Modeling Resource Sharing using FSM-SADF

João Bastos[1], Sander Stuijk[1], Jeroen Voeten[1,3], Ramon Schiffelers[1,2], Johan Jacobs[2], and Henk Corporaal[1]

[1]Eindhoven University of Technology, Eindhoven, The Netherlands
[2]ASML, Veldhoven, The Netherlands
[3]TNO-ESI, Eindhoven, The Netherlands
Email: {j.p.nogueira.bastos,s.stuijk,j.p.m.voeten}@tue.nl, {ramon.schiffelers,johan.jacobs-jacj}@asml.com, h.corporaal@tue.nl

***Abstract*—This paper proposes a modeling approach to capture the mapping of an application on a platform. The approach is based on Scenario-Aware Dataflow (SADF) models. In contrast to the related work, we express the complete design-space in a single formal SADF model. This allows us to have a compact and explorable state-space linked with an executable model capable of symbolically analyzing different mappings for their timing behavior. We can model different bindings for application tasks, different static-orders schedules for tasks bound in shared resources, as well as naturally capturing resource claiming/unclaiming using SADF semantics. Moreover, by using the inherent properties of dataflow graphs and the dynamic behavior of a Finite-State Machine, we can model different levels of pipelining, such as full application pipelining and interleaved pipelining of consecutive executions of the application. The size of the model is independent of the number of executions of the application. Since we are able to capture all this behavior in a single SADF model we can use available dataflow analysis, such as worst-case and best-case throughput and deadlock-freedom checking. Furthermore, since the model captures the design-space independently of the analysis technique, one can use different exploration approaches to analyze different sets of requirements.**

I. Introduction

In a traditional approach to system design, the first exploration phase focuses on iterating over different design alternatives to find the best solution that satisfies a given set of requirements and constraints. For example, real-time embedded systems have to perform under strict timing guarantees, or high-performance production systems focus on maximizing resource utilization to achieve maximal throughput. However, exploring different design alternatives is highly dependent on an efficient binding and scheduling of the application in order to analyze the performance of each design. Therefore, several approaches have been proposed to model and explore different mapping options. However, current modeling approaches for the binding and scheduling are not transparent, use different models for application and platform or rely on later transformations to different models and formalisms to solve the actual state-space exploration problem.

In this paper[1] we present a modeling approach based on Scenario-Aware Dataflow (SADF) that provides a single, compact and formal model of the system which reflects the complete state-space for all binding and scheduling options of an application to be mapped on a given platform. Our approach allows for the modeling of both application and platform constraints, keeping all deterministic behavior of the application within static dataflow graphs and model explicitly the different choices of scheduling and binding decisions in a Finite-State Machine. The approach uses a rich formalism capable of capturing different applications or different levels of application pipelining. The resulting model uses an underlying symbolic executable flow to analyze the performance or specific scenario sequences or perform state-space exploration for timing analysis.

Our model captures the following cases: 1) *Resource Sharing*, e.g., if two tasks are bound to the same resource; 2) *Resource Binding*, e.g., if one task is allowed to execute in two different resources; 3) *Application flow/Pipelining*, e.g., if the system allows for interleaved consecutive executions of the application, or if the system can have different bindings throughout application execution.

II. Related Work

Resource modeling has been explored in several works for different dataflow models. For SDFGs, a binding-aware model for task binding is proposed in [7]. It requires graph annotations to encode the binding and schedule and therefore a new model is required per scheduling or resource binding option. In contrast, we capture all options in a single model.

In [1], a modeling approach to find the maximal throughput for the problem of scheduling of SDFGs by converting the graph into a Timed Automata [2] model is presented. In contrast to our approach, it relies on a transformation of dataflow models to other models of computation for the analysis.

In [11], [10] an extended SDFG model, Resource-Aware Synchronous Dataflow (RASDF), is proposed to account for resource binding and scheduling decisions during the design flow. The work is extended for dynamic scheduling decisions using game-theory and SADF to synthesize controllers that meet timing and resource constraints. However, since these models are extensions of SDF and SADF models, existing analysis are no longer applicable. In our approach, we use a combination of SDFG models and finite-state automata. However, instead of embedding the annotations in the original graph, we define different scenarios to reflect the possible bindings and static-order schedules. This allows us to have a single model that reflects the whole mapping state-space and to orthogonally explore it using different analysis approaches.

III. Preliminaries

The semantics of dataflow models, such as Synchronous Dataflow Graphs (SDFGs), naturally capture the behavior of concurrent and dependency-driven applications, such as embedded or production systems. It allows the modeling of direct, cyclic, and pipelined dependencies. Several works have focused on the scheduling and timing analysis of streaming applications using SDFGs, such as [6], [5], [9]. We briefly introduce SDFG in this section (see [4] for details). Consider the example of Fig. 2(a), which depicts an application modeled as an SDFG. The application has 5 tasks, each of which has 1 time unit of execution time. Tasks are modeled as *actors*. Dependencies amongst tasks are modeled as directed *edges* between *actors*. Tokens are represented as black dots

[1]The extended version of this paper can be found in [3]

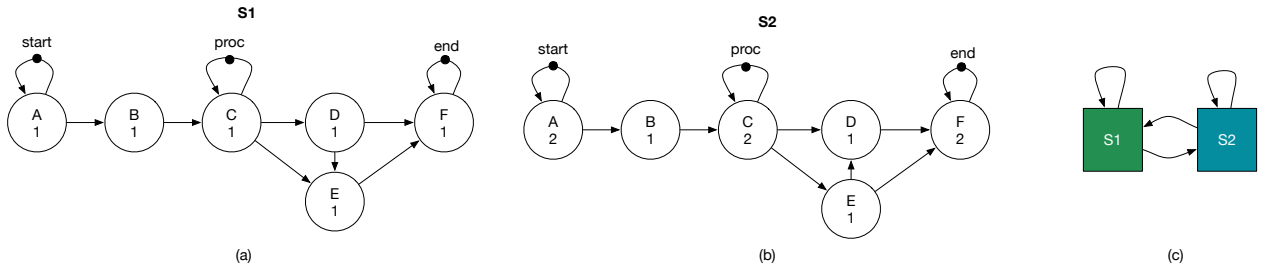

Fig. 1. Example FSM-SADF application graph: (a) Scenario S1 (b) Scenario S2 and (c) Finite-State Machine that models scenario flow

placed on edges between actors. An actor *fires*, i.e. executes its computation, when all its input edges have a number of tokens equal to the specified rate for that channel. For simplicity, throughout this paper we assume actors with a single-rate of 1. When an actor fires, it consumes all its input tokens and produces a number of tokens on its output edges equal to its defined rate. The act of firing takes a fixed amount of time called the execution time of the actor. The execution of 4 iterations of the example application is depicted in Fig. 2(b). In this case, since we are not assuming an execution platform for the example application, actors can fire as soon as possible. Execution of dataflow graphs can be captured by looking at the *start* and *execution* times of actor firings (Gantt Chart) or by analyzing the token production/consumption timeline (Token Timeline). The *Token Timeline* shows the production and consumption times of tokens throughout the graph execution. For each labeled token we define a timeline to register each consumption/production of that token. It provides less information in terms of the execution of the application graph, we do not see the individual firings of actors, but it allows us to visualize the transition between iterations of the application graph in time. We consider that an iteration of the graph is terminated once all the initial tokens are re-produced, and the graph returns to its initial state. Binding and scheduling of actors in a static dataflow graph is usually done by encoding the information on the graph by using extra actors and edges, or by performing separate analysis techniques, which often leads to building different graph models or transformations to other MoCs. In our work we use a richer dataflow MoC: Scenario-Aware Dataflow.

A. Scenario-Aware Dataflow

Scenario-Aware Dataflow is a MoC that combines SDFGs with finite-state automata. The dynamic behavior of a system can be captured using multiple scenarios, where each individual scenario (a SDFG) models a specific mode of operation of the system/application. Therefore, a possible execution of the application can be analyzed with a sequence of different scenarios. The possible orderings of scenarios are explicit in a Finite-State Machine (FSM). SADF exploits a combination of deterministic behavior in the scenario while allowing non-deterministic behavior in the selection of scenario sequences.

B. Example

We can use the example of Fig. 2 to define two different application scenarios as depicted in Fig. 1. In Fig. 1(a) we have an application scenario with a static-order for actors D and E, ($D \rightarrow E$), and all actors have an execution time of 1 time unit. In Fig. 1(b) we have a different scenario

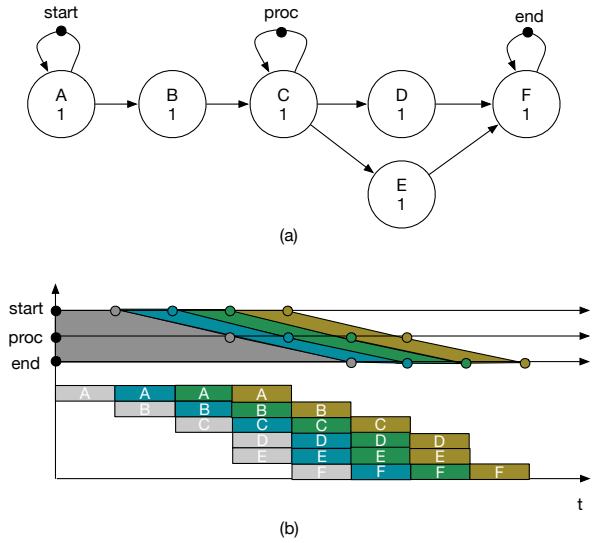

Fig. 2. (a) Example application graph modeled as an SDFG graph (b) Gantt chart execution of 4 iterations of the example application

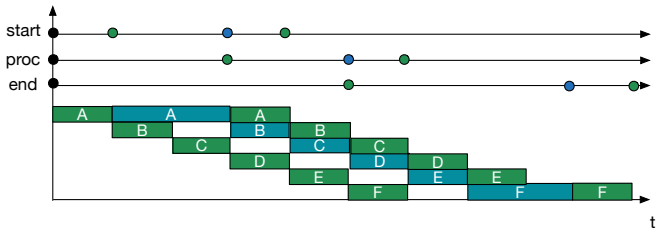

Fig. 3. Gantt Chart of Scenario Sequence: $S1 \rightarrow S2 \rightarrow S1$

where the static-order is ($E \rightarrow D$) and the execution times of A, C, and F are of 2 time units. These represent two possible modes of operation of the initial application of Fig. 2, for example, representing different power modes of an embedded application where actors A, C, and, F have different execution times. The possible orderings of scenario sequences are defined in the FSM (Fig. 1(c)). The states represent the possible application scenarios, while edges represent possible scenario transitions which result in the next scenario to execute. The flow of execution of scenarios is based on the token production/consumption of *labeled persistent tokens*. These tokens are represented by labeled black dots. The label is used to identify common tokens synchronizing between scenarios. This construct is fundamental to understand the execution and

scenario transitions. Fig. 3 depicts the execution of scenario sequence $S1 \rightarrow S2 \rightarrow S1$. If we look at the Gantt chart we can follow the scenario execution by using the coloring, green for S1 and blue for S2. We see that scenario executions overlap in time, however in some cases firings of actors between different scenarios get delayed. In a scenario sequence, a scenario starts executing as soon as the persistent tokens of the actors of that scenario become available. This means that multiple scenarios can overlap in time. For example, in the first transition $S1 \rightarrow S2$, scenario $S2$ starts executing as soon as the persistent token of actor A_{S2} *start* is produced by actor A_{S1}. In the same fashion, the execution of actors in a scenario might get delayed due to pending production of certain persistent tokens. For instance, in the second transition $S2 \rightarrow S1$, the execution of actor F_{S1} is delayed. Actor A_{S1} can start executing immediately after the firing of A_{S2} but actor F_{S1} is still dependent on the production of token *end*. Therefore, even-though F_{S1} could fire at time 9, it only fires at time 10, because it is the new production time of token *end*. This is the fundamental mechanism behind scenario transitions in an FSM-SADF graph and it allows us to model resource availability by using production and consumption of tokens as unclaiming and claiming of resources.

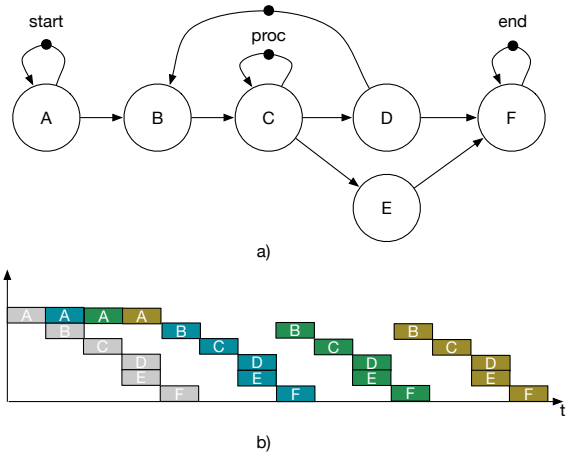

Fig. 4. Example with natural static-order between actors B and D

IV. Challenges in Modeling Resource Sharing

In this section, we explain in more detail the challenges in modeling resource sharing we want to capture in our model. For this purpose we will use the application of Fig. 2, as a running example. Assume a platform with 5 resources, 3 specific resources *start*, *proc* and *end* and 2 shared resources *R1 and R2*. Assume the binding of A, C and F to *start*, *proc* and *end*, respectively, while B, D and E are unbounded.

A. Resource Sharing

A resource sharing situation is the case when D and E, with no direct dependencies, are bound to the same resource. In this situation the order of execution of the actor is not static and can be either (D→E) or (E→D). Although throughput-wise there is no difference between the orders, there can be latency or deadline constraints applied to one of the actors. Therefore, we want to explicitly capture and explore both of these choices in a single model.

B. Resource Binding

Another case we might encounter when exploring the mapping options of an application concerns alternative bindings of actors. For instance, we may allow that actor B can be bound either to resource R1 or R2. Again, this can be expressed by using two different models where the binding choices are different. However, this approach poses two main issues: 1) we are required to create two distinct models and 2) in a binding-aware model the binding is static, while modern applications can exhibit behavior that allows the mapping to be changed dynamically throughout the execution. Therefore, we want to capture dynamic binding options in an explicit way.

C. Application Flow/Pipelining

A simple example is pipelining of different instances of the system application, which we call interleaved pipelining. This behavior is usually present in production or manufacturing systems, where throughput optimization focuses on the pipelining multiple processing steps of products. This cannot be captured in a static dataflow model, nor by using a traditional modeling approach with SADF since it requires the application to be executed in a partial fashion and not based on full graph iterations. Fig. 4 depicts a situation where actors B and D are sharing resource R1. In this case, since B and D have a direct dependency in the graph structure, we can impose a static-order. We can do so by adding a back edge from actor D to actor B with a single token. This enforces that B fires before D and that both actors cannot fire concurrently. If we observe the Gantt chart produced by this application graph, this indeed reflects the constraints imposed with the static-order schedule. However, if the modeled system allows for the pipelining of multiple data (or products), such is not captured with this static model. For example, notice that after the first firing of B we could have a second firing of B concurrently with the first firing of C, but due to the imposed static-order this is not possible. If this level of pipelining were captured then we could have an application execution as depicted in Fig. 5.

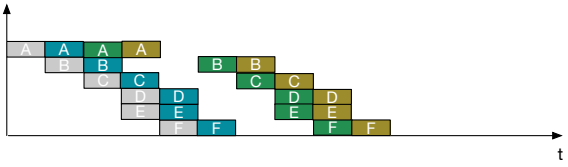

Fig. 5. Gantt chart of the execution of sequence 4 times of (A-B-C-D-E-F)

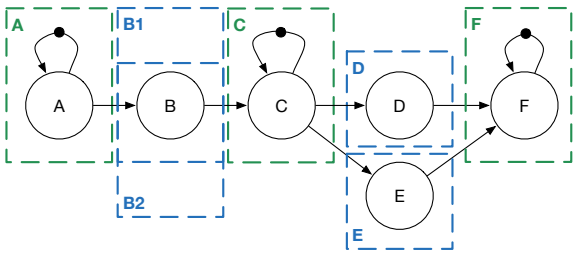

Fig. 6. Splitting of the original graph into deterministic (green) and non-deterministic (blue) parts into scenarios

V. Solutions to the Modeling Challenges

In this section we present our proposed modeling solution for each of the challenges presented previously by using SADF models to capture both application and platform resources. Traditional modeling using SADF would require the designer to build a unique scenario graph for each alternative in the system (binding or scheduling). Moreover, an execution of a scenario graph in SADF requires a full iteration of the graph, which does not allow the modeling of interleaved pipelining. Therefore, we propose a modeling approach where we *split* the application graph into multiple scenarios and use a Finite-State Machine (FSM) to model the application flow.

A. Splitting the Application

Fig. 6 depicts the division of the application graph into sets of actors that represent either static and deterministic executions (green) and sets of actors that can have scheduling freedom within their assigned binding (blue). This is the case of actors D and E, as well as, different bindings for actors, as is the case of actor B (B1 and B2). The actual splitting of the original graph to a set of scenario graphs requires the transformation of direct dependencies, modeled in the original application as edges between actors, as added *persistent tokens*. Fig. 8 depicts this transformations, where each individual dashed box in Fig. 6 is now fully described with dependencies as *persistent labeled tokens*.

In this transformation we use *persistent tokens* to model both the original direct dependencies of the original graph as well as binding decisions. For instance, in this transformation we assume actors B and D are mapped on resource R1, E on R2, and the remaining actors on *start*, *proc* and *end*. Note that the splitting of the original application graph does not have necessarily to be done per actor in the graph. Ideally any set of actors that represent a static and deterministic behavior in the original application can be grouped in the same scenario.

Let us consider scenario *A*, where the behavior of actor A is described. As in the original application, actor A has a resource self-dependency with a labeled token *start*, but, due to the dependency with actor B, also an added *dummy actor* dAB, with a labeled token *dAB* that represents the dependency with actor B. If we now look at scenario *B*, we see that this dependency is also present by the means of a dummy actor dAB and a labeled token *dAB* that precedes the firing of actor B. This way the correct execution of the graph is preserved, if the scenario sequence *A-B* is imposed. Notice that for each direct dependency we add a pair of dummy actors in each of the corresponding scenarios, however, these have no impact on the timing analysis of the model since their execution time is 0 s. The same principle is then applied to all the scenarios regarding its original dependencies. Crucial to the correct model of the system and its behavior is the FSM that accompanies the set of scenarios graphs, depicted in Fig. 7 (a). For simplicity, we keep the same coloring in the states of the FSM as in the scenarios. Green states represent static deterministic parts of the application, while blue states represent scenario graphs that model options in both binding and scheduling. The FSM-based SADF graph model can then capture the initial application flow of the original graph. See that the FSM transitions enforce the original application flow (A-B-C-D-E-F). A fundamental construct of our modeling approach is the ability to do symbolic executions of scenario sequences that reflect the actual binding and scheduling choices per scenario sequence. Fig. 5 depicts 4 iterations of the scenario sequence (A-B-C-D-E-F), given the scenario graphs of Fig. 8 and the FSM of Fig. 7(a). In this Gantt chart we see how the persistent tokens influence the execution of the application. All the *dummy* tokens representing the direct dependencies impose the sequentialization (e.g., A-B transition) or parallelization (e.g., D-E transition) of scenario executions. Furthermore, we see how modeling resource bindings as persistent labeled tokens allow us to naturally capture the resource utilization on the system. This is clear on the execution gap between firings of actor B, since at the third firing resource R1 is being used by actor D of the previous firing. Therefore, the claiming and unclaiming of tokens is correctly captured as well as the mutual exclusion of shared resources.

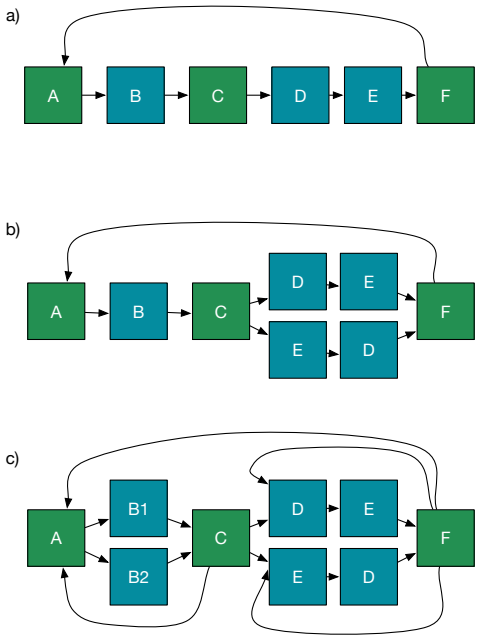

Fig. 7. FSM representing different stages of the modeling: a) Application flow with set bindings and actor order, b) Resource-Sharing (Orderings) and c) Resource Binding

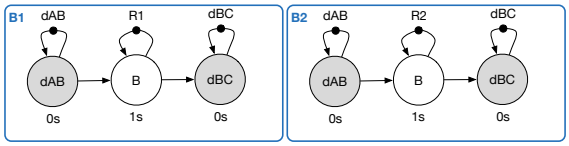

Fig. 9. Scenarios for each of the possible bindings of actor B

B. Resource Binding

Let us apply our approach to address the case where an actor has multiple bindings. Assume that actor B can have two bindings, R1 and R2, we can model this creating two different scenario graphs. Fig. 9 depicts these new scenarios to add to our state machine, depicted in Fig. 7(b). Each scenario includes a different binding for actor B, by using a different *persistent token* labels, in this case *R1* and *R2* labels. *Generalization*: Fig. 10 depicts an actor *a* with a self-edge and a labeled token *r*. In our modeling approach we use such a construct to model resource binding. The label on the token represents the resource where the actor has been

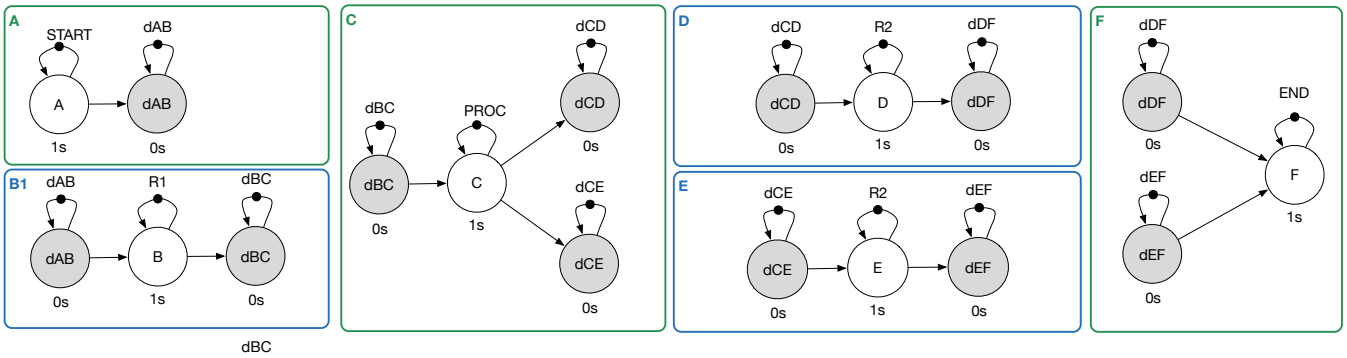

Fig. 8. Adaptation of the splitting of the SDFG of Fig. 6 into scenarios of the SADF model

bound and the self-edge imposes a non-concurrent claiming and unclaiming of the resource. Once the actor fires the labeled token is consumed. When a labeled token is consumed no other scenario actor can consume that same token. Once the token is produced again, the resource is released and therefore, the token is again available for any other scenario actor. We can also model the capacity of the resource (e.g. a resource that can simultaneously process two products), depicted in Fig. 10 by actor *b* by having multiple tokens in the self-edge, as long as each *persistent token* is labeled differently. For each binding a scenario graph has to be added to the FSM-SADF graph.

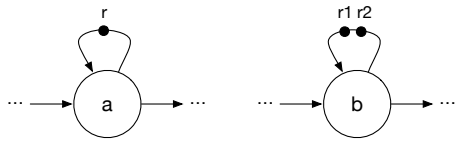

Fig. 10. Modeling resource binding in actors: (a) actor *a* is bound to resource *r* (b) actor *b* is bound two resource *r* with as a capacity of 2

C. Modeling Scheduling Options

If in our system application we have open scheduling options, such as the case of D and E, we can then use the FSM states as a mean to model both the options for the static-orders of actors D and E by using state *duplication*. This is captured in the FSM of Fig. 7 (b), where we duplicate the states to model the different execution orders. However, notice that when we duplicate a state we do not duplicate a scenario graph. This means that we move the decision point from the application model to the FSM exploration.

Generalization: If in the original application graph there are groups of actors bound on the same resource with no direct dependencies, then they have different static-orders. As a generalization rule, for each possible order for the group of actors we model it as a different path in the FSM, with the same initial and ending states. This is done by duplicating the states in the FSM, which represent the same scenario graph.

D. Modeling Application Pipelining

Finally, we can use the FSM-based SADF model to represent different application flows. This is done by adding transition edges in the FSM. Looking at all the FSM's depicted in Fig. 7 we see that the transition edge from (F-A) is always present. This is the transition edge that implies the full application pipelining, which is present in the original application. However, we would like to exploit pipelining in some systems. Therefore, we can add extra edges that reflect this behavior. For instances we can add the edges of Fig. 7(d), such that we allow the interleaving of scenario sequences of (A-B-C) and (D-E-F). This means that we allow the system to do partial executions of different instances of the application. If the system exhibits different modes of operations, such as power modes or execution times for certain tasks, we model these as different scenarios within the FSM-SADF model, as it is done on traditional FSM-SADF modeling approaches [8].

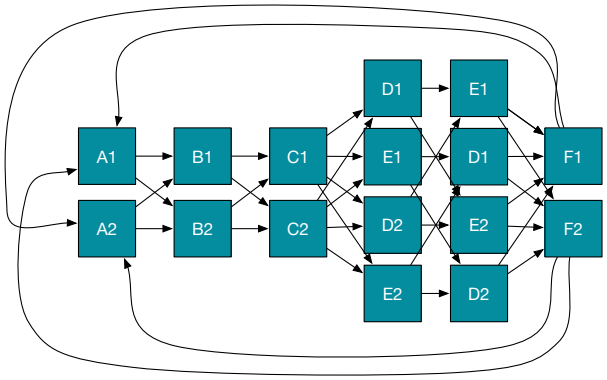

Fig. 11. FSM that holds all the behavior of our example application mapped on a 2-Homogenous resource platform

VI. Modeling Approach

In this section we collect the solutions for each of the modeling challenges and present them in a unified approach and define a set of construction rules. As an example of all the concepts presented so far, consider the FSM in Fig. 11. This describes the state-space of the application of Fig. 2 on a platform with two resources, $R1$ and $R2$, where we allow all the actors to be bound to either resource. We see that for each binding, every actor has a duplicate scenario. Furthermore, we also have actors D and E, that can be enabled simultaneously and therefore have to have specified all the allowed static-order schedules and binding combinations.

A. Construction Rules for the System Model

If we know the possible bindings for each resource (which can be done by matching *actor type* with *resource type*) we can then define the following rule:

Rule 1: For every binding of an actor we define an individual scenario graph, where the label of the persistent token in the self-edge of the actor is the label of the resource binding, and all the direct dependencies are transformed into inter-scenario dependencies using of persistent token labels.

However, we still need to define the possible orders and transitions to be allowed and explored by the FSM. For this purpose, we need to gather information from the application structure regarding its natural static order schedules. In other words, we need to identify the actors that may be fired simultaneously with no direct dependencies (unordered actors). Such that every unordered group of actors has explicitly in the model its different static order schedules. However, since we do not change the underlying scenario graph (for each actor) we duplicate the states of the FSM and force the static order schedules as different paths in the FSM transition sequences. This allows us to define the following rules:

Rule 2: For each unordered group of actors in the application and for each possible static-order of that group, a path in the FSM is created by duplicating existing states in the FSM.

Rule 3: If an ordered group of actors in the application is composed only of bound actors then they can be concatenated in a single scenario that represents the fully static and combined ordered set of actor firings.

Once all the scenarios and scheduling transitions have been added to the FSM, we can add the extra set of edges that reflect the execution of the application, that it can be instantiated as full application pipelining, or interleaved pipelining.

B. Scenario Sequences and Analysis

Once we have the system model built as explained above, we reach a FSM-SADF model such as the one composed by the set of scenario graphs of Fig. 8 combined with the FSM of Fig. 11 (to avoid overloading the Fig., we do not show the interleaved pipelining transition edges (C-A), (F-E) and (F-D) in the image). With this model we can now explore the state-space for several executions of the application. We do so by doing symbolic executions of scenario sequences. For example, if we explore *scenario sequence* (A1-B1-C1-D1-E1-F1) we are symbolically executing a full execution of the original application graph, where all the tasks are mapped on R1, and the static-order for D and E is $D \rightarrow E$. Furthermore, we can also explore *scenario sequences* for multiple consecutive executions of the application flow by using the transition edges which represent levels of pipelining in the system. In this case, it could be (A1-B2-C1-A1-B2-C1-D2-E2-F1-D2-E2-F1), which represents the interleaved pipelining of two products in the system. We then use SADF semantics to get a symbolic execution of this sequence that generates the Gantt Chart of Fig. 12. We can use the Gantt chart to follow the execution of different scenarios and use the Token Timeline for the resource utilization. When a token is consumed by an actor of a running scenario, the resource with that label is claimed. The duration of the claim is represented by the corresponding color bar on the timeline. When the actor terminates its firing, the token is produced again and the resource unclaimed. We can follow this trend throughout all the execution of the scenario sequence to view which resources are in use for the that sequence. Moreover, we see that the second firing of scenario B2 is executed interleaved with the still on-going first execution of scenario C1. For example, using this symbolic execution we can use analysis techniques to explore the state-space to find maximal throughout solutions. In this case, a solution to provide maximal throughput, would be to opt for a sequence where the second firing of scenario D is executed in resource R1 instead of R2. Furthermore, since we are able to capture the whole state-space in the model, we can use different exploration methodologies to address different requirements.

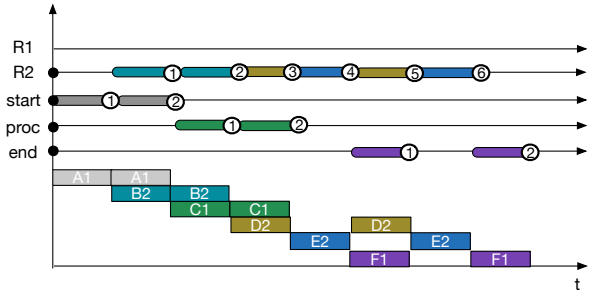

Fig. 12. Gantt chart and Token Timeline for the execution of scenario sequence (A1-B2-C1-A1-B2-C1-D2-E2-F1-D2-E2-F1)

VII. Conclusions

This paper presents a modeling approach using Scenario-Aware Dataflow to model a full system. We demonstrate that the binding and scheduling exploration for the modeled system can be completely captured with this approach without added constructs to SADF or transformations to another model of computation. We show how we model the behavior of shared resources, different bindings and interleaved pipelining by using persistent token labels across different scenarios graphs. The final model we propose is an FSM-based SADF graph, that consists in multiple scenarios to capture the full behavior of the application, but still allowing for a dynamic control of the application flow through the FSM. The fact that our model is representative of the full mapping problem and is as well an executable model, allows us to decouple modeling from analysis. Consequently making it possible to apply different analysis techniques to find scenario sequences that satisfy different sets of requirements.

Acknowledgments

This research is supported by the Dutch Technology Foundation STW under the Robust CPS project (12693).

References

[1] Ahmad, W., et al. *Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. Proc, ACSD'14.*
[2] Alur, R. et al. *A theory of timed automata. Theor. Comput. Sci.*, 126(2):183–235 (1994).
[3] Bastos, J., et al. *Modeling resource sharing using fsm-sadf.* Technical report, Technical University of Eindhoven, ESR-2015-03 (2015).
[4] Geilen, M. et al. *Worst-case performance analysis of Synchronous Dataflow scenarios. Proc, CODES+ISSS'10*, (C):125–134 (2010).
[5] Ghamarian, A., et al. *Throughput analysis of synchronous data flow graphs. Proc, ACSD'06*, 25–34 (2006).
[6] Lee, E. et al. *Static scheduling of synchronous data flow programs for digital signal processing. IEEE Transactions on Computers* (1987).
[7] Stuijk, S., et al. *Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. Proc, DAC '07.*
[8] Stuijk, S., et al. *Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. Proc, SAMOS'11*, 404–411.
[9] Stuijk, S., et al. *Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Trans on Comp.* (2008).
[10] Yang, Y., et al. *Iteration-based trade-off analysis of resource-aware SDF. Proc, DSD'11*, 567–574.
[11] Yang, Y., et al. *Playing games with scenario- and resource-aware SDF graphs through policy iteration. Proc. DATE'12*, 194–199 (2012).