Modular Static Analysis of Actor Programs

Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, Coen De Roover
{qstieven,jnicolay,wdmeuter,cderoove}@vub.ac.be
Vrije Universiteit Brussel, Belgium

***Abstract*—Existing static analyses for actor programs explicitely model all possible execution interleavings. Because the number of interleavings increases exponentially with the number of actors and messages exchanged, such analyses scale poorly. We describe the first modular analysis for actor programs, that analyzes each actor in separation of each other. This analysis over-approximate over the diferent interleavings without explicitly modeling them, rendering it scalable. We demonstrate its enhanced scalability by comparing the analysis of the Savina benchmark suite with a non-modular analysis and our new modular analysis. Our technique succeeds in analyzing all of the Savina benchmarks in a matter of seconds, while the non-modular analysis times out on more than half of these benchmarks. Moreover, we show that the precision of our modular remains on par with the precision of the non-modular analysis.**

I. Introduction

Actor programs consist of isolated processes, called *actors*, that run concurrently to each other and communicate through *messages*. At run time, actors can change their behavior, create an unbounded number of other actors, and communicate actor references through messages. This results in a possible highly dynamic topology for actor programs.

Existing static analyses for actor programs explicitely represent every possible interleaving of concurrent actor execution, and are therefore limited in scalability. Modular analyses allow for a scalable analysis, and have been explored in the context of shared-memory concurrent programs, limited to systems where the number of processes is fixed and known in advance. We explore modular analysis for actor programs. Compared to existing work for shared-memory concurrent programs, we need to account for dynamic creation of new actors.

The core insight behind our modular analysis is that the dynamic behavior of an actor is entirely defined by its code and the set of messages it receives. While the code is known beforehand, the set of messages received must be inferred by analyzing other actors. The goal of the analysis at the level of an actor is therefore to infer the set of created actors and messages sent to other actors for every possible message the actor understands. We refer to this analysis as the *intra-actor analysis*, an analysis that analyzes one actor in isolation and infers the interferences generated by this actor. The intra-actor analysis enables a modular analysis, at the level of the entire program, to infer the set of running actors and the set of messages received by each running actor. We refer to this analysis as the *inter-actor analysis*, an analysis that uses information from the intra-actor analysis of a set of actors to drive the next set of intra-actor analyses on a more complete set of actors. Repeated iterations of the inter-actor analysis will achieve a sound over-approximation of the behavior of all actors in the actor program in a scalable manner.

Our approach infers properties about concurrent actor programs that are required for tool support for pressing problems in software engineering such as program comprehension and bug detection. The inferred properties concern the actors created, the messages exchanged, and the behaviors changed at run time —in addition to the traditional data and control flow properties known from analyses for sequential programs. We demonstrate that our approach scales beyond the current state-of-the-art, as the first analysis to analyze the entire Savina benchmark suite [3] in a matter of seconds, with the same precision.

II. Dynamic behavior of actor programs

Consider the program in Listing 1, implemented in a Scheme dialect that supports concurrent actors [4]. The program computes and displays the factorial of a number.

After the definition of three behaviors through the `actor` construct (lines 2, 11, and 16), two actors are spawned through the `create` construct: actor `fact` with behavior `fact-actor` (line 20), and actor `displayer` that displays the messages it receives (line 21). On line 22 the number returned by `read-integer` (assumed non-negative) is sent to actor `fact` along with actor `displayer`, the *customer* that will receive the answer that `fact` computes.

When actor `fact` receives the `compute` message with $n = 0$, it sends 1 as answer to the customer (line 5). Otherwise $n \neq 0$, and `fact` spawns an actor with behavior `customer-actor` (line 6), passing n and `fact`'s customer as arguments. Actor `fact` then sends itself a `compute` message to compute $(n-1)!$, passing along the newly created customer (line 8).

Actors with the `customer-actor` behavior multiply the number they receive by the number given at their creation, and send the result to the customer given at their creation (line 13).

Putting everything together, when `fact` receives the initial `compute` message with number n and customer `displayer`, it creates a customer `c` with n and `displayer` as arguments, and computes $(n-1)!$ by sending a `compute` message to itself with $n-1$ and `c` as arguments. When the computation of $(n-1)!$ is completed, the result is sent to customer `c`, which computes $n*(n-1)! = n!$, and sends the result to `displayer`, which displays the result on the screen.

Figure 1 depicts the evolution of the actor topology throughout the execution of this program. We see that the `fact` actor creates a chain of customers that propagate and multiply a value until the computed factorial reaches the `displayer` actor.

```
1  (define fact-actor
2    (actor ()
3      (compute (n customer)
4        (if (= n 0)
5          (send customer result 1)
6          (let ((c (create customer-actor
7                           n customer)))
8            (send self compute (- n 1) c)))
9        (become fact-actor))))
10 (define customer-actor
11   (actor (n customer)
12     (result (k)
13       (send customer result (* n k))
14       (become customer-actor n customer))))
15 (define displayer-actor
16   (actor ()
17     (result (v)
18       (display v)
19       (become displayer-actor))))
20 (define fact (create fact-actor))
21 (define displayer (create displayer-actor))
22 (send fact compute (read-integer) displayer)
```

Listing 1: Program computing the factorial of a user-given number with actors, adapted from Agha [1].

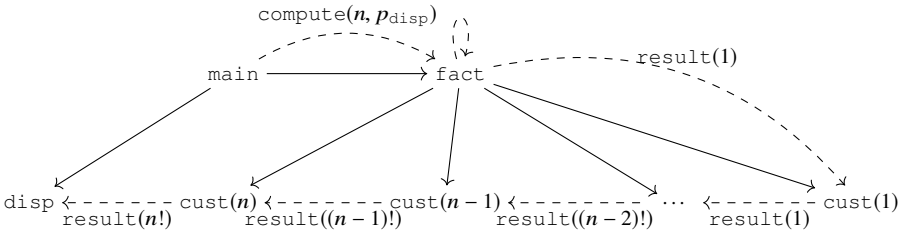

Fig. 1: Evolution of the actor topology during the execution of the factorial program in Listing 1. Plain edges represent actor creation, and dashed edges represent message sends. When annotated, the dashed edges indicate the message being sent. Actors denoted by `disp` and `cust` respectively denote actors with behaviors `displayer-actor` and `customer-actor`. The dots in the chain of `cust` actors represent a number of actors that depend on the value of n.

Despite its relative simplicity, this factorial program poses multiple challenges to static analyses due to its dynamicity. First, the number of actors created is not fixed nor known in advance. It depends on user input: n instances of the `customer` actor are created. Second, any actor can create other actors: `customer` actors are created in the `fact` actor, while the `displayer` actor is created from the main process. Third, actors exchange messages that contain actor references: the `compute` message is sent along with a reference to a `customer`. In general, actor programs can exchange messages that contain primitives such as integers or non-primitive values such as arrays and closures. Data flow and control flow information is required to reason about these values.

III. Approximating Dynamic Behavior

We give here a high-level overview of our modular analysis on the factorial program of Listing 1. Our approach is devised in an *inter*-actor analysis and an *intra*-actor analysis. Each iteration of the inter-actor analysis performs one intra-actor analysis on every actor that has been discovered. The intra-actor analysis infers the set of messages being sent and the set of actors being created, which are used in the next iteration. This process continues until a fixed point is reached, i.e., when no new message sends nor actor creations are discovered.

We describe the 4 iterations of the inter-actor analysis on the factorial program in Listing 1. The information computed by the analysis is summarized in Table I.

Iteration 1.

At the start of the first iteration no actor has been created yet, so only the main process is analyzed. Because the main process does not receive any messages, it can be fully analyzed in this first iteration. The intra-actor analysis deduces that the main process performs the following actions:

- it creates two actors: one with process identifier $p_{\texttt{fact}}$ and behavior `fact-actor` (line 20), and one with process identifier $p_{\texttt{disp}}$ and behavior `displayer-actor` (line 21), and
- it sends a message with tag `compute` and two arguments (an integer and $p_{\texttt{disp}}$) to $p_{\texttt{fact}}$ (line 22).

Iteration 2.

The main process does not have to be analyzed in this iteration, because it was already fully analyzed and cannot receive messages. As a message was sent to a newly created actor $p_{\texttt{fact}}$ in the previous iteration, this iteration now analyzes the code of this actor with behavior `fact-actor` and a mailbox containing the message `compute`(Int, $p_{\texttt{disp}}$). The body of the processor for message `compute` is analyzed with its parameters bound to the given values, and the intra-actor analysis of actor $p_{\texttt{fact}}$ infers the following actions:

- send the message `result`(Int) to `customer` (bound to $p_{\texttt{disp}}$),
- create a new actor with process identifier $p_{\texttt{cust}}$ and behavior `customer-actor`, and
- send the message `compute`(Int, $p_{\texttt{cust}}$) to itself.

This second iteration also has to analyze $p_{\texttt{disp}}$ created in the previous iteration. Because its mailbox is empty, no code can be executed in this actor and the analysis trivially finds no new effects. This trivial analysis is omitted from Table I.

Iteration 3.

The third iteration re-analyzes actor $p_{\texttt{fact}}$ because this actor

#	Pid	Behavior	Message received	Messages sent	Actors created
1	p_{main}	**main**	–	($p_{\texttt{fact}}$, compute(*Int*, $p_{\texttt{disp}}$))	($p_{\texttt{fact}}$, fact()) ($p_{\texttt{disp}}$, disp())
2	$p_{\texttt{fact}}$	fact()	compute(*Int*, $p_{\texttt{disp}}$)	($p_{\texttt{disp}}$, result(*Int*)) ($p_{\texttt{fact}}$, compute(*Int*, $p_{\texttt{cust}}$))	($p_{\texttt{cust}}$, cust(*Int*, p_{disp}))
3	$p_{\texttt{fact}}$ $p_{\texttt{disp}}$	fact() disp()	compute(*Int*, $p_{\texttt{cust}}$) result(*Int*)	($p_{\texttt{cust}}$, result(*Int*)) –	($p_{\texttt{cust}}$, cust(*Int*, $p_{\texttt{cust}}$)) –
4	$p_{\texttt{cust}}$ $p_{\texttt{cust}}$	cust(*Int*, $p_{\texttt{cust}}$) cust(*Int*, $p_{\texttt{disp}}$)	result(*Int*) result(*Int*)	($p_{\texttt{cust}}$, result(*Int*)) ($p_{\texttt{disp}}$, result(*Int*))	– –

Table I: Results of analyzing the program from Listing 1. The # column denote the inter-actor analysis iteration. Columns *Pid* and *Behavior* denote the process identifier under analysis with the given behavior. Column *Message received* denotes the message for which the actor is analyzed, and columns *Messages sent* and *Actors created* give the effects inferred by the intra-actor analysis.

received a new message compute(*Int*, $p_{\texttt{cust}}$). The behavior inferred by the intra-actor analysis is now more complete than in the previous iteration as the following additional behavior is discovered:

- send the message result(*Int*) to $p_{\texttt{cust}}$,
- create a new actor with process identifier $p_{\texttt{cust}}$ and behavior customer-actor(*Int*, $p_{\texttt{cust}}$).

The third iteration also has to analyze $p_{\texttt{cust}}$ with an empty mailbox, and trivially finds no new effects.

Iteration 4.

In the fourth and final iteration, $p_{\texttt{fact}}$ is not re-analyzed as no new effects affecting this actor have been discovered. Therefore, the analysis has reached a fixed point for this actor.

Actor $p_{\texttt{cust}}$ does have to be re-analyzed with a non-empty mailbox containing result(*Int*), and with two possible behaviors: customer-actor(*Int*, $p_{\texttt{cust}}$) and customer-actor(*Int*, $p_{\texttt{disp}}$). As two distinct behaviors can process the message, the intra-actor analysis also infers two possible message sends.

The newly sent message result(*Int*) has already been taken into account for $p_{\texttt{cust}}$ and $p_{\texttt{disp}}$ in the previous iteration, hence the analysis has reached a fixed point and no additional intra-actor analyses will be triggered.

Analysis result.

After reaching a fixed point, the analysis has discovered every actor that is created and every message that is sent to or received by each actor, with information about the values contained in the messages. Figure 2 depicts the corresponding actor topology that has been inferred, with the abstract actor states and abstract message values as expected. Note that the analysis has merged the unbounded chain of customers into a single abstract actor.

IV. Experimental evaluation

To evaluate our approach, we implemented the analysis in Scala on top of the Scala-AM static analysis framework [5]. Our implementation is available online[1]. We also implemented a concrete interpreter for the input language in Racket. This enables comparing the results of the analysis with the results of actually running a λ_α program.

[1]https://github.com/acieroid/scala-am/tree/modularactors

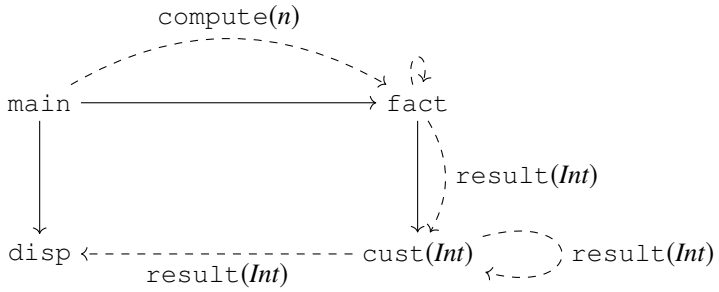

Fig. 2: An over-approximation of the actor topology's evolution, as inferred by the analysis. Actors denoted by disp and cust respectively denote actors with behaviors displayer-actor and customer-actor.

For our experiments, we translated all of the Savina benchmark programs into our Scheme-based input language. The original Savina benchmark programs, written in Scala (with fragments in Java), range from 102 to 616 lines of code. After translation into the input language, they range from 17 to 293 lines of code. Benchmarks of this size are beyond what is currently supported by existing static analyses. To support that claim, we compare our modular analysis with a non-modular analysis on the same set of benchmark programs and report on the results.

We ran each benchmark 1000 times with our concrete interpreter, and recorded every actor that is created, every become statement that is executed, and every message that is received. We then compared these observations with the information computed by the analysis. This enables the soundness and precision of our implementation to be respectively verified and measured empirically.

A. Performance

To assess performance of our modular analysis, we analyzed each Savina benchmark program two different analyses.

1) Our implementation of the modular analysis presented here.
2) Our previous non-modular analysis Mba [4], which provides several mailbox abstractions that vary in precision.

We used MBA_{Set}, that is MBA with the least precise, set-based, mailbox abstraction, to match the mailbox abstraction used by our modular approach.

Both analyses were executed under Scala 2.12.2 using Java 1.8.0_102 on a Mid-2014 MacBook Pro with a 2.8 GHz Intel Core i7 and 16 GB of RAM, and we report on the average timing of 20 runs after 10 warmup runs. We employed a timeout of two minutes, after which we denote an analysis time as infinite (∞). If the analysis completes under 0.01 second, we denote the analysis time as ϵ.

Table II lists the results of our experiments, which give a general idea of the scalability and precision of each analysis. Although the analyses differ slightly on what is computed, it is clear that our modular technique scales to the entirety of the Savina benchmark suite, analyzing it in a matter of seconds.

			Modular		Non-modular	
Benchmark	LOC	Observed	t	Spurious	t	Spurious
PP	27	9	0.03	0	0.11	0
COUNT	29	8	0.02	0	0.09	0
FJT	38	3	0.02	0	0.03	0
FJC	17	2	ϵ	0	ϵ	0
THR	43	5	0.02	2	∞	–
CHAM	81	10	0.05	0	∞	–
BIG	52	10	0.06	2	∞	–
CDICT	67	13	0.09	1	21.25	1
CSLL	61	16	0.08	1	∞	–
PCBB	98	13	0.66	0	102.99	0
PHIL	58	13	0.05	0	∞	–
SBAR	77	19	0.08	1	∞	–
CIG	49	9	0.05	1	1.40	1
LOGM	106	15	0.11	7	∞	–
BTX	61	9	0.08	0	2.50	0
RSORT	60	10	0.03	0	12.98	0
FBANK	143	38	0.15	0	∞	–
SIEVE	37	8	0.03	0	0.09	0
UCT	145	20	0.85	11	∞	–
OFL	293	13	5.75	0	∞	–
TRAPR	72	7	0.08	0	10.24	0
PIPREC	74	8	0.05	0	0.38	0
RMM	113	14	0.45	0	∞	–
QSORT	69	6	1.41	0	∞	–
APSP	188	5	1.06	1	∞	–
SOR	201	12	2.39	31	∞	–
ASTAR	92	11	0.26	0	∞	–
NQN	106	11	0.55	0	∞	–
Fully analyzed			28/28		12/28	

Table II: Scalability of our modular analysis and a non-modular analysis. The LOC column indicate the length of the benchmarks a Scheme-based input language. Column *Observed* reports the number of different elements observed at run-time (messages received, `become` statements, actors created) in 1000 runs of a benchmark program. For each analysis, we list the time taken to analyze each benchmark; and the number of spurious elements that have been reported by the analysis. Timings are in seconds where ϵ represents a time strictly smaller than 0.01s and ∞ a time out, which was set to two minutes (120s). The last row shows how many benchmarks could be analyzed by each analysis within the given time limit.

B. Soundness testing

We provide empirical evidence for the soundness of our implementation through *soundness testing* [2]. To this end, we verified that all information recorded during the concrete runs of each benchmark program is indeed over-approximated by the analysis of the same program. No unsound results were reported for any of the benchmarks, i.e., the analysis implementation over-approximated every value that was observed during the concrete executions.

C. Precision

Similar to the empirical verification of soundness, we measured the precision of the analysis. We compared the maximum-precision abstraction of the observed values in concrete runs of the program with the abstract values computed by the analysis.

Table II reports on the number of *spurious* values computed by the analysis. These are values that do not correspond to any value observed during any concrete execution. We also produced the same evaluation for our previous non-modular technique [4], and report on the spurious elements computed in benchmark programs for which the analysis does not time out. We observe that on these benchmarks, the precision of both techniques is identical: the same number of spurious elements appears in all benchmarks supportedby the non-modular technique. Overall, the precision of our modular analysis on the Savina benchmark suite is 84% (317 true positives and 61 false positives).

The reported numbers are merely an upper-bound on the number of potential false positives, as they correspond to elements that have not been observed in any concrete executions but that may be present in non-explored executions.

V. Conclusion

This paper presents a modular static analysis for concurrent actor programs that scales to all programs in the Savina benchmark suite. Relying on a sequential analysis to perform intra-actor analysis, we explain how a complementary inter-actor analysis uses the effects inferred by the intra-actor analysis to run the intra-actor analysis on other actors. Eventually, each actor is analyzed with an over-approximation of its mailbox, and the whole inter-actor analysis terminates. This yields a sound analysis that does not explicitely model every possible interleaving and, unlike existing techniques, scales well.

We evaluated our modular analysis through its implementation on top of an existing static analysis framework and show that, unlike non-modular approaches, it supports the entire Savina benchmark suite. The modular analysis generates few spurious elements and achieves a precision of 84% with respect to detecting messages sends, actor creations, and executions of `become` statements.

In summary, this paper demonstrates that it is possible to construct a scalable, sound, and precise analysis for actor programs. This is achieved by taking advantage of the fact that actor programs are made of actors that only interfere through messages. By relying on a sequential analysis for single actors, we are able to build a modular, scalable analysis for actor programs.

REFERENCES

[1] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1986.

[2] E. S. Andreasen, A. Møller, and B. B. Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 31–36. ACM, 2017.

[3] S. M. Imam and V. Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In E. G. Boix, P. Haller, A. Ricci, and C. Varela, editors, *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, pages 67–80. ACM, 2014.

[4] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover. Mailbox abstractions for static analysis of actor programs. In P. Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 25:1–25:30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[5] Q. Stiévenart, M. Vandercammen, W. De Meuter, and C. De Roover. Scala-am: A modular static analysis framework. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 85–90. IEEE Computer Society, 2016.