Towards Verification of a Service Orchestration Language

Tian Huat Tan
School of Computing,
National University of Singapore
{tianhuat}@comp.nus.edu.sg

***Abstract*—Recently, *Orc* is proposed as a powerful yet elegant language for distributed and concurrent programming which provides computational services such as distributed communication and data manipulation via sites. With a few concurrency primitives, programmers are able to orchestrate the invocation of sites to achieve a goal, and meanwhile, manage timeouts, priorities, and failures. To guarantee the correctness of *Orc* models, effective verification support is desirable. In this work, we present an automatic approach to verify different properties against *Orc* models using model checking techniques. To further improve the performance, advanced reduction techniques, like partial order reduction, are proposed.**

***Keywords*-Orc; Orchestration; Model Checking**

I. Introduction

Concurrency is an essential problem of the current world with the advent of multi-core and multi-CPU systems. However, it is not an easy task for programmers to maximize the benefit of concurrency, since programmers are burdened with the task of managing the threads and locks explicitly. *Orc* calculus [5] is designed to express orchestrations and wide-area computations in a simple and structured manner, and to address the problems mentioned above.

In this work, our approach contributes to providing direct verification for *Orc* language. *Orc* is developed under PAT framework [7]. Figure 1 shows the workflow of our approach. First, users can specify *Orc* models as well as various assertion properties via the editor. The input models are compiled into internal representations (i.e. LTS), based on the operational semantics [11], [4], [3]. Linear Temporal Logics (LTL) assertions are subsequently translated into Büchi automata. Users can visualize the system behaviors via an animated simulator, or perform verification using different verifiers.

II. Orchestration Language Orc

The fundamental of *Orc* calculus is the execution of expressions, which are built up recursively with concurrent combinators. During execution, an expression may call sites (i.e. external services) or publish values.

Sites are basic units of *Orc* language. They are considered as external services. A site can be an unreliable remote service (e.g. Google), or a predictable and well-defined local service (e.g. if). Henceforth, we will refer the former as remote site and latter as local site. The simplest *Orc* expression is a site call $M(\overline{p})$, where M is the service's name and $\overline{p}$ is a list of parameters.

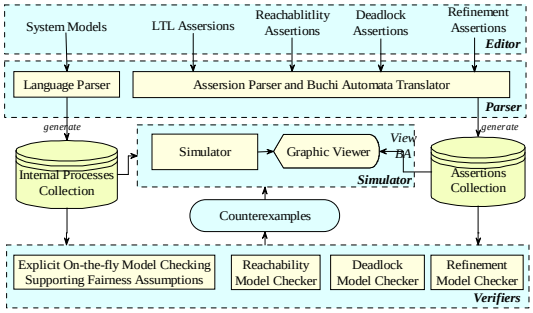

Fig. 1. System Work Flow

Combinators are used to combine various expressions. There are four combinators: parallel, sequential, pruning, and otherwise combinators. The parallel combinator $F \mid G$ defines a parallel expression, where expressions F and G execute independently. The sequential combinator $F > x > G$ defines a sequential expression, where each value published by F initiates a separate execution of G wherein x is bound to it. The execution of F is then continued in parallel with all these executions of G. The pruning combinator $F < x < G$ defines a pruning expression, where initially F and G execute in parallel. However, when F needs the value of x it will be blocked until G publishes a value to bind x and G terminates immediately after that. The otherwise combinator $F \,;\, G$ defines an otherwise expression, where F executes first. The execution of F is replaced by G if F halts without any published value. F halts if all site calls are responded or halted, and it doesn't publish any more value or call any more site.

We refer readers to [1] for detail introduction on Orc language, including various site definitions, and functional core language that have been added to *Orc*.

III. Verification

Now we present our approach in verifying *Orc*. We support all four combinators, functional core language, and a subset of local sites at the first stage. Behaviors of local sites are predefined and predictable. For non-timer sites, we support fundamental sites, such as *Ref*, *Buffer*, *SyncChannel*, and *Semaphore*. For these four sites, it is assumed that each call of them responses immediately with a predictable value v. For timer site, we support *Rtimer*, and it is assumed that it responses a *signal* value after t time units [11], where *signal* is a return value which carries no information.

A. Time constraint

In the expression $Rtimer(500) \gg$ "x" $\mid Rtimer(600) \gg$ "y", $Rtimer(500)$ should return before $Rtimer(600)$. In order to resolve time constraint, we use an approach similar to [9]. A clock variable will be assigned to each *Rtimer* when it is waiting for site returned value. In order to reduce the number of clock variables, the clock variable is created if necessary and pruned when it is not needed. To resolve the time constraint of active clocks, techniques based on different bound matrix (DBM) are used.

B. Deadlock-freeness

A deadlock occurs when the entire system is in terminal state, although some of the local processes might not be terminated. To verify whether a system is deadlock, we perform a depth-first search on the state-space for terminal state.

C. Reachability

A safety property is a property that states "bad thing will never occur". For example $\sim (cs0 \wedge cs1)$ specified that cs0 and cs1 cannot be both true at the same time. To verify a safety property p, we can negate the property $\sim p$, and check whether system reaches $\sim p$, i.e. the system is possible to enter into a state that satisfies property $\sim p$. If the system reaches $\sim p$, then the safety property is violated. A depth first search is performed on the state space for state that violates the property.

D. Linear Temporal Logic

A liveness property stating "something good will eventually happened". To model the liveness properties, Linear Temporal Logic (LTL) [6] is normally used.

Algorithms based on automata theoretic approach are used to avoid construction of the entire state-space. This tactic is called on-the-fly model checking [2].

E. Timed Refinement

In time-critical orchestrations, properties such as whether a particular process will be invoked at t time-units after a certain event e are crucial. We verify such properties by means of timed refinement checking. The idea is to construct a specification written in *Orc*, and to check whether the application to be verified conforms to the behavior of the specification including real-time constraints. For the time refinement algorithms, we are using an approach similar to [8].

IV. Reduction Techniques

In our work, we have discussed various aspects of verification, with the use of respective state-of-the-art verification algorithms. However, in verification of *Orc*, the main difficulty is the state-explosion problem. The operational semantics of *Orc* allows the execution to run in parallel in many places. For example, in pruning combinator $A < x < B$, parallel combinator $A \mid B$, and site call operators with two arguments $M(A, B)$, expression A and B can be run in parallel. This also implies that simple functional expressions like *if E then F else G* would need to run in parallel since it is translated into mixed of pruning and parallel expressions, ie. $(if(b) \gg F \mid if(\sim b) \gg G) < b < E$, before evaluation. Furthermore, when a function call is executed the function body and argument expressions are executed in parallel. Parallelism in *Orc* has made states grow exponentially and thus posed a challenge for its formal verification. Thus state reduction becomes a crucial task for verification of model.

Fortunately, due to the functional programming flavor of *Orc*, we can expect there would exist many independent execution, and they can be easily abstracted away using partial order reduction. This can be done by actively analyzing the structure of *Orc* program to understand whether it has made use of sites that support mutable storage (e.g. *Ref* site) or time (e.g. *Rtimer*), and handle them accordingly. Furthermore, it is frequent that behaviorally similar processes are run in parallel, thus we can use symmetry reduction or even process counter abstraction [10] to yield a smaller state space for verification. The general idea is grouping behaviorally similar processes while not keeping the process identifer. Lastly, there is a list of identities in *Orc* language that have been proved using strong bisimulation [12]. This identities are exploited for state reduction as well.

V. Conclusion

In this paper, we present our approach in direct verification of *Orc*. We believe this will help in improving the overall quality of the *Orc* language implementations. We are working on ways to further reduce the state space of the model. We would also consider to support verification of more local sites and include remote sites in the near future.

References

[1] Orc Language. http://orc.csres.utexas.edu/.

[2] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engeering*, 23(5):279–295, 1997.

[3] D. Kitchin. Operational and denotational semantics of the otherwise combinator. 2009.

[4] D. Kitchin, W. Cook, and J. Misra. A language for task orchestration and its semantic properties. pages 477–491. 2006.

[5] D. Kitchin, A. Quark, W. Cook, and J. Misra. The Orc programming language. In *FMOODS/FORTE*, pages 1–25, 2009.

[6] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[7] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 2009*, pages 702–708, Grenoble, France, June 2009.

[8] J. Sun, Y. Liu, J. S. Dong, F. Wang, M. C. Zhen, and L. A. Tuan. Verifying compositional safety critical systems by timed language inclusion checking. Technical report. http://www.comp.nus.edu.sg/~pat/cavreport.pdf.

[9] J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying stateful timed csp using implicit clocks and zone abstraction. In *ICFEM 2009*, pages 581–600. Springer, 2009.

[10] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking with process counter abstraction. In *FM'09*, pages 123–139. Springer, 2009.

[11] I. Wehrman, D. Kitchin, W. Cook, and J. Misra. A timed semantics of orc. *Theoretical Computer Science*, 402(2-3):234–248, August 2008.

[12] I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. Properties of the timed operational and denotational semantics of orc. Technical Report TR-07-65, The University of Texas at Austin, Department of Computer Sciences, December 2007.