Fair Termination for Resource-Aware Active Objects

Francesco Dagnino[2], Paola Giannini[3], Violet Ka I Pun[1], and Ulises Torrella[1]

[1] Western Norway University of Applied Sciences
[2] University of Genoa
[3] University of Eastern Piedmont

Active object systems [13, 14, 6] are the object-oriented instantiation of the actor model [2]. They provide a useful abstraction of distributed systems with asynchronous communications, which are represented as collections of objects (actors) interacting through asynchronous method calls. This means that a method invocation corresponds to sending a message to the receiver object that will eventually handle it, by running the method body. Thus, the invocation does not block the execution of the caller, but immediately returns a future, which can be subsequently used for synchronizing with the receiver and accessing the result of the call.

Among other applications, this model provides the formal basis for workflow modelling and analysis [3, 4], where models capture the behaviour of the internal (resource-sensitive) workflows of organisations. Workflows are processes that handle business cases and are primarily demanded to be terminating, hence resolve the business case (a customer order, a service ticket, etc.). In particular, the modelling language proposed in [3] is based on the ABS language [14], an active object language with multithreaded actors and a future-based cooperative scheduling paradigm. This means that each actor can handle multiple messages at a time, by explicitly yielding control on future **await** statements.

Besides this interaction mechanism, a workflow modelling language also needs to take into account *passive/informational resources* [19]. These resources move through processes while performing a workflow, undergo transformations, can be created or destroyed and crucially have a *limited availability* according to the specification domain. The interplay of asynchronous message passing, cooperative multithreading and resource management makes it challenging to ensure that a system can complete its task. For instance, if a thread tries to access a resource that is not available, it remains stuck as it cannot yield control, thus preventing the whole system from successfully terminating.

The contributions of this work are the development of a core calculus for resource-aware active objects together with a type system ensuring that well-typed programs are *fairly terminating* [9], that is, all their fair executions terminate, under a suitable fairness assumption. To achieve this, we combine techniques from graded semantics and type systems [7, 1, 8, 15, 5, 18], which are quite well understood for sequential programs, with those for fair termination [10, 11, 9, 12] , which have been developed for synchronous sessions.

More in detail, we model resources as *graded constants* r^g where r is a name identifying the resource and g is a grade describing the availability of the resource and thus constraining its usage. For instance, a resource can be used a fixed number of times or in a private or public mode. In our calculus, each actor owns a resource environment ρ, containing graded resources which can be accessed by threads of that actor. The language provides constructs through which a thread of an actor can hold and release resources from its resource environment. Notably, the reduction of a **hold** statement asking for r^g, i.e., "g copies" of the resource r, is stuck if the amount, i.e., the grade, of the resource r in the environment ρ of the actor is not enough to produce r^g. Finally, it is important to note that the introduction of graded resources has an impact also on the synchronization mechanism. Indeed, typically futures can be accessed an arbitrary number of times [17, 16, 14]. However, this is not the case in our setting because futures may contain graded resources and so, by copying the future, we would copy its content

as well, leading to a violation of the constraint on the resource usage expressed by the grade. To overcome this issue, we treat futures *linearly*, allowing for them to be read only once.

In order to ensure the correct use of resources, we endow our calculus with a graded type system [7, 1, 8, 15, 5, 18]. Besides basic types, we have graded types for resources and future types. Then, the typing judgment for expressions has the following shape: $\Phi;\ \Sigma;\ \Gamma \vdash e : T; \Phi'$, where the resource context Φ tracks the resource requirements the expression poses on each actor and it is handled like a graded context, the future context Σ tracks, in a linear way, the futures that the expression will read, and an almost standard variable context Γ where each variable is handled in a linear, graded or unrestricted way depending on its type. An expression is assigned a type T and a "release context" Φ' with the resources that the expression will release into the system. This judgment allows us to "chain" resource production and consumption in the sequential composition, enabling a form of reuse of resources, and also expose this information on the method type. Thus, given that the body of a method is an expression, its return type will be T, corresponding to the return value of the body, plus the resources released into the system Φ'. Then, on account of all methods being asynchronous, the type of a method call will be a Future $\mathbf{Fut}\langle T, \Phi'\rangle$. Hence, an **await** expression on a future of such type will have type T and release context Φ', so that the resources released by the method call can be reused by the process accessing its result.

Futures are the lone communication mechanism of processes. To avoid circular dependencies on futures we enforce a left to right future dependency on configurations by typing processes with judgments of shape: $\Phi;\ \Sigma \vdash P :: \Sigma'$, where there is a left-hand side resource context Φ for the required resources of the process, a left-hand side future context Σ for used futures, and a right-hand side future context Σ' for the produced futures. Processes are composed by parallel composition, whose typing rule sums up the resource contexts of the two parallel processes and checks that the process on the right only reads futures produced by the process on the left, thus ensuring that the dependency graph on futures is acyclic.

The main result of this work is a proof that well-typed configurations are fairly terminating. This is achieved applying a proof technique from [9] ensuring that to obtain fair termination it is enough to prove the standard subject reduction and weak termination of well-typed configurations. The latter means that every well-typed configuration admits a terminating execution. Note that, since our calculus supports a non-deterministic choice operator, this property does not forbid non-termination, but it ensures that termination is always possible. To prove this result, following [9], we annotate typing judgments with a measure, taken from a well-founded poset, and prove that there is always a reduction step making such measure decrease. The proof of subject reduction poses some challenges as well. Indeed, it does not hold for standard graded semantics and type systems [8, 5] because resource consumption in graded semantics usually is non-deterministic, hence only a form of "may subject reduction" can be proved, where well-typedness after a step is not guaranteed. Therefore, in order to recover subject reduction, we need to define a semantics where resources are consumed in a deterministic way and this requires an extension of the usual algebraic structure of grades adopted in the literature. Finally, by fair termination we guarantee that every well-typed system can always successfully terminate and so it is resource safe and never stuck.

This work is a first step towards integrating resource-awareness by grading in active objects systems, ensuring strong behavioural properties like fair termination. Notably, we would like to relax the linearity constraint on futures by treating them in a graded way as well. Moreover, it would be interesting to investigate a system where threads can yield control also on **hold** statements, so that they could be paused until the required resources are available, thus reducing the possibility for an actor of being stuck.

References

[1] Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proc. ACM Program. Lang.*, 4(ICFP):90:1–90:28, 2020.

[2] Gul A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems (Parallel Processing, Semantics, Open, Programming Languages, Artificial Intelligence)*. PhD thesis, University of Michigan, USA, 1985.

[3] Muhammad Rizwan Ali, Yngve Lamo, and Violet Ka I Pun. Cost analysis for a resource sensitive workflow modelling language. *Sci. Comput. Program.*, 225:102896, 2023.

[4] Muhammad Rizwan Ali, Violet Ka I Pun, and Guillermo Román-Díez. Easyrpl: A web-based tool for modelling and analysis of cross-organisational workflows. *CoRR*, abs/2502.20972, 2025.

[5] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. Resource-aware soundness for big-step semantics. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1281–1309, 2023.

[6] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.

[7] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014.

[8] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021.

[9] Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. *J. Log. Algebraic Methods Program.*, 139:100964, 2024.

[10] Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022.

[11] Luca Ciccone and Luca Padovani. An infinitary proof theory of linear logic ensuring fair termination in the linear π-calculus. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[12] Francesco Dagnino and Luca Padovani. small caps: An infinitary linear logic for a calculus of pure sessions. In Alessandro Bruni, Alberto Momigliano, Matteo Pradella, Matteo Rossi, and James Cheney, editors, *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, Milano, Italy, September 9-11, 2024*, pages 4:1–4:13. ACM, 2024.

[13] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017.

[14] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*,

volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.

[15] Ugo Dal Lago and Francesco Gavazzo. A relational theory of effects and coeffects. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022.

[16] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

[17] Siva Somayyajula and Frank Pfenning. Type-based termination for futures. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, volume 228 of *LIPIcs*, pages 12:1–12:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[18] Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. Effects and coeffects in call-by-push-value. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1108–1134, 2024.

[19] Michael zur Muehlen. Organizational management in workflow applications - issues and perspectives. *Inf. Technol. Manag.*, 5(3-4):271–291, 2004.