \qquad a_2 < a_3$$

The intuition we get from this example is that a stream causality violation, between activations a_1 and a_2, was the necessary element allowing this spurious deadlock to occur:

$$\begin{aligned} a_2 \preccurlyeq_{\text{SC}} a_1 \ \wedge\ a_1 \,\delta\, a_3 \,\delta\, a_2 \quad &\Rightarrow \quad \theta(a_1) < \theta(a_2) \leqslant \theta(a_1) \\ &\Rightarrow \quad \neg SC(\sigma) \end{aligned}$$

Though stream causality is a strong condition that is not decidable at compilation time, it is a semantically important condition for deadlock-freedom, and for some strong forms of serializability, as we will show in Section 6.2.

This shows stream causality is a relaxed form of Cilk strictness [11], which is defined as the absence of transverse dependences between different branches of the task creation tree, requiring that all dependences be synchronized by a task's parent. Semantically, strictness means that a task should always be executable (i.e., all its dependences should be satisfied) at the spawn point. In the CDDF model, this can be understood in a more relaxed way as requiring that all write stream accesses necessary for a task's execution be already *scheduled* by the control program when it reaches the task's activation point; in other words that all dependences should be forward in the control program, which ensures stream causality in each state of the CDDF program.

Theorem 32. *A CDDF program is free of all deadlocks if it is stream causal in each state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ where the control program reaches a barrier or terminates. All states σ satisfy:*

$$SC(\sigma) \qquad \Rightarrow \qquad \neg\, WD(\sigma)$$

4 Task Causality

While stream causality is sufficient to prove deadlock-freedom, we can prove less restrictive conditions, ensuring that only insufficiency deadlocks can occur. In this section, we present two alternative conditions for deadlock-freedom and the formal framework for their analysis.

4.1 CDDF Tasks

We have avoided, until now, the additional complexity of introducing a definition of tasks in the CDDF model, as they play a secondary role. We use a weak definition of tasks as equivalence classes of task activations, disregarding the work functions.

Definition 33 (CDDF task)**.** A task is an equivalence class in the set of task activations based on the following equivalence relation $\sim \in \mathcal{P}(\mathcal{A}^2)$ defined for $(a, a') \in \mathcal{A}^2$ as:

$$\begin{aligned} a \sim a' \quad &\triangleq \quad \begin{cases} \forall (u,s,i) \in a, \exists i' \in \mathbb{N},\ (u,s,i') \in a' \\ \forall (u,s,j') \in a', \exists j \in \mathbb{N},\ (u,s,j) \in a \end{cases} \\ &\triangleq \quad \mathcal{I}(a) = \mathcal{I}(a') \ \wedge\ \mathcal{O}(a) = \mathcal{O}(a') \end{aligned}$$

Two activations are equivalent in this relation iff they access exactly the same streams, both for input and for output.

The set of tasks in a CDDF program in a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ is the quotient space of the set of task activations:

$$T(\sigma) = (\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\})_{/\sim}$$

For a task activation a, we write $[a]_\sim$ its equivalence class on the $\sim$ relation; it is the CDDF task to which activation a belongs.

Using this definition, we can build the task graph of a program state. We note $S(\sigma)$ the set of streams used by a program in state σ, defined as the set of streams accessed by any activation in σ:

$$S(k_e, \mathcal{A}_e, \mathcal{A}_o) \triangleq \bigcup_{a \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}} \Big(\mathcal{I}(a) \cup \mathcal{O}(a)\Big)$$

Definition 34 (Task graph)**.** A CDDF program in a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ can be represented as a directed hypergraph:

$$H(\sigma) = \big(T(\sigma), S(\sigma)\big)$$

where tasks are vertices and streams are directed hyperedges connecting a set of producer tasks to a set of consumer tasks.

For a program in state σ and a stream $s \in S(\sigma)$, we define the sets of producer tasks $P(\sigma, s)$ and of consumer tasks $C(\sigma, s)$ as:

$$\begin{aligned} P(\sigma, s) &= \Big\{[a]_\sim \in T(\sigma) \mid s \in \mathcal{O}(a)\Big\} \\ C(\sigma, s) &= \Big\{[a]_\sim \in T(\sigma) \mid s \in \mathcal{I}(a)\Big\} \end{aligned}$$

Despite the fact that, by definition, $\mathcal{C}(k_e)$ is a producer and consumer of each stream, we do not count the continuation activation in any producer or consumer set. This task graph clearly depends on the current state of the control program, and is therefore dynamic.

4.2 Task Causality

We derive, from the control program order, a precedence relation on task activations, called *task order*. The stream access indexes of activations define a total order on the activations *within the task*. This order is induced by the definition of the activation point evaluation function ξ and the total order of activation points in the trace of the control program.

Definition 35 (Task order)**.** Task activations belonging to the same task in a program state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ are ordered by the relation $\preccurlyeq_{\text{T}} \in \mathcal{P}(\mathcal{A}^2)$, for $(a, a') \in \mathcal{A}_e \cup \mathcal{A}_o$ we define:

$$a \preccurlyeq_{\text{T}} a' \triangleq \big(a \sim a'\big) \wedge \Big(\forall s \in \mathcal{I}(a) \cup \mathcal{O}(a), \exists i, j \in \mathbb{N},\ j \leqslant i, (u, s, j) \in a, (u, s, i) \in a'\Big)$$

This relation is transitive as we restrict it to $a \sim a'$, reflexive because we use $j \leqslant i$ and antisymmetric by construction of task activations in the evaluation function ξ. However, its extension to all task activations is not an order relation as it lacks transitivity.

This order relation is very similar to the stream clock relation, $\preccurlyeq_{\text{SC}}$, that we introduced in the previous section, which was used to define stream causality. We will use here the task order relation to define task causality in the same way.

Definition 36 (Task causality)**.** A CDDF task is causal in a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ of a program if the task's activations appear in task order along all causality chains:

$$\forall t \in T(\sigma),\ \forall a, a' \in t, \quad \neg\big(a \preccurlyeq_{\mathrm{T}} a' \quad \wedge \quad (a'\ \delta^+\ a)\big)$$

A CDDF program is task causal in a state σ, and we note $TC(\sigma)$, iff each task in the program is causal in that state.

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\},\ \neg\big(a \preccurlyeq_{\mathrm{T}} a' \ \wedge\ (a'\ \delta^+\ a)\big)$$

Note that this definition revisits process monotonicity in Kahn networks [18], though the equivalent notion of process is slightly different from our quotient tasks. Indeed, our definition requires that no flow dependence chain goes back in task order, which would mean that the execution of task activation a' enables the latter execution of a. Because $a \preccurlyeq_{\mathrm{T}} a'$, the outputs of a are written in the prefix of the outputs of a', so Kahn's monotonicity is preserved if task $[a]_\sim$ is the single consumer and the single producer on its streams. Our definition of task causality generalizes Kahn's monotonicity to cases where other tasks share these streams. At the limit, as stream prefixes grow into complete streams, our definition reproduces the Kahn principle: continuity of the communicating processes.[3]

Furthermore, this property is weaker than stream causality, because the stream clock relation is defined across task activations producing data in a given stream, even if these activations are not in the same task. For $a, a' \in \mathcal{A}$, we have:

$$\begin{aligned} a \preccurlyeq_{\mathrm{SC}} a' \ &\Rightarrow\ \exists s \in \mathcal{S}, \exists j \leqslant i, (W, s, j) \in a \ \wedge\ (W, s, i) \in a' \\ &\Rightarrow\ \big(a \sim a' \ \Rightarrow\ a \preccurlyeq_{\mathrm{T}} a'\big) \end{aligned}$$

The only exception are sink tasks, that are only consumers, and in which activations are only in a stream clock precedence relation with the continuation activation. We extended the stream clocks relation in that case, but task causality does not allow the same simplification. Indeed, as we have discussed above, the continuation activation does not belong in any task, because tasks are defined as equivalence classes of task activations and the continuation activation is the only activation that has an infinite number of input and output streams. For this reason, the continuation activation can *never* be in a task order relation $\preccurlyeq_{\mathrm{T}}$ with task activations.

Definition 37 (Task causal schedule)**.** A scheduling function $\theta : \mathcal{A} \to \mathbb{N}$ enforces a task causal schedule iff:

$$\forall a, a' \in \mathcal{A} : \quad \begin{cases} a\ \delta^+\ a' & \Rightarrow \quad \theta(a) < \theta(a') \\ a \preccurlyeq_{\mathrm{T}} a' & \Rightarrow \quad \theta(a) \leqslant \theta(a') \end{cases}$$

Proposition 38 (Task causal CDDF program)**.** *A CDDF program is task causal in a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ that is not a weak insufficiency deadlock if and only if the program admits at least one task causal schedule in that state.*

Even though task causality is not a sufficient condition, it constitutes a fair starting point for searching new deadlock-freedom conditions. Furthermore, this condition is weaker than the restrictions of multiple programming models, including Cilk, OpenMP tasks and models based on Cyclo-Static Data-Flow, like StreamIt.

As mentioned above, it is also a slightly weaker version of Kahn's monotonicity, and we show below that unlike Kahn networks where communication channels are always single-producer and single-consumer, we can prove that only insufficiency deadlocks can occur in task causal programs where streams can be either multi-producer or multi-consumer, but not both.

[3] Lifting the Scott topology to streams, continuity is the monotonicity of stream prefix definitions at the limit.

Theorem 39. *A CDDF program can only experience insufficiency deadlocks if it is task causal and each stream in the program is either single-producer or single-consumer. For* $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$*:*

$$TC(\sigma) \wedge \big(\forall s \in S(\sigma),\ |P(\sigma, s)| = 1 \ \vee\ |C(\sigma, s)| = 1\big) \\ \Rightarrow \quad WID(\sigma) \ \vee\ \neg WD(\sigma)$$

4.3 Strongly Connected Components

In the absence of cycles in the program hypergraph, we can show that only insufficiency deadlocks can occur and Proposition 26 guarantees such deadlocks are never spurious deadlocks. While this condition may appear restrictive, it is satisfied by a wide class of streaming applications. Kudlur and Mahlke analyze a dozen such applications in [19] and find that the pattern never occurs and limit their study to acyclic task graphs.

Theorem 40. *A CDDF program can only have insufficiency deadlocks in a state* $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ *if its hypergraph* $H(\sigma)$ *contains no strongly connected components (SCC) in that state:*

$$SCC(H(\sigma)) = \varnothing \quad \Rightarrow \quad WID(\sigma) \ \vee\ \neg WD(\sigma)$$

5 Functional and Deadlock Determinism

One of the most important properties of CDDF, and it is one of the driving design goals, is to guarantee functional determinism. In this section we prove that our model guarantees not only functional determinism, but also that deadlocks occur deterministically, which is highly valuable for productivity in parallel programming environments. We focus in this section on data produced and consumed in streams, which means that functional determinism is defined as producing the same observable output *in streams* for a given input.

To prove determinism, we make the two following assumptions.

1. We assume that either no shared memory communication happens, or that it does not constitute a source of non-determinism (i.e., no race conditions). This assumption is natural as shared memory communication is only allowed as a convenience under the programmer's responsibility.
2. We assume that a task that declares writing in a stream does indeed define all of the elements it declares, leaving no undefined values in streams. If a task declares producing a given amount of data for a given output stream but writes less than the amount specified, then undefined memory locations may be read by consumers, breaking functional determinism guarantees.

5.1 Deterministic Data Schedule in Streams

Until now, we have disregarded the work functions of tasks, modeling task activations as a set of stream accesses where the actual data stored inside the stream is unimportant. Work functions are relevant in the current context because determinism concerns both data placement and the data itself. As a common assumption for sequential code, we postulate that all work functions and the control program itself are inherently deterministic. For a given set of stream accesses characterizing a task activation, if the data present in the input streams at the locations of all read accesses is the same, the data written on all output streams at the locations of write accesses

will be the same. If the data placement in all streams is deterministic, then the work functions' determinism will be sufficient to guarantee functional determinism for the program.

In our model, the schedule of data in streams is entirely determined by the control program. This choice is made in Section 2 specifically to guarantee determinism. As the control program is sequential and deterministic, the schedule of data will also be deterministic.

Lemma 41 (Deterministic data schedule)**.** *For a given input, the schedule of data is deterministic in all streams of a CDDF program.*

5.2 Program Functional Determinism

CDDF programs are deterministic by design, relying on a sequential deterministic control program to orchestrate the schedule of data rather than that of the execution.

Theorem 42. *CDDF programs are functionally deterministic.*

More specifically, for a given input, the data produced by the program in all streams is the same in a given program state σ, regardless of the execution schedule of task activations. Values in streams prefixes are functions of the program state and, in particular, the final state is a function of the initial state.

The proof of this theorem is based on Lemma 41, on the hypothesis that work functions are deterministic, and on the two assumptions made at the beginning of this section. Streams are single assignment structures. The only imaginable races would be associated with a read operation occurring before the single write operation to a given location. The ordering requirements to avoid races are given by the flow dependence relation δ. As the stream prefix order subsumes flow dependence requirements (see Proposition 16), the prefix order is stricter than necessary to avoid this type of races. It follows that there are no race conditions in a CDDF programs' stream communication, so this does not constitute a source of non-determinism.

5.3 Deadlock Determinism

We can go one step further than proving functional determinism of programs, and show that deadlocks occur deterministically.

Theorem 43 (Deterministic deadlocks)**.** *CDDF programs deadlock deterministically: for a given input set, the program either does not deadlock or it deadlocks in the same state in all executions.*

To prove this theorem, it is sufficient to show that if a CDDF program admits, for a given input, an execution schedule in which a deadlock occurs, then that deadlock state is the maximum state that can and will be reached in all executions.

We emphasize that deadlock determinism is only proven under the simplifying assumption of unbounded memory, which ensures that the maximum state can actually be materialized during execution. If the maximum state does not fit in the available memory, then the program may experience a resource deadlock before reaching this state; resource deadlocks are not deterministic. This is a difference with Kahn networks where bounded memory can be simulated with back-pressure channels. Detecting or disproving resource deadlocks in presence of dynamic task creation and barriers requires additional hypotheses and more complex criteria [23], escaping the scope of this paper.

5.4 Determinism, Productivity and Portability

An important result of Theorems 42 and 43 is that, when debugging a CDDF program, deadlock reproducibility is absolute. In most parallel programming environments, code instrumentation and debuggers lead to reduced deadlock or race reproducibility. In our model, this cannot happen.

Even more, our model guarantees that if a program experiences a deadlock-free run for a given input set, there can be no deadlocks in that program for the same input set. This is true irrespectively of the underlying architecture, so application testing becomes equivalent to that of sequential applications: changing the level of concurrency (or communication latencies, bandwidth, etc.) of the execution platform does not result in observing new spurious race conditions or deadlocks. A test suite that provides proper code coverage is therefore as efficient for uncovering errors in CDDF programs as it is for finding errors in sequential programs.

Lemma 41 and Theorem 42 further guarantee that stopping the control program (e.g., a breakpoint) will lead to a deterministic whole program state. This property in itself is very useful to facilitate the debugging process. The programmer can rely on this guarantee to deterministically observe the program state by stopping the control program, and allowing the outstanding task activations to execute until quiescence is reached.[4]

These properties strongly improve the productivity and portability of CDDF based programming models:

- Program testing is as effective as for sequential programs, independently of the architecture used for testing.
- Errors can be deterministically reproduced, when relying on stream communication.
- Deadlocks occur deterministically, on all execution platforms.
- Programs can be interrupted deterministically for debugging.

6 Consistency and Serializability

Current architectures involve increasingly relaxed memory consistency models. This choice is motivated by scalability and efficiency concerns, but it makes parallel programming increasingly more complicated. Programmers cannot ignore the consistency issues of shared memory communication. They must understand the memory models of target architectures and rely on memory fences, with varying patterns depending on the architecture [1], which makes parallel programming ever more complicated and races more elusive.

6.1 Strict Consistency of Stream Communication

Stream computing, and functional programming in general, hide the weak memory ordering issues from developers altogether. Because of the (perceived) *single assignment* behaviour of stream accesses, and the synchronization mechanisms enforcing flow dependences, all read operations in streams are guaranteed to return the last, and unique, value written at a given index in a stream. This is the strongest possible memory consistency guarantee: strict consistency [15].

In the case of CDDF programs, strict consistency is only guaranteed for stream accesses. As our model allows shared memory communication, it is still possible to be affected by relaxed memory consistency issues. Our position is that disallowing shared memory communication is too restrictive and would go against the design philosophy of many programming paradigms. For example, Cilk guarantees strict consistency for Cilk procedure call parameters and return

[4] Quiescence occurs when no further progress can be made. The time necessary to reach that point is undecidable, but finite as we have seen in the discussion of weak deadlock states.

value, and it allows shared memory communication with a relaxed, DAG-consistent memory model. Similarly the OpenMP specification requires users to insert the necessary memory fences, depending on the underlying architecture. We advocate only using shared memory if no stream communication patterns can be substituted.

6.2 Serializability of CDDF Programs

Serializability is a very important property for parallel programs as it allows efficient execution on non-concurrent platforms and sequential functional testing. In our case, as discussed in Section 5, the latter is less important because of the strong determinism guarantees our model offers. Nevertheless, sequential execution of CDDF programs is relevant and we need to ensure it is possible.

Intuitively, this should mostly be a question of whether there is a possible interference between the implicit synchronization of stream accesses of different task activations that would require an interleaving of operations that cannot be achieved in a sequential schedule of the execution of task activations.

Proposition 44 (Serializability)**.** *All deadlock-free CDDF programs admit at least one sequential schedule.*

If we only consider correct programs, so excluding all programs that have functional deadlocks, we can therefore guarantee serializability under the same conditions as spurious deadlock-freedom. However, this type of serialization is more akin to the dynamic scheduling of some ad hoc user-level threads, or fibers. This scheme incurs scheduling runtime overhead and disables many compiler optimizations that would apply to a static schedule.

Static serialization, where the sequential schedule is statically defined, is a much more powerful result, but proving the existence of a static serial schedule, let alone providing an automatic way of building it, is much harder in the general case. There is, however, a trivial static schedule for which we can prove a necessary and sufficient condition: the control program order. This schedule is semantically important as it directly correlates to Cilk strictness; it is the default serialization strategy of both Cilk and OpenMP.

Theorem 45 (Serializability – static schedule)**.** *A CDDF program can be sequentially executed in control program order if and only if it is stream causal in each state.*

Proof. Given the hypothesis that the program is stream causal in each state, we can add a barrier after each activation point in the control program and this barrier is guaranteed, by Theorem 32, to be able to pass, which means that each newly generated task activation can execute immediately.

Conversely, if the program can execute sequentially, with the execution of each task activation tied to the generation of the task activation, then the program can admit one barrier after each activation point without deadlocking. As the stream clock order is a subset of the order relation defined by the control program order, the execution of task activations was possible in strictly positive stream clock order for all streams in the program, while also respecting flow dependences, which is, by definition, a stream causal schedule. Proposition 30 allows to conclude that the program is stream causal in each state. □

7 Summary of CDDF Properties and Conditions

To give a better idea of the guarantees provided by the CDDF model, we present a summary of these properties and the necessary conditions on Table 1, where $D(\sigma)$, $ID(\sigma)$, $FD(\sigma)$ and $SD(\sigma)$

Condition on state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$	Deadlock Freedom properties				Serializability		Determinism	Strict
	$\neg D(\sigma)$	$\neg ID(\sigma)$	$\neg FD(\sigma)$	$\neg SD(\sigma)$	Dyn. ord.	CP ord.	Func^{al} & Dlock	Consist.
$TC(\sigma) \quad \wedge \quad \forall s, \neg MPMC(s)$ Weaker Kahn's monotonicity	no	no	yes	yes	if $\neg ID(\sigma)$	no	yes	yes
$SCC(H(\sigma)) = \varnothing$ Common case	no	no	yes	yes	if $\neg ID(\sigma)$	no	yes	yes
$SC(\sigma) \quad \vee \quad \Omega(k_e) \in \Pi$ Less rest. than strictness	yes	yes	yes	yes	yes	no	yes	yes
$\forall \sigma, SC(\sigma)$ Relaxed strictness	yes	yes	yes	yes	yes	yes	yes	yes

Table 1: Properties of CDDF programs.

respectively mean that the program can experience a deadlock,[5] an insufficiency deadlock, a functional deadlock or a spurious deadlock, all relative to a program state σ. The conditions presented range from the weakest to the strongest. Each condition is sufficient to ensure spurious deadlock freedom, which is one of our main objectives, as well as functional and deadlock determinism and strict consistency. We describe the conditions, which are presented in an abbreviated form in the table, and we discuss the results below.

1. The first condition corresponds to Theorem 39. It requires that no stream in the program be multi-producer and multi-consumer (MPMC) in that state and that the program be task causal, $TC(\sigma)$. As Kahn channels are single-producer single-consumer, the Kahn principle is more restrictive than this condition. Programs meeting these requirements can only experience insufficiency deadlocks and benefit from serializability when no insufficiency deadlocks occur.
2. The second condition, from Theorem 40, requires the hypergraph $H(\sigma)$ to be free of strongly connected components. This condition corresponds to the common case identified by Kudlur and Mahlke in [19]. It is sufficient to avoid all but insufficiency deadlocks. It also allows dynamic serializability in the absence of insufficiency deadlocks.
3. The third condition requires stream causality of the program state whenever the control program reaches a barrier or terminates. This strong condition is sufficient, in Theorem 32, to prove the absence of any type of deadlocks and to ensure serializability in all cases. As only barrier states need stream causality, this condition is less restrictive than Cilk strictness.
4. Finally, the last and strongest condition, from Theorem 45, requires stream causality $SC(\sigma)$ in each state σ of the program, providing all of the possible properties, including static control program (CP) order serializability. This condition is a relaxed form of strictness where the constraints only bear on the control program, not on task activation execution.

As a final note, the two causality conditions, stream and task causality, are alternatives to the Kahn principle when streams can have multiple producers and/or multiple consumers. When restricting CDDF to single-producer single-consumer streams, it is easy to verify that both conditions are equivalent, and also equivalent to the Kahn principle at the limit.

8 Conclusion

We presented CDDF, a new, general model of computation for stream-computing programs. It is built around the notion of control program, a process that constructs a dynamic, yet deterministic, schedule of data in streams. We proved a set of important properties provided

[5] This general deadlock case is the union of all sub-cases.

by programs implementing this model, like deadlock-freedom, determinism and serializability. We proved that programming models relying on CDDF can be implemented efficiently, allowing to synchronize streams over closed prefixes without spurious deadlocks, and exposing excellent productivity features: functional and deadlock determinism, independently of the execution platform, serializability, and strict consistency on streams.

Let us evaluate the impact of CDDF on language design. Language constructs matching the native task creation, stream communication and barriers of CDDF inherit its determinism and deadlock-freedom. To ensure the soundness of a new language construct, the designer is only required to prove the absence of spurious deadlocks in the formalization of this construct with the prefix ordering on streams; the other properties are important for productivity and for enabling optimizations, but not mandatory for soundness.[6] For any new constructs, such productivity properties as serializability, determinism and strict consistency can be proven using the CDDF formalization and existing proofs as blueprints.

Finally, establishing the strong properties of CDDF involved a few simplifying assumptions, notably unbounded memory, a sequential control program, and the absence of feedback from task activations to the control program. The relaxation of these restrictions will be the topic of a distinct publication.

References

[1] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 258–272. Springer, 2010.

[2] Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, 1989.

[3] E. A. Ashcroft and W. W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[4] A. Bernstein. Program analysis for parallel processing. *IEEE Transactions on Computers*, 15:757–762, October 1966.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Data Flow. *ICASSP*, 5:3255–3258, 1995.

[6] P. Caspi, G. Hamon, and M. Pouzet. *Real-Time Systems: Models and verification – Theory and tools*, chapter Synchronous Functional Programming with Lucid Synchrone. ISTE, 2007.

[7] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ICFP*, pages 226–238, New York, NY, USA, 1996. ACM.

[8] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[9] A. L. Davis. The architecture and system method of DDM1: A recursively structured data driven machine. In *ISCA*, pages 210–215, New York, NY, USA, 1978. ACM.

[10] J. B. Dennis and D. Misunas. A preliminary architecture for a basic data flow processor. In W. K. King and O. N. Garcia, editors, *ISCA*, pages 126–132. ACM, 1974.

[6] If we consider programs with functional deadlocks to be incorrect.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM Symp. on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Quebec, June 1998.

[12] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller. The Sisal Model of Functional Programming and its Implementation. In *PAS*, pages 112–, Washington, DC, USA, 1997. IEEE Computer Society.

[13] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, San Jose, CA, Oct 2006.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[15] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL*, pages 13–26, New York, NY, USA, 1987. ACM.

[16] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27:1–164, May 1992.

[17] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.

[18] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug. 1974. North Holland, Amsterdam.

[19] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI*, pages 114–124, New York, NY, USA, 2008. ACM.

[20] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–25, 1987.

[21] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In *PPOPP*, 2010.

[22] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *Intl. J. on High Performance Computing Architecture*, 23(3):284–299, 2009.

[23] A. Pop. *Leveraging Streaming for Deterministic Parallelization: an Integrated Language, Compiler and Runtime Approach*. PhD thesis, MINES ParisTech, Sept. 2011.

[24] A. Pop and A. Cohen. A stream-computing extension to OpenMP. In *HiPEAC*, pages 5–14, New York, NY, USA, 2011. ACM.

[25] A. Pop and A. Cohen. Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs. Rapport de recherche RR-8001, INRIA, June 2012.

[26] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. S. III. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS*, pages 277–288, 2008.

[27] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[28] The StreamIt language. `http://www.cag.lcs.mit.edu/streamit/`.

[29] I. Watson and J. R. Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, 1982.

A Detailed Proofs of CDDF Properties

Proof of Lemma 14. Definition 13 states that a deadlock occurs when no rule can be fired in a given state of the program:

$$D(\sigma) \quad \Leftrightarrow \quad \neg(\text{GEN}) \ \wedge \ \neg(\text{BAR}) \ \wedge \ \neg(\text{TERM}) \ \wedge \ \neg(\text{EXEC})$$

From the definition of the execution rules on Figure 1 and using $\big(\Omega(k_e) \neq \mathit{barrier} \ \wedge \ \Omega(k_e) \neq \top\big) \ \Leftrightarrow \ \Omega(k_e) \in \Pi$ to merge the conditions for (BAR) and (TERM), we can derive the following equivalence, which simplifies in the desired property:

$$D(\sigma) \quad \Leftrightarrow \quad \begin{cases} \Big(\big(\mathcal{A}_o \neq \{\mathcal{C}(k_e)\}\big) \ \vee \ \big(\Omega(k_e) \in \Pi\big)\Big) \\ \wedge \ \Big(\Omega(k_e) \notin \Pi\Big) \wedge \ \Big(\forall a \in \mathcal{A}_o : \ \neg\mathcal{A}_e \ltimes a\Big) \end{cases}$$

□

Proof of Lemma 21. The proof relies on Lemma 17, which ensures that the continuation activation is never executable with (EXEC_Δ):

$$\neg\big((\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}) \,\Delta\, \mathcal{C}(k_e)\big)$$

Definition 20 can be rewritten and expanded from the definition of flow dependences, $ID(\sigma) \ \Rightarrow \ \big(\forall a \in \mathcal{A}_o, \ \mathcal{C}(k_e) \,\delta^+\, a\big)$:

$$\begin{aligned}
&\Rightarrow \ \big(\forall a \in \mathcal{A}_o, \exists a' \in \mathcal{A}_o, \ \mathcal{C}(k_e) \,\delta^*\, a' \ \wedge \ a' \,\delta\, a\big) \\
&\Rightarrow \ \big(\forall a \in \mathcal{A}_o, \exists a' \in \mathcal{A}_o, \exists (s,i), \ (W,s,i) \in a' \ \wedge \ (R,s,i) \in a\big) \\
&\Rightarrow \ \big(\forall a \in \mathcal{A}_o, \exists (R,s,i) \in a, \forall a' \in \mathcal{A}_e, \ (W,s,i) \notin a'\big) \\
&\Rightarrow \ \big(\forall a \in \mathcal{A}_o, \ \neg\big(\mathcal{A}_e \,\Delta\, a\big)\big)
\end{aligned}$$

□

Proof of Lemma 24. By Definition 23 there exists $\sigma' = (k'_e, \mathcal{A}'_e, \mathcal{A}'_o)$ such that:

$$\mathit{WID}(\sigma) \Rightarrow \mathit{ID}(\sigma') \ \wedge \ |\sigma| \leqslant |\sigma'| \ \wedge \ k_e = k'_e$$

Which can be expanded using Definition 20 for $ID(\sigma')$:

$$\Big(\big(\Omega(k'_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}'_o \neq \{\mathcal{C}(k_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}'_o, \ \mathcal{C}(k_e) \,\delta^+\, a\big)\Big)$$

Knowing that $\mathcal{A}'_o \neq \{\mathcal{C}(k_e)\}$, and in any program state $\{\mathcal{C}(k_e)\} \in \mathcal{A}'_o$, there is $a' \in \mathcal{A}'_o, a' \neq \mathcal{C}(k_e) \ \wedge \ \mathcal{C}(k_e) \,\delta\, a'$. As $k_e = k'_e$, there can be no task activation generation in between the two states of the program, so $\mathcal{A}'_o \subset \mathcal{A}_o$, and therefore $a' \in \mathcal{A}_o$, which allows us to conclude:

$$\mathit{WID}(\sigma) \ \Rightarrow \ \Omega(k_e) \notin \Pi \ \wedge \ \big(\exists a \in \mathcal{A}_o, \ a \neq \mathcal{C}(k_e) \ \wedge \ \mathcal{C}(k_e) \,\delta\, a\big)$$

To prove the $\Leftarrow$ direction, let us consider state $\sigma' = (k_e, \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e), a\}, \{\mathcal{C}(k_e), a\})$. As $\{\mathcal{C}(k_e), a\} \subset \mathcal{A}_o$, we deduce that $|\mathcal{A}_e| \leqslant |\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e), a\}|$, which yields:

$$|\sigma| \ = \ |k_e| + |\mathcal{A}_e| \quad \leqslant \quad |k_e| + |\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e), a\}| \ = \ |\sigma'|$$

As we know that $\Omega(k_e) \notin \Pi$ and both states have the same trace k_e, we only need to show that $ID(\sigma')$ is true. $\mathcal{A}'_o = \{\mathcal{C}(k_e), a\}$ satisfies $\mathcal{A}'_o \neq \{\mathcal{C}(k_e)\}$ and the only remaining activation $a \in \mathcal{A}'_o$ satisfies $\mathcal{C}(k_e)\ \delta\ a$, which concludes this second part of the proof.

Finally, to prove the equivalence of weak deadlock states defined by either of the δ or $<$ relations, we show that:

$$\forall a \in \mathcal{A}_o,\ \mathcal{C}(k_e) < a \quad \Leftrightarrow \quad \mathcal{C}(k_e)\ \delta\ a$$

Proposition 16 yields $\forall a \in \mathcal{A}_o,\ \mathcal{C}(k_e) < a \quad \Leftarrow \quad \mathcal{C}(k_e)\ \delta\ a$, so we need only prove the reverse in order to conclude this proof. By definition of the stream prefix order, $\mathcal{C}(k_e) < a$ iff:

$$\exists s \in \mathcal{S}, \exists i, j \in \mathbb{N},\ j \leqslant i\ \wedge\ (W, s, j) \in \mathcal{C}(k_e)\ \wedge\ (R, s, i) \in a$$

From the Definition 9 of the continuation activation, we deduce:

$$\forall s \in \mathcal{S}, \exists \alpha \in \mathbb{N} :\ \forall k \in \mathbb{N}, (W, s, k) \in \mathcal{C}(k_e)\ \Rightarrow\ k \geqslant \alpha$$

Which allows us to conclude: $(W, s, j) \in \mathcal{C}(k_e)\ \Rightarrow\ j \geqslant \alpha$. As $j \leqslant i$, we further deduce that:

$$(W, s, j) \in \mathcal{C}(k_e) \quad \Rightarrow \quad i \geqslant j \geqslant \alpha \quad \Rightarrow \quad (W, s, i) \in \mathcal{C}(k_e)$$

By definition of the flow dependence relation, $(W, s, i) \in \mathcal{C}(k_e) \wedge (R, s, i) \in a\ \Rightarrow\ \mathcal{C}(k_e)\delta a$. □

Proof of Lemma 25. We only present the proof for $WD(\sigma)$, that of $WFD(\sigma)$ is identical. We first show that, in a weak deadlock state, we can always either find a dependence cycle or a dependence chain starting at $\mathcal{C}(k_e)$.

Let us consider $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ such that $WD(\sigma)$ which by definition implies that $\exists \sigma',\ D(\sigma')\ \wedge\ k_e = k'_e$, and we expand:

$$\exists \sigma',\ \big(\Omega(k_e) \notin \Pi\big)\ \wedge\ \big(\mathcal{A}'_o \neq \{\mathcal{C}(k_e)\}\big)\ \wedge\ \big(\forall a \in \mathcal{A}'_o, \neg \mathcal{A}'_e \ltimes a\big)$$

We use Definition 11 of the stream prefix order, yielding:

$$\forall a \in \mathcal{A}'_o, \exists (R, s, i) \in a, \exists j \leqslant i,\ \forall a' \in \mathcal{A}'_e,\ (W, s, j) \notin a'$$

Proposition 10 ensures that all stream accesses are covered by an activation in $\mathcal{A}'_e \cup \mathcal{A}'_o$ and since (W, s, j) is not present in activations from $\mathcal{A}'_e$, it necessarily belongs to an activation in $\mathcal{A}'_o$:

$$\begin{aligned} & \forall a \in \mathcal{A}'_o, \exists (R, s, i) \in a, \exists j \leqslant i,\ \exists a' \in \mathcal{A}'_o,\ (W, s, j) \in a' \\ \Leftrightarrow\ & \forall a \in \mathcal{A}'_o, \exists a' \in \mathcal{A}'_o :\quad a' < a \end{aligned}$$

We can recursively build a chain $... < a'' < a' < a$ by applying this last proposition to a, then a' which satisfies $a' < a$ and so on. As $\mathcal{A}'_o$ is a finite set, we deduce that either we stop once we reach the continuation activation or there must be at least one cycle. As $\mathcal{A}'_o \neq \{\mathcal{C}(k_e)\}$, we can choose $a \neq \mathcal{C}(k_e)$, which concludes the first part of the proof.

The proof of the reverse simply requires building a state σ' with $\mathcal{A}'_o = \{\mathcal{C}(k_e), a_1, a_2, ..., a_n\}$, where $a_1 ... a_n$ are the activations forming the cycle or the dependence chain, which is a deadlock state as the cycle or the dependence on the continuation prevents any activation to be executable:

$$\forall a \in \mathcal{A}'_o, \Big(\exists a' \in \mathcal{A}'_o,\ a' < a\Big) \quad \Rightarrow \quad \neg\ \mathcal{A}'_e \ltimes a$$

□

Proof of Proposition 26. The first part of this proof is a direct result of Lemma 21:

$$WID(\sigma) \;\Rightarrow\; \exists \sigma', ID(\sigma') \;\Rightarrow\; FD(\sigma') \;\Rightarrow\; WFD(\sigma)$$

To prove that $WFD(\sigma) \;\Rightarrow\; WD(\sigma)$, we need only consider Lemmas 14 and 19, which respectively provide the characterization of a deadlock and a functional deadlock state.

$$\begin{aligned}
&WFD(\sigma) \;\Rightarrow \exists \sigma', FD(\sigma') \\
&\Rightarrow \exists \sigma', \Omega(k_e) \notin \Pi \;\wedge\; \mathcal{A}'_o \neq \{\mathcal{C}(k_e)\} \;\wedge\; \forall a \in \mathcal{A}'_o, \neg \mathcal{A}'_e \,\Delta\, a \\
&\Rightarrow \exists \sigma', \Omega(k_e) \notin \Pi \;\wedge\; \mathcal{A}'_o \neq \{\mathcal{C}(k_e)\} \;\wedge\; \forall a \in \mathcal{A}'_o, \neg \mathcal{A}'_e \ltimes a \\
&\Rightarrow \exists \sigma', D(\sigma') \;\Rightarrow\; WD(\sigma)
\end{aligned}$$

On the third line, we use Proposition 16 which gives $\mathcal{A}'_e \ltimes a \Rightarrow \mathcal{A}'_e \Delta a$ or conversely $\neg(\mathcal{A}'_e \Delta a) \Rightarrow \neg(\mathcal{A}'_e \ltimes a)$. □

Proof of Proposition 30. We first prove that program stream causality is necessary to the existence of a stream causal schedule, according to our definitions. Let us assume, by way of contradiction, that the program is not causal in a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$. There exist $a, a' \in \mathcal{A}_e \cup \mathcal{A}_o$ such that:

$$\neg(a = a' = \mathcal{C}(k_e)), \quad a \preccurlyeq_{\text{SC}} a' \;\wedge\; a' \,\delta^+\, a$$

Which means that any scheduling function θ, that enforces a causal schedule in this state, meets the following constraints:

$$(\theta(a) \leqslant \theta(a')) \;\wedge\; (\theta(a') < \theta(a))$$

This contradiction allows us to conclude to the impossibility.

Let us consider a scheduling function θ that enforces all flow dependences. This schedule, which is obtained by using the $(EXEC_\Delta)$ transition rule, satisfies, by definition:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o : \quad a \,\delta^+\, a' \quad \Rightarrow \quad \theta(a) < \theta(a')$$

We need to show that this schedule is admissible (i.e. that it can indeed be executed) and that $a \preccurlyeq_{\text{SC}} a' \Rightarrow \theta(a) \leqslant \theta(a')$ is verified in this schedule. We remark that $a \preccurlyeq_{\text{SC}} a$ is always true for activations that contain at least one write access, which is necessarily the case if an activation is the source of a flow dependence, which we inject in Definition 28, deducing that:

$$\exists a \in \mathcal{A}_e \cup \mathcal{A}_o, a \neq \mathcal{C}(k_e),\; a \,\delta^+\, a \quad \Rightarrow \quad \neg SC(\sigma)$$

This means that stream causality in a state precludes flow dependence cycles between task activations. Furthermore, as all activations are in a stream clock precedence relation with the continuation activation:

$$\exists a \in \mathcal{A}_e \cup \mathcal{A}_o, a \neq \mathcal{C}(k_e),\; \mathcal{C}(k_e) \,\delta^+\, a \quad \Rightarrow \quad \neg SC(\sigma)$$

As there can be no cycles and no dependence chains containing the continuation activation, Lemma 25 allows us to conclude that this state is not a weak functional deadlock state, irrespectively of the fact that the control program has reached a barrier or not. This means that this schedule built on $(EXEC_\Delta)$ is able to schedule all task activations in $\mathcal{A}_e \cup \mathcal{A}_o$. As a side note, this already proves that a CDDF program that is stream causal every time the control program reaches a barrier cannot experience functional deadlocks.

Let us verify that, in a stream causal program state, this schedule is itself stream causal. From the definition of stream causality, we have $\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o$:

$$\neg(a = a' = \mathcal{C}(k_e)), \quad a \preccurlyeq_{\text{SC}} a' \quad \Rightarrow \quad \neg(a' \,\delta^+\, a)$$

We deduce that $a \preccurlyeq_{\text{SC}} a' \Rightarrow \neg(a' \, \delta^+ \, a) \Rightarrow \neg(\theta(a') < \theta(a)) \Rightarrow \theta(a) \leqslant \theta(a')$, which concludes the proof. □

Proof of Theorem 32. We prove this theorem by showing that $SC(\sigma) \wedge WD(\sigma)$ is impossible. Let us assume, by way of contradiction, that $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ is such a state.

We rewrite the Definition 11 of the stream prefix order in terms of flow dependences and the stream clock relation:

$$\begin{aligned} &\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o, \ a < a' \\ &\Leftrightarrow \quad \exists (s,i) \in \mathcal{S} \times \mathbb{N}, \exists j \leqslant i, \ (W,s,j) \in a \ \wedge \ (R,s,i) \in a' \end{aligned}$$

Proposition 10 yields $\exists a'' \in \mathcal{A}_e \cup \mathcal{A}_o, (W,s,i) \in a''$, which we add to the expression above to conclude that:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o, \ a < a' \ \Leftrightarrow \ \exists a'' \in \mathcal{A}_e \cup \mathcal{A}_o, \ a \preccurlyeq_{\text{SC}} a'' \ \wedge \ a'' \, \delta \, a'$$

As the program is causal in this state, Proposition 30 guarantees the existence of at least one causal schedule in this state. Let θ be a scheduling function for such a schedule. By Definition 29 of a causal schedule, for $a, a' \in \mathcal{A}_e \cup \mathcal{A}_o$ we have:

$$\begin{aligned} a < a' \quad &\Rightarrow \quad \exists a'' \in \mathcal{A}_e \cup \mathcal{A}_o, \ \theta(a) \leqslant \theta(a'') \ \wedge \ \theta(a'') < \theta(a') \\ &\Rightarrow \quad \theta(a) < \theta(a') \end{aligned}$$

Expanding Lemma 25 for the weak deadlock state σ yields:

$$\begin{aligned} \exists (a_1, ..., a_n) \in \mathcal{A}_o^n, a_1 \neq \mathcal{C}(k_e), \quad &\big(a_1 < ... < a_n < a_1\big) \\ &\vee \ \big(\mathcal{C}(k_e) < a_1 < ... < a_n\big) \end{aligned}$$

Where we substitute the scheduling function constraints on $<$:

$$\begin{cases} \theta(a_1) < \theta(a_2) < ... < \theta(a_n) < \theta(a_1) \quad \Rightarrow \quad \theta(a_1) < \theta(a_1) \\ \theta(\mathcal{C}(k_e)) < \theta(a_1) < ... < \theta(a_n) \quad \Rightarrow \quad \theta(\mathcal{C}(k_e)) < \theta(a_n) \end{cases}$$

Both results are contradictory. The first trivially, the second because $a_n \preccurlyeq_{\text{SC}} \mathcal{C}(k_e)$ is always true, by Definition 27, so $\theta(a_n) \leqslant \theta(\mathcal{C}(k_e))$ and $\theta(\mathcal{C}(k_e)) < \theta(a_n)$, concluding this proof. □

Proof of Proposition 38. The proof of this proposition is very close to that of Proposition 30, with a simple substitution of relations and properties. The only difference comes from the fact that contrary to stream clocks, task order does not allow ordering task activations with respect to the continuation activation. This means that the existence of a dependence chain starting at the continuation activation does not allow to conclude to a violation of causality:

$$\exists a \in \mathcal{A}_e \cup \mathcal{A}_o, a \neq \mathcal{C}(k_e), \quad \mathcal{C}(k_e) \, \delta^+ \, a \quad \not\Rightarrow \quad \neg TC(\sigma)$$

For this reason, the application of Lemma 25 only allows to conclude that, in a task causal state σ, all functional deadlocks are insufficiency deadlocks. The existence of the schedule depends on the absence of the weak insufficiency deadlock condition $\mathcal{C}(k_e) \, \delta^+ \, a$. □

Proof of Theorem 39. Let us consider the state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ of a program that satisfies the hypotheses. We use the results from Lemmas 25 and 24 instead of the definition to characterize weak deadlock states. Merging the two, we obtain:

$$WD(\sigma) \ \Rightarrow \ WID(\sigma) \ \vee \ \big(\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(k_e), \quad a <^+ a\big)$$

Replacing this expression in the theorem shows that our objective is to prove that no task activation cycle on the stream prefix order relation can exist in such states:

$$TC(\sigma) \wedge \big(\forall s \in S(\sigma) :\ |P(\sigma,s)| = 1 \ \vee\ |C(\sigma,s)| = 1\big)$$
$$\Rightarrow \quad \neg\big(\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(k_e),\ a <^+ a\big)$$

Let us assume, by way of contradiction, that the program is in a weak deadlock state σ, but not in a weak insufficiency deadlock, and therefore that $\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(k_e),\ a <^+ a$. As the program is also task causal in this state, it admits a task causal schedule. Let θ be such a schedule. Expanding this cycle yields:

$$\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(k_e), \exists n \in \mathbb{N}, n > 0, \exists (a_1, ..., a_n) \in \mathcal{A}_o^n :$$
$$\big(a_1 = a_n = a\big) \ \wedge\ \big(a_1 < ... < a_n\big)$$

From the definition of the stream prefix order, we deduce that:

$$\forall k \in [1, n-1],\ a_k < a_{k+1}$$
$$\Rightarrow\ \exists (s,i) \in \mathcal{S} \times \mathbb{N}, \exists j \leqslant i,\ (W,s,j) \in a_k \ \wedge\ (R,s,i) \in a_{k+1}$$

The stream s is either single-producer or single-consumer, so we will handle the cases separately.

If $|P(\sigma,s)| = 1$, then we look at the producer task activation a_k and its equivalence class. Either the precise write operation (W,s,i) belongs to a task activation in this class, or this operation is not yet scheduled and therefore belongs to the continuation activation:

$$\big(\exists a'_k \in [a_k]_\sim, (W,s,i) \in a'_k\big) \quad \vee \quad (W,s,i) \in \mathcal{C}(k_e)$$

If the write operation is in the continuation activation, then we would have $\mathcal{C}(k_e)\ \delta\ a_{k+1}$, which is a weak insufficiency deadlock condition and contradicts the hypothesis. We can therefore assume that $(W,s,i) \in a'_k$ and $a'_k\ \delta\ a_{k+1}$. We deduce that:

$$a_k \preccurlyeq_\mathrm{T} a'_k \ \wedge\ a'_k\ \delta\ a_{k+1} \ \Rightarrow\ \theta(a_k) \leqslant \theta(a'_k) \ \wedge\ \theta(a'_k) < \theta(a_{k+1})$$

We conclude that, in the case of a single-producer stream connecting a_k and a_{k+1}, we have:

$$a_k < a_{k+1} \qquad \Rightarrow \qquad \theta(a_k) < \theta(a_{k+1})$$

If $|C(\sigma,s)| = 1$, then we look at the consumer task activation a_{k+1} and its equivalence class. As the higher read access operation (R,s,i) has already been scheduled to a task activation, the lower index operation (R,s,j) is guaranteed to have also been scheduled, so we know that:

$$\exists a'_{k+1} \in [a_{k+1}]_\sim, \quad (R,s,j) \in a'_{k+1}$$

We have $a_k\ \delta\ a'_{k+1}$ and $a'_{k+1} \preccurlyeq_\mathrm{T} a_{k+1}$, which leads to:

$$\theta(a_k) < \theta(a'_{k+1}) \ \wedge\ \theta(a'_{k+1}) \leqslant \theta(a_{k+1}) \ \Rightarrow\ \theta(a_k) < \theta(a_{k+1})$$

By aggregating the results of the two cases, we conclude the proof with the contradiction:

$$a_1 < ... < a_n \quad \Rightarrow \quad \theta(a_1) < ... < \theta(a_n) \quad \Rightarrow \quad \theta(a) < \theta(a)$$

□

Proof of Theorem 40. The proof of this theorem derives from Lemma 24, which characterizes a weak insufficiency deadlock state, and Lemma 25, characterizing weak deadlock states as dependence cycles. To prove this theorem, we show that:

$$SCC(H(\sigma)) = \varnothing \ \wedge \ \neg WID(\sigma) \quad \Rightarrow \quad \neg WD(\sigma)$$

Let σ be a CDDF program state where $H(\sigma)$ contains no cycles and $\neg WID(\sigma)$. Lemma 24 yields:

$$\neg WID(\sigma) \ \Leftrightarrow \ \Omega(k_e) \in \Pi \ \vee \ (\nexists a \in \mathcal{A}_o, \ a \neq \mathcal{C}(k_e) \ \wedge \ \mathcal{C}(k_e) < a)$$

As $\Omega(k_e) \in \Pi$ gives $\neg WD(\sigma)$, we only keep the condition $\nexists a \in \mathcal{A}_o, \ a \neq \mathcal{C}(k_e) \ \wedge \ \mathcal{C}(k_e) < a$, which means that $\mathcal{C}(k_e)$ cannot appear in any task activation cycle in this state.

We need to show that the absence of cycles in the task graph $H(\sigma)$ means that there can be no task activation cycles in $\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}$. This is intuitively simple as the program hypergraph is an over-approximation of the communication patterns in task activations where we lose the precise information on stream access indexes.

Let us assume by way of contradiction that there is a cycle of task activations in $\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}$:

$$\exists a_1, ..., a_n \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\} \mid a_1 < a_2 < ... < a_n \ \wedge \ a_n = a_1$$

Each stream prefix order relation translates, by Definition 11, into:

$$\begin{aligned} &\forall k \in [1, n-1] : \quad a_k < a_{k+1} \\ &\Leftrightarrow \quad \exists (R, s_k, i_k) \in a_{k+1}, \exists j_k \leqslant i_k, \ (W, s_k, j_k) \in a_k \end{aligned}$$

From the previous relation, we deduce that the set of streams $(s_1, s_2, ...s_{n-1})$ satisfies:

$$\begin{aligned} &\begin{cases} (R, s_1, i_1) \in a_2 \ \wedge \ (W, s_2, j_2) \in a_2 \\ (R, s_2, i_2) \in a_3 \ \wedge \ (W, s_3, j_3) \in a_3 \\ ... \\ (R, s_n, i_n) \in a_n = a_1 \ \wedge \ (W, s_1, j_1) \in a_1 \end{cases} \\ \Rightarrow \quad &\begin{cases} C(\sigma, s_1) \cap P(\sigma, s_2) = \{a_2\} \neq \varnothing \\ C(\sigma, s_2) \cap P(\sigma, s_3) = \{a_3\} \neq \varnothing \\ ... \\ C(\sigma, s_n) \cap P(\sigma, s_1) = \{a_1\} \neq \varnothing \end{cases} \end{aligned}$$

This contradicts the hypothesis of no cycles in $H(\sigma)$ and therefore concludes the proof. □

Proof of Lemma 41. We have modeled the control program's execution, see Definition 8, with the oracle function Ω and the evaluation function ξ that builds task activations from activation points, determining the stream access indexes of each task activation and therefore the data schedule in streams. As both Ω and ξ are deterministic, the former by definition and the latter because it computes a prefix sum on a deterministic trace, the construction of stream accesses of task activations is entirely deterministic as well. □

Proof of Theorem 42. Let us assume, by way of contradiction, that there is a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ where two different executions can lead to different data in some stream location $(s, i) \in \mathcal{S} \times \mathbb{N}$. Let $a_p \in \mathcal{A}_e$ be the task activation that produced data at this location, so $(W, s, i) \in a_p$. Lemma 41 guarantees that the same task activation produces the data for a given location since this is not dependent on the execution schedule.

As we consider that all work functions are deterministic, the execution of a_p can only produce different data for location (s, i) if its inputs differ between the two executions. So there exists $(R, s', i') \in a_p$ such that the value read by a_p at location (s', i') differs between the two executions.

As $a_p \in \mathcal{A}_e$, the execution rule (EXEC) guarantees that we have $\mathcal{A}_e \ltimes a_p$. By Definition 11, of the stream prefix order, there is another task activation $a'_p \in \mathcal{A}_e$ such that $(W, s', i') \in a'_p$ and $a_p \neq a'_p$ or we would have a functional deadlock, which would contradict the hypothesis that $a_p \in \mathcal{A}_e$.

We can recursively build an infinite chain of task activations, causally tracing upstream the source of the inconsistency, and all elements in this chain are in $\mathcal{A}_e$, which is a finite set, built by the execution of task activations. We deduce that there must be a cycle in the chain, which once again leads to an impossibility. Any cycle of flow dependences represents a functional deadlock, see Lemma 25, and all activations in the cycle cannot have executed, so they do not belong in $\mathcal{A}_e$. This contradiction concludes the proof. □

Proof of Theorem 43. Let us consider such a program that deadlocks in state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ with a schedule modeled by a scheduling function θ. Let us now assume, by way of contradiction, that there exists another schedule θ' that, irrespectively of deadlocks, allows the program to execute up to a state $\sigma' = (k'_e, \mathcal{A}'_e, \mathcal{A}'_o)$ such that $|\sigma| < |\sigma'|$.

By definition, this means that $|k_e| + |\mathcal{A}_e| < |k'_e| + |\mathcal{A}'_e|$, which leads to the following two relevant cases to handle: (1) traces have different lengths, $|k_e| < |k'_e|$; or (2) the set of executed task activations is different, $|\mathcal{A}_e| < |\mathcal{A}'_e|$.

Case (1): $|k_e| < |k'_e|$. As σ is a deadlock state, we know from Lemma 14 that $\Omega(k_e) \notin \Pi$. As we have argued in the proof of Lemma 41, the control program trace is necessarily deterministic for the same input, so $|k_e| < |k'_e|$ implies that k_e is a prefix of k'_e. We deduce that $\Omega(k_e) = \mathit{barrier}$, as the trace continues, and therefore that in the schedule θ' the program was able to pass the barrier while in schedule θ, it was unable to make further progress.

From Lemma 14, we also know that $\mathcal{A}_o \neq \{\mathcal{C}(k_e)\}$. As k_e is a prefix of k'_e and task activations are created deterministically, we have:

$$\begin{aligned}&\big(\pi \in k_e \;\Rightarrow\; \pi \in k'_e\big)\\ &\quad\Leftrightarrow\; \big(a \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\} \;\Rightarrow\; a \in \mathcal{A}'_e \cup \mathcal{A}'_o \setminus \{\mathcal{C}(k'_e)\}\big)\end{aligned}$$

In order to pass the barrier all activations in $\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}$ must be executed. As in the second execution the barrier has been able to pass, and $\mathcal{A}_o \neq \{\mathcal{C}(k_e)\}$, we have:

$$\exists a \in \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\},\; a \in \mathcal{A}'_e$$

The rest of the proof is similar with the second case, where we first prove this same property before continuing.

Case (2): $|\mathcal{A}_e| < |\mathcal{A}'_e|$. If at this point $|k_e| < |k'_e|$, then we just consider this to fall in Case (1), so we consider here that $|k_e| \geqslant |k'_e|$. Using the same reasoning as above, we deduce that:

$$a \in \mathcal{A}'_e \cup \mathcal{A}'_o \setminus \{\mathcal{C}(k'_e)\} \qquad \Rightarrow \qquad a \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}$$

We also have:

$$|\mathcal{A}_e| < |\mathcal{A}'_e| \quad \Rightarrow \quad \exists a \in \mathcal{A}'_e,\; a \notin \mathcal{A}_e$$

From these two expressions we also get the same property as in Case (1):

$$\exists a \in \mathcal{A}'_e, \quad a \in \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}$$

Cases (1) and (2) continued.

The last property we have from Lemma 14 is that $\forall a \in \mathcal{A}_o, \neg(\mathcal{A}_e \ltimes a)$, which is true in particular for the activation we have been able to find in both cases:

$$\exists a \in \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\} \quad | \quad \left(a \in \mathcal{A}'_e\right) \wedge \left(\neg(\mathcal{A}_e \ltimes a)\right)$$

As $a \in \mathcal{A}'_e$, this activation was executed in the second execution schedule, so its dependences were satisfied by some subset of $\mathcal{A}'_e$ that does not contain a'. Let $\mathcal{A}''_e \subsetneq \mathcal{A}'_e$ be the smallest set of activations such that $\mathcal{A}''_e \ltimes a$. As $\neg(\mathcal{A}_e \ltimes a)$, we deduce that:

$$\exists a' \in \mathcal{A}''_e \quad | \quad a' < a \ \wedge \ a' \notin \mathcal{A}_e$$

If $a' \notin \mathcal{A}_o \wedge a' < a$ then the barrier was impossible to pass as the deadlock was an insufficiency deadlock. If $a' \in \mathcal{A}_o$, then we found the same property that allowed us to continue from cases (1) and (2):

$$\exists a' \in \mathcal{A}''_e \quad | \quad a' \in \mathcal{A}_o \setminus \{\mathcal{C}(k_e)\}$$

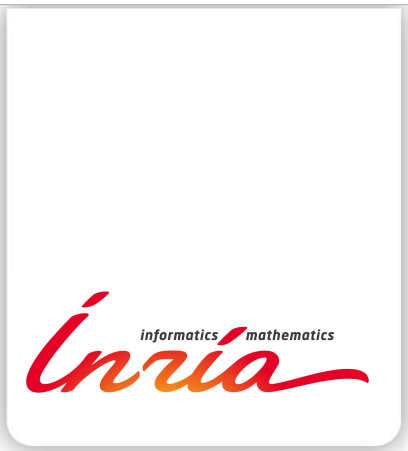
We can recursively apply this reasoning and, as $\mathcal{A}''_e \subsetneq \mathcal{A}'_e$, build an infinite sequence of strictly decreasing sets of task activations, which is impossible and therefore concludes this proof.

The fact that this particular maximum state is actually reached in all executions is deduced by swapping the role of states in the proof. If any execution cannot reach the state σ, but stops in state σ'' instead with $|\sigma''| < |\sigma|$, then the proof can be applied using σ'' as the base deadlock state. □

Proof of Proposition 44. This property is a result of the definition of the execution rules on Figure 1, where the (EXEC) rule aggregates all the dependences of task activations in the $\ltimes$ relation, therefore only allowing the execution of task activations that have all their dependences satisfied before they can start executing. This means that a task activation behaves atomically, in the sense that it cannot enable the execution of another task activation until all of its own dependences are satisfied.

Let us consider that, in a CDDF program in a state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$, there are two outstanding task activations $(a, b) \in \mathcal{A}_o^2$ that cannot be serialized, so it is impossible to schedule a before b or b before a. By definition of the scheduling constraints, we have $a <^+ b$ and $b <^+ a$, which means that there is a cycle $a <^+ a$ and therefore a deadlock according to Lemma 25. This contradicts the deadlock-freedom hypothesis.

A trivial sequential schedule, in an unbounded memory abstraction, consists in running the control program until it reaches a barrier or terminates, then executing all task activations in any order allowed by the $\ltimes$ relation, then repeat until termination. □



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399