Feature-Oriented FSMs for FPGAs

Justin Deters
Peyton Gozon
Max Camp-Oberhauser
Ron K. Cytron

Justin Deters, Peyton Gozon, Max Camp-Oberhauser, and Ron K. Cytron, "Feature-Oriented FSMs for FPGAs," in *Proc. of IEEE High-Performance Extreme Computing Conference* (HPEC), September 2023.
DOI: 10.1109/HPEC58863.2023.10363511

SimpleRose
St. Louis, MO

Dept. of Computer Science and Engineering
Washington University in St. Louis

Feature-Oriented FSMs for FPGAs

Justin Deters*, Peyton Gozon†, Max Camp–Oberhauser†, and Ron K. Cytron†
*SimpleRose
St. Louis, MO, USA
justin@simplerose.com
†Department of Computer Science and Engineering
Washington University
St. Louis, MO, USA
{peyton.gozon, c.max, cytron}@wustl.edu

***Abstract*—In this paper we consider a feature-oriented approach for specifying finite-state machines, which form the basis of cache controllers (and other components) for RISC-V implementations, and which are commonly found in hardware designs. Using a library we constructed for Chisel, developers can apply features at will, with the resulting machine containing only the circuitry needed to support the desired features.**

Our library offers two constructs for building features. The first, inspired by aspect-oriented programming, applies incremental changes to the states and edges of a finite-state machine to alter and customize its behavior in response to features of interest. The second construct couples the behavior of separate finite machines into a single machine that processes its inputs simultaneously. We illustrate each construct separately using a vending machine and the game of Nim, respectively.

Our approach offers significant leverage in supporting both the number and size of the generated designs. We present results from synthesis that show the size of the design endpoints compared with the much smaller size of their specification.

I. Introduction

Projects such as RocketChip [3] and RISC-V Mini [8] allow customization of RISC-V [4] processors that can then be deployed economically on FPGAs. These resulting systems can offer better power and area performance than general-purpose processors while achieving a price point well below generation of an ASIC (application-specific integrated circuit).

While some portions of the RISC-V characterization are easily included or excluded at will, based on a given application's needs, components such as cache features, branch-prediction circuitry, and superscalar support are much more difficult to weave in or excise from a given characterization.

In the RISC-V implementations cited above, components of the processor such as its cache and bus protocols rely on finite-state machines (FSMs) to control the sequencing of the associated logic. Hardware implementations commonly rely on FSMs at the core of their design, due to their simplicity and efficiency.

In this paper, we consider FSM controllers in which features should be included based an application's needs. In a system with n such independent features, there are 2^n possible endpoint designs that could be generated. Maintaining those as distinct projects is unwieldy and inefficient. Instead, we use techniques from aspect-oriented programming to weave features into a design on demand. The resulting code base is significantly smaller, while maintaining the ability to generate any endpoint. The resulting characterization must still undergo synthesis, but the system in its feature-factored form is easier to maintain. Moreover, testing of all feature combinations can be easily automated.

Hardware-*generation* languages such as Chisel [2] allow a designer to write a program whose execution generates the hardware design. The program can be authored using paradigms that promote efficiency, reuse, rigorous testing, and clarity of expression. Our work builds on the hardware-generation language Chisel [15], which is in turn built on Scala [12]. Chisel is a Scala-embedded domain-specific language with libraries that generate Verilog [1] when a Chisel program is executed.

In this paper, we consider a feature-oriented programming (FOP) approach to generating hardware, specifically *finite-state machines* (FSMs). Our contributions are as follows:

- In Section III-A, we describe an approach to formulating features in FSMs based on aspect-oriented programming (AOP). We apply this idea to a featureful vending machine in Section IV and present results on the automatically generated FSMs with specific feature sets in Section VI-A.
- In Section III-B we describe a generative *cross-product* technique (due to Harel [6]) that composes larger FSMs from smaller, behavior-specific machines, illustrated using the game of Nim. Results from its application to variations of Nim are shown in Section VI-B.
- The code we have developed for this work integrates with Chisel and is available via `github`.

The FSMs generated using our approach enjoy benefits similar to feature-oriented software systems:

- Smaller footprints than the fully-featured design are obtained by including only those features of interest. The resulting hardware designs have fewer states and less logic, supporting only the features of interest.
- Logic related to inactive features is completely absent from the resulting designs. Since the clock rate of a hardware system is determined in part by the amount

Supported under NSF CISE award CNS-1763503 Performant Architecturally Diverse Systems via Aspect-oriented Programming

of logic that must complete in a clock cycle, less logic could lead to faster clock rates.

II. Prior work and motivation

We review here the concepts and prior work upon which our work is based.

A. Aspect-oriented programming

Our work builds on a programming paradigm available in the software community that efficiently supports the expression and application of *cross-cutting* concerns in a software system. Aspect-Oriented Programming (AOP) [7] and its realization in systems such as AspectJ [5] allow developers to express ideas that affect multiple components of a system in support of a common idea or feature.

Aspects have been applied to finite-state machines for sequence diagrams in a modeling (UML) setting [14]. Our use of aspects extends that work by taking advantage of types in Scala to formulate advice and guide FSM modifications.

B. Benefits of feature-oriented programming

Our work concerns the generation of FSMs that incorporate desirable subsets of features. We show that FSMs can be specified much more efficiently using this approach. Moreover, the FSMs ultimately generated by the Chisel toolchain contain only those features of interest, requiring fewer resources than would a full-featured, monolithic specification whose undesirable features are disabled.

An example from the software world of the benefits obtainable from an FOP approach to a featureful system concerns the CORBA [10] Event Channel [11]. The standard implementation was *monolithic*, offering all possible features in all allowable combinations. A decomposition of the Event Channel in terms of its features has demonstrated that useful subsets of those features use significantly fewer resources when formulated generatively [13].

III. Generative FSM specifications

We next illustrate our two FOP constructs for generating complex FSMs. The first uses aspect-oriented advice to incorporate features selectively into an FSM. The second builds a *cross-product* FSM from the synchronous simulation of smaller FSMs.

A. Feature introduction via AOP

As an example of a featureful hardware design, we consider an FSM implementation of a vending machine. A state in our design carries the necessary (Scala) *traits* to represent its role in the machine's operation: the funds inserted and the potential products dispensed. Our generative approach described below offers the following advantages over a monolithic design:

- The design itself is simpler and clearer when described using FOP. A monolithic design that includes all features can be realized, but the resulting FSM does not readily make the features apparent. Moreover, the work to create and maintain that monolithic implementation is tedious and error-prone.
- Modification of the FSM is greatly simplified. For example, introduction of a new value of coinage automatically creates the necessary additional states and transitions.
- Scala traits allow elegant expression of an application's behaviors in support of FOP hardware design.

For this example and the results we present, the features of interest for a vending machine are as follows:

- **Add Currency** introduces a value of coinage.
- **Dispense Product** introduces a vendible item and its price.
- **Print Funds** causes the machine to display the total funds after each state change.
- **Insufficient Funds** introduces a prompt to advise the consumer to insert more funds to buy a particular item.
- **Change Return** introduces a button (input to the FSM) that causes the machine to return unspent funds.
- **Peanut Warning** requests confirmation of purchase for items that contain peanuts.
- **Buy More** allows the consumer to continue purchasing items if funds remain in the machine. The **Change Return** feature, if present, allows the consumer to request return of the remaining funds.

The dependencies of these features are shown in Figure 2, but this graph is not needed for construction: the advice for a given feature is applicable only when its associated *join points* exist in the FSM. As is typical with aspect-oriented approaches, all advice is presented to a *weaver* (our runtime library for Chisel), and the aspects are continually applied until no changes occur.

For example, the advice for **Add Currency** of coinage k specifies that for any state representing that n cents have been inserted, a state representing $n + k$ cents must exist, with a transition from state n to state $n + k$ based on the insertion of coinage k. This advice fails to terminate if not capped by some upper bound on funds, which could be related to the most expensive product sold. For example, Figure 1 shows a machine that

- Accepts only 5 cent coinage
- Accepts up to 15 cents
- Vends peanuts that cost 10 cents

The FSM is automatically generated by the advice for **Add Currency** and **Dispense Product**. Continuing with this example, consider the **Buy More** feature, intended to incentivize consumers to spend more money. This feature causes the machine to retain remaining funds after a purchase to encourage subsequent purchases. Without this feature, the machine in Figure 1 would return 5 cents if 15 cents are used to purchase peanuts costing 10 cents. With the feature, the 5 cents of remaining value would be held by the machine for subsequent purchases. The advice for this feature modifies *every* purchase to move to a state representing currently held funds. In Section IV we discuss application of other features to this FSM.

A monolithic approach requires designers to specify all states and transitions for each feature subset, which is tedious and error-prone. With our approach, designers can verify the correctness of much smaller designs using standard FSM verification techniques and then obtain much larger generative designs that are correct by their construction.

In terms of leverage, consider an FSM for which there are n orthogonal and independent features. A valid system could thus be written or generated with or without each of those n features. This leads to 2^n feature-specific implementations. While it is unlikely that each of those implementations would find an application, the ability to generate *any* of them automatically offers significant leverage.

B. Generating FSMs via cross-product composition

Nim [17] is a broad class of impartial mathematical strategy games, which traditionally involve multiple heaps of tokens (*e.g.*, sticks) and two or more alternating players. The current player removes an allowable number of sticks from a subset of the heaps. The winner is usually defined as the player taking the last token. In a *misère* version of the game, that player would lose.

In contrast with the usual monolithic solution, even the most basic game of Nim can be regarded as the composition or simultaneous operation of two simpler machines: one that represents only the allowable subtractions of tokens in a heap (such as the 5-token heap shown in Figure 4(a)) and one that represents only the alternation of players (such as the two-player alternation shown in Figure 4(b)). Transitions not shown in those machines are errors, such as Player A taking two consecutive turns.

Following is our feature decomposition of Nim:

- **Heap Bounds** encodes the initial and winning number of tokens for each heap.
- **Legal Moves** encodes permissible combinations of adding or removing tokens from each heap.
- **Num Players** specifies how many players take turns in the game.
- **Win Type** specifies whether the game is misère play or normal play.

The dependencies of these features is shown in Figure 3.

The game is won when both conditions are met:

- Players alternate correctly, as in Figure 4(b). For example, the sequence ABA leads to an accept, but the sequence AAB cannot.
- All tokens have been taken, for example using the sequence 212 in Figure 4(a).

Using a construction technique due to Harel [6] and well documented in Ptolomy [9], we obtain the basic game of Nim shown in Figure 5. That algorithm simulates the simultaneous, lock-step execution of the machines shown in Figures 4(a) and (b).

In terms of leverage, consider the cross-product generation of an FSM from two identical FSMs each of size m (states+transitions). The resulting machine's worst-case size is $O(m^2)$. An n-way cross product generates a machine of size $O(m^n)$, where m is viewed as a constant here. The structures we can generate with this approach are (in the limit) exponential in the size of their specifications.

IV. Formalism for cross-cutting features

We begin with an FSM M, typically defined as follows:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q is a set of states, Σ is a set of tokens, δ is the transition function, q_0 is the start state, and F is the set of accepting states. A state is typically denoted by an upper-case letter; lower-case letters denote tokens and strings. The symbol λ denotes the empty string. When an FSM is drawn as a graph, the start state receives an edge with no sources, and an accepting state is drawn with two concentric circles.

The formalism presented here is implemented in Chisel as a library we call Foam, described in Section V. The examples and results presented in this paper were created using Foam.

We follow [14] in the treatment of aspects for FSMs. Essentially, a state is like a method and a transition between states is like a method call. The usual forms of before, after, and around advice are available (*cf.* AspectJ [16]). A cross-cutting feature is implemented using advice that modifies an FSM's behavior before, after, or during a transition between states.

As described below, a feature is comprised of *advice* applied to *pointcuts* of an FSM, which can formally change the language of the machine. More broadly and usefully, the advice can affect actions taken by the machine as its inputs are processed.

a) Pointcuts: These specify *where* advice should be applied in a targeted FSM. The generative nature of Chisel eliminates any need for new syntax to express pointcuts. Instead, we can select states or symbols using simple set quantifiers and predicates, written in Chisel/Scala and executed along with the rest of the Chisel code that generates a circuit. For example, the **Print Funds** feature can be generated in a vending-machine FSM through *after* advice applied to any token that adds value to the machine. Such properties are supported nicely in Scala using *traits*. To implement this feature, the base code likely requires refactoring to include the value trait. However, the effort is worthwhile because the refactoring and associated advice make the resulting product both clearer and more easily able to exclude or include actions taken at different inputs.

In AOP terminology, a pointcut yields a set of *join points* at which advice is applied. A join point associated with the above example would be a single token "5", such at the one between state "10" and "15", at which the value increases in the machine by "5" cents. Because this is an *after* pointcut, the join point has context that includes the state "15" that follows the token "5", as shown in Figure 1.

b) Advice: This specifies *what* changes to the FSM should be applied at a join point. In the **Print Funds** feature, a new state is inserted following each token "5" in the FSM. The exact print statement generated by the advice is determined by the context contained within the join point. For example,

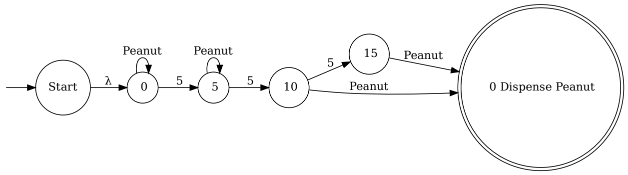


Fig. 1. FSM for a vending machine that accepts 5 cent coins and dispenses peanuts that cost 10 cents.

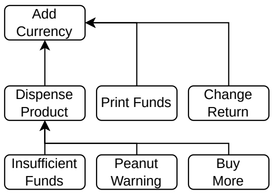


Fig. 2. Dependencies between vending machine features.

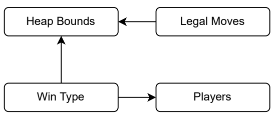
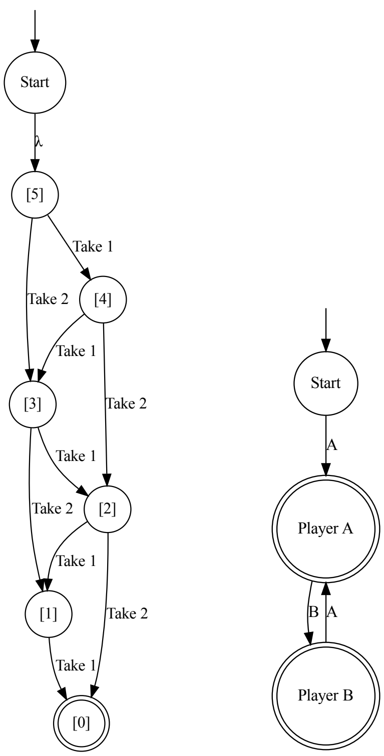


Fig. 3. Dependencies between Nim features.



(a) (b)

Fig. 4. (a) FSM for a 5-token heap that allows one or two tokens to be removed in a turn; (b) FSM specifying alternation of Players A and B.

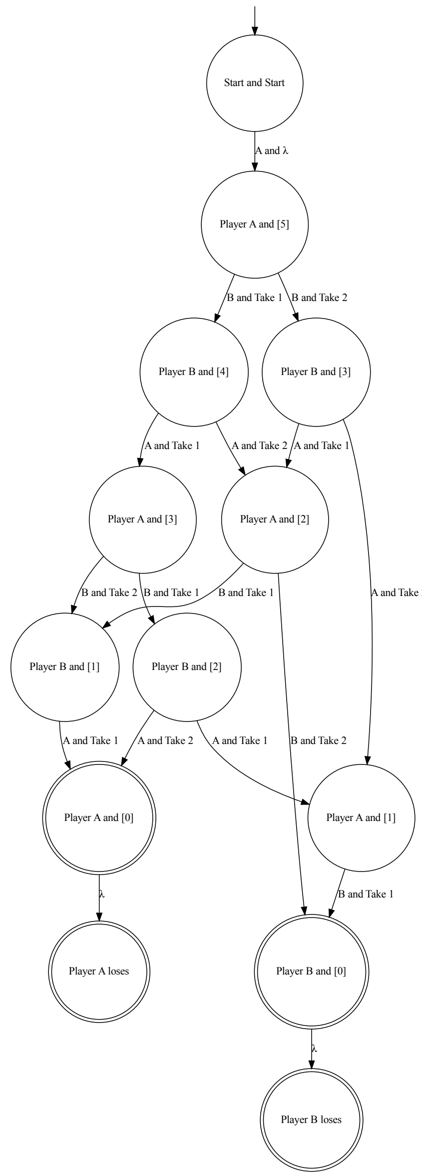


Fig. 5. The resulting Nim finite-state machine. The edge transitions are labeled with the player who acts to take the specified number of tokens. The state is labeled with the player who just completed a turn and the number of remaining tokens.

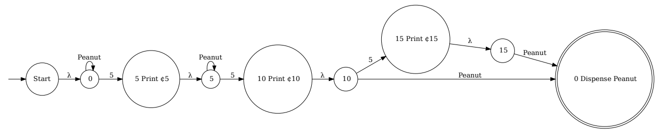


Fig. 6. The resulting FSM of the application of Print Funds to the FSM from Figure 1.

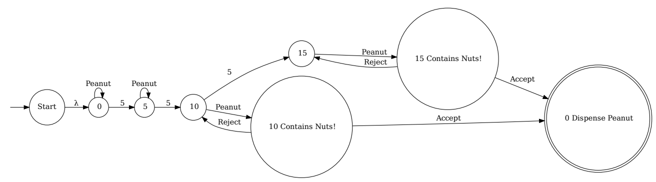


Fig. 7. The resulting FSM of the application of Peanut Warning to the FSM from Figure 1.

`Print ¢15` is generated because the state "15" follows the token "5" discussed earlier. Furthermore, to prevent unending application of features, the advice for **Print Funds** checks to see if the context in the join point is already a printing state. If so, no advice is applied. Like pointcuts, advice is written in Chisel/Scala.

Not only can advice insert new states, but it can also insert new symbols as well. Consider the **Peanut Warning** feature as an example. The pointcut is predicated upon a dispense state having a "peanut" trait. Because this is a *before* pointcut, each join point has context that includes transition information that targets the state. In Figure 7 this is $10 \xrightarrow{\text{Peanut}}$ and $15 \xrightarrow{\text{Peanut}}$. The advice will insert a new "Contains Nuts!" state and "Accept" token for each of the join points. It also inserts a new transition on the "Reject" token whose destination is determined by the context contained in the join point.

V. Aspect-Oriented Finite-State Machine Library

We have implemented an aspect-oriented finite-state machine library in Scala that we call Foam. Here we discuss the library as well as code generation.

While Foam is not realized as a domain-specific language, we have modeled the interface after the well-established aspect-oriented extension to Java, AspectJ [5]. The intention is to provide aspect-oriented practitioners a familiar interface for interacting with the FSMs. Because the library is implemented in ordinary Scala, we have access to and can utilize the full power of its type system.

We provide a series of extendable base classes to represent FSMs. The library builds pointcuts, applies aspects, and performs the cross-product construction. Like AspectJ, our library provides access via reflection to a join point and its context. For examples, our full library can be found at https://github.com/wustl-frisc/faust.

A. Code Generation

The library currently supports emitting DOT and Verilog code. Since the library produces Verilog, standard validation techniques can be applied. Code generation is decoupled from the creation of the FSMs, as the library generates code based off the internal data structures, not the Scala code itself.

VI. Case Studies and Results

Here we present three case studies to demonstrate the generative ability of our framework: a Vending Machine, the game of Nim, and a SIMD cache with a coherence protocol.

A. Vending Machine

We implemented all the features from Section III-A in our library. The resulting FSMs were then emitted as Verilog. The Verilog was synthesized on a xc7a35tcpg236-1 FPGA using Vivado 2022.1. Below we report the number of generated states, transitions in the FSM, and the space in LUTs used by the FSM on the FPGA.

Figure 8 shows the results for different endpoints generated by our library. For our tests, we held the currency threshold at 100. Every machine contains 5, 10, and 25 value coins; and 4 products of value 25, 50, 75, and 100. This is captured by feature set "None". In the first set of results, each feature is shown by itself. Even single features can greatly increase the components in the FSM. The **Buy More** feature (denoted B) by itself more than doubles the number of states and transitions. This impressive leverage is further exemplified when combining features.

In these cases the number of states increase by 2.5x in the simplest endpoint, up to 4.6x in the most complex, and the transitions by 2.5x and 5x respectively. Recall, this is a relatively simple vending machine that can only accept up to 100 units of value. Simply doubling the amount of accepted value to 200 creates a machine with 284 states (10.5x increase over the base) and 3113 transitions (16x increase over the base). However, this is accomplished in our library with relatively few lines of code. The largest feature in terms of code is **Peanut Warning**, which is implemented in just 39 lines.

Despite the growing number of generated states and transitions as the features increase in Figure 8, the resulting hardware resources, in this case Look Up Tables (LUTs), used by the FSM are relatively modest. This is because hardware synthesis tools can represent states using a linear encoding scheme. For an FSM with n states, each state takes only $O(\log n)$ space when encoded as an integer. However, the *specification* of that circuit to a synthesis tool must be expressed state-by-state. If the number of transitions per state is bounded by a constant, then the specification (*e.g.*, lines of Verilog) takes $O(n)$ space. Our generative approach to specifying FSMs allows designers to implement much larger FSMs than are currently sustainable with a hand-coded approach.

Features	States	Transitions	LUTs
None	27	208	24
P	47	368	38
I	47	368	46
C	48	423	70
W	33	320	60
B	57	448	60
PI	67	528	73
PICB	118	1053	186
PICW	94	1023	160
PICWB	124	1353	220

Fig. 8. Number of generated states, transitions, and LUTs depending on selected features. The features are as follows: P = Print Funds, I = Insufficient Funds, C = Change Return, W = Peanut Warning, and B = Buy More.

B. Nim

We implemented all the features described in Section III-B and present two variations of Nim: traditional Nim and circle Nim. All results assume each game is misère play with at least one heap and one player. We further assume that the players alternate in a round-robin fashion. The details of each variation are described below, along with the number of generated states and transitions within the created FSMs.

a) Traditional Nim: The classical game of Nim contains two players and three heaps, though there are no restrictions on the number of players or quantity of heaps. The defining characteristic of traditional Nim is that, each turn, the current player may only remove sticks from a single heap of their choosing, and must take between 1 and all the remaining sticks within that heap. The game ends when all heaps contain zero sticks.

Figure 9 shows the results for different endpoints of the traditional game of Nim, varying both the quantity of heaps, the number of sticks within each heap, and the number of players. In the simplest variation, containing only a single heap with three sticks and one player, a total of six states and 30 transitions were generated. Juxtapose this with the most complex variation, containing three heaps – with three, four, and five sticks, respectively – and four players: this variation contains 419 states and 20950 transitions. With minimal changes to the specification of the game, this represents a nearly 70x increase in the number of states and 698x increase in the number of transitions. For a fixed number of players, the number of transitions experiences growth by an order of magnitude moving from 3 to 4 to 5 heaps.

b) Circle Nim: Circle Nim is given its name due to the layout of its heaps: the heaps are placed around a circle, with their adjacencies affecting gameplay. It is traditionally played with a finite number of heaps, each containing a single stick, and two players. Each turn, the current player is allowed to take the sticks from between one and a pre-set number of consecutive heaps. We allow the player to take from between one and three consecutive heaps.

Figure 10 shows the results for different endpoints of circle Nim, varying both the number of players and the number of heaps. In the simplest variation, with three heaps and a single player, 10 states and 110 transitions are generated. In the most complex variation, with nine heaps and four players, 1429 states and 157190 transitions are generated, representing a 143x and 1420x increase, respectively. Only two numbers were changed in code to realize this exponential increase in output complexity.

Heaps			Players	States	Transitions
3	4	5			
✓			1	6	30
✓			2	9	63
✓			4	11	88
✓	✓		1	22	198
✓	✓		2	39	624
✓	✓		4	60	1800
✓	✓	✓	1	112	1708
✓	✓	✓	2	235	6110
✓	✓	✓	4	419	20950

Fig. 9. Number of generated states and transitions by selected features for traditional Nim.

Heaps	Players	States	Transitions
3	1	10	110
3	2	15	255
3	4	17	340
6	1	66	1320
6	2	113	4294
6	4	150	11100
9	1	514	14906
9	2	951	53256
9	4	1429	157190

Fig. 10. Number of generated states and transitions by selected features for Circle Nim. Each heap contained one stick; players could take from between one and three consecutive heaps.

VII. Summary and Future Work

We have described an FOP approach to constructing complex finite-state machines from much simpler ones. We have illustrated our ideas using two pedagogical examples and one real-world setting, namely an SIMD cache. We have presented an algorithm that creates a cross-product FSM, which simulates the lock-step simultaneous execution of its two input FSMs. Our results confirm that this FOP approach provides significant leverage in terms of the size of the generated products, as compared with the relatively smaller effort of authoring the individual features.

We are currently working to create a fully feature-oriented cache using our system. Hardware caches are ripe for feature-oriented design as they contain many orthogonal features. For example, if we wanted to build a cache model even closer to the RDNA architecture, the original cache FSM would need to be write-through, 4-way set associative, and utilize an LRU replacement policy. Instead of forcing hardware designers into choosing an initial design and refactoring, write policy, allocation policy, replacement policy, and associativity could all be selectable features of the microarchitecture.

References

[1] IEEE Standards Association et al. Ieee standard for verilog hardware description language (ieee 1364-2005). *http://standards. ieee. org/*, 2006.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery.

[3] UC Berekely. Rocketchip generator, 2022. https://github.com/chipsalliance/rocket-chip/blob/master/README.md.

[4] UC Berkeley. RISC-V international, 2023. https://riscv.org/.

[5] Eclipse Foundation. Aspectj, 2022. https://www.eclipse.org/aspectj/.

[6] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[8] Donggyu Kim. riscv-mini. https://github.com/ucb-bar/riscv-mini, 2022.

[9] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2009. Technical Report No. UCB/EECS-2009-151.

[10] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, October 2000.

[11] Object Management Group. *Notification Service Specification*. Object Management Group, OMG Document formal/2002-08-04 edition, August 2002.

[12] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[13] Ravi Pratap, Frank Hunleth, and Ron Cytron. Building fully customisable middleware using an aspect-oriented approach. *Software*, 151(4):199–218, 2004.

[14] Roy Rønmo, Ragnhild Kobro Runde, and Birger Møller-Pedersen. Confluence of aspects for sequence diagrams. *Software & Systems Modeling*, 12:729–824, 2013.

[15] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019.

[16] The AspectJ Organization. Aspect-Oriented Programming for Java. www.aspectj.org, 2001.

[17] Wikipedia contributors. Nim — Wikipedia, the free encyclopedia, 2022. [Online; accessed 9-August-2022].