A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools*

Stefan Marr
Johannes Kepler University
Linz, Austria
stefan.marr@jku.at

Carmen Torres Lopez
Vrije Universiteit Brussel
Brussels, Belgium
ctorresl@vub.be

Dominik Aumayr
Johannes Kepler University
Linz, Austria
dominik.aumayr@jku.at

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.be

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract

Today's complex software systems combine high-level concurrency models. Each model is used to solve a specific set of problems. Unfortunately, debuggers support only the low-level notions of threads and shared memory, forcing developers to reason about these notions instead of the high-level concurrency models they chose.

This paper proposes a concurrency-agnostic debugger protocol that decouples the debugger from the concurrency models employed by the target application. As a result, the underlying language runtime can define custom breakpoints, stepping operations, and execution events for each concurrency model it supports, and a debugger can expose them without having to be specifically adapted.

We evaluated the generality of the protocol by applying it to SOMns, a Newspeak implementation, which supports a diversity of concurrency models including communicating sequential processes, communicating event loops, threads and locks, fork/join parallelism, and software transactional memory. We implemented 21 breakpoints and 20 stepping operations for these concurrency models. For none of these, the debugger needed to be changed. Furthermore, we visualize all concurrent interactions independently of a specific concurrency model. To show that tooling for a specific concurrency model is possible, we visualize actor turns and message sends separately.

CCS Concepts • **Software and its engineering → Concurrent programming languages**; *Software testing and debugging*;

Keywords Debugging, Tooling, Concurrency, Breakpoints, Stepping, Visualization

*Submitted for Review

DLS'17, October 24, 2017, Vancouver, Canada
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference format:
Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools. In *Proceedings of Dynamic Languages Symposium, Vancouver, Canada, October 24, 2017 (DLS'17)*, 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Building and maintaining complex concurrent systems is a hard task. Some developers combine different high-level models to solve problems with a suitable tool [Tasharofi et al. 2013]. Unfortunately, debugging support for combined concurrency models is minimal, which makes building and maintaining complex concurrent systems even harder.

For more than three decades, debugging support for concurrency models has been studied for each model in isolation [McDowell and Helmbold 1989]. As a result, thread-based languages such as Java and C/C++ have debuggers aware of threads and locks. Similarly, ScalaIDE and Erlang provide support for actors and message sending.

However, support for debugging combined concurrency models is still missing. The main challenge is to identify a common representation for concurrency models so that a debugger does not need specialized support for each model. For example, there are four main interpretations of the actor model, each of which has been implemented in different variations [De Koster et al. 2016]. For comprehensive debugging support of all these variations, a debugger needs to abstract from the concrete concurrency model and provide a common set of abstractions instead. This would allow us to use the same debugger without changes for the different concurrency models and their numerous variations.

This paper presents the Kómpos protocol, a *concurrency-agnostic protocol* to enable debuggers to support a wide range of concurrency models. Using the Kómpos protocol, we built the Kómpos debugger for online debugging of complex concurrent systems that combine *communicating event*

loops (CEL) [Miller et al. 2005], *communicating sequential processes* (CSP) [Hoare 1978], *software transactional memory* (STM) [Harris et al. 2005], *fork/join* [Blumofe et al. 1995], and shared-memory *threads and locks.* Based on the concurrency-agnostic protocol, Kómpos supports a rich set of breakpoints for the various concurrency abstractions, a rich set of stepping semantics to explore program behavior, a generic visualization of interactions between concurrent entities, as well as an actor-specific visualization of turns and message sends. This evaluation shows that the Kómpos protocol is (1) general enough to support advanced debugger features for shared-memory and message-passing models, and (2) that it supports tools using the provided data independently of any concurrency model, while preserving the ability to build tools specific to a single model.

Kómpos is a debugger for SOMns, a Newspeak implementation [Bracha et al. 2010] based on Truffle [Würthinger et al. 2012]. SOMns' debugger support is built on Truffle's tooling and debugger features [Seaton et al. 2014; Van De Vanter 2015]. SOMns supports the five aforementioned concurrency models and implements breakpoints, stepping, and a tracing mechanism for them. Because of the concurrency-agnostic design of the Kómpos protocol, the Kómpos debugger is independent from these concurrency models.

The contributions of this paper are:

- An analysis of the major shared-memory and message passing concurrency models to identify abstractions for a concurrency-agnostic debugger protocol.
- A concurrency-agnostic debugger protocol that enables custom breakpoints, stepping, and visualization.
- An implementation of the Kómpos protocol as part of the Kómpos debugger and SOMns.
- An evaluation of the protocol based on CEL, CSP, STM, fork/join, and threads and locks.

2 Background

This section discusses existing debugger protocols and introduces the Truffle debugger, on which we build our work.

2.1 Debugger Protocols

Runtime systems and integrated development environments (IDE) typically communicate via a *debugger protocol.* This includes the Java Debug Wire Protocol (JDWP),[1] the GDB machine interface,[2] the Chrome DevTools protocol,[3] and the Visual Studio Code debug protocol.[4]

These protocols define commands to interact with the program, to request information about threads, stack frames, local variables, objects, memory. They also communicate events, e.g., when a breakpoint was hit. While the protocols differ in format and structure, they cover a similar set of common debugger features. For instance, they allow a user to define a breakpoint for a specific source location, possibly with filters and conditions attached to it.

The use of a debugger protocol also decouples runtime systems and debuggers facilitating the construction of new tools. However, these protocols are often designed for sequential or threaded languages. As such, their support for debugging concurrency abstractions is rather limited. For instance, JDWP and GDB can show a list of running threads or control execution of a particular thread. GDB also has support to request information about Ada tasks. In Chrome, a step-into-async operation is the only explicit support for concurrent stepping. In short, the protocols are limited to these specific concurrency concepts.

Thus, custom breakpoints or stepping operations for concurrency models are not supported. Instead, the protocols support a fixed set of breakpoint and stepping operations. Any extension requires changing the protocol.

In this paper, we argue that a protocol similar to the ones mentioned above forms a foundation for the basic debugger features, i.e., to interact with the program and request basic information such as values or local variables and objects. However, in contrast to the classical debugger protocols, we propose a *concurrency-agnostic protocol* that supports a wide range of concurrency models. Our implementation in SOMns is inspired by the Visual Studio Code protocol (cf. section 5), but as detailed later, the Kómpos protocol abstracts completely from different breakpoints and stepping operations and thereby provides the necessary flexibility to support custom semantics for different concurrency models.

2.2 Truffle Debugger: A Language-Agnostic Debugging Framework

SOMns is built on top of Truffle, a framework for AST-based interpreters [Würthinger et al. 2012]. Part of the framework is support for interpreter instrumentation, which is used, e.g., for language-agnostic debugging and execution monitoring. The framework provides Truffle-languages with a classic breakpoint-based debugger for sequential code [Seaton et al. 2014], which we use in SOMns for the basic sequential stepping and breakpoint support.

One key element of the framework is its use of tags for the AST nodes [Van De Vanter 2017]. For the Truffle debugger support, a language annotates AST nodes with tags for `Statement`, `Call`, and `Root`. Based on these tags, the debugger determines the target nodes for line breakpoints, single stepping, step over, and returning from a method. SOMns

[1] *Java Debug Wire Protocol*, Oracle Inc., access date: 2017-05-16, https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwp-spec.html
[2] *GDB/MI Interface*, The GNU Project Debugger, access date: 2017-05-16, https://sourceware.org/gdb/onlinedocs/gdb/GDB$_0$02fMI.html
[3] *Chrome DevTools Protocol*, Google, access date: 2017-05-16, https://chromedevtools.github.io/devtools-protocol/
[4] *VS Code Debug Protocol*, Microsoft, access date: 2017-05-16, https://github.com/Microsoft/vscode-debugadapter-node/tree/master/protocol

uses the same mechanism for AST node tags to encode additional information, which is used to recognize concurrency-related operations as well as generic syntax information such as keywords. The Kómpos debugger can use this information for debugging and syntax highlighting.

3 Which Concurrency Concepts are relevant for Debugging?

This section analyzes concurrency models to identify the basic concepts that need to be supported by a debugger protocol to enable concurrency-related breakpoint and stepping operations. As a basic categorization, we distinguish shared-memory and message-passing models [Almasi and Gottlieb 1994]. We analyze instances for both types of models including threads and locks, STM, and fork/join parallelism as shared-memory models, and CEL and CSP as message-passing models. To account for more advanced debugger features, we also analyze what information would be needed to visualize these concepts.

3.1 Threads and Locks (T&L)

Shared-memory models, as supported by e.g., C/C++, Java, C#, have a wide range of concepts for simultaneous execution and access control to shared resources. For brevity, this analysis includes only threads, locks, condition variables, and object monitors. Other constructs are left to future work.

Threads are the active entities that execute code. Locks and object monitors enable the synchronization of threads, i.e., they restrict thread interactions on shared resources to enforce correctness properties. Condition variables are used to communicate between threads that certain conditions have changed. Furthermore, object monitors, as known from Ada or Java, are a structured synchronization mechanism that uses a lock to protect some shared resource in the dynamic scope of some object method or code block. In contrast, locks and condition variables are entities with which a thread can interact in unstructured ways.

For debugging, it needs to be possible to step through the execution of a thread and set breakpoints on statements. This should include the standard operations to step into or over a call to, and return from a method. However, the debugger should also provide stepping and breakpoints for concurrency abstractions such as locks, object monitors, and conditions variables. This would allow developers to check for incorrect synchronization. For locks, the debugger should be able to step from an `acquire` operation to the corresponding `release` operation to see how they relate. That also helps to detect unbalanced acquire/release operations which, e.g, can lead to starvation if locks are not released. Similarly, for object monitors, the debugger should allow to set a breakpoint on entering and exiting the monitor. For condition variables, stepping between the `wait` and `notify` operations allows to observe their effects on, and communication with other threads. From the point where a thread is created in a program, the debugger should allow to set a breakpoint on its execution, or to step into its execution. Similarly, when a thread terminates, the debugger should allow to suspend execution of the thread that joins with it and waits for termination.

The high-level interactions of threads entering monitors and using locks or condition variables are also relevant for visualization, which is useful for identifying unintended interactions or missing synchronization.

3.2 Communicating Event Loop Model (CEL)

The CEL model is a variation of the actor model, used by languages such as E [Miller et al. 2005] and AmbientTalk [Van Cutsem et al. 2014]. The main concepts are actors, messages, promises, and turns. Actors are the active entities, each with its own event loop. Actors execute messages as *turns*, i.e., one by one, in the order they are received. They contain a set of objects and interact via asynchronous messages because actors do not share memory. Promises are *eventual values*. They can establish a data dependency between actors, e.g., as placeholders for the return value of an asynchronous message. Since the CEL model includes only non-blocking abstractions, promise values can be accessed only via callbacks, which are executed as turns on an actor.

When debugging such CEL actors, developers should be able to step through turns of a specific actor to see which messages are received. This means, the debugger should combine normal sequential stepping within a turn with the ability to skip sequential operations and step to the next turn. When sending messages, suspending the actor's execution before a message is sent would allow developers to inspect its parameters and target object. Similarly, following the execution to observe how promises are resolved and how callbacks on them are executed after resolution can help developers to identify communication issues or unexpected values. For all these operations, the debugger should be able to either explicitly step through them or define breakpoints.

For a visualization, the high-level interactions of actors with messages and promises are relevant. For example, a visualization of the order in which turns, i.e., messages are executed could help to identify synchronization issues and bad message interleavings.

3.3 Communicating Sequential Processes (CSP)

The main concepts in CSP are processes, channels, and messages. Processes are the active entities that execute code. Channels are first-class elements that connect processes and allow them to communicate by passing messages with rendezvous semantics. A message is a specific datum exchanged via a channel. Like CEL, CSP is a message passing model. However, CSP uses blocking semantics and has no notion of turns, which makes it very different to CEL.

For debugging, it should be possible to follow the sequential execution of a process, from its creation to the end, like

in threads. But, similar to CEL, the debugger should be able to step through message sends, i.e., in this case channel operations, to identify the communication partners and their state, or put breakpoints on these operations. Furthermore, it should account for the rendezvous semantics of channels, to step from a receiver to the continuation in the sender. This is useful since channels can be passed around, which might lead to the wrong processes communicating with each other.

For visualization, the communication between processes and the run-time network of channels is relevant, e.g., to identify communication partners or lack-of-progress issues.

3.4 Software Transactional Memory (STM)

STM provides a wide range of notions for transactions, e.g., open or closed, optimistic or pessimistic. For brevity, this analysis includes only the basic concepts of threads and transactions. Threads are the same as for other shared-memory models. Transactions, on the other hand, introduce a dynamic scope, in which all modifications to shared state are applied either atomically or not at all.

The debugger should be able to interact with transactions in a way that makes it possible to observe the behavior when transactional conflicts occurs. Thus, developers should be able to set breakpoints or step through the execution of transactions to the final commit operation. Being able to stop right before a transaction commits allows developers to examine transaction interactions. The developer should also be able to step between transactions to follow the high-level flow of program elements interacting on shared state.

For a visualization, the transactions executed on threads and their ordering could help to identify missing synchronization, unintended dependencies, or performance issues.

3.5 Fork/Join Parallelism (F/J)

Fork/Join programming enables parallel divide-and-conquer algorithms. The main abstraction is an asynchronously executing task, which produces a result eventually. The model is only concerned with decomposing problems into a structure of tasks that synchronize based on fork and join operations. Other forms of synchronization are left out of the model.

A debugger should focus on these fork and join operations. Breakpoints and stepping should enable developers to explore the recursive structure of spawns and joins. Visualizing these spawn and join dependencies may help to understand the recursive nature of complex fork/join programs.

3.6 Analysis and Conclusion

The above discussion identified the main concepts for CSP, CEL, F/J, STM, and T&L. This section categorizes them to establish abstractions for a debugger protocol.

Activities are the active entities executing code. This includes threads, actors, processes, and fork/join tasks. **Dynamic scopes** are well-structured and nested parts of a program's execution during which certain concurrency-related

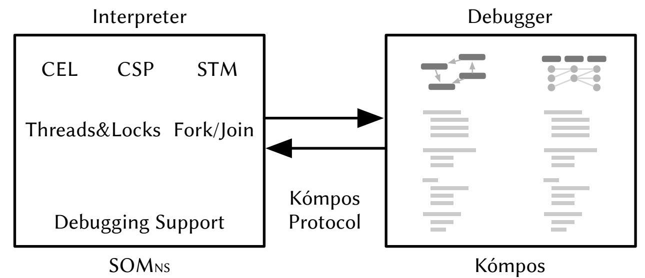

Figure 1. General architecture: Interpreter and debugger communicate via the concurrency-agnostic Kómpos protocol. The interpreter provides the implementation of the different concurrency models and debugging support.

properties hold, e.g., during a transaction, while executing code under an object monitor, or during an actor message turn. **Passive entities** are concurrency entities that are relevant for a program's execution, which do not act themselves, but are acted upon. For example, we consider messages and promises as passive entities of the CEL model, while the ones of CSP are channels and messages. Note that we do not consider normal objects as passive entities, because it was not needed.

To model interactions between these entities, we use send and receive operations. A **Send Operation** is an interaction that initiates communication or synchronization. A **Receive Operation**, is an interaction that reacts to a communication or synchronization operation. Consequently, we consider acquiring a lock or signaling a condition to be send operations and joining with a thread is a receive operation.

Table 1 gives an overview of the identified categories per concurrency model. These categories of entities and operations are used as foundation for the Kómpos protocol and detailed in the following section.

4 A Concurrency-Agnostic Debugger Protocol

To build a concurrency-agnostic debugger, we devise a protocol for communication between the debugger and interpreter that can support the breakpoints, stepping operations, and visualizations envisioned in section 3, without merely enumerating all the concurrency concepts. The goal is that only the language implementation needs to know the specifics for each concurrency model, while the debugger remains independent of them. Figure 1 shows the architecture of such a system. An interpreter with support for various concurrency models is connected to a debugger via the concurrency-agnostic Kómpos protocol.

This section discusses the design of the Kómpos protocol based on the analysis in section 3.6. First, it gives a high-level description, then discusses an example, and finally details each of the protocol elements.

449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504

Table 1. A Taxonomy of Concurrency Concepts Relevant for Debugging.

	T&L	CEL	CSP	STM	F/J
Activities	threads	actors	processes	threads	tasks
Dynamic Scopes	object monitors	turns		transactions	
Passive Entities	conditions locks	messages promises	channels messages		
Send Operations	acquire lock signal condition	send message resolve promise	send message		
Receive Operations	release lock wait for condition join thread		receive message join process	join thread	join task

4.1 High-Level Overview of the Protocol Concepts

From section 3.6 we derive that concurrency concepts relevant for breakpoints, stepping operations, and visualization can be modeled based on *activities*, *dynamic scopes*, *passive entities*, *send operations*, and *receive operations*. Using these basic notions, a protocol is independent of any specific concurrency model. However, the debugger requires meta data to match debugging operations and concrete concurrency concepts. Thus, when the debugger connects to the interpreter, it receives meta data with details on the supported concurrency models, the entities they define, their breakpoints, and their stepping operations. Note, this information is mostly opaque to the debugger. For breakpoints and stepping, it is sufficient to match opaque identifiers, as is explained in section 4.3.

Concurrency Concepts. As discussed in section 3.6, *activities* are active entities that execute code. The debugger protocol uses this notion, e.g., to offer stepping operations that are specific to an activity type. *Dynamic scopes* are used, for instance to determine possible stepping operations. Some stepping operations are only available during a transaction or while holding an object monitor.

Passive entities, *send*, and *receive operations* are used for the visualization of concurrent interactions. However, they are currently not used in the context of pausing/resuming program execution or performing step-by-step execution.

Debugger Concepts. For debugging, the protocol includes various other concepts. For brevity, we discuss only the ones distinct from other debugger protocols (cf. section 2.1).

A *source* is either a file or some other form of source text. The source text has to be annotated with *source tags* to identify the semantic elements contained in a source range. For instance, tags can indicate the source locations of message sends or lock operations so that the debugger can show breakpoints or stepping operations. As mentioned in section 2.2, SOMns employs Truffle's tagging mechanism to annotate AST nodes in the interpreter. The Kómpos protocol is used to send this information to the debugger.

```
                                      actType    scopes
   Atomic
(1) atomic {                          thread
  (2) int b = this.fieldB;            thread     transaction
      this.fieldA = b + 1;
  (3)                                 thread     transaction
   }
```

Figure 2. Example program using an atomic transaction. The debugger recognizes the `atomic` keyword via the `Atomic` tag. When execution suspends at one of the three program points indicated with a number, the debugger receives location, indicated activity type, and active dynamic scopes.

A *breakpoint* type defines suspension points, which may be related to concurrency concepts. For example, one breakpoint type could be for the point where a message is received by an actor. They are distinguished by name and define which source tags they apply to. Thus, the debugger does not need to know the relationship between tags and breakpoints. Instead, it can treat tags as opaque identifiers.

A *stepping operation* type defines an operation to follow program execution sequentially or concurrently. Similar to breakpoint types, stepping operation types are distinguished by name. Furthermore, they define criteria to determine whether the operation is applicable in the current dynamic context. Such applicability criteria include source tags, the activity type executing the currently suspended code, as well as the dynamic scopes active for the current execution.

4.2 Example: Breakpoints and Stepping for an Atomic Block

This section discusses an example to illustrate the protocol. Consider the program fragment in fig. 2 that uses an `atomic` block to ensure that the `fieldA` of an object is updated without interference by other threads.

The figure indicates three possible program points with a number, in which execution can be suspended for the atomic block. Further, it shows that the atomic block is known to the debugger via the `Atomic` tag, which it received as part

505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560

of the meta data and source information. When execution is suspended, the debugger received the source location, the activity type, and active dynamic scopes, which are depicted in the right hand side of the figure. From the meta data and the `Atomic` tag associated with the source location for ①, the debugger can derive that it can offer a breakpoint that is triggered right before a transaction is started.

Setting the breakpoint sends a `BreakpointUpdate` message to the SOMNs interpreter, which includes the coordinate for source location and the chosen breakpoint type. Afterwards, the program can stop at the `atomic` keyword, and the interpreter sends a `Stopped` message to the debugger. The message says that execution is suspended at location ①, and that the current activity is a thread with a specific id. Based on this location information, source tags, and execution information, the debugger can derive the applicable stepping operations. Note that in this case there is no concurrent stepping applicable. However, all stepping operations are handled uniformly. When the debugger determines the stepping operations for ①, the resulting set contains only the sequential stepping operations that always apply, e.g. step into and step over.

When the developer chooses the step-into operation, the debugger sends a `Step` message to the interpreter, which includes the thread id and the chosen stepping operation. After the interpreter performs this step, execution is suspended at ② and the debugger receives again the current location and activity. It also receives the information that a dynamic scope for a transaction is active. Based on this scope, it can offer extra stepping operations, e.g., to step right before or after the commit for the transaction. When executing these stepping operations, execution would continue either to ③, or to the first statement after the `atomic` block.

4.3 Detailed Description of the Kómpos Protocol

This section provides a more detailed overview of the protocol. While we discuss the semantics of the protocol, we refrain from prescribing a specific implementation, since we consider the main ideas to be applicable to wide range of concrete debugger protocols. A concrete prototype implementation is discussed in section 5.

The protocol assumes bidirectional communicate between interpreter and debugger, which could be realized with remote function calls, messages on sockets, etc.

Figure 3 shows an overview of the main concepts used by the Kómpos protocol. We distinguish between (1) debugging-specific messages exchanged between debugger and interpreter, (2) trace data sent from the interpreter for visualization, and (3) a general meta-data description used to interpret the exchanged messages. We detail each of them below.

Debugger Messages. For stepping and breakpoints, the Kómpos protocol uses the first four debugger messages in fig. 3.

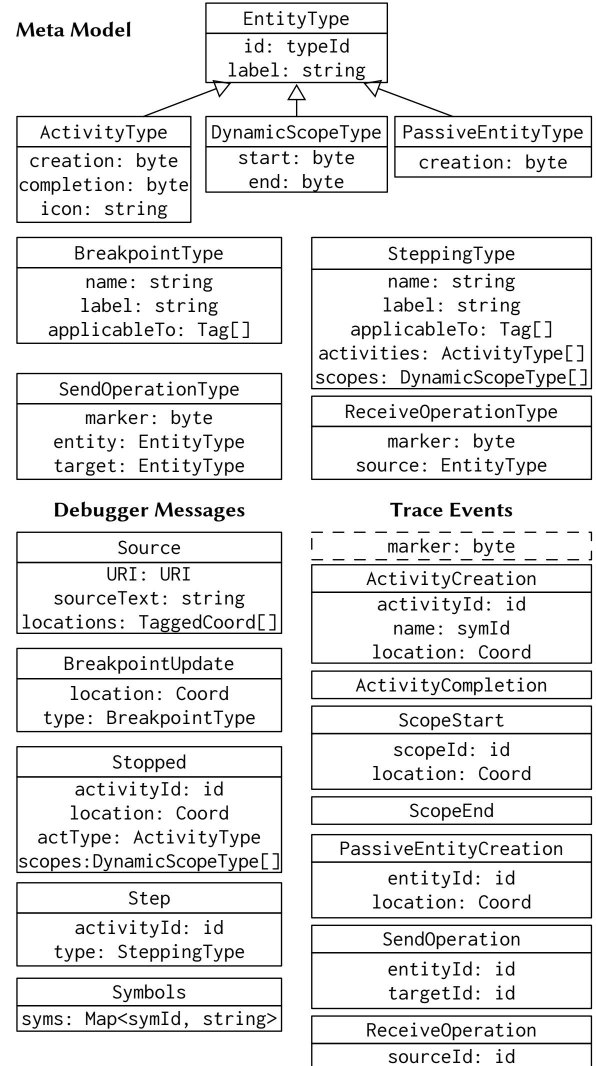

Figure 3. Class diagram of the main elements of the Kómpos protocol. The meta model describes the concurrency and debugger concepts supported by the interpreter, and provides the debugger with the meta data to identify when and where breakpoints and stepping operations are applicable. Debugger messages are used to update the debugger or interpreter. Trace events encode an execution trace used for visualization. Trace events are prefixed with a `marker` byte.

The `Source` message provides the source information to the debugger. Since the debugger is to be agnostic from specific concurrency concepts, as well as incidentally from a specific language, we provide source information explicitly. The message includes a URI to identify the source file or resource, the source text, and a list of tagged source locations. Source locations specify the exact coordinates, for instance based on a line number, column number, and character length. The tags are merely opaque identifiers which identify concurrency operations, e.g., as seen with the `Atomic` tag in fig. 2.

The BreakpointUpdate message is used to communicate breakpoints from the debugger to the interpreter. It encodes the source location and the breakpoint type.

The Stopped message is sent from the interpreter to the debugger to indicate that either a breakpoint was hit or a stepping operation completed. It identifies the current location and the suspended activity with id and type. Furthermore, it includes a list of currently active dynamic scopes for this activity. Note that the activity type and active dynamic scopes can also be determined from the trace data, but providing them explicitly simplifies the debugger implementation.

Finally, the Step message is sent from the debugger to the interpreter to instruct the latter to resume execution of a specified activity with a given stepping type.

The last message listed in fig. 3, called Symbols message, is an optimization. It avoids sending long strings repeatedly by sending a symbol table from the interpreter to the debugger.

Execution Trace Data. To provide details about the execution of a concurrent program, the Kómpos protocols uses trace events that encode the program's behavior with 7 different trace entries. We use these trace events for instance to visualize concurrent interactions. In general, each trace event starts with a marker, which is indicated by the dashed line in fig. 3. The relation between the concrete marker and a concurrency concept is defined via the meta data discussed in the following subsection.

An ActivityCreation event records the id of the created activity, its name, and the source location of the creation operation. An ActivityCompletion event is merely a marker recording that an activity terminated. The corresponding activity id can be determined from the complete trace. A ScopeStart event records the beginning of a dynamic scope. It records the id of a scope, which corresponds to, e.g., the message id for an actor turn. It also includes the source location for the scope, e.g., the method invoked for a turn, or the atomic code block for a transaction. A ScopeEnd event is also a marker that can be matched to the scope start implicitly. A PassiveEntityCreation event records the id of the passive entity created and the source location of the operation.

Interactions are recorded as SendOperations with the id of the involved passive entity, e.g., channel or message, and the target entity id, e.g., the receiving actor. Information about the sending entity can be inferred from the trace based on the dynamic scope or current activity. ReceiveOperations encode merely the id for the source entity which is for instance a channel or fork/join task.

Meta Data Description. The debugger messages and trace events discussed above are completely independent from concurrency models. To distinguish different types of entities and interactions, the interpreter sends meta data to the debugger when the connection is initialized. The meta data consists of the 8 concepts shown at the top of fig. 3.

There exist three types of entities: ActivityType, DynamicScopeType, and the PassiveEntityType. EntityType defines data common to all entity types. All entities have a label, i.e. a name, and a unique id to distinguish them. ActivityType additionally defines unique trace event marker for activity creation and completion, as well as an identifier for an icon to be used in the user interface. DynamicScopeType defines the start and end markers for scopes, and PassiveEntityType defines creation marker.

The BreakpointType defines the possible breakpoints. Each type has a unique name, a label to be used in the user interface, and an applicability criterion based on source tags. If a source location has one of the listed tags, then it supports the breakpoint. If a breakpoint type does not specify any source tags, it applies to all source locations.

The SteppingType defines the stepping operations. Each type has a name and label. As for breakpoints, source tags also define whether a stepping operation is applicable. For instance, the operation to step to the receiver of a message is only available on a message-send operation. An additional applicability criterion is the current activity type, e.g., to enable stepping to the next turn for actors. Similarly, stepping can be conditional to the current dynamic scope. As such the third applicability criterion is the current scope type. For instance, some transaction-related stepping operations are only useful within a transaction (cf. section 4.2).

Finally, the meta data specifies how to interpret SendOperations and ReceiveOperations. For each operation, a unique marker is defined. For a send, the operation type specifies the entity types for the involved entity and target. This allows to interpret the id and identify which set of entities it belongs to. Similarly, for a receive operation, the type of the source entity is specified.

This meta data makes it possible to handle breakpoints and stepping operations in an abstract manner. Furthermore, it becomes possible to interpret the trace events either generically or specific to a concurrency model to visualize them. We evaluate both aspects in section 6.

5 Implementation

This section provides some basic details of our implementation, which is used for the evaluation in the next section. The Kómpos debugger is a web application running in a browser. The SOMns interpreter implements the necessary support for the concurrency models, their breakpoints, stepping semantics, and execution tracing. As shown in fig. 1, the SOMns interpreter and the Kómpos debugger communicate via a bi-directional connection, for which we use Web Sockets.[5] JSON is used to send the meta data and debugger messages between the two components. For efficiency, the trace events are sent through a separate binary web socket.

[5] *The WebSocket Protocol*, IETF, access date: 2017-05-16, https://tools.ietf.org/html/rfc6455

When the Kómpos debugger connects to SOMns, the interpreter sends the meta data to initialize the debugger. The debugger then processes the meta data to enable efficient parsing of trace events, to initialize breakpoints, stepping operations, and visualizations. Based on the labels provided as part of the meta data, the Kómpos debugger also interprets the meta data to enable filtering and querying data for specific concurrency models, which can be used to build tools specific to a concurrency model.

When a program executes in the SOMns interpreter, it sends source code and source tags as part of the `Source` message to the debugger. The Kómpos debugger uses this data to display the code, indicate possible locations for breakpoints, and also apply syntax highlighting based on the tags. This approach makes the debugger completely language-agnostic.

When the program hits a breakpoint or completes a step, the debugger uses the data provided in the `Stopped` message to highlight the source location. It also uses the meta data and information about current activity type and active dynamic scopes to select the possible stepping operations. To obtain information on the stack trace and local variables, we use messages similar to the Visual Studio Code debugger protocol (cf. section 2.1). As a result, we can also use Visual Studio Code as debugger for SOMns,[6] at least for sequential debugging.

One challenge for the correct implementation of a concurrent debugger such as Kómpos is that interactions between the interpreter and debugger need to handle data races. For instance, there is a race between the two web socket connections, because the order, in which messages are received between the two connections, is not guaranteed. This can be problematic because we might hit a breakpoint for which the debugger does not yet know the corresponding activity. We solve this in the debugger by using promises for the activities, which delays handling for the debugger message until all data is available. Similarly, a trace event can also use a symbol id, for which the full string was not yet received via the `Symbols` message. For trace events, we handle these races by waiting for all dependent data elements before a trace event can be used further.

6 Evaluation

This section evaluates the Kómpos protocol with respect to its ability to support breakpoints, stepping operations, and visualizations. The goal of the evaluation is to demonstrate that the protocol is agnostic of specific concurrency abstractions and general enough to support a wide range of concurrency models. We base the evaluation on the five aforementioned models: CEL, CSP, fork/join, STM, and threads and locks. These models are chosen for their different concurrency characteristics, and for being the main programming models in the field. All reported experiments are implemented as part of SOMns[7] and the Kómpos debugger [Marr et al. 2017].

6.1 Breakpoints

To evaluate the flexibility of the system to represent various breakpoints, we apply the analysis results of section 3 to SOMns. Specifically, we identify and implement 21 different breakpoints that can be used to pause execution based on the concurrency abstractions and their interactions. As a general principle, we consider the point in time right before or after a concurrent operation as potentially relevant. The goal is to allow developers to observe the effects of an operation that might interleave with other operations in the system. The breakpoints are listed with a brief description in table 2.

With the concepts of the Kómpos protocol presented in section 4.3, we were able to model all breakpoints solely by specifying the source tag to identify the source locations they apply to. No specific support was required in the debugger. The breakpoint implementation is thus completely confined to the interpreter, where the concurrency operations are implemented. Arguably, the Kómpos protocol allows developers to define arbitrary breakpoints specific to concurrency operations, or other kind of language constructs.

6.2 Stepping Operations

To evaluate the Kómpos protocol's support for standard and advanced stepping operations, we apply the results of the analysis in section 3 and implement 20 different stepping operations. Guided by the breakpoints in table 2, we identified stepping operations that allow one to follow the execution flow between various potential points of interest. The stepping operations are listed with a brief description in table 3.

With the Kómpos protocol, we were able to model all those stepping operations and customize their applicability based on a current source location, the type of the current activity, or active dynamic scopes. Other than these generic concepts, no support is required in the debugger. Similar to the breakpoint support, the stepping operations are defined completely in the interpreter, where the concurrency operations are implemented. This demonstrates that the Kómpos protocol provides the desired flexibility to define arbitrary stepping operations.

6.3 Visualization

Finally, we evaluate whether the data provided by the Kómpos protocol can be used for visualizations. To this end, assess whether it is possible to built a visualization that is agnostic to the concurrency models as well as one that is specifically designed for one concurrency model. The goal is to identify where the boundary is between concurrency-agnostic aspects and special-purpose constructs. We built: (1) an agnostic visualization of interactions between entities

[6] *SOMns VS Code Extension*, access date: 2017-05-16, https://github.com/smarr/SOMns-vscode

[7] *SOMns and the Kómpos Debugger Protocol*, access date: 2017-05-16, https://github.com/metaconc/SOMns/tree/uniform-trace-format

Models	Name	Description	Source Tag
all	activity creation	before an actor, process, task, or thread is created	`ActivityCreation`
CSP, F/J, T&L	activity execution	before first statement of the new activity is executed	`ActivityCreation`
CSP, F/J, T&L	before join	before waiting that a process, task, or thread completes	`ActivityJoin`
CSP, F/J, T&L	after join	after a process, task, or thread completed execution	`ActivityJoin`
CEL	actor message send	before an actor message is sent	`EventualMessageSend`
CEL	actor message receiver	before first statement of message is processed in the receiver	`EventualMessageSend`
CEL	before async. method activation	before the first statement of a method activated by an async. msg send	`EventualMessageSend`
CEL	after async. method activation	before returning from a method activated by an async. msg send	`EventualMessageSend`
CEL	before promise resolution	before a promise is resolved with a value or error	`PromiseCreation`
CEL	on promise resolution	before the first statement of all handlers registered on promise	`PromiseCreation`
CSP	before channel send	before executing the send on a channel (set on send operation)	`ChannelWrite`
CSP	after channel receive	after receiving a message from a channel (set on send operation)	`ChannelWrite`
CSP	before channel receive	before receiving a message from a channel (set on receive operation)	`ChannelRead`
CSP	after channel send	after sending on a channel (set on receive operation)	`ChannelRead`
STM	before transaction	before starting a transaction	`Atomic`
STM	before commit	before attempting to commit changes of a transaction	`Atomic`
STM	after commit	after committing the transaction succeeded	`Atomic`
T&L	before acquire	before attempting to acquire a lock	`AcquireLock`
T&L	after acquire	after acquiring a lock	`AcquireLock`
T&L	before release	before releasing a lock	`ReleaseLock`
T&L	after release	after releasing a lock	`ReleaseLock`

Table 2. Set of breakpoints implemented in SOMns. None of the breakpoints requires special support in the Kómpos protocol. Instead, they are all implemented based on meta data that includes name and source tag.

Model	Name	Description	Criteria
all	resume	continue execution of current activity	
all	pause	pause execution of current activity	
all	stop	terminate program	
all	step into	step into method call	
all	step over	step over method call	
all	return	return from method call	
CSP, F/J, T&L	step into activity	halt new activity before execution of the first statement	source tag: `ActivityCreation`
CSP, F/J, T&L	return from activity	halt activity that joins with the current one, after joining	current activity: Process, Task, Thread
CEL	step to message receiver	halt activity before executing the first statement of a received message	source tag: `EventualMessageSend`
CEL	step to promise resolver	halt activity before resolving a promise	source tag: `PromiseCreation`
CEL	step to promise resolution	halt all activities before executing the first statement of handlers registered on a promise	source tag: `PromiseCreation`
CEL	step to next turn	continue current actor's execution and stop before the first statement of the next executed message	current activity: Actor
CEL	return from turn to resolution	continue current actor's execution and stop before the execution of the first statement of all handlers registered on a promise that is resolved by the current turn	current activity: Actor
CSP	step to channel receiver	halt activity reading from a channel to receive the sent message	source tag: `ChannelWrite`
CSP	step to channel sender	halt activity sending to a channel	source tag: `ChannelRead`
STM	step to next transaction	halt activity before starting the next transaction	
STM	step to commit	halt activity before committing a transaction	dynamic scope: Transaction
STM	step after commit	halt activity after committing a transaction	dynamic scope: Transaction
T&L	step to release	halt activity before releasing a lock	dynamic scope: monitor
T&L	step to next acquire	halt the next activity right after acquiring the current lock	dynamic scope: monitor

Table 3. Set of stepping operations implemented in SOMns. None of these stepping operations require special support in the Kómpos protocol. Instead, they are realized solely based on the applicability criteria provided as part of the meta data.

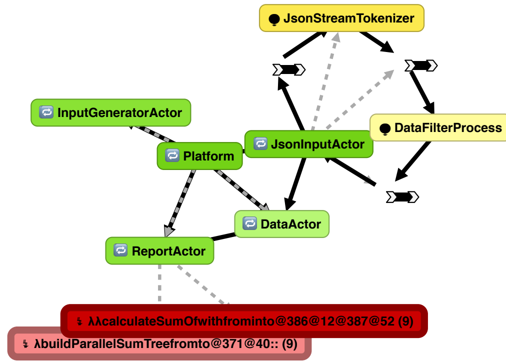

Figure 4. Screenshot of the system interaction visualization. Activities are represented as rectangles. The debugger assigns color ranges to an activity type, e.g., green for actors, yellow for processes, and red for tasks. Color shades distinguish between activities of the same type. Black arrows represent messages sent and gray dashed arrows indicate who created an entity. Black bars with two white arrows are a custom visualization for channels. The visualization is chosen via an optional map using an entity type's label.

in a program, and (2) a visualization specific to the actor model which shows the execution of actor turns and their causal relationships based on the messages sent in a turn.

6.3.1 System Interaction Visualization

The system interaction visualization shows how entities communicate with each other. Figure 4 shows a screenshot of the visualization. Activities are visualized as rectangles with rounded corners. Depending on the number of activities created from the same source location, the visualization groups the activities. Furthermore, the debugger chooses a different color range depending on the activity type. On the other hand, the icon in front of an activity's name is directly specified as part of the meta data (cf. fig. 3). An `ActivityType` includes a name for an icon, for which the debugger can then determine a suitable visualization.

Passive entities are visualized with custom SVG graphics. Figure 3 shows channels as two white arrows on top of a black bar. The visualization is generated in the debugger and matched to a `PassiveEntityType` based on its label. The goal was to make these entities easier to recognize. The design tradeoff here is between including more meta data in the protocol and leaving room for the debugger to add custom visualizations like this. We decided that the simplest would be to have an extensible map in the debugger to select a specific visualization for entities it is aware of.

The gray-dashed arrows between entities, i.e., activities or passive entities, are determined based on their creation information. The visualization does not show dynamic scopes. It merely uses them to identify the connection between entities. Actor and channel messages are not shown because we do not record their creation, but rather the specific send/receive operations. Thus, the visualization itself is agnostic from the concurrency models, but it depends on how the data is encoded in trace events for a specific concurrency model.

Send/receive events are used for the black arrows. Entities exchanging more messages are displayed closer together.

Overall, the system interaction visualization is independent of specific concurrency models. It just uses the knowledge about activities, dynamic scopes, passive entities, creation operations, and send/receive operations to generate a graph representing the systems interaction. For most aspects, the debugger independently visualizes the elements, colors, and icons considering only meta data provided by the interpreter. However, to provide the extra bit of polish, i.e., to provide an iconic representation of channels, it includes a map of additional visualizations that matches entity type labels. We consider this design a reasonable tradeoff that shows that small customizations are possible, but a concurrency-agnostic visualization is feasible.

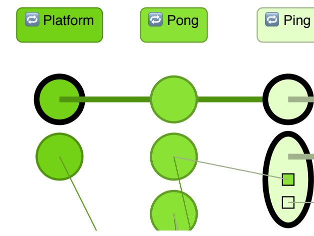

Figure 5. Screenshot of the actor turn visualization. Each actor is shown on a lane with its turns indicated as circles. Lines between turns indicate message sends. When expanding a turn, it shows the order of the messages sent.

6.3.2 Actor Turn Visualization

Figure 5 shows our second visualization which is inspired by the processes view in Causeway [Stanley et al. 2009]. The goal of this visualization is to show the causality between turn executions and messages. It visualizes each actor in the system on a lane, on which its turns are indicated as circles. A line indicates the message that caused a turn. When inspecting a specific turn, it unfolds into an ellipse and shows sent messages (as rectangles) in the order they were sent. The messages connect with arrows to the turns on the receiving actor that processes them.

While this visualization is specific to the actor model, parsing and interpreting of the trace events is still done in an agnostic way. Only after obtaining the data, the Kómpos debugger uses the meta data to determine which activities are actors, which dynamic scopes are turns, and which send operations are actor messages.

This visualization is specific to the notion of communicating event loops, and its implementation makes assumptions about which interactions are possible. Nonetheless, it is based

on Lamport's general *happens-before relationship* [Lamport 1978], which can be applied to other concurrency models, too. Moreover, its implementation in Kómpos is based on the abstract notions of the protocol, and merely filters out the actor-related trace events. Thus, it seems feasible to extent it to other concurrency models, especially if they use for instance transactions or object monitors as dynamic scopes, which would allow to indicate their causal relations.

6.4 Conclusion

The evaluation shows that the Kómpos protocol is abstract enough to support arbitrary breakpoints and stepping operations independent from a specific concurrency model. Furthermore, the provided data is generic enough for tools that are agnostic of the concurrency models as shown with our system interaction view. However, it remains possible to build tools specific to a concurrency model by interpreting the meta data, as we have shown with the actor turn view.

7 Related Work

This section discusses concurrent debuggers and novel IDE designs that influenced our work or that are closely related. Debugger protocols similar to ours and their limitations have been discussed in section 2.1. Generally, their support for concurrency models is minimal and they do not provide any facilities for custom breakpoint or stepping types, while the Kómpos protocol is designed for this purpose.

7.1 Concurrent Debuggers

Debuggers for concurrent and parallel systems have a long history [McDowell and Helmbold 1989]. This includes support for breakpoints, stepping, and visualizing of parallel systems. However, to the best of our knowledge, so far, no debugger supports a wide range of concurrency models.

REME-D [Gonzalez Boix et al. 2014] is the closest related work. It is also an online debugger focusing on distributed communicating event-loop programs, that uses a meta-programming API to realize breakpoints and stepping semantics. Our actor breakpoint and stepping operations are reminiscent of REME-D's ones. However, REME-D's API is specific to the actor model, and does not abstract from concurrency concepts as the Kómpos protocol does.

Erlang[8] and ScalaIDE[9] support basic debugging of actor programs with sequential stepping and breakpoints. ScalaIDE also includes an option to follow a message send and stop in the receiving actor. However, neither of them attempts to go beyond this basic debugger functionality.

Zyulkyarov et al. [2010] introduced a debugger for a transactional system. The focus of their work is to ensure that the STM implementation does not interfere with the debugging experience, and that stepping over or into transactions works naturally. Furthermore, they provide mechanisms for conflict-point discovery and debug-time transactions. Our work, however, focuses on advanced breakpoint and stepping semantics. Their advanced debugging mechanisms would be highly interesting for Kómpos, too.

Early prototypes of the Kómpos debugger were presented by Torres Lopez et al. [2016] and Marr et al. [2017]. However, this was only an exploration of initial ideas and did not yet include any work on the Kómpos protocol.

7.2 Novel IDE Designs

Projects such as the Language Server Protocol,[10] which is implemented by Visual Studio Code, and Monto [Keidel et al. 2016] try to change how we think about integrated development environments (IDEs). Instead of using the plugin approach common to Eclipse or Visual Studio, they provide support for languages by providing a common protocol to exchange information for code completion, code errors, and other common IDE services. We consider their design an inspiration for this work. However, neither the language server protocol nor Monto support debugging at this point.

With respect to flexible debuggers, the work of Chiş et al. [2015] on a debugger framework for domain-specific debuggers might be the most advanced. They support domain-specific breakpoints, stepping operations, and debugger views. For example, they have a debugger for a parser framework, which allows to step through the parsing process on the level of the parser rules instead of the parser implementation. Similarly, they have a debugger for a complex notification system, which allows stepping through the activations of the subscriptions to notifications instead of working on the basic notion of method calls and callbacks. Instead of providing a framework for building debuggers, our work focuses on the protocol between the debugger and the interpreter. To our understanding, our protocol supports all required elements to also support their domain-specific breakpoint and stepping operations. However, we do not provide a framework to build custom debugger interfaces as they do.

8 Conclusion and Future Work

To enable better debugging tools for complex concurrent applications, we propose the Kómpos protocol, a concurrency-agnostic debugger protocol. The protocol abstracts from specific concurrency models to support custom breakpoints, stepping operations, and visualizations, without requiring support for the specific concurrency models.

Based on our study of shared-memory and message-passing models, the protocol represents concurrency concepts in terms of activities, dynamic scopes, and passive entities. It uses opaque meta data to allow the debugger to determine

[8] *Debugger*, Ericsson AB, access date: 2017-05-16, http://erlang.org/doc/apps/debugger/debugger$_c$hapter.html

[9] *Asynchronous Debugger*, ScalaIDE, access date: 2017-05-16, http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html

[10] *Language Server Protocol*, Microsoft, access date: 2017-05-16, https://github.com/Microsoft/language-server-protocol

where breakpoints or stepping operations are applicable. The protocol also includes the notion of send and receive operations to, e.g., visualize concurrent interactions.

To evaluate the protocol, we implemented it in the Kómpos debugger and SOMns. SOMns supports the five major concurrency models: threads and locks, communicating event loops, communicating sequential processes, fork/join parallelism, and software transactional memory. We implemented 21 breakpoints and 20 stepping operations in SOMns for these models, without requiring any modifications to the debugger, which shows that the protocol is concurrency agnostic. We also implemented two visualizations. The first one shows the concurrent interactions independently of the concurrency models. The second one shows causalities between actor turns, messages, and their ordering, which is specific to the communicating event-loop model. This demonstrates that the protocol is flexible enough to enable advanced debugging tools that are concurrency agnostic, while it remains possible to build tooling specific to a concurrency model.

Based on this work, existing debugger protocols could be extended to provide advanced debugging support for concurrent programming, without requiring support for specific concurrency models. This provides a foundation for better tooling and debuggers for complex concurrent systems that combine concurrency models.

For future work, we would like to study how to enable arbitrary libraries to benefit from such a generic protocol. The challenge here is to expose the relevant data about concepts and their relation to library methods to the interpreter so that it can be communicated to the debugger. Especially in dynamic languages, it needs to be able to expose this information at runtime.

Further work is also required to make the visualization scalable to large applications. We need to find ways to explore complex systems and focus on relevant details, and we need to investigate ways to provide the relevant data efficiently. Future work also needs to study how to effectively expose the large number of concurrency-specific debugger features to users, and whether they help to debug concurrent applications more effectively.

Acknowledgments

We would like to thank Sander Lenaerts for the implementation of the actor turn view, and Manuel Rigger and Richard Roberts for comments on an early draft. Stefan Marr and Dominik Aumayr were funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31. Carmen Torres Lopez was funded by a grant of the Research Foundation Flanders (FWO), project number G004816N.

References

George S. Almasi and Allan Gottlieb. 1994. *Highly Parallel Computing* (2nd ed.). Benjamin-Cummings Publishing Co., Inc.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proc. of PPoPP*, Vol. 30. ACM.

Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *Proc. of ECOOP*. LNCS, Vol. 6183. Springer, 405–428.

Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. 2015. Practical domain-specific debuggers using the Moldable Debugger framework. *Computer Languages, Systems & Structures* 44, Part A (2015), 89–113.

Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proc. of AGERE!'16*. ACM, 31–40.

Elisa Gonzalez Boix, Carlos Noguera, and Wolfgang De Meuter. 2014. Distributed debugging for mobile networks. *Systems and Software* 90 (2014).

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proc. of PPoPP'05*. ACM, 48–60.

C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677.

Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. 2016. The IDE Portability Problem and Its Solution in Monto. In *Proc. of SLE'16*. ACM, 152–162.

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. Kómpos: A Platform for Debugging Complex Concurrent Applications. (2 April 2017), 2 pages.

Charles E. McDowell and David P. Helmbold. 1989. Debugging Concurrent Programs. *ACM Comput. Surv.* 21, 4 (Dec. 1989), 593–622.

Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Symposium on Trustworthy Global Computing (LNCS)*, R. De Nicola and D. Sangiorgi (Eds.), Vol. 3705. Springer, 195–229.

Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. 2014. Debugging at Full Speed. In *Proc. of DYLA'14*. ACM, Article 2, 13 pages.

Terry Stanley, Tyler Close, and Mark Miller. 2009. *Causeway: A message-oriented distributed debugger*. Technical Report. HP Labs. 1–15 pages. HP Labs tech report HPL-2009-78.

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with other Concurrency Models?. In *Proc. of ECOOP (LNCS)*, Vol. 7920. Springer, 302–326.

Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. 2016. Towards Advanced Debugging Support for Actor Languages: Studying Concurrency Bugs in Actor-based Programs. (30 October 2016). Presentation, AGERE! '16.

Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Com. Lan., Sys. & Struct.* 40, 3–4 (2014), 112–136.

Michael Van De Vanter. 2017. Building Flexible, Low-Overhead Tooling Support into a High-Performance Polyglot VM: Extended Abstract. (2 April 2017), 3 pages. Presentation, MoreVMs'17.

Michael L. Van De Vanter. 2015. Building Debuggers and Other Tools: We Can "Have It All". In *Proc. of ICOOOLPS'15*. ACM, Article 2, 3 pages.

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proc. of DLS'12*. 73–82.

Ferad Zyulkyarov, Tim Harris, Osman S. Unsal, Adrían Cristal, and Mateo Valero. 2010. Debugging Programs That Use Atomic Blocks and Transactional Memory. In *Proc. of PPoPP'10*. ACM, 57–66.