Opis: Reliable Distributed Systems in OCaml

Pierre-Évariste Dagand

ENS Cachan-Bretagne, France

pierre-evariste.dagand@ens-cachan.org

Dejan Kostić

EPFL, Switzerland

dejan.kostic@epfl.ch

Viktor Kuncak

EPFL, Switzerland

viktor.kuncak@epfl.ch

Abstract

Concurrency and distribution pose algorithmic and implementation challenges in developing reliable distributed systems, making the field an excellent testbed for evaluating programming language and verification paradigms. Several specialized domain-specific languages and extensions of memory-unsafe languages were proposed to aid distributed system development. We present an alternative to these approaches, showing that modern, higher-order, strongly typed, memory safe languages provide an excellent vehicle for developing and debugging distributed systems.

We present Opis, a functional-reactive approach for developing distributed systems in Objective Caml. An Opis protocol description consists of a reactive function (called event function) describing the behavior of a distributed system node. The event functions in Opis are built from pure functions as building blocks, composed using the Arrow combinators. Such architecture aids reasoning about event functions both informally and using interactive theorem provers. For example, it facilitates simple termination arguments.

Given a protocol description, a developer can use higher-order library functions of Opis to 1) deploy the distributed system, 2) run the distributed system in a network simulator with full-replay capabilities, 3) apply explicit-state model checking to the distributed system, detecting undesirable behaviors, and 4) do performance analysis on the system. We describe the design and implementation of Opis, and present our experience in using Opis to develop peer-to-peer overlay protocols, including the Chord distributed hash table and the Cyclon random gossip protocol. We found that using Opis results in high programmer productivity and leads to easily composable protocol descriptions. Opis tools were effective in helping identify and eliminate correctness and performance problems during distributed system development.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Design, Reliability, Verification

Keywords Arrows, Distributed Systems, Functional Programming, Model-checking

1. Introduction

Concurrent and distributed systems play an increasingly important role in the infrastructure of modern society. Ensuring reliability of such systems is difficult because they combine a number of challenges in design, algorithms, and implementation. The assumption behind this paper is that programming languages, models, and tools are key generic mechanisms that can greatly help in developing concurrent and distributed systems, making them more widespread and more reliable.

Among several models of concurrency, message passing has been successful in theory and practice of distributed systems [33]. For example, the Singularity operating system design based on message passing proved reliable and sufficiently efficient [21]; software based on Erlang scales to industrial systems with large numbers of distributed processes [1].

In this paper we explore message passing in the context of distributed systems deployed over wide-area networks. In such systems, message passing naturally describes the underlying physical constraints, where network communication delays and failures unavoidably affect system performance. We draw the examples of our paper from *peer-to-peer* distributed systems, in which each participant acts both as a client and as a server. This symmetry in principle offers extreme scalability and has been applied to peer-systems with thousands of nodes. To manage the complexity of this task, a peer-to-peer system often builds a logical network over the physical one: this is called an *overlay*. Although simple in principle, building and maintaining an overlay is a complex and error-prone task. In a real environment, peers continuously enter and leave the network, and communicate through asynchronous, unreliable channels. Consequently, even a seemingly innocent mistake in these systems can lead to dramatic and difficult to reproduce errors.

Until recently, many peer-to-peer applications were built essentially from scratch, using general-purpose programming languages. However, the complexity involved in recent systems has grown to a point where low-level details, such as manipulating network sockets or dealing with timers, burden the development effort. A recent advance is the use of domain-specific languages such as P2 [31] and Mace [22] to enable the same description to be used for simulation and deployment. Moreover, tools such as MaceMC help the developer find subtle semantic errors through explicit-state model checking of code. We believe that approaches that incorporate advanced analysis techniques are essential for building reliable distributed systems of the complexity that matches future requirements. Our paper presents Opis, a new system that comes with a number of advanced analysis tools. Unlike systems such as Mace that are based on C++, Opis uses a strongly typed language, Objective Caml (OCaml), as the
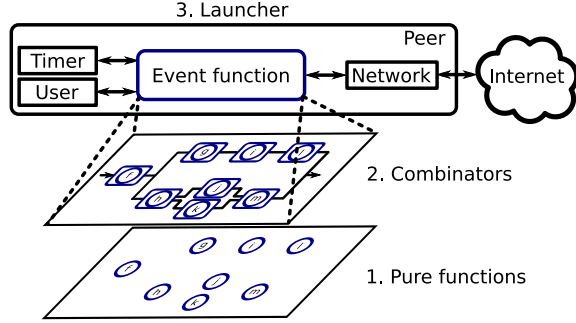
Figure 1. Opis programming model

implementation language for distributed systems. We next highlight key aspects of the Opis approach.

1.1 Modern Programming Language Platform

We argue that a modern tool for distributed system development should, first of all, benefit from memory safety, type safety, modularity, and parametrization that have already proved effective in sequential programs. The implementation language of choice in Opis is OCaml, a popular language that offers type safety and good performance, and has an encouraging history of use in distributed applications [16, 43]. Our current implementation of Opis takes the form of a library. The library contains both combinators for developing Opis applications and tools for deploying and analyzing such applications.

1.2 Layered Programming Model

Opis supports a programming model in which distributed systems are built in three layers (Figure 1):

1. pure functions in OCaml are used to describe main steps of computation

2. Arrow combinators [20] are used to compose such steps into message handlers and introduce encapsulated local state

3. a set of higher-order *launchers* enables such message handlers to be deployed as a distributed system (exhibiting the usual non-determinism), simulated, debugged, model checked, or analyzed for performance properties.

Note that each subsequent level builds on the previous one, and captures increasingly more complex semantic domain: the first step starts with pure functions, the second one incorporates a disciplined form of local state, while the final one incorporates non-determinism of the real environment.

1.3 Comprehensive Programmer's Toolkit

In addition to the use of Arrow combinators for distributed systems, the main contributions of Opis are the simulator, debugger, model checker and performance debugger. Together with OCaml code generation facilities in Coq [3] and Isabelle [34] theorem provers, this enables a wide range of verification and analysis tools to be applied to Opis systems.

We have found the design of Opis to lead to high productivity, both when developing distributed applications in Opis, and when developing Opis infrastructure itself.

When developing applications in Opis, strong typing and pattern-matching of OCaml supported type-safe horizontal (stack of protocols) and vertical (post-processing of proto-

cols outputs) composition of protocols, and enabled defining highly parametrized protocols.

When developing Opis infrastructure, the combinator approach allowed us to cleanly reuse existing infrastructure; the use of higher-order functions made it possible to deploy and analyze the application using different tools; whereas the use of pure functions and state encapsulation made backtracking in the model checker efficient.

1.4 Contributions

We summarize the contributions of this paper as follows:

- We introduce a paradigm for developing distributed systems on top of OCaml. We use functional-reactive programming approach to naturally separate specifications of 1) pure function-building blocks, 2) functions encapsulating local states, and 3) the (implicit) non-determinism present in the network. To our knowledge, this is the first time functional-reactive programming has been applied to the domain of distributed systems.

- We present tools that we built to make the development of programs in this approach easier: simulator, debugger, model checker, and performance analyzer.

- We describe our experience in using Opis to develop two non-trivial overlay applications: a sophisticated gossip protocol, and a distributed hash table.

The rest of this paper focuses on illustrating different aspects of Opis through a detailed example of developing an overlay protocol that combines Cyclon [41] and Vicinity [42]. We then describe the combinators of Opis in detail, and describe Opis simulator, debugger, model checker, and performance analyzer. We further elaborate on our experience with Opis by describing the results of developing the Chord [40] distributed hash table in Opis.[1]

2. Opis Overview through an Example

In this section we show how we can use Opis to implement two protocols for peer-to-peer networks: Cyclon [41], a random gossip protocol, and Vicinity [42], a semantic-aware gossip protocol. We first show how to develop a *parametrized* gossip protocol in Opis. We then implement both Cyclon and Vicinity by instantiating this parametrized description with protocol-specific functions.

Moreover, we illustrate type safe composition of protocols in Opis by composing these two protocol implementations into a combined protocol that we call 'Cyclonity'. This combined protocol [42] has proved to be more efficient for searching and shown to have close to optimal convergence, faster than both Cyclon and Vicinity by themselves.

2.1 Parametrized Gossip Protocol in Opis

We start by describing a *generic* gossip protocol, which specifies the structure of gossip protocols in general. Our specification has the form of an OCaml module signature. Based on this signature, we define a functor that instantiates any particular gossip specification into a complete protocol implementation. (In general, an OCaml functor [25] is a higher-order module that takes a module as an argument and defines a new module.)

---

[1] Additional information, including the full source code for Opis, the case studies, and formal Isabelle proofs are available from `http://perso.eleves.bretagne.ens-cachan.fr/~dagand/opis/`

(a) Gossip Protocol: gossip   (b) Gossip Maintenance: maintenance

(c) View maintenance: maintain_my_view   (d) Gossip Reaction: reaction
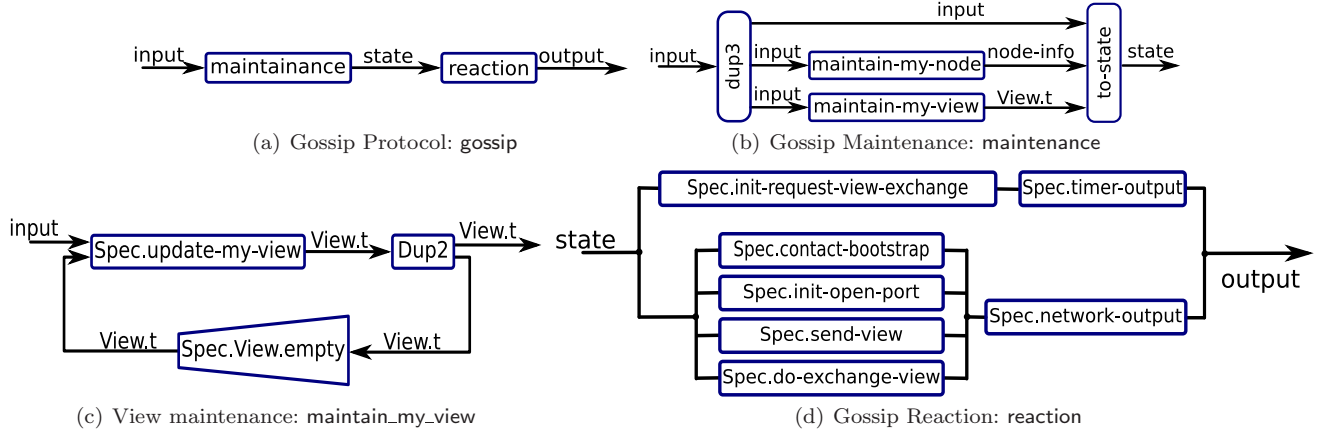
Figure 4. Generic Gossip Protocol Diagrams

```
module type GOSSIP_SPECIFICATION =
sig
    (∗ (a) ∗)
    module View : sig type t
                         val empty : t end
    (∗ (b) ∗)
    type event
    type my_node_info
    val update_my_view : ((event ∗ View.t), View.t) event_function
    val update_my_node : ((event ∗ my_node_info), my_node_info) event_function

    (∗ (c) ∗)
    type state  = (my_node_info ∗ View.t ∗ event)

    (∗ (d) ∗)
    type net_output
    type timer_output
    val send_view                  : (state , net_output) event_function
    val do_exchange_view           : (state , net_output) event_function
    val contact_bootstrap          : (state , net_output) event_function
    val init_request_view_exchange : (state , timer_output) event_function
    val init_open_port             : (state , net_output) event_function

    (∗ (e) ∗)
    type output
    val network_output : (net_output, output) event_function
    val timer_output : (timer_output, output) event_function
end
```

Figure 2. Gossip-protocol signature

2.1.1 Specifying a Gossip Protocol

Gossip protocols are based on the "world is small" observation. To find information stored in the network, this observation suggests that it suffices to know a small number of node's neighbors (peers) [23]. Each node in a gossip protocol maintains a *view* data structure storing the information about its neighbors.

Every version of the gossip protocol has its own view data structure and its own implementation of view maintenance operations. To formalize the notion of a gossip protocol, we

```
module Gossip (Spec: GOSSIP_SPECIFICATION) =struct
    let dup2 =  arr  (fun x → (x,x))
    let  maintain_my_view = dloop Spec.View.empty
                            (Spec.update_my_view >>> dup2)
    ...
end
```

Figure 3. Generic Gossip Protocol Code

therefore define the minimal signature that a view module should satisfy (Fig. 2(a)). We only require an abstract type for the view data structure and its initial, empty value.

Each node in a gossip protocol updates its view according to the information it receives. Given the view data structure, the protocol developer specifies the operation (update_my_view) to update the view based on the previous view and on the current input event. Furthermore, the developer specifies the operation (update_my_node) to store information about the node itself, such as its IP address (Fig. 2(b)). The overall state of the node (Fig. 2(c)) is therefore a tuple containing node-specific information, the neighbors view and the input event.

Finally, each node *reacts* to events, taking into account its current state. The reaction depends on the specific protocol and is parametrized by five event functions (Fig. 2(d)). Some of these functions target the Network subsystem, using the type net_output, whereas some of them target the Timer subsystem, using the type timer_output. To send them through the output wire, we need a unique output type and the corresponding injection functions network_output and timer_output (Fig. 2(e)).

2.1.2 From Specification to Event Function

To create an event function from gossip protocol specification, we simply wire the different components of the signature. Therefore, we define a functor working on any gossip-protocol specification (Fig. 3).

Figure 4(a) shows a high-level view of a gossip event function: the first task is to maintain the state of the peer, then based on this state, the peer reacts to the event. The maintenance part stores information about the peer, such as its IP address, and keeps its neighbor view up to date, as presented in Figure 4(b).

View maintenance in turn consists of enclosing maintain_my_view provided by the specification into a feedback loop, where the initial view is View.empty. To give an overview of a typical Opis code, Figure 3 shows this feedback loop; its meaning becomes clear in Section 3.1.

However, this data-flow is best represented in a graphical form, without loss of information, as shown in Figure 4(c). We adopt this graphical form in the rest of the paper. We define the maintain_my_node function similarly to maintain_my_view, using Spec.update_my_node and dloop.

The *reactive* part is more complex and is summarized in Figure 4(d).

2.2 Instantiating the Cyclon Protocol

The Cyclon protocol aims at building a random, strongly connected graph between peers. To achieve this, peers populate their neighbor view through periodical exchanges with their neighbors. First, a node randomly chooses one of its peers. Second, it selects a subset of its neighbor view, excluding the destination peer. Finally, the destination peer receives this view, updates its own view and acknowledges the exchange by answering with a subset of its own, previous view.

Our Cyclon implementation contains a view data structure that grows up to a bounded number of neighbors. Given this data structure, it is straightforward to define functions that satisfy the signature of Figure 2, producing a concise implementation of Cyclon.

2.3 Instantiating the Vicinity Protocol

Whereas the random graph eases statistical analysis of average case performance, it might poorly behave in a situation where the queries are not randomly distributed. In a file-sharing system, for instance, users are most likely to share the same kind of files, based on their interest. This interest is measured by a *semantic value*. Aggregating peers according to their habits gives more relevant and faster results: it is therefore relevant to cluster the semantically close nodes together.

Vicinity is a gossip protocol that aggregates peers based on their semantic value. The view has been designed to take this semantic value into account and its operations has been modified to make the view converge to the semantically closest node possible. Then, the events have been extended to carry the semantic values of nodes, to be able to, efficiently, fill the view. The overall design rules remain the same but we manipulate this semantic value, and it is again straightforward to instantiate the generic specification of Figure 2.

2.4 Type-Safely Composing Cyclon and Vicinity

It has been shown [42] that Vicinity does not converge to the optimal overlay. Even worse, Vicinity can break the overlay into isolated partitions, therefore weakening its resilience to failures. The solution to this problem is for the Cyclon overlay to regularly feed the Vicinity overlay with some random nodes: in a sense, we *compose* them to strengthen Vicinity.

Although the idea of composing protocols is not new [42], in a global state, handler-based system, this can lead to many subtle bugs. First, we have to ensure that handlers of both systems are independent: if handlers interfere, this can lead to performance or even behavioral bugs. Second, relying on global, mutable states increases the programming effort: although both systems worked perfectly in isolation, once composed they may badly interact during their manipulation of the shared, mutable data structures. Therefore, the behavior of the composed system cannot be deduced from the behavior of each system taken in isolation.

In our design, there is no such *hidden* communication channels: we reuse the Cyclon and Vicinity implementations developed above, and compose them by pre-processing the general input messages (g-input) into protocol-specific ones. Therefore, we are able to run both systems in isolation, by separating Cyclon and Vicinity-specific events. In the resulting system Vicinity benefits from Cyclon activ-

ity: while pre-processing Cyclon events, we will also route them through Vicinity that will then be able to enrich its own view with new, potentially interesting peers (Fig. 5). Figure 14(a) shows the improved efficiency of the resulting implementation.
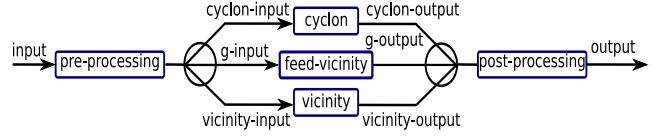
Figure 5. Composition of Cyclon and Vicinity: Cyclonity

2.5 Testing in the Large using the Opis Simulator

After type checking our code (and eliminating many errors to start with), we would next like to test our implementation. For example, in Vicinity we are concerned with the convergence of each peer toward its semantically closest neighbors: starting from any initial configuration, the nodes should progressively organize themselves in clusters of similar interest.

Therefore, we execute Vicinity in Opis network simulator and, informally, check that each peer strictly converges toward a closest semantic neighborhood. A key invariant of the semantic view is that *the semantic view should always be sorted by semantic order*, maximizing the likelihood that neighbors of a node contain the most relevant information.

Our first implementation of Vicinity contained a logical error that broke this invariant. Consequently, our buggy implementation of Vicinity was not able to converge to reasonable semantic value: this behavior appears clearly in Figure 14(b), which was drawn from the trace produced by Opis simulator.

Whereas the simulator indicates the issue, the containment of state imposed by the combinators helps us identify that the problem is caused by update_my_view. The next challenge is to find out which event breaks the invariant. We use the Opis debugger for this purpose.

2.6 Testing in the Small using the Opis Debugger

To identify the precise nature of the problem , we would like to manipulate single events as well as inspect the state of each peer. We are also interested in being able to go backwards in time, to revert a debugging decision and explore other options.

We can accomplish this by compiling Vicinity with Opis debugger support. We then simulate the execution of the complete network for 2 rounds. Every peer is initialized and ready to process network events. Hence, we check the value of the view of a peer in Figure 6(a). This view appears correctly sorted: the list of peers (second item, the IP addresses) is ordered by increasing number of semantic value (the number after each address). The first item corresponds to a vector in the semantic space: for example, it can represent the list of files shared by this peer.

However, if we play the first network event, we reach the state shown in Figure 6(b). After checking the new view we discover that it is no longer sorted. In maintain_my_view, we just have to look at the code guarded by this network event: the merge function that merges two semantic views. Because merge is a pure function, we can easily analyze it and find the bug that caused the problem.

```
(a)    > show peer 4 state my_view
       State :
        [ 0, 0, 2, 6, 8, | 10.0.0.0:0.4 ; 10.0.0.5:0.6 ]
```

```
(b)    > step peer 4 event n0
       > show peer 4 state my_view
       State :
        [ 0, 0, 2, 6, 8,
          | 10.0.0.5:0.6 ; 10.0.0.1:0.2 ; 10.0.0.0:0.4 ]
```

```
(c)    > backstep
       > show peer 4 state my_view
       State :
        [ 0, 0, 2, 6, 8, | 10.0.0.0:0.4 ; 10.0.0.5:0.6 ]
```

Figure 6. Debugging session

As an illustration, note that we could still play the second network event at this point: we simply go backward in time (Fig. 6(c)) and arrive exactly at the previous state.

2.7 Finding Errors using the Opis Model Checker

One of the essential Cyclon requirements is that it builds a strongly connected graph among peers. We use the Opis model-checker to test the following property: *either there exists a peer with an empty view, or the graph given by the views of all nodes is strongly connected.* The first part of the property captures the initialization stage, where nodes build their view. The second part states that the overlay will not be fragmented into several independent overlays.

Based on this property we have model-checked our implementation of Cyclon for a network of four peers, each maintaining a view containing one neighbor. There is a reason to expect that maintaining the global connectivity with only *one* neighbor is more difficult than with several neighbors: as soon as this unique link is wrong, the property will be violated.

After running the Opis model checker on our first implementation, in 0.01 seconds we obtain the output shown in Figure 7. The output points to a subtle bug in the view management of our implementation. The intention is to ensure the strong connectivity through the exchange of views, but, in our implementation, we also chose to insert the sender of any message in the view. Therefore, peer 1 considers peer 2 as a neighbor after the first message. Then, peer 2 inserts peer 1 in its view. Later, peer 3 contacts peer 0 that inserts peer 3 in its view, and answers to peer 3 that considers peer 0 as a neighbor. Finally, we end up with two disjoint graphs respectively composed by peers $(1, 2)$ and $(0, 3)$.

Avoiding the insertion of message senders solved this problem: the view is only populated by the view exchanges, as prescribed by the protocol. After this correction, we can run the model checker, which, in 0.5 seconds, explores the complete state-space. This provides a fairly high confidence that Cyclon ensures the strong-connectivity of the overlay.

```
[] Setup the network... Done.
Stack [3] : Host 3 <- Network_in: UDP_in(10.0.0.0:1000,
                My_view_is ( 10.0.0.0 , [ 10.0.0.3 ]))
Stack [2] : Host 0 <- Network_in: UDP_in(10.0.0.3:1000,
                                Join ( 10.0.0.3 ))
Stack [1] : Host 2 <- Network_in: UDP_in(10.0.0.1:1000,
                My_view_is ( 10.0.0.1 , [ 10.0.0.2 ]))
Stack [0] : Host 1 <- Network_in: UDP_in(10.0.0.2:1000,
                                Join ( 10.0.0.2 ))
Safety broken
Number of explored states: 5
```

Figure 7. Cyclon Model-checking

2.8 Formal Proofs of Event Function Properties

The fact that core Opis functionality is expressed using pure functions enables us to use theorem provers to prove properties of such functions that hold for all executions. As a simple illustration of this capability, we have developed and verified the Cyclon view data structure using the Isabelle interactive theorem prover. Figure 8 illustrates how straightforward it is to prove in Isabelle that the size of the view is bounded. The figure defines the pure function merge for updating the view, and then shows that its execution always returns (as the resulting view) a list of size bounded by the parameter n of the protocol. The actual proof is performed automatically by the Isabelle simplifier. Such boundedness properties are important to ensure good event function performance and (along with the garbage collector) absence of certain memory leaks. Note that even a language with type safety and garbage collection could contain a 'semantic memory leak' where views grow arbitrarily with the size of the system. The OCaml code generated by Isabelle can be integrated naturally with the manually developed code.

Proving termination. A specialized use of interactive provers and their code generation facility is ensuring termination of Opis event functions. Termination of a protocol depends on the 'building blocks', pure functions given as arguments to the 'arr' combinators. We derive the overall termination proofs from two facts. The first is the property of Arrow combinators used to execute Opis systems: *given an event function input, the computation of the output will execute each building block a finite number of times.* The second is the fact that Isabelle always proves the termination of definitions of total functions, so writing a function definition in Isabelle ensures its termination. We wrote all building blocks of Cyclon in Isabelle and exported the code to OCaml. The Isabelle and the generated OCaml code were syntactically almost identical and the termination was proved automatically by Isabelle. This ensured that the execution of a reactive function of a Cyclon node on any single event terminates, and can be used as a starting point for establishing stronger functional correctness properties.

```
theory Bounded_list imports Main begin
constdefs
  merge :: "nat => 'a :: linorder  list  => 'a  list  => 'a  list "
  "merge n xs ys ≡ take n ( sort ( xs @ ys ) )"
  insert  ::  "nat => 'a :: linorder  => 'a  list  => 'a  list "
  " insert  n x  xs ≡ take n ( insort  x xs)"
  empty :: "'a :: linorder   list "
  "empty ≡ []"

theorem merge_bounded_size:
  "length  (merge n xs ys) ≤ n"
apply(simp only: merge_def, simp) done

theorem insert_bounded_size :
  "length ( insert  n x  xs) ≤ n"
apply(simp only:  insert_def , simp) done

export_code "merge" " insert "  "empty" in "OCaml" file "bounded_list .ml"
end
```

Figure 8. Isabelle implementation of a bounded list

2.9 Performance Debugging

In addition to verifying correctness of the system using simulator, debugger, model checker, and theorem prover, Opis also supports profiling and performance debugging by indicating the performance of node's reactive functions. The performance of reactive functions, along with the knowledge of

	function name	time total	# executions	time average	time std. deviation	complexity estimate
(a) first version	update_my_view	1.09	21048	0.00005210	0.00006837	O(n^2)
	post_treatment	0.18	105240	0.00000172	0.00000661	O(n)
	to_network	0.13	84192	0.00000165	0.00000464	O(n)
	function name	time total	# executions	time average	time std. deviation	complexity estimate
(b) w/ precomputation	update_my_view	0.48	21048	0.00002314	0.00003012	O(n)
	post_treatment	0.18	105240	0.00000172	0.00000637	O(n)
	to_network	0.13	84192	0.00000165	0.00000388	O(n)

Figure 9. Output of Opis performance debugger on selected functions of Vicinity protocol (times are in seconds). The debugger correctly classified the initial version of update_my_view as a quadratic function in terms of time complexity.

high-level protocol behavior, determines the efficiency of the overall protocol. By leveraging the Arrow abstraction, Opis profiler indicates the performance of components of nodes reactive functions both in terms of 1) the concrete average of running time, and in terms of 2) an empirical estimate of function's computational complexity. Large concrete running time along with asymptotically high complexity suggest functions that may require algorithmic improvements.

After we applied Opis profiler to our first implementation of Vicinity, we obtained the results sketched in Figure 9(a). Realizing that the update_my_view function accounts for most of the processing time, we examined this function in detail and found that the semantic values (used to sort the list of neighbors) were being recomputed unnecessarily. By pre-computing these values we arrived at a more efficient implementation which led to the improved results in Figure 9(b). Note that the analyzer automatically computes the expressions $O(n)$ and $O(n^2)$, and correctly detects the asymptotic improvement in the running time.

Summary of the Example. Overall, the total development time for Cyclonity (including the generic gossip protocol) was 13 hours, and resulted in 775 lines of code. We believe that the programming language, programming model, and a palette of analysis tools were essential in making the development of a non-trivial overlay protocol in Opis this effective.

3. Opis Design

Our goal when designing Opis was to produce a system in which: i) the programmer can informally and formally reason about the code, ii) the type system catches a large number of errors, and iii) the programmer is given a range of tools for analysis and debugging of code functionality and performance. The rest of this section shows how Opis meets these goals.

3.1 Arrows for Composable Event Functions

Many of the benefits of Opis are a result of our decision to express event functions using the concept of *Arrows* [20, 35, 36]. Arrows are a generalization of Monads and abstract the notion of computation taking inputs of a given type and producing outputs of another type. We represent an event function (an instance of Arrow) graphically as in Figure 10(a), and denote it af.

The event_function type is abstract; to manipulate values of type event_function we use Arrow combinators. We use a notation inspired by [35] to represent the combinators. The first, essential, combinator is *arr*, which takes a pure function and turns it into an event function. Figure 10(b) shows the *arr* signature and graphical representation. We next define the *composition* combinator (Figure 10(c)) that lets us compose computations. To allow for multiple inputs,

we define the combinator *first* (Figure 10(d)). To enable choices to be made, we define the sum type either :

type (α,β) either = Left of α | Right of β

We use either to express conditionals: the *choice* combinator (Figure 10(e)) applies the first computation if the input is tagged Left and applies the second computation otherwise. Then, n-ary choices can be implemented by a cascade of binary choice combinators.

The combinators so far are restricted to memory-less computations. To encode computations with memory we define the *loop* combinator (Figure 10(f)) that feeds an output back to the input of the event function. Then, we define the *delay* combinator (Figure 10(g)) that, given an initial value always outputs the value of the previous input, thus remembering the current input. Together, *loop* and *delay* let us express memories. We use this construct to encapsulate protocol state.

To permit generation of a sequence of events based on a single input, we define the *mconcat* combinator in Figure 10(h): the list of outputs generated by a computation will be transformed, at the wire level, in a sequence of impulse-events. The event functions that follow will successively process these events, individually. The *mappend* combinator (Figure 10(i)) is a special case of *mconcat*: for one single input event, *mconcat* processes it through two event functions and transfers the results at the wire level, in two successive events with no particular order.

3.2 Deployment and Analysis using Launchers

Opis aims to provide to the programmer a complete set of tools to develop, simulate, debug, model-check and deploy an application. To achieve this goal, Opis uses a set of *launchers*. A launcher is a function satisfying the signature

```
val launcher : (in, out) event_function -> ()
```

The in and out types are described in Section 4.2. Once we have built an event function of this type, we are able to deploy it transparently on any launcher.

The role of a launcher is to provide the event function with events and to execute its outputs: it acts as an *interface* to the world. Our definition of a launcher is sufficiently large to encompass many approaches to deploy and analyze event functions. We next describe the current set of launchers available in Opis; additional launchers could easily be incorporated.

3.2.1 Runtime Launcher

First of all, Opis provides a *runtime launcher* that interprets the event function in real execution. The network, timer and user inputs are directly provided to the event function whereas its outputs are interpreted in terms of network actions (opening a connection, sending a message, etc.), timer commands (setting up a new timer, disabling a previous
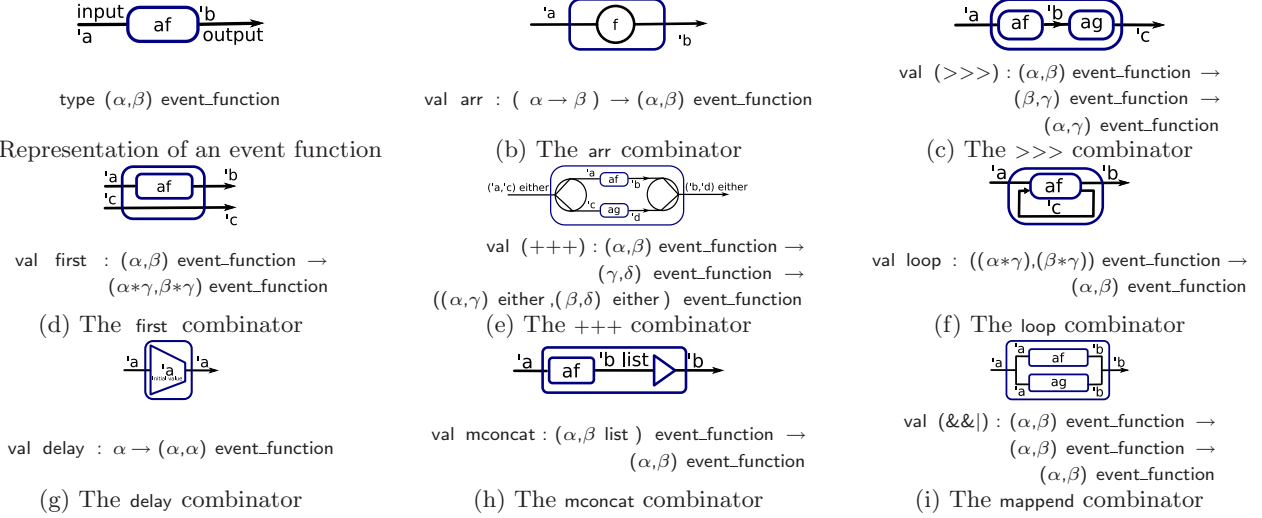
(a) Representation of an event function

type (α,β) event_function

(b) The arr combinator

val arr : ($\alpha \rightarrow \beta$) $\rightarrow$ (α,β) event_function

(c) The $>>>$ combinator

val $(>>>)$: (α,β) event_function $\rightarrow$
(β,γ) event_function $\rightarrow$
(α,γ) event_function

(d) The first combinator

val first : (α,β) event_function $\rightarrow$
$(\alpha*\gamma,\beta*\gamma)$ event_function

(e) The $+++$ combinator

val $(+++)$: (α,β) event_function $\rightarrow$
(γ,δ) event_function $\rightarrow$
$((\alpha,\gamma)$ either $,(\beta,\delta)$ either $)$ event_function

(f) The loop combinator

val loop : $((\alpha*\gamma),(\beta*\gamma))$ event_function $\rightarrow$
(α,β) event_function

(g) The delay combinator

val delay : $\alpha \rightarrow (\alpha,\alpha)$ event_function

(h) The mconcat combinator

val mconcat : $(\alpha,\beta$ list $)$ event_function $\rightarrow$
(α,β) event_function

(i) The mappend combinator

val $(\&\&|)$: (α,β) event_function $\rightarrow$
(α,β) event_function $\rightarrow$
(α,β) event_function

Figure 10. The Arrow Combinators

timer, etc.), or user interactions. Using this launcher, we can deploy any event function on real, live systems. Section 5 shows that the deployed systems have good performance.

3.2.2 Simulator Launcher

Although the use of a safe programming language and programming model in Opis eases the development and eliminates a class of errors, a number of other issues may remain in the distributed system after compilation. These include logical errors, protocol misinterpretation, and emergent behaviors arising in systems with large number of nodes. The use of a software debugger, such as ocamldebug, is not effective at that point: bugs might appear only after a large number of interactions among peers that are hard to reproduce in a standard software debugger. This is why our approach to initial testing is to use the Opis *simulator*. The simulator can realistically and efficiently simulate large distributed systems on a single machine, exposing a sample of message interleavings. In our experience, Opis simulator substantially speeds up the edit-compile-test loop.

3.2.3 Debugging Launcher

To pinpoint the precise source of identified defects, we need a more fine-grained tool than the simulator. For this purpose we use the *debugger* launcher. Our replay-debugger is built on top of a modified simulator and allows the programmer to: i) show the state of the whole network, ii) inspect the events pending for some peers, iii) inspect the states maintained by some peers, iv) take forward steps, either at the entire network-level or only at a subset of its peers, and v) take backward steps and rollback previous actions. The debugger benefits from our definition of event function and its suitability for backtracking algorithms.

3.2.4 Model Checking Launcher

Due to concurrency in the distributed system, the sample of system behaviors exposed by the simulator or the debugger is limited. To obtain higher confidence in the correctness of implementation and find subtle bugs, we include a *model checking* launcher in Opis. The model checker can efficiently explore the specified finite portion of the application state space, considering different non-deterministic choices and checking the given property in the explored states. Our current implementation supports checks of local and system-wide safety properties and supports partial-order reduction for node-local properties.

3.3 Using Formally Proved Code

The fact that the event functions in Opis must be built from pure functions provides well-known benefits of purely functional programming. This includes informal reasoning (knowing that computations are deterministic, using equational reasoning). Moreover, given the simplicity of semantics of pure functions, we can develop and verify the critical functions of a protocol in a theorem prover such as Coq or Isabelle. We can then use code generation facilities of these provers to generate executable OCaml code that is called directly from within event functions of the Opis protocol implementation. Such proof-assisted programming has already proved its feasibility with large scale projects such as compilers [26]. Verifying distributed systems with theorem provers also has a long and fruitful history, as in the formalization of the Ensemble system [24, 17].

Unlike these complete formalization systems, Opis provides the developer with a *pay as you go* approach where the programmer focuses on the critical functions and proves their correctness. Because the Arrow combinators are fixed and their algebraic properties can be proved once and for all, most of the work is on reasoning about protocol-specific pure functions for which interactive provers are quite effective [34].

3.4 Performance Analysis Support

Opis also provides support for performance analysis, which we found very useful in practice. Although the use of a model-checker and a theorem prover offers guarantees about the correctness of an implementation, it does not address the performance-related issues (at least not with the usual embedding of pure functions into higher-order logic). Profiling tools such ocamlprof are too fine-grained to expose any meaningful information in our context: instead of working at the OCaml-function level we are operating at the event-function level. Therefore, given an event function, we are interested in the pure functions that consume most of

the processing time or that are biggest memory consumers. Thanks to the Arrow combinators, we easily retrieve three fundamental metrics while executing a pure-function building block: 1) the size of its input, and 2) its processing time, and 3) the size of the computed output. Our profiler uses this information to help the developer find the performance bottlenecks. Specifically, the profiler tries to match the data sets to logarithmic, linear, quadratic, and cubic functions, and reports the computational complexity that fits the data best. The programmer can then use this information to identify unexpected behaviors. This form of function profiling and characterization was made possible thanks to state encapsulation using Arrow combinators, and would be more difficult to obtain in a system where functions can access any global data.

4. Implementation Highlights

We next highlight some aspects of Opis implementation and substantiate the claim that Opis approach is not only easy to use but also easy to implement and extend.

4.1 The Event Function Library

Our implementation of Arrow combinators is based on OCaml Objects, which is in contrast to the existing implementation of reactive, arrow-based systems in call-by-need functional languages [29, 19]. The use of objects is motivated by the need to be able to execute, debug, simulate, model-check, or profile an event function without modification. Hence, objects allow us to, transparently, overload the Arrow combinators to support all these possible interpretations.

We define an event function as an object providing a call method, used internally to execute the event function, and a copy method, used to clone the state of an event function. The raison d'être of copy comes from the side-effectful definition of the loop combinator below. Being able to clone an event function allows us to regain referential transparency when needed, such as during model-checking for instance. The signature of this object is the following:

```
class type [α,β] event_function =
object
  method call : α → β
  method copy : (α,β) event_function
end
```

To build an event function out of a pure function, we simply define an object that is built with the given function. A call is translated into this function and the copy simply duplicates the object:

```
class [α,β] event_function (f : α → β) =
object ( self )
  method call = f
  method copy = self
end
```

The remaining combinators follow almost the same design rule, the only difference being the computation in the call method. Therefore, we simply describe the *compose* combinator. This combinator accepts two event functions af and ag, which are both event_function objects. The call method applies the composition of both functions by first computing af of x, then computing ag of the previous result. The copy method recursively copies both sub-objects and returns a fresh, duplicated object:

```
let (>>>) af ag : (α,β) event_function =
object
```

```
  val af : (α,γ) event_function  =  af
  val ag : (γ,β) event_function  =  ag
  method call x = ag#call (af#call x)
  method copy = {< af = af#copy; ag = ag#copy >}
end
```

Implementing a loop combinator in a call-by-value language is challenging: the result of a computation should already be defined as an input to this very same computation. Moreover, loop is, by definition, not *total* and, the developer would always need to verify that the loop calls a delay on its feedback wire. To remove this burden and simplify the implementation, we only provide a dloop combinator, which takes an initial state value and is total. To leverage standard laws for reasoning about Arrows [35] we define dloop by

```
dloop x af ≡ loop ( second ( delay x ) >>> af )
```

Our actual implementation of dloop is the following:

```
let dloop init rf : ('b,'c) event_function =
object
  val mutable state  =  init
  val rf : ('a*'b,'a*'c) event_function  =  rf
  method call x =
    let state ', y = rf#call (state, x) in
      state ← state '; y
  method copy = {< rf = rf#copy >}
end
```

4.2 The Launcher Architecture

We next elaborate the in and out types mentioned in Section 3.2. The in type can be either a Network event, a Timer event or a User event. The out type that can be a command destined for the Network, the Timer system, or the User. Hence the following definitions:

```
type in = NetIn of networkInput
        | UserIn of userInput
        | TimerIn of timerInput
type out = NetworkOut of networkOutput
        | UserOut of userOutput
        | TimerOut of timerOutput
        | DoNothing
```

Network events convey a typed payload that can be transported using either TCP (quasi-reliable channel) or UDP (unreliable channel), leading to the following definition of networkInput and networkOutput:

```
type networkInput =
  | NetReceive of (peerAddr * portNumber *transport * payload)
  | Sent of (peerAddr * portNumber *transport * payload)
  | SendFailed of (peerAddr * portNumber *transport * payload)
  | ListeningOn of (portNumber *transport)
  | ConnectionClosed of (portNumber *transport)
type networkOutput =
  | Send of (peerAddr * portNumber *transport * payload)
  | ListenOn of (portNumber *transport)
  | CloseConnection of (portNumber *transport)
  | NetworkDoNothing
```

Note that the payload of network messages is typed. This ensures that the communication channels used by our system are typed as well. Therefore, the type system guarantees that network events will not generate run-time errors, especially during marshaling and unmarshaling.

Timers are either periodic (occur every t seconds) or unique (occur once), so type timerType = Periodic | Unique. A timer is defined by the following record:

```
type timer = { id : id; clock : time; delay : time;
               timerType : timerType; command : timerCommand }
```

where id identifies the timer, to be able to cancel it later; clock is the absolute time at which the timer must be fired, delay is used when re-scheduling a periodic timer, the clock being set to the current absolute time plus the delay; and command is transmitted to the event function when the timer fires. A timerInput and timerOutput are defined by:

```
type timerInput  =  TimeOut of timer
                 |  TimerReady of timer | TimerKilled  of  timer
type timerOutput = LaunchTimer of timer | KillTimer of id
                 |  TimerDoNothing
```

Timers are set up or canceled using LaunchTimer and KillTimer. These commands are acknowledged by TimerReady and TimerKilled. When a timer times out, it raises a TimeOut event.

4.2.1 Runtime Implementation

Opis provides an interface to the networked world using the Runtime launcher, that keeps track of open network sockets as well as of pending timers. In complex overlays, the number of open connections and pending timers can be arbitrarily large. However, the user interface would not tolerate slow-down or freeze while the launcher does its bookkeeping. Therefore, the runtime relies heavily on threads, based on the Fran model [10]. We run user interface, network interactions and timers isolated in threads of their own. The *Event* module of OCaml [37] ensures linearizability of these concurrent executions.

4.2.2 Model Checking with Encapsulated State

Several properties of Opis make it a platform of choice for efficient model-checking techniques. First, clear containment of states, imposed by the combinators, helps efficiently define and compute the state of an event function. Second, given that Opis is purely functional and is written in a functional language, backtracking is efficient and simple to implement. Finally, because the event functions are first-class citizens in the host language, we can easily manipulate them in the model checking launcher.

The properties of Opis enabled us to developed an abstraction on top of the network that lets us manipulate a complete network of peers as an automaton. Thus, we can efficiently compute the *signature* corresponding to a unique network state. Second, given a network, we can compute all activated *transitions* and decide to take any of them. Using this abstraction, an implementation of depth-first safety checking is possible using standard algorithms [18].

The expressiveness offered by the automaton abstraction also allowed us to easily implement more advanced model-checking techniques, such as *partial-order reduction (POR)* [13, 11]. By restricting the exploration to *dependent* transitions and by removing many irrelevant interleavings of events, POR was able to avoid the exponential blowup during exhaustive search in some protocols we developed (see Section 5.2.4).

Opis model checker also explores possible firings of timers, by maintaining the notion of 'current time'. At every model-checking step, it either 1) executes the earliest timer event in the set of pending timers, or 2) does not fire any timer. This approach enabled us, for example, to explore the complete state-space of Vicinity (Section 2).

4.3 Performance Analyzer Implementation

Opis performance analyzer identifies performance issues in reactive functions by collecting information about running times of pure-function building blocks in an Opis system, as well as the sizes of their inputs. The analyzer then identifies functions with largest average running times and displays them to the user. Moreover, the analyzer estimates function complexity by examining the measured input sizes and the corresponding running time for the invocations of these functions. To estimate the asymptotic complexity, the analyzer considers a set of common mathematical functions including log, linear, quadratic, and exponential. For each function it computes the sum-of-squares error and displays to the user the function with the least error.

5. Evaluation

Building on the overview of Section 2, we next evaluate Opis in terms of programmer productivity, efficiency of developed systems, and efficiency of some of the analysis tools in Opis.

5.1 Scope of Use and Programmer Productivity

Peer-to-peer overlays are roughly divided in two families: the structured ones, embodied by Chord [40], and the unstructured ones, embodied by Cyclon (the subject of our example in Section 2). To demonstrate that Opis is not restricted to unstructured, highly-reactive systems, we have implemented the Chord protocol. Our implementation of Chord highlights the correctness points as well as the conciseness of the Arrow-based formulation: whereas P2 relies on 47 logic rules and Mace implementation is expressed in 320 statements, our implementation is built upon 30 event functions, with an event function being composed of 10 lines of code on average.

The map of our Chord event function is shown in Figure 11. To improve legibility, we do not draw the body of the *React* event function. This function is a mappend on a set of functions that each react to a specific event, while ignoring the others. This can be viewed as a form of *event-matching*.

These two implementations show that Opis can easily express both categories of overlays: structured and unstructured. Moreover, we have tried to quantify the expressiveness offered by Opis in Figure 12. The number of lines of code corresponds to the arrow-ized code as well as the data structures needed by the protocol. In comparison, the Opis toolkit itself is implemented in 5'000 lines of code.

Protocol	LOC	# Event functions	Developer time
Ping-pong	134	12	2 hours
Generic Gossip	54	4	1 hour
Cyclon	265	13	5 hours
Vicinity	365	13	6 hours
Cyclonity	91	4 + 13 + 13	1 hour
Chord	812	30	2 days

Figure 12. Expressiveness measures

5.2 Experimental Results

We next present additional experience that shows the effectiveness of Opis in developing distributed applications.

5.2.1 Event Functions Performance

For satisfactory performance, we must ensure that the performance overhead of this design remains low. To measure it, we instrumented our simulator to measure event function execution time. We measure the most complex event function, the one in Chord.

The results are shown Figure 13(a). In this experiment, the overlay is composed of 1000 nodes, with one node joining
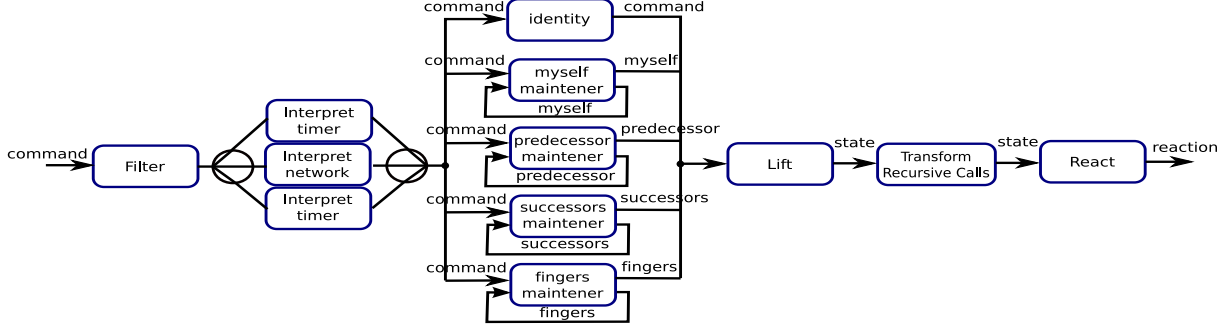
Figure 11. The Chord event function

every round. The first step corresponds to the Join phase, when a lot of messages are sent through the network, while the following steps correspond to the processing time under normal load. Given the latencies encountered in wide area networks (tens, even hundreds of milliseconds), we consider this processing speed to be sufficiently fast.

5.2.2 Gossip Protocol Performance

One of the most important features of semantic-aware gossip protocols is convergence speed toward semantically closest peers. We therefore use the Opis simulator to examine average node semantic value in Vicinity and Cyclonity (combination of Vicinity and Cyclon) protocols we highlighted in our example (Section 2). The average semantic value is the average ratio of identical files between a node and its neighborhood, and it captures the ability of the protocol to bring peers with similar interests closer. This property translates into higher query success rate and lower network overhead for future searches.

We see in Figure 14(a) that Cyclonity exhibits superior performance relative to standalone Vicinity. Figure 14(b) depicts the difficulty our implementation of Vicinity had before a bug fix. In these simulations, each node owned 8 files, randomly chosen among a total of 16 possible files.

5.2.3 Chord Performance

Finally, we demonstrate that using Opis to implement overlays does not reduce their performance in real, live deployment. We show that in a realistic setting, Opis Chord must behave no worse than its previously developed imperative counterparts.

We have deployed our Chord implementation on the ModelNet network emulator. ModelNet allows us to realistically emulate 1000 hosts that are running live code over an Internet-like INET [7] topology. The ModelNet emulator we used is composed of 5 dual-core Intel Xeon 5140 at 2.33GHz, with one machine acting as a ModelNet core. Each machine is provisioned with 2 GB of RAM and runs GNU/Linux 2.6.17. These machines are linked with a full-rate gigabit Ethernet switch.

We have focused our experiment on the convergence speed of finger tables. Therefore, we are able to compare Opis Chord with the previously published performance results for both MIT *lsd* (a hand-tuned C++ implementation) and MACEDON Chord (a highly-optimized implementation in the MACEDON [38] overlay programming language) that were also obtained using ModelNet over a 1000-participant INET topology. Figure 13(b) shows the convergence speed for the three implementations, running live on 1000 hosts.

During the bootstrapping phase of the experiment, Opis Chord exhibits good performance. This highlights the highly-reactive behavior of our overlay. However, over time, the number of correct finger entries in Opis Chord grows more slowly than Macedon or LSD Chord. We attribute this to the fact that our implementation has not been optimized for fast convergence speed on long term, but for quickly reacting to dynamic network conditions.

To conclude, we believe that the results offered by our Chord implementation are fully satisfactory. While mainly targeting simplicity and reliability, our implementation is able to compete with highly optimized code. Moreover, the overhead of the Arrow abstraction is negligible while all its benefits are validated: ease of development, ease of debugging, efficiency, and a greater trust in the code.

5.2.4 Model-Checker Efficiency

To measure the efficiency of our model-checker, we have used it to explore the complete state-space of a simple Ping-pong protocol. The ping-pong setup consists of n peers which periodically send a *ping* message to a central peer. The central peer responds with a *pong* to every *ping*. The execution terminates when all peers have sent 5 *ping*s that have all been answered. This application exhibits a lot of interleaving and, in this sense, is representative of most distributed systems.

Although simple, this system allows us to compare the exhaustive search strategy with the partial-order one for relatively large number of peers. Figure 15(a) shows the number of explored states. Figure 15(b) shows the duration of the model-checking using Opis model checker. Note that we use the log scale for measuring model checking behavior. The results in the presence of partial-order reduction are promising and the technique appears essential for this class of distributed systems. Note that the size of the state space is kept relatively small for a software system, which is thanks to the fact that node behavior is given by large-granularity, deterministic reactive functions.

Overall, we have found Opis model checker to be a very effective tool for finding subtle errors arising due to the asynchronous nature of distributed systems.

6. Related Work

The first reactive systems have been built using synchronous data-flow languages such as Signal [12], Lustre [15], or Esterel [2]. These languages are used to design human-machine interfaces as well as communication protocols, and have a long and successful history in the domain of safety-critical systems.
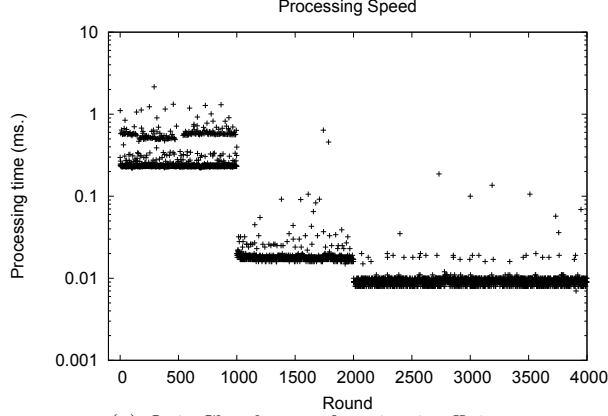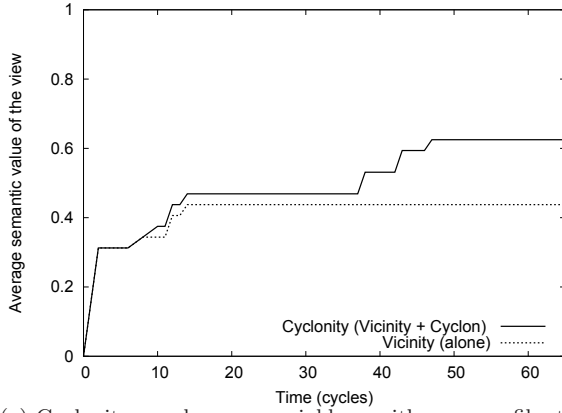
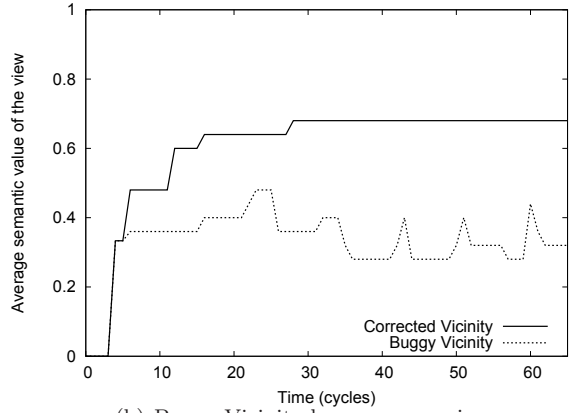(a) Opis Chord event function is efficient
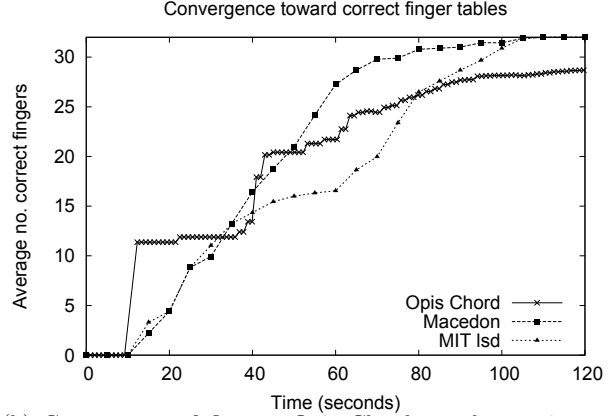
(b) Convergence of fingers: Opis Chord matches native and MACEDON Chord (live)

Figure 13. Chord performance

(a) Cyclonity peer has more neighbors with common files than in Vicinity

(b) Buggy Vicinity has a narrow view

**Figure 14.** Semantic convergence (simulation)

The introduction of reactive programming into functional programming gave birth to many libraries such as Fudget [6] for graphical user interface. This concept has also been applied to robot programming with tools like Frob [14]. Finally, all these implementations have inspired Yampa [8], which intends to be their 'arrow-ized' generalization. Like Opis, these systems are not dedicated programming languages but embedded into a host functional language.

One of the recent overlay programming and deployment efforts is P2 [31]. In P2, the programmer uses the Over-Log declarative language to specify the protocol in abstract terms, without referring to low-level details such as network messages. Nevertheless, thanks to the abstractions provided by Opis, we believe that reasoning about Opis code is no more difficult than reasoning about P2 OverLog statements. Moreover, working at that level of abstraction of OverLog implies performance compromises that make P2 effective primarily for prototyping. In contrast, Opis semantics preserves enough of the physical constraints of a distributed system to enable the developer to hand-tune the protocol.

MACEDON [38] / Mace [22] framework is based on finite-state machines that builds a domain specific language on top of C++. Mace (a reimplementation of MACEDON) leverages the object-oriented features of C++ to manage finite-state machines through interfaces, while exporting standard handlers. Mace achieves very good performance and provides an intuitive approach to overlay programming. Model-checking techniques have been directly applied to Mace code, and Mace model checker found liveness bugs in several overlays. The handler-based design and the "code once, run many ways" philosophy has also been adopted by the WiDS [28] system. One advantage of Opis relative to these efforts is memory and type safety, and its basis is the embedding of Opis into OCaml. Mace and WiDS both have additional drawbacks: hidden data-flow and unobservable user data structures. Another Opis advantage comes from the abstraction offered by Arrows, which allowed us to implement a superior performance debugging tool.

Another successful handler-based system is Ensemble [30] that builds distributed systems by stacking simpler protocols together. Whereas Opis allows to compose systems both vertically (stacking) and horizontally (post-processing), Ensemble demonstrated that OCaml can be used for distributed systems programming.

(a) Total number of explored states. (b) Total model-checking time
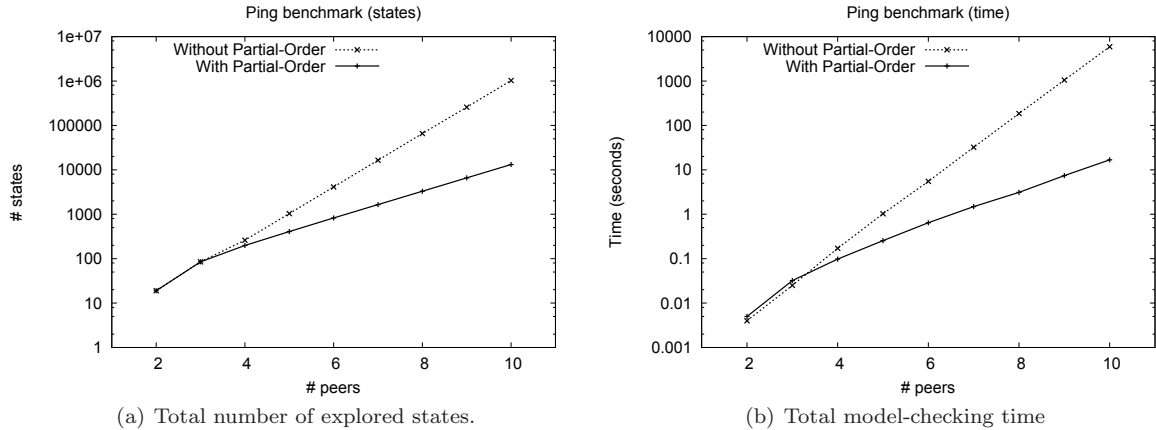
Figure 15. The illustration of partial-order reduction on the Ping-Pong benchmark.

While one could be concerned about the efficiency of OCaml for network programming, Melange [32] already proved that OCaml can be even faster than hand-written, C implementations of complex protocols. Opis can be naturally connected to Melange and thereby benefit from this speed up.

Acute [39] is a programming language for distributed computing that supports migration of executable code. In contrast, Opis uses Arrow combinators to advances techniques for implementing reliable large-scale distributed systems such as transport services.

Erlang [1] demonstrates the suitability of strict, functional programming concepts for programming concurrent and distributed systems. Erlang uses the Actor model to build asynchronous message-passing systems. Erlang uses strong dynamic typing discipline, allowing users to send arbitrary values over the channels. In contrast to Erlang, the data flow of events in Opis is explicit, which clearly identifies the path taken by any given event. We are not aware of a comprehensive set of tools (that include theorem provers and performance analyzers) that provide the benefits of Opis to Erlang programmers.

The implementation of a TCP stack in Standard ML [4] has shown that network programming is, first, possible in a high-level language and, second, that it enhances safety. This is thanks to functors for modularity and a strong type-system that catches many errors. The Eel library [9] extends the event-driven programming model with a set of tools to ease the task of reading, writing and verifying handler-based code. By working on imperative, side-effectful languages, eel has to resort to complex analysis techniques whereas Opis completely avoids some of this effort.

Recent work [27] demonstrates that efficient, lightweight network interfaces can be written in functional languages. The Network interface of Opis could easily benefit from these performance-related efforts, because its implementation is orthogonal to the event processing part.

The Flux [5] system defines a data-flow oriented language to compose C/C++ components into networked application. Whereas Flux features a type-checker as well as a simulator, the use of C/C++ makes the implementation of debugging or model-checking tools more difficult. Opis is able to do performance prediction and it benefits from the widely used, well-tested OCaml type-checker.

The Singularity project [21] has also addressed the problem of safely implementing message-passing based systems. It introduces the concept of *exchangeable types* that capture values transmitted through communication channels. In Opis, network channels must be typeable by OCaml type-system, providing similar benefits. Opis does not support Singularity's *channel contracts*. Instead, Opis provides the ability to apply an explicit-state model checker to the entire distributed system, which is less scalable yet more precise approach.

7. Conclusion

We have presented Opis, an approach for developing distributed systems in OCaml. Opis supports a programming model for distributed systems where node behavior is given by event functions built from simple pure functions using arrow combinators. The development based on combinator library means that Opis benefits from the type safety and memory safety of OCaml, along with the well-understood composition mechanisms such as higher-order functions and parametrized module system.

Opis complements this solid programming language base with a number of tools (implemented as higher-order 'launcher' functions) for analyzing the systems to uncover emergent semantic and performance behaviors. Specifically, given node behavior description, Opis launchers enable the developer to deploy, simulate, debug, model check, and analyze performance of the distributed system.

Using the Opis toolkit, we have developed an advanced gossip overlay protocol and a distributed hash table. Compositionality allowed us to factor out common portions of protocols, reducing the development time and code base size. We found the OCaml type checker to prevent a large number of errors, and found Opis tools extremely useful in uncovering subtle semantic and performance problems. Our experience with generating code from Isabelle was also encouraging. The total development time was short compared to our experience with related systems, the system description was very compact, and the resulting systems have good performance. This suggests that Opis is a promising and flexible approach for reliable distributed system development.

8. Acknowledgements

References

[1] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, 2003.

[2] G. Berry. *The Foundations of Esterel*. MIT Press, 2000.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[4] E. Biagioni. A structured TCP in standard ML. *SIGCOMM Comput. Commun. Rev.*, 24(4):36–45, 1994.

[5] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: a language for programming high-performance servers. In *USENIX ATC*, page 13, 2006.

[6] M. Carlsson and T. Hallgren. FUDGETS - A graphical user interface in a lazy functional language. In *FPCA*, 1993.

[7] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*, June 2002.

[8] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Haskell Workshop*, 2003.

[9] R. Cunningham and E. Kohler. Making events less slippery with eel. In *HOTOS*, page 3, 2005.

[10] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, 1997.

[11] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[12] T. Gautier, P. L. Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In *FPCA*, 1987.

[13] P. Godefroid and P. Wolper. A partial approach to model checking. In *LICS*, 1991.

[14] G. Hager and J. Peterson. Frob: A transformational approach to the design of robot software. In *ISRR*, 1999.

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

[16] M. Hayden. Distributed communication in ml. *J. Funct. Program.*, 10(1), 2000.

[17] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In *TACAS*, 1999.

[18] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[19] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *LNCS*, 2002.

[20] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37, 2000.

[21] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Op. Sys. Review*, 41(2):37–49, 2007.

[22] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007.

[23] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *STOC*, 2000.

[24] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In *CADE-15*, 1998.

[25] X. Leroy. A modular module system. *J. Funct. Program.*, 10(3), 2000.

[26] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.

[27] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199, June 2007.

[28] S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo. WiDS: an integrated toolkit for distributed system development. In *HOTOS*, 2005.

[29] H. Liu and P. Hudak. Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.*, 193:29–45, 2007.

[30] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *SOSP*, 1999.

[31] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5), 2005.

[32] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: creating a "functional" internet. *SIGOPS Oper. Syst. Rev.*, 41(3), 2007.

[33] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[34] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[35] R. Paterson. A new notation for arrows. In *ICFP*, 2001.

[36] R. Paterson. *The Fun of Programming*. Palgrave, 2003.

[37] J. H. Reppy. Higher–Order Concurrency. Technical Report TR92-1852, Cornell Univ, Ithaca, NY, 1992.

[38] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. MACEDON: methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI*, 2004.

[39] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: high-level programming language design for distributed computation. In *ICFP '05*. ACM, 2005.

[40] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[41] S. Voulgaris, D. Gavidia, and M. Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13, 2005.

[42] S. Voulgaris and M. van Steen. Epidemic-style management of semantic overlays for content-based searching. In *EuroPar*, 2005.

[43] MLDonkey, the p2p client for Linux/Unix/Windows. http://mldonkey.sourceforge.net/.