Criteria-Based Merging of Dynamic CAL Actors

Florian Krebs and Klaus Schneider
Department of Computer Science
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany
{florian.krebs, klaus.schneider}@rptu.de

Abstract—**The dataflow model of computation is well-established in many application areas, including image encoding and decoding. This model consists of actors that process data and communication channels that transmit data between actors. However, when mapping these models to the systems on which they will be executed and synthesizing the necessary code, the number of actors often exceeds the number of execution units by a significant amount. This results in increased overhead for data buffering and communication between actors. To address this issue and adapt the model to the target execution system, we propose a technique that combines static and dynamic actors defined in the CAL actor language. This technique is based on actors' composition within the dataflow network and creates a new composite CAL actor. When merging actors, we consider the actions of the actors, including their CAL scheduling mechanisms, finite state machines, priorities, and guards. We derive conditions under which connected actors can be merged to form a new CAL-compliant actor. Preliminary experiments show performance improvements of up to 3.6x compared to unoptimized code.**

Keywords — Dataflow Model of Computation, Composition, Actor Merge, Model Optimization

I. Introduction

Using models is a well-established method for managing the inherent complexity of large systems. Such models can be created at a higher level of abstraction, which simplifies the development process. With model-based approaches, software is designed modularly rather than being developed directly as a target-specific implementation. In particular, the dataflow model of computation (MoC) offers these benefits and enables the separation of different components and the utilization of parallelism. This model is based on directed graphs, where the edges represent unidirectional, point-to-point communication channels of infinite size that transmit tokens, and the nodes represent data-processing elements, called "actors". An actor can contain several actions, that it can execute, called "firing", based on the availability of tokens, token values, the actions' scheduling conditions, or by randomly selecting one of the actions. An action can consume an arbitrary but fixed number of tokens from the input channels and produce an arbitrary but fixed number of tokens for the output channels. The number of tokens consumed and produced does not need to be the same for each channel and can be zero.

Due to the modular nature and visualization capabilities of dataflow models, they are well suited to embedded software design, such as signal or image processing. For example, MPEG codecs are dataflow models [1]. However, when it comes to code synthesis and mapping these models to real execution units, their modular structure may not align with the chosen hardware architecture. In particular, a model may have many more actors than processing elements. Although the communication channels allow for the distribution of actors across processing elements, they can also cause significant performance overhead. Mapping actors that exchange a lot of data to the same processing element can reduce this effect but cannot eliminate it completely. Consequently, methods are required to merge actors executed by the same processing element for the synthesis of efficient software.

Two popular classes of dataflow models are Dataflow Process Networks (DPNs) [2] and Static Dataflow (SDF) [3]. Unlike SDF, which has constant token rates across all firings, DPNs do not constrain their actors' (or processes') behavior. Each actor can consume and produce different numbers of tokens at any given time, a property known as dynamic behavior. Since the token flow is unknown at compile time, scheduling dynamic dataflow actors necessitates runtime scheduling.

A. Cal Actor Language

The Cal Actor Language (CAL) [4] is a domain-specific language for specifying dataflow actors. It provides a comprehensive set of features for this purpose. Here, the focus is on a popular subset of CAL that is sufficient for specifying dataflow actors that conform to the basic definitions of the aforementioned dataflow models. Fig. 1 shows an example of a CAL dataflow actor.

An actor can be instantiated multiple times within the same model. These instances are referred to as an actor instances. A defined actor has a certain number of input and output ports separated by "==>" (line 1); these ports are connected to communication channels within the overall model definition. Actions can access these ports, such as action *a* (line 2), which reads a token from input port *x* (input pattern) and writes a token to each of the output ports (output expression). An action can consume and produce multiple tokens. For the sake of brevity, action bodies that can perform more complex operations known from common programming languages are omitted here. Guards (line 5) define additional conditions that must be met for the corresponding action to fire. The schedule FSM (lines 5-9) and priorities (line 10) can further specify the scheduling for the given actor. The FSM defines the initial state (line 5) *–s_one*

Manuscript received July 23, 2025; revised September 24, 2025; accepted September 9, 2025. Published October 14, 2025.
Issue category: Regular
Paper category: Short
DOI: doi.org/10.64552/wipiec.v11i2.81

```
1   actor example()   uint in ==> uint out1, uint out2 :
2         a: action in:[x] ==> out1:[x+1], out2:[x] end
3         b: action in:[x] ==> out1:[x+2], out2:[x+1]
4                   guard x > 5 end
5         schedule fsm s_one :
6                   s_one ( a ) --> s_two;
7                   s_two ( a ) --> s_one;
8                   s_two ( b ) --> s_two;
9         end
10        priority  b > a;  end
11  end
```

Figure 1 CAL Example

– and the state transitions. Each state transition lists the actions that can be fired in the given state and cause that state transition. Actions that are not listed cannot be fired in this state. Priorities define which action shall be fired in case several are ready to be fired, e.g., action *a* can only be fired if the scheduling conditions for action *b* are not fulfilled.

Consequently, in addition to the schedule FSM, priorities and guards must be considered for merging, otherwise the functionality of the actors might be lost or altered.

B. Contribution

In this paper, we present a methodology for merging CAL actors based on their composition in the dataflow model with the resulting actors again specified in CAL (Section III). This methodology considers guards, priorities, and the schedule FSM during merging. The generation of new CAL-compliant actors has the benefit of a seamless integration into existing toolchains for code synthesis. Since actions consume and produce fixed numbers of tokens and are bound to certain scheduling constraints, we derive concrete criteria for a successful (correct) CAL actor merging (Section II). In Section IV, we show preliminary results of executions time improvements achieved actor merging.

II. Actor Merging Criteria

Merging dataflow actors, especially dynamic ones, is not always possible in an arbitrary manner. In dynamic dataflow models with data-dependent actors, such as switches, the token flow is not known in advance, so it is impossible to determine the token rates.

In the following, we define criteria that the actor instances α_i in a connected subgraph of actor instances Z must meet in order to be merged into a single composite actor. For the purpose of this analysis, we define the set of channels connected to actors $\alpha_i \in Z$ as C, the set of channels with the source actor not in Z and the sink actor in Z as C_{in}, and the set of channels with the source actor in Z and the sink actor not in Z as C_{out}. Furthermore, we use the symbols $\mathrm{con}_{c_j}(a_i)$ and $\mathrm{prod}_{c_k}(a_i)$ to denote the consumption and production rate of an action $a_i \in \alpha_i$ for the corresponding channels $c_j, c_k \in C$.

Criterion 1 (Internal Buffering): The merging of actors must not buffer tokens within the composite actor across different firings. For each channel c_i between actors $\alpha_i, \alpha_j \in Z$ each combination of actions of $a_i \in \alpha_i$ and $a_j \in \alpha_j$ fired in sequence must satisfy $x * \mathrm{prod}_{c_i}(a_j) = y * \mathrm{con}_{c_i}(a_i)$ with $x, y \in N$.

If buffering of tokens within the composite actor is permitted, the boundedness of this buffering must be guaranteed. Additionally, during scheduling, the buffered tokens must be considered such that consumption of these tokens takes precedence over generation of new tokens for the same internal buffer. Therefore, buffering is avoided to reduce the complexity of the actor merging process and the resulting actor. The token rates of two adjacent actors to be merged must therefore match, and multiples are allowed. For multiples, loops must be added.

Criterion 2 (Scheduling Conditions): The guard conditions of the actions must be propagatable. A guard condition is propagatable, if it depends only on tokens consumed from channels C_{in} or internal state variables.

This criterion simplifies the generation of guard conditions for composite actors. Generating a guard condition that covers the actions of two actors whose guard conditions depend on each other's input tokens would involve propagating the guard condition back to the actor that produces the token. This is done by replacing the token value with the calculations that lead to the production of the token. Therefore, generating a guard condition for a large set of actors is not feasible since, in the worst case, the guard conditions of the composite actor already include most of its behavior, and it is not possible to consume or produce additional tokens based on dynamic behavior. Therefore, this criterion does not apply to static actors, which can make a scheduling decision without altering the dynamics.

Criterion 3 (Minimal Schedulability): Any token numbers in C_{in} that enables the actors in Z to produce tokens for C_{out} must also produce the same behavior in the composite actor.

If the actors in Z can produce tokens for a given set of input tokens in C_{in}, but the composite actor cannot, then other actors may require tokens that were previously produced by actors in Z to continue processing. However, the new composite actor may be unable to produce these tokens, for example, in the case of feedback cycles. If such a situation occurs, the merge could render the previously functional network unusable.

III. Actor Merging Methodology

The actor merging process consists of four steps: dataflow configuration enumeration, merge criteria checking, actor generation and network adjustment. The merge criteria cannot be checked in the first step because they only apply to valid token flows through Z. Therefore, they must first be detected.

After the enumeration step, the merge criteria are checked. Only then can the actual actor merging take place. This includes generating the composite actor, which includes a new schedule, finite state machine (FSM), priorities, and actions with guards. The final step is replacing the merged actors with the composite actor in the original model.

A. Dataflow Configuration Enumeration

A dataflow configuration is a relation between consumed input tokens from C_{in} and produced output tokens for C_{out} and the corresponding (actor instance, action)-pairs and their execution rate, executed to satisfy the input-output relation with respect to the actors internal scheduling states. Consequently, a dataflow configuration contains the necessary information to generate a new action, including the current and next FSM states in which these actions can be executed in.

The enumeration process involves first identifying the largest connected subgraphs of Z that do not contain actors with FSMs. These subgraphs are then pre-computed to facilitate the subsequent enumeration of all possible token flows. The actual enumeration starts with the initial state of all the actor instances, which are used as the starting point of the enumeration. Based on this state combination, all schedulable actions that consume only tokens from C_{in} are determined. For each valid combination of them a new dataflow configuration is created by iterating through Z in token flow order with respect to the produced tokens of already added (actor instance, action)-pairs. For each processed actor instance, add all schedulable actions. If adding actions results in different token flows, split the dataflow configuration and proceed with this procedure for both. Consequently, when processing a static actor, all its actions are added to the dataflow configuration, as the scheduling decision can be made at runtime without affecting the token flow. If one of the pre-computed clusters is discovered during iteration, add the entire cluster. For each detected dataflow configuration, identify the FSM state combinations and execution rates that prevent internal buffering. Then, proceed with the procedure for all unknown follow state combinations.

Precomputing the token flow through the connected subgraphs of Z that do not contribute to the combined state makes it possible to reuse these parts for generating every dataflow configuration. This reduces the effort required for this step, as enumeration can lead to exponential behavior in the worst case, which is the main driver of the runtime of the entire procedure.

B. Composite Actor Generation

The composite actor consists of the ports to connect to C_{in} and C_{out} and the state variables of the original actor instances. All identifier names in use are checked for possible collisions and renamed if necessary. The scheduling priorities, schedule FSM, and actions are generated based on the dataflow configurations.

A dataflow configuration is converted into an action by adding the contained actions' operations in token flow order. If a processed action consumes tokens from C_{in} or produces tokens for C_{out}, the input patterns and output expressions are added to the generated action. Otherwise they are converted to variable assignments and added to the action body. Guard conditions, must be propagatable and are therefore added to the guard condition of the generated action. For actor instances with an execution rate greater than one, a for loop is added.

The new FSM is generated by creating a state for each different schedule FSM state composition present in the dataflow configurations. The state transitions are generated based on the following state compositions, which are also stored for each dataflow configuration.

New scheduling priorities are generated for actions schedulable in the same FSM state based on the initial priority definitions. A new priority relation is added between two generated actions if one of them only consists of incorporated actions that have a higher or no priority assigned than the actions incorporated in the other generated action. At least one priority relation between the incorporated actions has to exist to create a priority relation between the generated actions. Transitive priority relations can be omitted.

IV. Evaluation

The main purpose of actor merging is to improve performance. To demonstrate the effectiveness of our merging procedure, we compare the performance of unmodified and optimized applications, both of which are generated from dataflow specifications. First, we merge all actors. If a merge is not feasible due to the criteria, we reduce the set until a feasible merge is found. This process repeats with the remaining unmerged actors until no feasible merges can be found. The merges performed serve only to demonstrate the functionality of the described approach and may not be optimal.

The selected applications are ZigBee Multitoken, the Digital Pre-Distortion (DPD), and an adaptive LMS filter from the Open RVC-CAL Applications repository[1], as well as the Fast Fourier Transformation (FFT)[2]. The channel size used is 512 in all cases. The preliminary experimental results are shown in Tab. 1.

The dataflow models were translated to C++ and compiled using MSVC version 19.40.33813 with the compile options /O2 and /Ob2. Only single-threaded code was created to eliminate scheduling and multi-core timing effects from our performance measurements as we focus on actor merging as optimization for actor clusters mapped to the same core. A Core i7-8550U with 16GiB of RAM running Windows 11 Pro 10.0.22631 was used for the performance measurements.

The ZigBee example uses FSMs, priorities, and guards in a dynamic dataflow scenario, but it did not demonstrate

[1] https://github.com/orcc/orc-apps

[2] http://www.averest.org/

TABLE I MEASURED EXECUTION TIMES FOR OPTIMIZED AND UNMODIFIED MODELS

Benchmark	Runtime [ms]		Speed Up
	Unmodified	*Optimized*	
ZigBee	15063	15042	0
LMS	2386	979	1.44
DPD	2449	871	1.81
FFT	14390	3070	3.69

significant performance improvement. Due to restrictions imposed by the merge criteria, only a small portion of the model could be merged. The remaining three examples—LMS, DPD, and FFT—show performance improvements ranging from 1.4x to 3.6x. These results demonstrate that merging dataflow actors is an effective optimization method.

I. RELATED WORK

Unlike the approach presented here, which allows for the inclusion of dynamic actors in the merging process, most actor merging approaches rely on SDF. Ali et al. [5] use (C)SDF and transform it into HSDF. Using a heuristic based on the worst-case execution time of the actors and their time constraints, they select and merge actors to reduce the graph. Falk et al. [6] present an approach in which static nodes are clustered to bind dataflow nodes to a specific processor in a multiprocessor system. These clusters are then used to create cluster finite state machines (FSMs) based on quasi-static schedules. In [7] and [8], a set of rules is specified to avoid state space enumeration when generating composite actors with quasi-static schedules. Janneck [9] introduces a formal actor model and its composition through composers. Composers generate different composite actors from the same set of actors. In [10], Janneck introduces actor machines, which are another representation of dataflow actors. The composition and computation of SDF schedules of actor machines is achieved through an abstract simulation that explores computation paths through the actor machines. Cedersjö and Janneck [11] use these actor machines and their composition to build a compiler for RVC-CAL dataflow networks, translating them into C code. Boutellier et al. [12] present actor merging for RVC-CAL dataflow networks, including dynamic actors, by creating a schedule FSM composed of the FSMs of the merged actors based on a chosen lead actor. Ersfolk et al. [13] use control tokens to model a schedule that makes most of its decisions at compile time. The resulting schedule can be used for actor composition.

II. CONCLUSION AND FUTURE WORK

Our preliminary results indicate that actor merging is a feasible optimization technique. However, the current approach is not beneficial in all cases. For instance, the ZigBee example did not yield any performance improvements despite [14] showing that improvements are possible. One reason for this is that the merging criteria are too restrictive, allowing only some actors to be merged. These criteria are driven by the property of fixed input and output token rates of actions. Therefore, the dynamics of actions without buffering tokens in state variables and using auxiliary actions to cover missing parts are limited. This is also why an extensive enumeration is required to find valid dataflow configurations. These two disadvantages make applying the current approach to a code synthesis toolchain for automatic optimization difficult. An approach based on intermediate representation transformations might better lift these restrictions. With such an approach, no enumeration is necessary, and the full dynamics of the target language can be utilized.

REFERENCES

[1] "Information technology — mpeg systems technologies — part 4: Codec configuration representation," International Organization for Standardization, Geneva, CH, Standard, 2017

[2] E. A. Lee and T. Parks, "Dataflow process networks," Proceedings of the IEEE, vol. 83, no. 5, pp. 773–801, May 1995

[3] E. A. Lee und D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987

[4] J. Eker and J. Janneck, "CAL language report," EECS Department, University of California at Berkeley, Berkeley, California, USA, ERL Technical Memo UCB/ERL M03/48, December 2003.

[5] H. I. Ali, S. Stuijk, B. Akesson, and L. M. Pinho, "Reducing the complexity of dataflow graphs using slack-based merging," *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, no. 2, pp. 24:1–24:22,2017, https://doi.org/10.1145/2956232

[6] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications," in *Embedded Software (EMSOFT)*, L. de Alfaro and J. Palsberg, Eds. Atlanta, Georgia, USA: ACM, 2008, pp.189–198.

[7] J. Falk, C. Zebelein, C. Haubelt, and J. Teich, "A rule-based static dataflow clustering algorithm for efficient embedded software synthesis," in *Design, Automation and Test in Europe (DATE)*.Grenoble, France: IEEE Computer Society, 2011, pp. 521–526.

[8] J. Falk, C. Zebelein, C. Haubelt, and J. Teich, A rule-based quasi-static scheduling approach for static islands in dynamic dataflow graphs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, pp. 74:1–74:31, 2013

[9] J. W. Janneck, "Actors and their composition," *Formal Aspects of Computing*, vol. 15, pp. 349–369, 2003.

[10] J. W. Janneck, "A machine model for dataflow actors and its applications," in *Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*. Pacific Grove, CA, USA: IEEE Computer Society, 2011, pp. 756–760.

[11] G. Cedersjö and J. Janneck, "Software code generation for dynamic dataflow programs," in *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, H. Corporaal and S. Stuijk, Eds. Sankt Goar, Germany: ACM, 2014, pp. 31–39.

[12] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén, "Actor merging for dataflow process networks," *IEEE Transactions on Signal Processing*, vol. 63, no. 10, pp. 2496–2508, May 2015

[13] J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli, "Modeling control tokens for composition of CAL actors," in *Design and Architectures for Signal and Image Processing (DASIP)*, P. Meloni and C. Jégo, Eds. Cagliari, Italy: IEEE Computer Society, 2013, pp. 71–78

[14] J. Boutellier, A. Ghazi, O. Silven, und J. Ersfolk, „High-performance programs by source-level merging of RVC-CAL dataflow actors", in *SiPS 2013 Proceedings*, 2013, S. 360–365.