Leveraging Sequential Computation for Programming Efficient and Reliable Distributed Systems

Ivan Kuraj[1] and Armando Solar-Lezama[2]

1 **MIT CSAIL, Cambridge, MA, USA**
2 **MIT CSAIL, Cambridge, MA, USA**

Abstract

While sequential programs represent a simple and natural form for expressing functionality, corresponding distributed implementations get considerably more complex. We examine the possibility of using the sequential computation model for programming distributed systems and requirements for making that possible. The benefits of such an approach include easier specification and reasoning about behaviors in the system, as well as a possibility to directly reuse existing techniques for checking correctness and optimization of sequential programs to produce efficient and reliable distributed implementations.

1998 ACM Subject Classification D.1.3 [Concurrent Programming] Distributed Programming, D.3.4 [Processors] Code Generation, Compilers, Optimization, F.1.1 Models of Computation

Keywords and phrases distributed systems, sequential computation, verification

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.7

1 Introduction

The sequential model of computation is often the most natural way to think about programming. Sequential programs represent computations which are performed by evaluating expressions in a predefined order. By contrast, programming distributed systems is much more challenging because programmers have to worry about controlling concurrent computations and consistency of their results, as well as a number of additional aspects, such as communication between nodes and managing data across the system.

Programming models for distributed systems need to provide means to deal with this additional complexity [5, 2]. Many modern programming models and languages for developing distributed systems allow expressing behaviors of individual nodes of the system in the sequential model, while using specialized abstractions and language constructs for specification of aspects such as communication [25, 13, 12, 7, 5]. For example, flexible and general models that allow controlling communication, expose it and require implementing it directly at a low level of abstraction, and force programmers to split behaviors into distinct program units with separate message sends and handlers [15, 26, 22]. The underlying programming models usually make trade-offs between the expressiveness and support for certain distributed aspects [5, 3, 11]. Finding an optimal trade-off, however, is difficult because models that are too low-level increase the complexity for the programmer, but models that are too high-level take away control from the programmer and risk producing code that does not match the programmer's expectations.

The paper presents a new programming model that aims to simplify the development of event-driven distributed programs. The key novelty of our programming model is that it

decomposes the problem of developing such applications into three distinct steps. In the first step, the programmer defines a data model and a collection of sequential routines that define the behavior of the system and are effectively treated as transactions. The programmer can reason about the semantics of each such transaction sequentially, without worrying about concurrency or data distribution. In the second step, the programmer uses type annotations to describe how the data is distributed across a collection of computational nodes. Based on these type annotations, the system determines where to execute the defined transactions and derives the necessary communication and synchronization in order to guarantee the sequential semantics of their executions. Finally, in the third step, the programmer defines a set of logical triggers, which dictate when the transactions should execute. The triggers can launch a transaction based on the global state of the distributed system.

We argue that the separation of concerns afforded by our approach leads to a programming model that is not just simple and expressive for developing distributed programs, but also simplifies reasoning about their behaviors, and enables various optimizations to produce *efficient implementations*. For example, the paper illustrates the potential for applying existing techniques for *verification and optimization of sequential programs* in this model. Moreover, the model effectively allows adding and changing both behaviors and distributed aspects without the need to change the code for existing behaviors.

Although demonstrated through a few running examples, the paper focuses on the ideas and a few key characterizations that are needed to define the new programming model. This paper does not present a fully-developed language that is general and applicable for producing efficient implementations for a wide range of distributed systems. Moreover, the paper focuses on the characterization and semantics of the programming model in terms of distribution of data and behaviors, and tying such behaviors to external stimuli. In turn, multiple aspects of realistic distributed systems, which include security and failure-tolerance, were omitted, making the model applicable only to certain scenarios of distributed systems where nodes operate in reliable and trusted environments.

2 From Sequential to Distributed Programs

2.1 Overview of the Approach

We illustrate the ideas behind the approach by implementing the functionality of a simple distributed application for managing storage and supplies of a central warehouse with multiple independent stores. We focus on the functionality of ordering an item from the warehouse (which possibly resides on a remote location) and updating the store's inventory accordingly. The developers start by specifying this behavior as a sequential program, from the perspective of only one store and the warehouse, ignoring distributed aspects of the system. In the subsequent step, the developers transforms the sequential program into a distributed implementation by specifying how the sequential behavior is instantiated for every store in the system, and distributed across all stores and the warehouse.

The developers start by defining the data used by the program. Given the chosen starting point of a sequential program with a single store, the developers declare a map from items (identified by `String`) to their quantities and a single variable that reflects the quota for items, which belongs to a particular store, but will later be instantiated at each store in the final distributed implementation[1]:

```
var quantities: Map[String, Nat]; var quota: Nat
```

[1] We use Scala-like syntax, with variables declared with `var`.

Afterwards, the developers implement the function for ordering some quantity of a particular item:

```
def order(item: String, quant: Nat): Boolean =
  if (quantities(item) >= quant && quota >= quant) {     // check if order eligible
    quantities(item) = quantities(item) - quant;        // update (inventory) state
    quota = quota - quant;
    true                                                      // order successful
  } else false
```

which, given the item and quantity to order, performs a simple check: if the given quantity is available in the warehouse and the store has sufficient quota left, it updates the corresponding variables and returns `true`, otherwise returns `false`. For simplicity, we assume each item takes exactly one unit of the available quota.

An important insight behind the new approach is that, modulo distribution, this simple function faithfully reflects the behaviour of item ordering. The functionality does not depend on how the data and computation are distributed; adding distributed aspects effectively provides a different view and instantiation of the functionality. Note that unlike some distributed algorithms that inherently require programming low-level aspects like communication (e.g. next actions depend on the received messages at particular nodes during execution), functionality in our example can fully be captured by ignoring distributed aspects [20]. Even though distributed implementations differ from "pure" behaviors expressed with sequential computation, given the necessary information about distributed aspects, they can be used as specifications of behaviors that can be automatically transformed into final distributed implementations.

To that end, since the sequential computation model is not sufficient to characterize such distributed applications, we identify components of specification that can, when coupled with sequential programs, completely characterize distributed implementations:

- allocation of data (used in the function) to different nodes in the system
- specifying how is the behavior (defined by the function) invoked in the system
- consistency of data and behaviors (in the presence of concurrency) in the system

Provided this additional information, developers can capture the desired behaviours in the system and allow the compiler to produce the appropriate distributed implementation. The compiler, which produces the final low-level implementation, can then decide to allocate computation and communication in a way such that the given specification – of data allocation, consistency and triggering of behaviors – is satisfied. Note that low-level details of the resulting implementation, such as communication and computation allocation, can still be decided and implemented in potentially multiple different ways by the compiler. We consider possibilities for providing and satisfying these specifications in the subsequent sections.

2.2 Location-dependent Types

We propose specifying data (and computation) allocation through types. To that end, we allow declaring nodes that participate in the system and enrich the type system to specify location of data or computation represented by the given expression. For any expression, besides associating standard type to it (e.g. in simply typed lambda calculus), we associate an additional label that designates the node the expression resides on or is computed at.

For our running example, the developers declare two sets of nodes, *Server* and *Client*, where the *Server* is a singleton set (which represents the single, centralized, warehouse). Node types are effectively instances of a higher-level type; they serve as labels that allow

distinguishing between nodes. Developers can then annotate declarations and expressions to specify the intended data allocation in the final implementation: having type `T`, with $\texttt{T}_{\diamond node}$ the developer designates allocation of the expression of the given type to *node*, which belongs to some node type. Then, developers assign `quantities` to the (single) *Server*, and `quota`, the arguments, and the result of the function `order` to the *Client* nodes:

```
var quantities: Map[String,Nat]◇Server; ∀client : Client, var quota: Nat◇client
∀client : Client, def order(item: String◇client, quant: Nat◇client): Boolean◇client = ...
```

To designate that `order` might be invoked by any store, developers write ∀ to quantify over the set of store (*Client*) nodes (similarly to classical type-dependent systems). Note that the definition of `order` is omitted; it remains the same as before.

It is interesting to consider how the location information given in types affects the resulting implementation. Specifically, the program states that both arguments, as well as the result of the function, are allocated to the client node. However, `quantities` – variable used in the function – is allocated to the server, thus communication between the nodes is inevitable. To produce an implementation that type-checks (performs allocation as specified), the compiler has to update quantities on the server and return the result back to the client. Thus, the intended implementation where a client node (i.e. a store) sends arguments to the server (i.e. the warehouse) and awaits a response would typecheck successfully and can be produced as the resulting implementation.

Given the previous definition, since the state update has to happen on the server, the following has to typecheck:

```
(quantities(item) = quantities(item) - quant): Unit◇Server
```

meaning the expression is evaluated at the *Server*. However, value of the assignment can be computed also on the client store (if e.g. forced by a typing annotation):

```
(quantities(item) - quant): Nat◇client
```

which would produce a correct, but less efficient implementation, since it would incur additional two-way communication between the client and the server (for `quant` and the result). Effectively, location-dependent types allow developers to dictate allocation of data and computation within the system. Moreover, they can be used to prevent certain communication (e.g. for security reasons), where unwanted implementations, which could lead to potential data leaks, would not typecheck.

2.3 Triggering Behavior with Logical Formulas

In order to capture how are behaviours in the system invoked, we propose defining triggers as logical formulas, which can talk about events occurring across the system, as well as data, i.e. state, located at different nodes in the system. Whenever the condition defined by the trigger becomes true, the associated behavior is invoked. By combining events that can be bound to arbitrary internal and external actions (or stimuli) in the system, with formulas, we gain flexibility for specifying various, possibly reactive, distributed systems.

In our running example, we would like to trigger `order` as a consequence of an explicit user action (e.g. user interaction). We assume to have event $EvOrder_{Client}(quant : Nat)$ that can occur at any store in the system and is parametrized by a value for quantity (populated at the time of the instantiation of the event). In our current prototype, events are declared with a special construct in the language, while compiler generates function calls for each declared event (with arguments that match parameters of the event), which, when called, "fire" the associated event in the system.

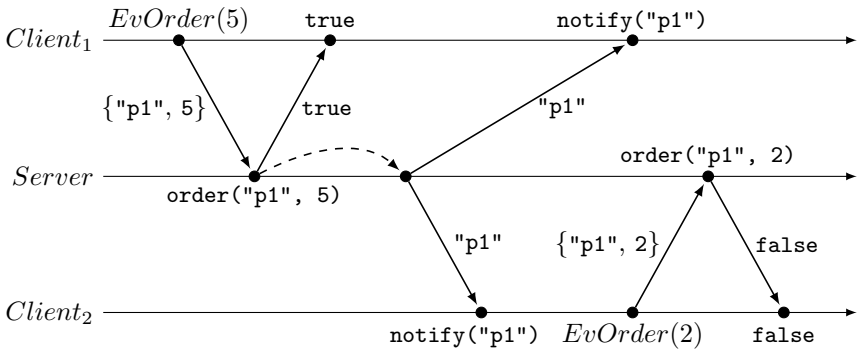

Figure 1 Distributed execution of two instances of `order` and one of `notify` behavior.

After declaring a variable for the selected item at every store, the developers can specify that `order` gets triggered whenever *EvOrder* fires at any client node (i.e. store):

$\forall client : Client$, `var selected: String`$_{\diamond client}$ `// designates selected item`
$\exists c : Client.\ EvOrder_c(quant) \rightarrow$ `order(selected`$_{\diamond c}$`, quant)`

where `order` is called with `selected` variable located on the client and the `quant` parameter from the event. (With `e`$_{\diamond n}$ we specify allocation of expression `e` to node n without specifying the full type.) The event expression on the left of the arrow can be thought of as pattern matching: variable `quant` captures the parameter value carried by the event.

Our programming model allows using logical formulas to specify triggering of behaviors when a certain condition becomes true in the system. If we consider a new functionality of notifying stores when the item they have selected gets out of stock, the following logical formula can specify such trigger. (We omit the function definition, which only updates the store-local state.)

`def notify(item: String): String`$_{\diamond client}$
$\exists client : Client,\ item : String.\ selected_{\diamond client} = item\ \wedge\ quantities(item) = 0\ \rightarrow$ `notify(item)`

The behavior returns a notification string (that is saved at the store client) and gets invoked whenever a store has selected an item (by changing `selected`) with quantity 0. (Note that a shorter condition with $quantities(selected_{\diamond client}) = 0$ can be written as well.) The semantics of triggering dictates that behaviors are invoked only once, whenever the condition changes from false to true in the system. Note that the produced implementation needs to incur communication between stores and the warehouse to check if the given condition became true, due to the $\exists$ quantifier (at least when the selected item at some store changes or quantity for some item becomes 0). To guarantee this, the compiler emits additional checks that check the condition whenever it might become true; more specifically, after every change to `quantities` on the server and `selected` on the clients.

2.4 Semantics and Consistency of Resulting Implementations

In order to characterize possible valid resulting distributed implementations, the model should constrain their behavior to match the behavior defined with sequential program, as well as the constraints of distributed aspects. Effectively, a valid implementation should project the given sequential programs onto the distributed system, respecting specified allocation and triggering conditions.

A possible execution of a distributed implementation of behaviors defined by `order` and `notify` is illustrated in Figure 1. The system includes one *Server* and two *Client* nodes.

We assume the initial value of `selected` is `"p1"` at both store clients and its quantity is `5` at the warehouse. The dots on the timelines designate the points in time of either firing of a trigger or evaluation of the given expression at the given node. Note that *EvOrder* events fire (after being invoked programatically) on the store client nodes, while the dashed line designates that the execution of `order` triggers the `notify` behavior; all three invoke the behavior defined by the appropriate function. Labels on the edges denote values that are being communicated between nodes. Note that behaviors are split, by the compiler, into multiple executions on potentially multiple nodes in the system.

An important aspect for enabling natural and convenient reasoning about (possibly concurrent) executions of behaviors is guaranteeing consistent execution. We propose allowing reasoning about end-effects of executions as if behaviors specified with sequential programs executed atomically, in the same order observed anywhere in the system (alike guaranteeing linearizable executions [14]). In Figure 1, the system executes the behaviors in a consistent way, relative to the linear order of triggering of each of the behaviors; more specifically, fist invoking `order` on $Client_1$ which causes `notify` to be invoked afterwards, followed by another `order` invoked on $Client_2$.

Even though providing such strong guarantees in the distributed setting might be costly (atomicity requirement might require effectively locking all nodes participating in the behavior beforehand [4]), we demonstrate the possibility for avoiding such overheads by analyzing the defined behaviors and their possible concurrent executions. We analyze three different cases of the resulting implementation, going from the variant that uses the most pessimistic mechanisms to variants that leverage the specifics of the behaviors to relax the used mechanisms arriving at a more efficient implementation that achieves the same results in terms of the intended semantics and strong consistency guarantees.

Let us consider the running example, with a single server for the warehouse and multiple store clients, with a small addition. In addition to the presented operations `order` and `notify`, for the purpose of examining negative effects of reordering of behaviors observed at store clients, we assume an additional operation `notifyAvailable`, which is similar to `notify`, but notifies the store that the selected item became available (its quantity became positive):

$\exists client : Client,\ item : String.\ selected_{\diamond client} = item\ \wedge\ quantities(item) > 0\ \rightarrow$
`notifyAvailable(item)`

Note that, as shown in Figure 1, since the state is allocated according to specified location-dependent types, behaviors consist of code execution on both types of nodes, together with message sending and handling. Consequently, this allows inconsistent execution orders in which executions of `notify` and `notifyAvailable` are observed in different order on the server and clients. To guarantee strong consistency, the compiler needs to emit distributed implementations that prevent such inconsistent executions. We discuss different mechanisms the compiler might choose to use in this case, depending on the analysis of the semantics of the involved behaviors:

Distributed Locking. A pessimistic method for ensuring strong consistency can be achieved by using distributed locking. Commonly used in transactional processing, distributed locking tries to arrange a particular set of nodes to agree on a particular transaction, avoiding inference from other transactions, effectively locking those nodes for exclusive rights of the transaction [4]. Although achieving strong consistency (through strong serializability, where the order of transactions is defined by the acquisition of locks), algorithms for achieving such locking are prohibitively expensive and often unusable in practice.

In our example, this method can straightforwardly be applied to all the nodes in the system, such that behavior in the system can be invoked only after "acquiring" the global lock. This approach clearly ensures strong consistency, albeit at a prohibitive cost of allowing execution of only one behavior (either that of `order` or notifications) at any point in time across the whole system.

Central Point of Serialization. A simple observation in our example, where behaviors overlap at the single warehouse server, reveals the possibility to achieve more efficient, but also strongly consistent implementation. More specifically, it is sufficient to rely on the warehouse server to enforce ordering between executions of behaviors that might interfere, at the point they are executed on the server. A common mechanism for achieving this is to simply assign an index to messages that correspond to behaviors at the central point of serialization (in our case, the server), so that all nodes in the system can order messages, and thus corresponding executions [2, 4].

In our example, since the functionality of ordering and notifications depend only on the state that is located on the server, the server assigns an index to messages that carry the resulting values (e.g. a Boolean value for `order`) so that clients deliver (and end) behaviors in the same order as on the server. Guaranteeing the same order of observing behaviors is sufficient to guarantee linearizability, where the effects are the same as if behaviors were executed in a serial manner. Note that, e.g. when a notification is issued for an out-of-stock item (`notify`), it is issued after the `order` that caused it, so that store clients always receives confirmations of their orders and corresponding out-of-stock messages in the right order (with the need to store messages that are received out-of-order to deliver them later).

Removing Redundant Executions. A further observation is that even though ordering of behaviors is sufficient, it is not necessary to execute all parts of behaviors in certain cases; some parts of executions can simply be ignored, while preserving the semantics.

Behaviors for notifications gets invoked whenever quantity of an item becomes 0 or gets increased from 0. If an item quantity becomes 0 and then gets increased, it is incorrect to observe the two different associated notifications in a different order on any of the store clients. (This clearly cannot happen if either of the two previously presented mechanisms is used.) Interestingly, for any set of notification behaviors that is executed, since both `notify` and `notifyAvailable` just mutate per-store state (perform destructive updates), the store clients can simply execute (i.e. observe) only the latest one, ignoring all the previous ones. Therefore, a simple optimization of the previous implementation is to always execute the latest notification behavior on the store clients, regardless of their order determined on the server and discard any out-of-order notifications. This does not violate the semantics and guarantees linearizable executions, while achieving better performance in cases of concurrent notification behaviors. Note that the compiler can perform this optimization in any such case of destructive updates (without side-effects).

A key insight behind our proposal is that the compiler, after analyzing defined behaviors and their concurrent executions, can discover that not only the most pessimistic implementation is possible, but also two further optimizations, arriving at a more efficient implementation that satisfies the semantics and consistency requirements of the given program. In the worst case, if the compiler cannot discover any optimizations, the emitted implementation can use the most pessimistic mechanism. We leave further concerns of handling semantics and consistency concerns, as well as possible optimizations the compiler can perform in the general case open, while assuming such strong guarantees in the rest of the paper. We did not fully explore the extent to which such a program analysis can detect and perform

optimizations to arrive at efficient implementations for a wider range of distributed systems. It would be interesting to examine the possibility of employing various existing lower-level distributed algorithms and systems in the produced implementations.

2.5 Defining Dynamic Structure of Distributed Systems

Some distributed systems dynamically maintain structure which conceptually corresponds to a (sequentially defined) datastructure. In such cases, changes in the structure of the system reflect modifications of the corresponding datastructure. To demonstrate flexibility of the model in such cases, we incorporate a notion of a mapping between a datastructure and the structure of the desired distributed system.

Let's assume we want the structure of our system to correspond to a search tree (which is not uncommon, e.g. in large-scale computing [23]). Having declared nodes that store data in the system with *Node* and client nodes with *Client*, the developers can designate the mapping between the defined binary search tree abstract datatype to nodes with $\leftrightarrow_\sigma$[2]:

```
Node ↔σ BST, trait BST
case object Leaf extends BST; case class Inner(l: BST, v: Int, r: BST) extends BST
```

which associates every `Leaf` and `Inner` instance to a node (of type *Node*) in the system. Afterwards, the developers can refer to a node associated with an expression `e: BST` with $\sigma(e)$. The programming model creates one mapping for every designated datastructure; it's purpose is to provide implicit associations between instances of the datastructure and labels that denote location. Therefore, location of tree instances `tree = Inner(l, v, r)` and `l` are, $\sigma(tree)$ and $\sigma(l)$, respectively (where, by default, the model treats them as different physical nodes as well).

Next, the developers implement insertion of a new element into the tree, where each key is assigned to a separate (newly created) node, with the following function:

```
∃c : Client, n : Node. EvInsert(c,n)(key) →
def insert(tree: BST◇n, key: Int◇c): Inner = tree match {
  case Leaf => Inner(Leaf, key, Leaf): Innerσ(tree)
  case Inner(l, v, r) => if (v < key) Inner(l, v, insert(r, key))
    else if (v > key) Inner(insert(l, key): Inner◇σ(l), v, r)
    else Inner(l, v, r) }
```

where the event *EvInsert* represents an action on some client node c that targets a data node n. (Where the most common case for n is the node that represents the root of the tree.) Due to the declared mapping between nodes and trees, when a new tree node is created, a new data node in the distributed system is bound to it. Effectively, this code implements a distributed version of the tree, where each tree node is located on a separate physical node.

For illustration, the developers have annotated two expressions in the function, both of which if typechecked, should execute on nodes defined with σ. For example, insertion into the left sub-tree executes on, and creates, data node $\sigma(l)$, where $\sigma(l)$ is different from $\sigma(tree)$, thus insertion into sub-trees executes across different nodes in the system. Note that a valid implementation, which matches the datatype, needs to (know how to) create new tree nodes and assign appropriate values to them (i.e. initialize their state). (Interestingly, for an implementation closer to realistic scenarios, data nodes can be mapped to elements that can be easily created and migrated between different physical nodes, such as actors in the actor model [1].)

[2] Again, we use Scala syntax for defining abstract datatypes as class hierarchies.

3 Verifying Distributed as Sequential Programs

One of the insights behind our approach is that by allowing developers to specify distributed systems with sequential programs we can eliminate much of the complexity that stems from handling distributed aspects, and as a consequence, decrease the amount of programming errors, as well. Moreover, we will demonstrate that such an approach allows incorporating existing techniques for analyzing and verifying correctness of sequential programs into development of distributed applications.

3.1 Checking Correctness of Application Logic

In addition to simpler reasoning about the behavior of the system, alleviating the concurrency, communication, and other low-level details enables direct application of techniques for checking correctness of sequential programs. Here, we demonstrate that we can easily check functionality of the system with a standard verification technique, solely by the virtue of relying on sequential programs for specifying behavior.

Even though our running example is simple, we can imagine verifying the property that no order can be made if the given item is out of stock, regardless of the quantity. To check this property, we formalize it with the following logical formula:

$$\forall quant, item.\ quantities(item) = 0 \wedge (res = order(item, quant)) \rightarrow\ res = \bot$$

which can easily be translated into a verification condition and checked with an off-the-shelf SMT solver. After encoding this condition as an SMT instance, we verified it in less than half of a second with the CVC4 SMT solver.

Verifying the condition as a low-level distributed implementation would need to take into account the introduced distributed aspects and would become considerably more complex. This example hints that by separating functionality from specifying distributed aspects, we can leverage existing verification tools for sequential programs and effectively translate all the obtained guarantees to the resulting distributed implementations.

3.2 Checking Correctness of Concurrent Behaviors

The fact that we can rely on behaviors faithfully translated into strongly consistent executions of a distributed system affects the extent to which the system can be tested and checked. As one of the possible techniques that offer potential for scalability, we will consider model-checking concurrently executing behaviors and demonstrate an immediate gain in scalability, relative to model-checking low-level distributed implementations.

Let's add functionality of item transfer to our warehouse application: stores can transfer specified items (for simplicity, we allow only a single transfer of a predefined quantity) to other stores, while those items should be ordered at some point later from the warehouse (to account the transfer). Transfers effectively transfer quotas between stores, so the store that received a transfer can use it for orders, while the store that made the transfer needs to settle it with the warehouse (i.e. to decrease its quota accordingly). We omit some definitions: transfer is tracked with `tStat`, while the code for `order` now uses and updates a transfer, if any transfer at the store exists. The developers implement this functionality with:

$\forall client : Client$, `var tStat: String`$_{\diamond client}$ `// track a transfer`
$\exists c1, c2 : Client.\ EvTransfer(c1, c2)$(`item`) $\wedge\ c1 \neq c2\ \rightarrow$ `transfer(item,` `tStat`$_{\diamond c1}$`,` `tStat`$_{\diamond c2}$`)`

where they ensure that transferring an item can occur between two stores (here, located at client nodes `c1` to `c2`), as long as the two nodes differ. The condition should prevent the

item from being transferred back to the original store and avoid chain of transfers without accounting the transfer with the warehouse (i.e. artificially inflating the quotas).

The developers can then formulate the correctness condition as an LTL formula, where the transfer status is appropriately encoded with predicates *pending* (transfer sent from a store), *received*, and *settled* (the pending transfer gets settled):

$$\forall item.\ \mathbf{G}(received(item) \rightarrow \mathbf{X}(pending(item)\ \mathbf{U}\ settled(item)))\,.$$

The formula states that for all items *item*, it is always (globally) true that, if *item* is *received*, starting from the next step, the transfer status of *item* will be *pending* until it is *settled*. Note that such a formula is sufficient since we assume only one transfer is possible. Model-checking this formula can reveal that the condition in the trigger is not sufficient: a store can get it's own item transferred back, simply by giving it and receiving it back, and use it without the necessary accounting.

By checking behaviors as transactional executions of given sequential programs (which is sound due to strong consistency guarantees), we can discover this bug in considerably less iterations than when checking low-level implementations, due to the combinatorial explosion of the search space caused by intertwined low-level steps, including message sends and receives.

4 Program Transformations as Optimization

Having functionality expressed with sequential programs enables applying program analysis not just for checking functional correctness, but also for optimizations; many semantic-preserving transformations that apply to sequential programs can be reused in order to generate more efficient distributed implementations.

4.1 Optimizing with Data Allocation

Data allocation greatly influences possible resulting implementations and their performance. In many cases, by invoking behaviors only when needed the compiler can optimize away much of the communication in the system, while preserving the specified functionality.

In our running example, when implementing `order`, instead of sending data to perform the check on the warehouse server, the compiler can generate an implementation with:

```
(quota >= quant): Boolean◇client
```

which checks the given condition on the store client and thus avoids incurring unnecessary communication in case the condition is false (i.e. the store does not have sufficient quota to make the order and the order fails immediately).

Note that in this case as well, this optimization can be discovered and performed by the compiler automatically, without any intervention from the developers. At this point, the compiler only considers optimizations that decrease the amount of communication in the system, while in many cases other optimization metrics could be used as well. (The compiler can choose implementations that make different trade-offs; the implementation might incur less communication, but transfer more data overall.) The programming model offers possibilities for extending the compiler to consider different optimization metrics.

4.2 Removing Unnecessary Triggers

Conditions, which invoke behaviors, might trigger at any point and place in the system; in the general case, the compiler needs to insert code that checks the condition at many places

in the implementation, pessimistically. However, often, there are much fewer places where conditions can actually trigger. In addition to only checking condition at places at which the variables mentioned in the formula of the condition might change, if proved that the condition cannot become true regardless of the value of variables, the condition checking at that place can be eliminated altogether. This optimization becomes more significant in cases where to check the condition, communication between nodes needs to be incurred.

In the running example, lets consider the distributed warehouse with previously defined behaviors, including notifications for an item becoming available for ordering (`notifyAvailable`, as presented before). In addition, developers add functionality for adding a certain quantity of an item to the warehouse (refilling the warehouse) as `addItem` (which is similar to `order`; we will omit the definition for brevity and assume it can be invoked at store clients similarly):

$\exists c : \mathit{Client}.\ \mathit{EvAdd}_c(\mathit{quant}) \rightarrow$ `\mathit{addItem}(\mathit{selected}`$_{\diamond c}$`, \mathit{quant})`

Having all these sequential definitions, in the worst case, the resulting implementation would need to check conditions for the two notifications (`notify` and `notifyAvailable`) both at the place where the item quantity gets decreased (in `order`) and increased (in `addItem`), since at those places item quantities change (and notifications might potentially need to be invoked). However, only two checks are (provably) needed: for the case of out-of-stock notifications, they surely cannot be triggered when item quantity is increased (in `addItem`). The compiler can guarantee this by checking the satisfiability of the following implication:

$$\exists \mathit{item}.\ \neg(\mathit{quantities}(\mathit{item}) = 0) \wedge (\mathit{res} = \mathit{quantities}(\mathit{item}) + 1) \rightarrow \mathit{res} = 0$$

which states that there exists an item, for which if the quantity was not zero (due to triggering only conditions that become true), after incrementing the item's quantity, the quantity can become 0. This logical formula is clearly not satisfiable. Therefore, an optimized implementation can completely omit checking the condition and the notification functionality in that case. In the produced implementation, this halves the total number of invocations of the functionality for both notifications, on average.

4.3 Inferring and Generating Contexts

Some distributed applications behave in specific ways depending on the current context: most notably, some behaviors might be enabled only under a specific context. In our programming model, such contexts can be specified implicitly in triggering conditions. However, one possible optimization that compiler can perform is to infer more general contexts from the specified triggering conditions, and maintain and propagate them across the system to optimize certain behavior executions, e.g. to avoid unnecessary communication.

As an illustration of the idea, in our running example, if developers change the definition of `order` and add an additional expression to the triggering condition, such that the functionality depends on the warehouse being non-empty (since otherwise the order will fail), as:

$\exists c : \mathit{Client}.\ \mathit{EvOrder}_c($`quant`$) \wedge (\exists \mathit{item}.\ \mathit{quantities}(\mathit{item}) > 0) \rightarrow$ `order(selected`$_{\diamond c}$`, quant)`

and add other functions that depend on the same condition or the negation of it (i.e. that the store is empty, $\forall \mathit{item}.\ \mathit{quantities}(\mathit{item}) = 0$), the program analysis can infer this as a context. If so, the compiler can then produce an implementation that propagates the state of the warehouse as the context, within messages for other behaviors, and prevent unnecessary executions of further `order` requests.

Effectively, given the specifications are satisfied, program analysis can abstract the resulting system as a state machine. In addition to being more efficient, due to prohibiting

unnecessary executions depending on the context, the relation between behaviors and identified contexts can make reasoning and verifying the system easier.

5 Related Work

Many approaches presented in prior work focus on using sequential computation to some extent while introducing additional abstractions, such as remote procedure calls, reactive values, and conflict-free replicated data type, for handling distributed aspects of the system [7, 17, 25, 9, 24]. A related line of research includes programming platforms based on writing sequential programs that aim at abstracting away infrastructure concerns to allow focusing on the application logic [3, 18]. An overview of different programming models and the influence of the sequential model on programming distributed systems is given in [5, 2]. In general, even though these models abstract away some of the complexity, due to the close match between the program and the final distributed implementation, expressing certain complex behaviors requires low-level reasoning and careful structuring of the program [28, 26].

Our approach is aligned with the idea of using high-level specifications of distributed aspects and offloading the search for low-level implementations to the compiler. Some approaches lift the abstraction of specifying behaviors by using similar mechanisms to the ones employed by our approach, including logical formulas (used for triggering in our approach) in the form of event guards and await statements, and the concept of location, which allows automatic data distribution according to specifying computations [20, 16, 10]. Prior work discusses the importance of preserving semantics of sequential computation and its effects on possible optimizations, as well as the potential role for programming distributed systems [21, 19]. In the similar spirit, this work tries to motivate lifting the level of abstraction by demonstrating potential gains in simplicity and performance. Moreover, it provides a different perspective on formalization of sequential computation and specifications to allow additional means for ensuring correctness and efficiency of the resulting implementation.

While our approach focuses on implementing behaviors which can be conceptually expressed as sequential programs, it lacks expressiveness for programming distributed algorithms that inherently require dealing with aspects like processes and messages, and require control of low-level concerns [20, 26]. While re-implementing such algorithms is rarely needed, they often cannot be used directly via an external library (e.g. if modifications to some of its internals are needed); our approach aims at utilizing different existing algorithms as means to an end whenever necessary, even in cases their code needs to be customized for specific needs of the intended distributed application.

Our approach shares some of the high-level goals with the following lines of research on programming distributed systems:

Tierless Programming Models. Similar in spirit of avoiding the complexity and breaking the underlying programing model, tierless programming models focus on simplifying specification of aspects that cut across different tiers and unify them into a single model (and traditionally, focus on web development) [8, 27, 7]. Although these models simplify some of the aspects considered in this work, including communication, storage and interaction, their focus is to remove the complexity that arises due to handling different tiers of the system, rather than on preserving the semantics and structure of sequential computation within the same tier. Note that tierless models usually adopt existing mechanisms and constructs, such as client-server architecture and RPC for communication [7].

Actor-Based Programming Models. Despite being flexible and providing clean abstractions for programming distributed event-driven systems that can easily be mapped to actual physical systems, actor models suffer from being close to the low-level implementations, where the structure of the system and behaviors need to match closely with the declared programs, making them complex and hard to reason about [1, 15, 13, 26]. Interestingly, actor-based programming frameworks represent a good fit for a low-level model that can be leveraged in the final emitted implementations [19, 26].

Partitioned Global Address Space Partitioned global address space (PGAS) models aim to provide a simple programming model, and consequently allow better performance, for parallel programs by unifying the support for data and task parallelism, and abstracting the data model through a global address space [6, 10]. The concept of a "place" in these models allows allocating computations and data across the global address space, at a level that can be closer to the intended (sequential) behavior. Although places allow assigning a cost model to data accesses (based on the topology), automatic data distribution is usually restricted to partitioning of regular and dense data structures such as arrays; some PGAS languages require explicit distribution of data objects to remain expressive for irregular and sparse structures [10]. Nodes in our model are similar to places in PGAS in that they contain running computations, which in turn might be spread across multiple different nodes. However, our model does not rely on specific patterns of data distribution and parallelism; it analyzes defined behaviors to emit event-driven implementations that need to satisfy consistency guarantees, and appropriately allocate both computation and data.

6 Concluding Remarks and Vision

The sequential model of computation provides a natural way for expressing computation. However, the sequential model alone is not sufficient for programming distributed systems. As such, it is either heavily ignored, or to large extent complicated, in modern programming models and languages for distributed systems, due to the need to accommodate distributed aspects such as data allocation and communication.

This paper explores fully reusing sequential computation model for expressing behavior, while characterizing intended distributed systems with orthogonal specifications. With separation of concerns of expressing behavior and specifying distributed aspects of the system, by writing orthogonal constraints, we can achieve development of distributed systems without breaking the simplicity of writing and reasoning about sequential programs. We have shown an approach to specifying data and computation allocation through enhancing the type system and defining behavior invocations with logical formulas. We motivated the new approach by demonstrating potential benefits in the development process, not just in terms of simplicity in writing programs, but also checking their correctness and applying semantic-preserving optimizations for emitting efficient distributed implementations.

A number of challenges remains for completely characterizing the programming model and transforming it into a programming language expressive for development of realistic distributed systems. We only briefly discussed the strong semantics and consistency guarantees that the model should provide as an interface for developers, while demonstrating an approach that can emit efficient implementations in certain scenarios. Achieving strong guarantees, together with efficiency, in the general case, remains an open problem, for which a solution would potentially require combining multiple techniques and results from the domain of programming languages and distributed computing. As hinted in the paper,

providing a high-level interface for specifying behaviors through sequential programs opens up possibilities for many lower-level design choices in the final implementation; one interesting venue to explore represents not just more flexible data allocation, but also data sharing and replication, and the needed mechanisms the compiler would need to utilize.

References

1 Gul Agha. Actors: a Model of Concurrent Computation in Distributed Systems. *MIT Press*, 1986.

2 Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 1989.

3 Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. Cloud-native, event-based programming for mobile applications. In *MOBILESoft*, 2016.

4 Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 1981.

5 Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and Distribution in Object-oriented Programming. *ACM Computing Surveys*, 1998.

6 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, Vivek Sarkar, Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

7 Adam Chlipala. Ur/Web: A simple model for programming the Web. In *POPL*, 2015.

8 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *FMCO*. 2007.

9 Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *PLDI*, 2013.

10 Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, 2015.

11 Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *OOPSLA*, 2014.

12 Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with High-Level Programming Languages. *ESOP*, 2001.

13 Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2009.

14 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 1990.

15 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, 1973.

16 K. R. Jayaram and Patrick Eugster. Program analysis for event-based distributed systems. In *DEBS*, 2011.

17 JMacroRPC – reactive client/server web programming. URL: `http://hackage.haskell.org/package/jmacro-rpc`.

18 Emre Kiciman, Benjamin Livshits, Madanlal Musuvathi, and Kevin C. Webb. Fluxo: a system for internet service programming by non-expert developers. In *SoCC*, 2010.

19 Ivan Kuraj and Daniel Jackson. Exploring the role of sequential computation in distributed systems: motivating a programming paradigm shift. In *Onward!*, 2016.

20 Yanhong A. Liu, Scott D. Stoller, Bo Lin, Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *OOPSLA*, 2012.

21 Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. The Silently Shifting Semicolon. *SNAPL*, 2015.

22 M. Felleisen Matthews, J., R. B. Findler, P. T. Graunke, S. Krishnamurthi. Automatically Restructuring Programs for the Web. In *ASE*, 2003.

23 Mehul Nalin Vora. Hadoop-HBase for large-scale data. In *ICCSNT*, 2011.

24 Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *PPDP*, 2015.

25 Meteor – pure javascript web framework. URL: `http://meteor.com`.

26 Aleksandar Prokopec and Martin Odersky. Isolates, Channels, and Event Streams for Composable Distributed Programming. *Onward!*, 2015.

27 Manuel Serrano and Gérard Berry. Multitier programming in Hop. *Communications of the ACM*, 2012.

28 Andrew Stuart Tanenbaum and Robbert van Renesse. A critique of the remote procedure call paradigm. Technical report, Vrije Universiteit, 1987.