Journal of Innovations in Computer Science and Engineering
Shahid Beheshti Unversity

Jan 2024, Volume 02, Issue 01

System-Level Modeling of Dynamic Applications with Scenario-Aware Dataflow Graphs

Seyed-Hosein Attarzadeh-Niaki ORCID: 0000-0002-2171-1528[a],
Mohammad Vazirpanah ORCID: 0000-0003-3960-674X[b]

[a]*Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran, email: h_attarzadeh@sbu.ac.ir*
[b]*Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran, email: mohammad.vazirpanah@gmail.com*

Abstract

This paper presents a comprehensive modeling framework for managing the complexities inherent in modeling dynamic and intelligent embedded and cyber-physical systems (CPSs). Leveraging the scenario-aware dataflow (SADF) model of computation (MoC), our framework effectively captures CPS dynamism through controlled scenario representations. We establish denotational-style semantics within the Formal System Design (ForSyDe) framework and operational-style semantics tailored for practical industrial implementation. Integration of SADF MoC into ForSyDe-SystemC exploits modern C++ language features, offering type- and size-safety, model introspection, parallel simulation, and foreign model integration. The contributed SADF extension possesses the capability to seamlessly interconnect with other MoCs, thereby facilitating heterogeneous system modeling. Demonstrational examples, including an encoder/decoder system and an MPEG-4 decoder algorithm for the simple profile, attest to the framework's correctness and practicality. We also introduce a tool flow for automated synthetic benchmark generation, essential for assessing the scalability and performance of ForSyDe-SystemC SADF models in diverse conditions. The extended modeling framework, examples, and supporting tools are available as public domain code.

Keywords: Modeling Abstractions, Scenario-Aware Dataflow, Embedded Systems, Cyber-Physical Systems, Formal System Design, Streaming Applications.

1. Introduction

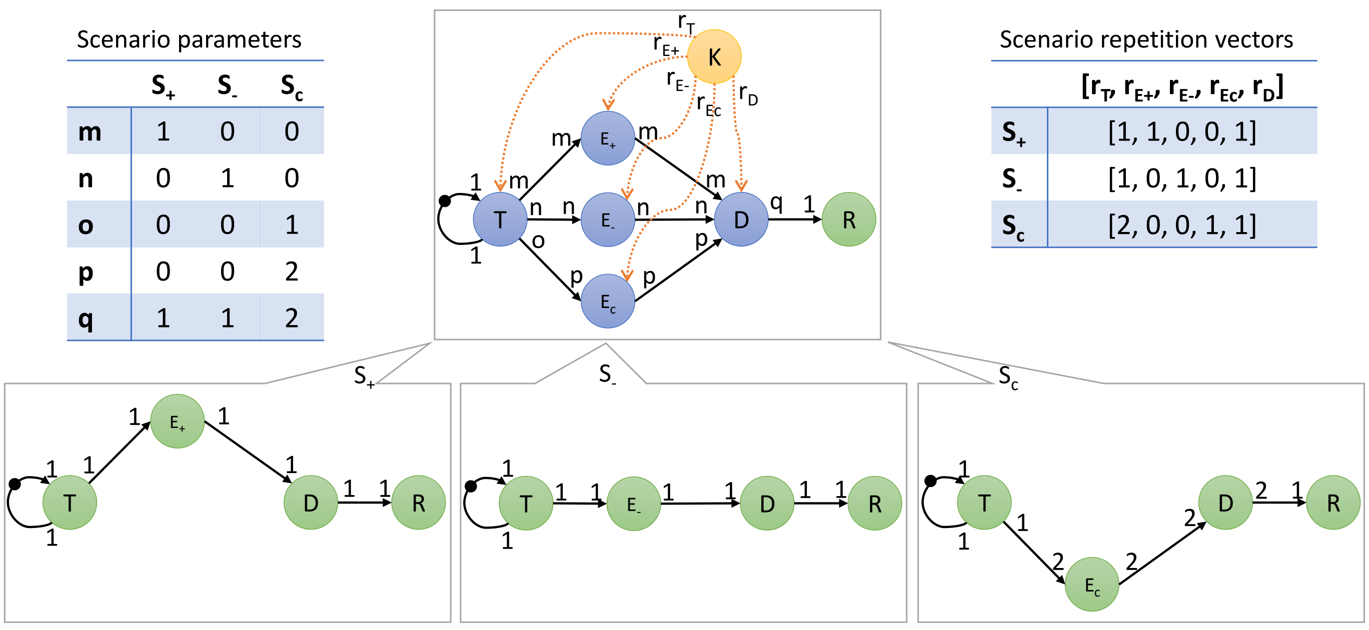
The design of dynamic embedded and cyber-physical systems (CPSs), as well as systems on chip (SoCs), presents significant challenges due to their complex and heterogeneous nature, coupled with real-time constraints and changing operating environments. Traditional application models and design methods based on static structure and pre-determined execution order of components are insufficient to capture the expected dynamic behavior of these systems in response to their changing environment [21]. This problem is intensified in modern smart systems equipped with machine learning and adaptive behavior [6].

In the context of electronic system-level (ESL) design, system-level models are a means for expressing the functional behavior of the system, together with its desired properties and requirements, to be satisfied throughout the design flow [12]. In this case, very often a standard language such as SystemC [2] is used to model the application behavior and the target platform. Unfortunately, these languages are unable to capture both the required dynamic behavior of the system and the desired properties such as realtimeness, safety, etc. for a robust system design flow.

During execution of a SystemC model, an elaboration phase is perfomed followed by a discrete-event (DE) simulation [2]. While it is possible to spawn new processes during simulation, the model structure cannot be modified dynamically after elaboration. To lift this constraint, language and library extensions to SystemC are suggested [1, 15, 20]. However, these approaches concentrate on providing an infrastructure for modeling a dynamic functional behavior for the application while the expressed models lack the required analyzability and synthesizability.

Other works approach the problem at a higher level and map dynamic models of computation (MoCs) on top of SystemC [10, 3]. Dataflow MoCs are a proper candidate for this purpose and there is a spectrum of them which trade-off expression of dynamic behavior with analyzability and implementation efficiency [18]. More classic dataflow models such as synchronous dataflow (SDF) are supported in many SystemC-based frameworks; however, they completely lack the dynamic aspect. In contrast, the scenario-aware dataflow (SADF) MoC [22] provides a flexible and adaptable framework that can capture the dynamic behavior of these systems and also enable efficient mapping of system components to hardware resources. This modeling approach can facilitate ESL design flow by allowing designers to efficiently explore different design trade-offs and optimize system performance. Moreover, it can enable rapid prototyping and testing of complex systems, reducing design time and improving overall system reliability. Therefore, the SADF MoC is of high interest for ESL design flows, particularly for dynamic embedded systems, CPSs, and SoCs.

The SADF MoC is an instance of the more generic concept of system-scenario-based design [8]. Instead of coping with the dynamic nature of applications by considering the corner case or designing for separate use cases separately, this approach combines several statically analyzable scenarios in a systematic manner. System-scenario-based design can be incorporated in all



Scenario parameters

	S_+	S_-	S_c
m	1	0	0
n	0	1	0
o	0	0	1
p	0	0	2
q	1	1	2

Scenario repetition vectors

	$[r_T, r_{E+}, r_{E-}, r_{Ec}, r_D]$
S_+	[1, 1, 0, 0, 1]
S_-	[1, 0, 1, 0, 1]
S_c	[2, 0, 0, 1, 1]

Fig. 1: An illustration of a scenario-aware dataflow graph. The top figure portrays the SADF model of a simple encoder/decoder system, featuring detectors (in orange), kernels (in blue), and ordinary actors. In the sub-graphs presented at the bottom are the synchronous-dataflow equivalents of the model in each scenario. Additionally, the graph parameters and the schedule of the graph (repetition vectors) for each scenario are visually depicted.

design layers from high-level models, which is the focus of our work, down to hardware platform layers. Recently, this approach has been investigated for novel applications such as deploying deep neural networks on the edge [13].

Unfortunately, there is no available modeling framework with appropriate support of the SADF MoC based on standard ESL languages. The closest reported work is by Bonna and others [7], which provides a functional model, together with an implementation based on the functional language Haskell. Although the authors sketch a template for mapping to what they call high-level languages, their solution is more appropriate for code generation with the sole purpose of simulation, rather than directly capturing models early in the design flow. We provide a more comprehensive comparison to this approach using a case study in Section 4.

In a study focused on the analysis of formal models for avionics systems design [9], the authors advocate for the extended framework above [7]. They assert that a set of MoCs, each tailored for specific aspects of avionics systems, aligns well with the requirements of this domain. According to the authors, the SADF MoC is deemed more suitable for implementing safety, components with runtime reconfiguration, and supporting modern architectures within avionics systems. In comparison, our work, which shares the same underlying **For**mal **Sy**stem **De**sign (ForSyDe) formalism, also supports the synchronous (SY), SDF, and SADF MoCs, among others. Notably, our approach leverages a more industry-friendly system design language, provides MoC interfaces (MIs) as a mechanism for seamless integration of these MIs as heterogeneous models, and supports co-simulation wrappers for legacy IP integration.

The ForSyDe-SystemC modeling framework [3] provides the formal modeling abstractions of ForSyDe in the de-facto system modeling language SystemC. This framework supports the general untimed (UT) MoC with a high level of dynamism, however lacking analyzability, and the highly analyzable, yet restrictive SDF MoC. This paper extends ForSyDe-SystemC with the SADF MoC. The new MoC is integrated with already available ones using MoC interfaces to support heterogeneous embedded and CPSs design. Another advantage of our approach is that the new extension is mapped onto the common abstract MoC layer of the framework which provides facilities such as parallel simulation [5], foreign-model integration [4], and introspection. The latter is a highly useful facility which allows exporting the expressed models into an intermediate representation for further analysis and synthesis by external tools. We also extend the introspection mechanism in this work to report the runtime state of the dynamic models captured in the newly added SADF MoC.

The rest of this paper is organized as follows. Section 2 provides the background on the modeling concepts required for an in-depth understanding of this work. Section 3 contains the main contribution of the work and provides the semantics and syntax of SADF models for implementation in ForSyDe-SystemC. Section 4 provides empirical results to justify the usefulness of the framework and Section 5 concludes the paper.

2. Basic Concepts

2.1. Scenario-Aware Dataflow

Dataflow MoCs are frequently employed to articulate the behavior of signal processing and streaming applications. Earlier iterations of dataflow models, exemplified by the one introduced in Kahn's seminal paper [11], serve the purpose of delineating and expressing parallel programs intended for execution or simulation. Nevertheless, designers often harbor an interest in assessing the absence of deadlock in such programs and formulating a scheduling strategy characterized by bounded buffers for unbounded execution. Regrettably, these issues prove to be undecidable within the domain of Kahn process networks (KPNs) and general dynamic dataflow graphs [14]. In contrast, more constrained dataflow models, such as the SDF MoC, strike a balance between expressiveness and analyzability, as well as implementation efficiency. A comprehensive examination of these aspects can be found in the work of Stuijk et al. [18].

The scenario-aware dataflow (SADF) MoC was initially introduced by Theelen and colleagues [22]. SADF represents an augmentation of the SDF MoC, systematically incorporating dynamic behavior into the model while preserving a robust level of analyzability. Within an SADF model, two fundamental components take center stage: *kernels* and *detectors*.

Kernels serve as the fundamental building blocks within SADF models. They share similarities with actors found in other dataflow models, although the behavior of a kernel is contingent upon the presently active scenario. Kernels exhibit the capability to exhibit distinct consumption and production rates, execution functions, as well as execution durations, all contingent upon the specific scenario in play. In contrast, actors within SDF models maintain fixed rates and execution functions [22].

Detectors assume the pivotal role of orchestrating the behavior of kernels and discerning the current system scenario. These detectors take the form of Mealy-like finite-state machines (FSMs), consuming data tokens and generating control tokens. Within the realm of SADF, each kernel is endowed with multiple scenarios, but only one scenario can be activated for a given kernel at any given moment. It is the task of detectors to ascertain the active scenarios and relay this information to the kernels. The detector offers versatility in implementing various control strategies, including state machines, decision trees, and Petri nets. In each scenario, the collective behavior of kernels within the graph resembles that of an SDF graph. It is worth noting that, in the original definition of SADF models, detectors are represented using a Markov chain. Nonetheless, for the purposes of this work, we adhere to the FSM-based FSM model, which aligns more seamlessly with functional modeling objectives [22].

Each kernel features a solitary control input that is under the purview of a detector. This arrangement arises from the fact that, at any given moment, each kernel is capable of executing only a single scenario. Importantly, the detector has the capacity to generate control commands at varying rates across different scenarios.

Fig. 1 serves as an illustrative example of an SADF graph, which models an encoder/decoder system. First, this system generates a sequence of data tokens designated for transmission (T). Subsequently, depending on the currently active scenario, these tokens undergo encoding through either an increment operation (E_+), decrement operation (E_-), or a process that combines two consecutive tokens (E_c). On the receiving end, the transmitted tokens are decoded (D) and subsequently emitted via the receiving channel (R).

At the top of Fig. 1, the SADF graph is depicted, featuring orange circles to symbolize detectors, blue circles to represent kernels, and green circles denoting classic SDF actors. Notably, dashed arrows interconnecting the detectors and the kernels serve as control signals, facilitating the transmission of the current scenario information. It is worth noting that the consumption rate of the control signal on the kernel side, although not explicitly illustrated in the figure, is consistently set to one. Within this framework, scenario functions, potentially implemented in the form of tables, govern the operational scenario of the actors, specifying the data production-consumption rates for each kernel and detector across various scenarios.

It's important to observe the various modeling patterns evident in this example. Kernel T consistently maintains the same functionality, generating input tokens. However, these tokens are directed to different destinations in each scenario. On the encoder side, encoding algorithms are realized through distinct kernels–E_+, E_-, and E_c–with one active in each scenario. Conversely, a singular decoding kernel, denoted as D, applies the reverse algorithm, contingent upon the currently active scenario.

2.2. The ForSyDe Framework

ForSyDe is a powerful design framework for complex embedded systems, SoCs, and CPSs [16]. It is based on a formal and mathematical foundation, which makes it well-suited for verification and validation. ForSyDe is designed with the following goals in mind.

ForSyDe stands as a potent design framework tailored for intricate embedded systems, SoCs, and CPSs [16]. It finds its roots in a rigorous and mathematical underpinning, rendering it highly adept for purposes of verification and validation. The overarching design objectives of ForSyDe are as follows.

- System design should begin at a high-level of abstraction. The designer concentrates on the holistic functionality of the system and not worry about low-level implementation details. This approach empowers the designer to render decisions that remain untethered to the constraints of particular implementations.

- A design ought to establish a robust groundwork for the application of formal methods, which can subsequently be employed to ascertain the correctness of a system's design.

- The gap between the abstract specification of the system and its tangible implementation should be traversed through meticulously defined transformations and refinements. This implies that the designer should employ methodologies to transmute the high-level system specification into a low-level implementation that not only demonstrates efficiency but also adheres to the system's requirements.

The ForSyDe modeling framework initially found its implementation in the functional programming language Haskell, which aligns seamlessly with the foundational principles of ForSyDe. However, Haskell may be less familiar within the realm of industrial engineering. To enhance the practical applicability of ForSyDe, it has been transposed onto the system-level design language SystemC [3]. ForSyDe-SystemC harnesses the advanced capabilities inherent in C++, including template meta-programming, lambda functions, and more, to actualize the core tenets of the ForSyDe modeling framework.

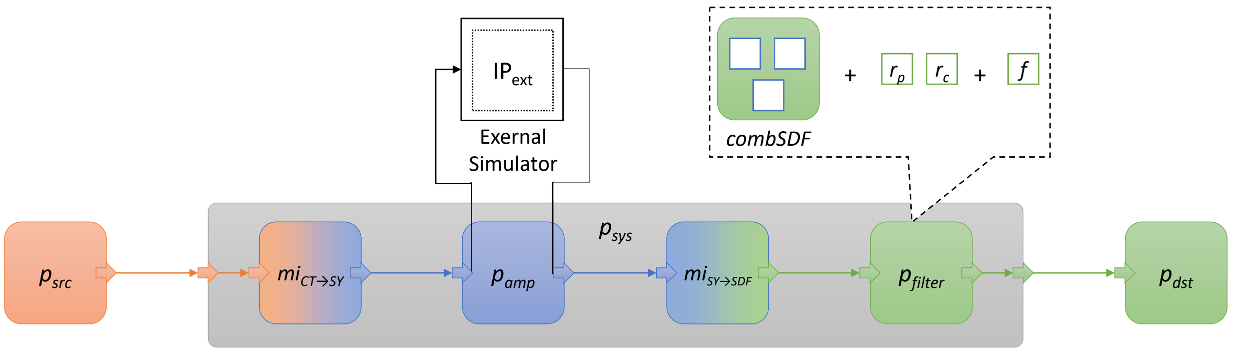


Fig. 2: An illustrative instance of a heterogeneous ForSyDe-SystemC model including processes and MoC interfaces with structural hierarchy. Leaf processes are instantiated using process constructors. Wrapper processes have the potential to facilitate co-simulation with external models.

As depicted in Fig. 2, a system model within ForSyDe is structured as a hierarchical network of processes. Heterogeneous modeling is essential for CPSs, thus, ForSyDe exploits MoCs to describe the functional behavior of different parts of a system using different semantics. As a result, the process network consists of processes belonging to specific MoCs, plus MIs between processes of different MoCs. The only means of inter-process communication and synchronization in ForSyDe is through signals. ForSyDe-SystemC supports a range of MoCs, such as UT MoC, including its dataflow variants, SY MoC, continuous-time (CT) MoC, and the distributed discrete-event (DDE) MoC, among others.

The process network depicted in Fig. 2 comprises three processes p_{src}, p_{sys}, and p_{dst}. The composite process p_{sys}, in turn, is composed of two leaf processes and two MIs. The amplifier process p_{amp} acts as a *wrapper* to incorporate an external model, running in an external simulator into the modeling framework. Note that all signals, process ports, and leaf processes built using process constructors belong to a specific MoC.

A noteworthy characteristic of the ForSyDe modeling framework lies in its exclusive use of *process constructors* for the instantiation of all leaf processes and MIs. For instance, consider the construction of the SDF actor p_{filter} through the employment of $combSDF$, along with the specification of production and consumption rates denoted as r_p and r_c, and the actor function f. This utilization of process constructors empowers designers to leverage a collection of pre-designed templates, readily accessible within the library, to define various components within the system. However, it is imperative to recognize that this constraint enhances the model's analyzability, as it ensures that the semantics governing communication and execution are predetermined and distinct from the core computational operations performed.

3. Main Idea

3.1. The SADF MoC in ForSyDe

Within the framework of ForSyDe, a generic untimed (UT) MoC is established, boasting a degree of expressiveness on par with that of the general dynamic dataflow model. In this context, we introduce the SADF MoC, characterized by two pivotal process constructors: $kernel_{mn}$ and $detector_{mn}$. These constructors are instrumental in the creation of SADF kernels and detectors, respectively, featuring multiple inputs and outputs. It is pertinent to note that these processes operate on UT signals denoted as $\dot{s} \in \dot{S}$, wherein event consumption and production are contingent upon events denoted as $\dot{e} \in \dot{E}$.

Fig. 3 provides the comprehensive specification of these process constructors. The $kernel_{mn}$ constructor accepts two parameters and yields a process $p_k : \dot{S} \times \dot{S}^m \to \dot{S}^n$ with characteristics as follows: it possesses one control input, m data inputs, and n data output signals. The first parameter entails a function f, expressed as $f : \dot{E} \times \dot{S}^m \to \dot{S}^n$. This function operates on the scenario event $\dot{e} \in \dot{E}$ to map m input sequences $\dot{S}^m$ to n output sequences $\dot{S}^n$ based on the scenario event. The second parameter, denoted as $\psi : \dot{E} \to \mathbb{N}^m \times \mathbb{N}^n$, represents the kernel scenario mapping. This mapping associates each scenario event $\dot{e}_\psi \in \dot{E}$ retrieved from the control input with input token consumption rates $r_{c,j}; 1 \le j \le m$ and output token production rates $r_{p,k}; 1 \le k \le n$. The partitioning function π subdivides the signals in accordance with the aforementioned rates. In this manner, during the i^{th} iteration, a sequence of control events $a_{i,c}$ of length one, along with m sequences of data inputs $a_{i,j}$, is provided to the function f. Consequently, n sequences of data outputs $a'_{i,k}$ are generated.

As for $detector_{mn}$, the process constructor requires five arguments to create an m-input n-output detector process $p_d : \dot{S}^m \to \dot{S}^n$. First, the detector scenario function $f_{ds} : \dot{E} \times \dot{S}^m \to \dot{E}$, determines the next scenario w_{i+1} based on the current scenario w_i and m input sequences $a_{i,j}$. The kernel scenario function $f_{ks} : \dot{E} \times \dot{S}^m \to \dot{S}^n$ uses the updated scenario along with the same input sequences to generate scenario events $a'_{i,k}$ for n kernels on its outputs. Next, the detector scenario mapping $\psi : \dot{E} \to \mathbb{N}^n$ determines the output productions rates $r_{p,k}$ based on the updated scenario. The initial scenario w_0 and a tuple of fixed input consumption rates $r_c \in \mathbb{N}^n$ are the last arguments to $detector_{mn}$.

Regarding the $detector_{mn}$ process constructor, the creation of an m-input n-output detector process $p_d : \dot{S}^m \to \dot{S}^n$ necessitates the provision of five arguments, leading to its definition as in Fig. 3b. Initially, we encounter the "detector scenario function", denoted as $f_{ds} : \dot{E} \times \dot{S}^m \to \dot{E}$, which plays the pivotal role of determining the subsequent scenario w_{i+1} based on the current scenario w_i and m input sequences $a_{i,j}$. Following this, the "kernel scenario function" denoted as $f_{ks} : \dot{E} \times \dot{S}^m \to \dot{S}^n$ utilizes

$$
\begin{aligned}
\mathit{kernel}_{mn}(f,\psi) &= p_k\\
&\text{where}\\
p_k(s_c, s_1,\dots,s_m) &= (s'_1,\dots,s'_n)\\
f(c_i, a_{i,1},\dots,a_{i,m}) &= (a'_{i,1},\dots,a'_{i,n})\\
\psi(c_i) &= (r_c, r_p), r_c = (r_{c,1},\dots,r_{c,m}), r_p = (r_{p,1},\dots,r_{p,n})\\
\pi(\nu_c, s_c) &= \langle c_i\rangle, \nu_c(i) = 1\\
\pi(\nu, s_j) &= \langle a_{i,j}\rangle, \nu_j(i) = r_{c,j};\ 1 \le j \le m\\
\pi(\nu'_k, s'_k) &= \langle a'_{i,k}\rangle, \nu'_k(i) = \mathsf{length}(a'_{i,k}) = r_{p,k}, \mathsf{rem}(\pi,\gamma'_k,s'_k) = \langle\rangle;\ 1 \le k \le n\\
s_c, s_j, s'_k, c_i, a_{i,j}, a'_{i,k} &\in \dot{S};\ i, r_{c,j}, r_{p,k} \in \mathbb{N}
\end{aligned}
$$

(a)

$$
\begin{aligned}
\mathit{detector}_{mn}(f_{ds}, f_{ks}, \psi, w_0, r_c) &= p_d\\
&\text{where}\\
p_d(s_1,\dots,s_m) &= (s'_1,\dots,s'_n)\\
f_{ds}(w_i, a_{i,1},\dots,a_{i,m}) &= w_{i+1}\\
f_{ks}(w_{i+1}, a_{i,1},\dots,a_{i,m}) &= (a'_{i,1},\dots,a'_{i,n})\\
\psi(w_{i+1}) &= r_p, r_p = (r_{p,1},\dots,r_{p,n})\\
\pi(\nu, s_j) &= \langle a_{i,j}\rangle, \nu_j(i) = r_{c,j};\ 1 \le j \le m\\
\pi(\nu, s'_k) &= \langle a'_{i,k}\rangle, \nu'_k(i) = \mathsf{length}(a'_{i,k}) = r_{p,k}, \mathsf{rem}(\pi,\gamma'_k,s'_k) = \langle\rangle;\ 1 \le k \le n\\
s_j, s'_k, , a_{i,j}, a'_{i,k} &\in \dot{S};\ w_i \in E;\ i, r_{c,j}, r_{p,k} \in \mathbb{N}
\end{aligned}
$$

(b)

Fig. 3: Definition of SADF process constructors for kernel (a) and detector (b) with m inputs and n outputs.

the updated scenario, in conjunction with the same input sequences, to generate scenario events $a'_{i,k}$ for n kernels on its outputs. Subsequently, the "detector scenario mapping" denoted as $\psi : \dot{E} \to \mathbb{N}^n$ comes into play, effectively ascertaining the output production rates $r_{p,k}$ based on the updated scenario. Finally, the initial scenario w_0 and a tuple of fixed input consumption rates $r_c \in \mathbb{N}^n$ conclude the set of arguments required for $\mathit{detector}_{mn}$.

Consistent with the principles outlined in reference [16], the combination of the aforementioned process constructors, coupled with the *delay* (or *init*) function, which introduces initial tokens onto signals (SADF channels), alongside the sequential, parallel, and feedback process composition operators, collectively formulate the foundation of the SADF MoC.

3.2. Mapping SADF Processes onto the Refined Layer of ForSyDe

Fig. 4 shows mapping the key process types in the SADF MoC—as defined in Fig. 3—onto the ForSyDe abstract semantics. The *Init* stages simply allocate space for input and output tokens with fixed rates. The *Prep* stage for the kernel process, reads a scenario token a^c from the control port, determines the consumption/production rates of input/output ports, resizes the corresponding buffers in the process, and reads the required number of tokens from each of the input ports. The prepared input buffers, together with the current scenario are passed to the kernel function to produce the outputs in the corresponding buffers in the *Apply* stage. Finally, in each iteration, the *Prod* stage writes out the output buffers to the output ports.

Fig. 4 elucidates the mapping of key process types within the SADF MoC, as previously defined in Fig. 3, onto the abstract semantics of ForSyDe. Within this context:

- The *Init* stages are primarily responsible for the allocation of space, setting fixed rates for input and output tokens.
- The *Prep* stage associated with the kernel process reads a scenario token a^c from the control port. Subsequently, it determines the consumption and production rates of input and output ports, adjusts the corresponding buffers within the process, and retrieves the requisite number of tokens from each of the input ports.
- The prepared input buffers, in conjunction with the current scenario, are then forwarded to the kernel function, where they are utilized to generate the outputs within the corresponding buffers during the *Apply* stage.
- Lastly, in each iteration, the *Prod* stage is responsible for writing out the contents of the output buffers to the output ports.

Regarding the detector process, the *Prep* stage consistently reads a predetermined number of tokens from the input sources. Within the *Apply* stage of the detector, the process commences by updating the detector scenario. This update relies on the current scenario and the acquired input data, with the aid of the detector scenario function f_{ds}. Subsequently, production rates are extracted, contingent upon the current scenario, thereby determining the appropriate sizing of the output buffers. Finally, the f_{ks} function, representing the kernel scenario function, is applied to the current scenario and input buffers, leading to the generation of output tokens. The *Prod* stage for the detector closely mirrors the one established for the kernel process.

```
$Init_{kernelSADF_{mn}}$
ALLOC($a^c$)

$Prep_{kernelSADF_{mn}}$
$a^c \leftarrow$ READ($in_c$)                              ▷ read the control token
$(r^c_{1...m}, r^p_{1...n}) \leftarrow \psi(a^c)$           ▷ consumption/production rates
for all $1 \le i \le m$ do
    RESIZE($a^{in}_i$,$r^c_i$)                             ▷ resize input buffers
    for all $1 \le j \le r^c_i$ do                           ▷ read inputs
        $a^{in}_i[j] \leftarrow$ READ($in_i$)
    end for
end for
for all $1 \le i \le n$ do
    RESIZE($a^{out}_i$,$r^p_i$)                            ▷ resize output buffers
end for

$Apply_{kernelSADF_{mn}}$
$(a^{out}_1, \ldots, a^{out}_n) \leftarrow f(a^c, a^{in}_1, \ldots, a^{in}_m)$

$Prod_{kernelSADF_{mn}}$
for all $1 \le i \le n$ do                                   ▷ write output tokens
    for all $1 \le j \le r^p_i$ do
        WRITE($out_i$, $a^{out}_i[j]$)
    end for
end for
```

(a)

```
$Init_{detectorSADF_{mn}}$
$\forall 1 \le i \le m$ : ALLOC($a_i$, $r^c_i$)
$a^s \leftarrow w_0$

$Prep_{detectorSADF_{mn}}$
for all $1 \le i \le m$ do                                   ▷ read input tokens
    for all $1 \le j \le r^c_i$ do
        $a^{in}_i[j] \leftarrow$ READ($in_i$)
    end for
end for

$Apply_{detectorSADF_{mn}}$
$a^s \leftarrow f_{ds}(a^s, a^{in}_1, \ldots, a^{in}_m)$
$(r^p_{1...n}) \leftarrow \psi(a^s)$                         ▷ production rates
for all $1 \le i \le n$ do
    RESIZE($a^{out}_i$,$r^p_i$)                            ▷ resize output buffers
end for
$(a^{out}_1, \ldots, a^{out}_n) \leftarrow f_{ks}(a^s, a^{in}_1, \ldots, a^{in}_m)$

$Prod_{detectorSADF_{mn}}$
for all $1 \le i \le n$ do                                   ▷ write output tokens
    for all $1 \le j \le r^p_i$ do
        WRITE($out_i$, $a^{out}_i[j]$)
    end for
end for
```

(b)

Fig. 4: Mapping SADF process constructors of Fig. 3 to the refined layer of ForSyDe for kernel (a) and detector (b) processes.

3.3. Realization in SystemC

Following the mapping of processes to the refined layer, the SADF MoC is introduced as a novel addition to ForSyDe-SystemC. This extension adheres to and builds upon the foundational principles initially delineated in the original work [3]. The tangible implementation of the SADF MoC is contributed back to the principal repository of the framework[1].

Notably, the modeling framework places significant emphasis on ensuring type and size safety for the data. Consequently, several decisions made during the implementation deviate from a rudimentary or simplistic approach. Some of these distinct considerations are briefly highlighted below.

- The `kernelMN` class, serving as the embodiment of the *kernel*$_{mn}$ process constructor, necessitates robust static type safety across a variable number of arguments for both inputs and outputs. To accomplish this, variadic templates and partial template specialization within C++ are judiciously employed to define such a class. Presented below is an excerpt of the class declaration for `kernelMN`, wherein `TOs` represent output types, `TIs` denote input types, and `TC` corresponds to the control token (scenario) type—each dynamically specified upon instantiation.

```
template<typename TO_tuple, typename TC, typename TI_tuple>
class kernelMN;

template<typename... TOs, typename TC, typename... TIs>
class kernelMN<std::tuple<TOs...>,TC,std::tuple<TIs...> >
    : public SADF_process
{
  // Implementation
}
```

- In contrast to analogous methodologies documented in [7], scenario mappings denoted as ψ are implemented within the processes as local tables. This approach is adopted to uphold static safety, all while ensuring compatibility. These scenario mappings are realized through the use of C++ Standard Template Library (STL) types. For instance, consider the definition of the scenario table type for the kernel, structured as a `std::map` that maps from the control token type `TC` to a `std::tuple` comprising fixed-size `std::array`, outlined as follows.

```
typedef std::map<TC,std::tuple<
                std::array<size_t, sizeof...(TIs)>,
                std::array<size_t, sizeof...(TOs)>
        >> scenario_table_type;
```

[1]`https://github.com/forsyde/ForSyDe-SystemC`

- Traversing variadic template types can prove challenging in classic C++, often necessitating compile-time recursion through template meta-programming. However, with the advent of modern extensions in C++17, notably `std::apply` and fold expressions [19], it becomes feasible to apply a function to the elements of a tuple in a type-safe manner during compile time. To illustrate this concept, consider the following code snippet, which showcases the potential implementation of the *Prod* stage within the constructors. In this context, `oport` represents a `std::tuple` consisting of *n* output ports, while `ovals` signifies a `std::tuple` comprising *n* output buffers intended for writing.

```
std::apply([&](auto&&... port){
  std::apply([&](auto&&... val){
    (write_vec_multiport(port, val), ...);
  }, ovals);
}, oport);
```

As an illustration of the library's practical application, we can model the decoding kernel process "D" for the system depicted in Fig. 1 as demonstrated below. Notably, automatic type deduction is employed, alleviating the need for explicit specification of template types, which streamlines the modeling process. In addition, structured binding feature of C++17 allows a more expressive assignment of inputs and outputs.

```
auto d_func = [](auto&& out, const auto& sc, const auto& inp) {
    const auto& [inpEp, inpEm, inpEc] = inp;
    auto&& [outR] = out;

    switch (sc) {
        case Sp:
            outR[0] = inpEp[0] - 1;
            break;
        case Sm:
            outR[0] = inpEm[0] + 1;
            break;
        case Sc:
            outR[0] = (inpEc[0] + inpEc[1]) / 2;
            outR[1] = (inpEc[0] - inpEc[1]) / 2;
            break;
    }
};

SADF::make_kernelMN(
    "d",
    d_func,
    {
        {Sp,{{1,0,0},{1}}},
        {Sm,{{0,1,0},{1}}},
        {Sc,{{0,0,2},{2}}}
    }, // d_table
    tie(dtor),
    ktod,
    tie(eptod, emtod, ectod)
);
```

3.4. Extending Introspection for Dynamic Models

ForSyDe-SystemC models are endowed with the capability of self-reflection, a feature known as *introspection*. Introspection comes into play during the elaboration phase of SystemC, where it extracts the static structure of the models. This extraction manifests as a hierarchical graph comprising processes, each uniquely identified by its process constructors and the arguments supplied during instantiation. This intermediate representation has the capacity to be exported in an XML format, which proves invaluable for integrating ForSyDe-SystemC into various stages of a design flow, including but not limited to synthesis and verification processes.

The implementation of the SADF MoC is structured to facilitate the utilization of introspection, enabling the export of specified models for analysis by external tools. In particular, the output XML representation can be seamlessly converted into the XML format compatible with the SDF^3 tool [17]. This tool is proficient in conducting various analyses, including but not limited to deadlock freedom checks, throughput calculations, model transformations, visualization, and more. Fig. 5 serves as an exemplar, showcasing the introspection output of the system depicted in Fig. 1, effectively converted for visualization purposes using the `f2dot` tool.

In addition to the static introspection capabilities, a complementary dynamic introspection mechanism has been meticulously developed and harnessed by the SADF MoC. This dynamic feature empowers each of the kernel and detector processes to report their present scenario of operation. This valuable information can be amalgamated with the overarching model structure, providing insights into the SADF graph's composition and the model's status at any given moment. To mitigate undue computational overhead, all aspects related to dynamic introspection, including definitions and activities, can be selectively excluded from the model's implementation through the utilization of a compile-time macro definition. This allows for the efficient management of introspection features based on the specific requirements of the model.

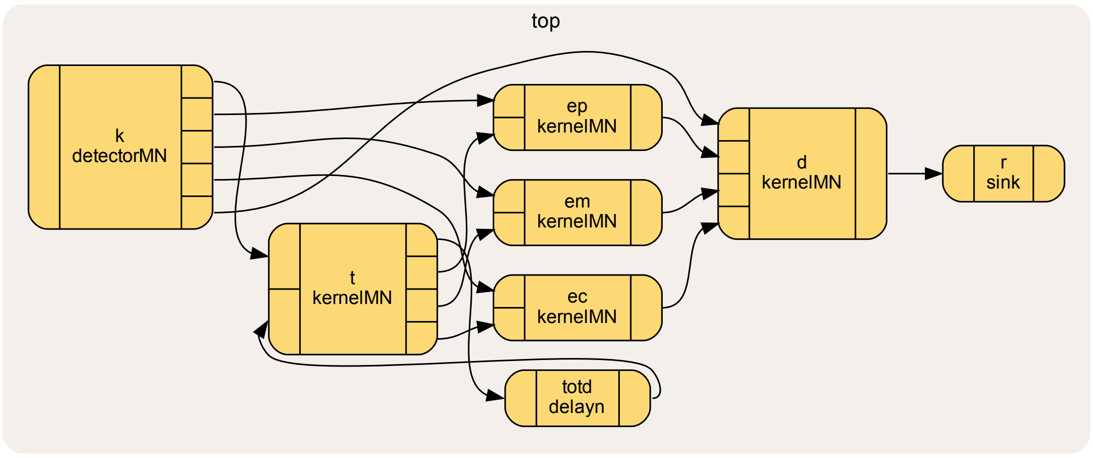


Fig. 5: Visualization of the introspection output for the tutorial example.

4. Experimental Results

In order to substantiate the assertions made in the paper, a series of experiments have been devised. First and foremost, to demonstrate the framework's precise functionality, we construct a model of the simple-profile MPEG-4 decoder (MPEG-4 SP), as initially introduced in the early SADF paper [22] and also used in similar works [7], employing the modeling framework presented in this paper. Furthermore, we undertake an assessment of the framework's performance and scalability by employing synthetic models generated by SDF^3. These synthetic models serve as valuable benchmarks for evaluating the framework's efficiency and its ability to handle increasingly complex and resource-intensive scenarios.

4.1. The MPEG-4 Decoder

A video decoder designed to support MPEG-4 SP is specifically tailored to handle video streams exclusively comprising intra-coded (I) and predicted (P) frames. These frames are constructed from multiple macro blocks, each necessitating various operations, including Variable Length Decoding (VLD), Inverse Discrete Cosine Transformation (IDCT), Motion Compensation (MC), and the Reconstruction (RC) of the resulting image.

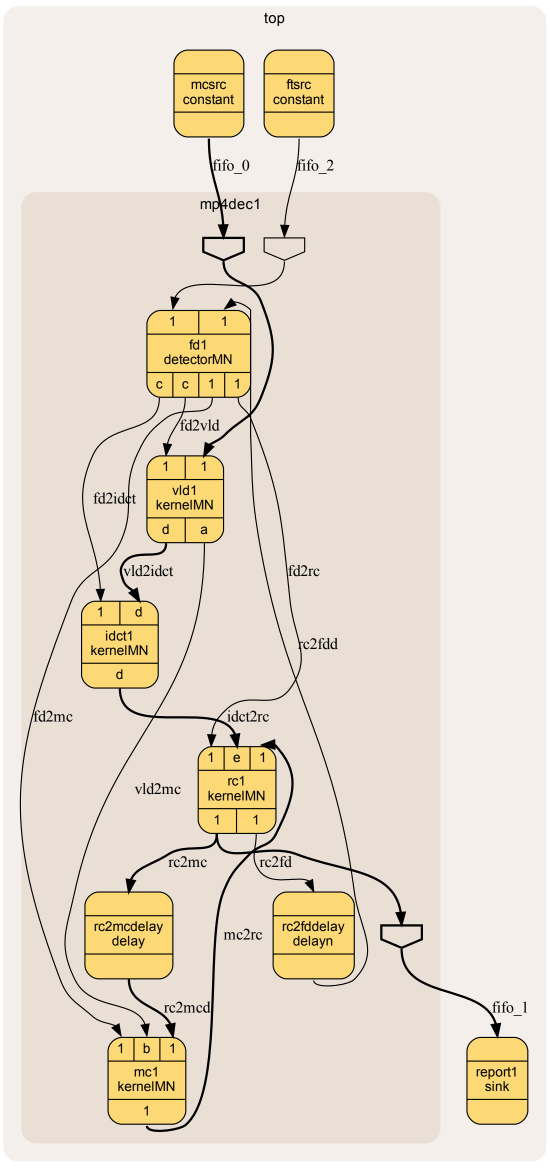
The crucial observation regarding MPEG-4 decoding lies in the significant variation in computational demand and resource requirements when decoding different frame types. For instance, during the decoding of I-frames or P-frames for still video, certain functions such as VLD and MC may be entirely omitted. Prior works [22] have demonstrated that designing such an application using conventional MoCs like SDF necessitates a conservative allocation of CPU time and memory buffers for the worst-case scenario, resulting in an inefficient implementation. An alternative design approach, discussed below, involves partitioning the behaviors into a finite set of scenarios and analyzing the application's behavior, performance, and cost metrics in each scenario.

We have created a ForSyDe-SystemC model representing an MPEG-4 simple profile decoder. The introspection output of this model is depicted in Fig. 6a, comprising four kernel processes, each corresponding to one of the aforementioned operations, alongside the frame detector (FD). This introspection output has been slightly adjusted to include details regarding the input and output rates of both the kernels and detectors, offering a more comprehensive view of the model's behavior and characteristics.

When decoding an I frame, all macro blocks need to undergo VLD and IDCT processes, while the resulting image is straightforwardly reconstructed through RC. Assuming an image size of $m \times n$ pixels, there are a total of $mb_I = \frac{m \times n}{k^2}$ macro blocks of size $k \times k$ to be decoded for I frames. For example, for QCIF images, $m = 176$, $n = 144$, $k = 16$, and the macro block set size for I frames is $mb_I = 99$. However, I frames do not require any motion vectors. Motion vectors are only relevant for MC when processing P frames, which involves determining the new position of macro blocks based on the previous frame. In such cases, the resulting image is adjusted using the pixel data obtained from the newly decoded macro blocks. The number of motion vectors and macro blocks to decode can vary for different P frames, ranging from zero to mb_I. When there are no motion vectors (zero motion vectors), it indicates the decoding of still video, where VLD and IDCT processes are not required, and MC simply copies the previously decoded frame.

As stated earlier, such an application exhibits a high level of dynamism and resource requirements in different scenarios and thus, an SADF model can capture this diversity more precisely. Assuming a separate scenario for decoding I frames, and P frames with a limited set of $|mb_I|$ values, Fig. 6b provides the token rates of kernels for each scenario type. For the purpose of throughput analysis, the designer must also estimate the execution times of the actors in each scenario. However, since this information is not directly relevant for functional modeling purposes, we recommend referring to the literature [22], for instance, for example data.

The FD detector receives the frame type signal and a synchronization signal consisting of Booleans, and configures all kernels with one of the predefined scenarios, as outlined in Table 6b. The VLD kernel decodes the input streaming of macro blocks and sends the decoded macro blocks to the IDCT kernel. In the case of decoding P-frames, the VLD kernel also sends the motion vectors to the MC kernel. During the decoding of I-frames, motion compensation is unnecessary. As a result, the MC kernel does not utilize motion vectors and generates a blank image consisting of a frame filled with zeros. In the case of P-frames, the MC kernel consumes the previously decoded frame along with a specific number of motion vectors corresponding to the type of



(a)

	Scenario		
Rate	I	P_0	P_x
a	0	0	1
b	0	0	x
c	99	1	x
d	1	0	1
e	99	0	x

$x \in \{30, 40, 50, 60, 70, 80, 99\}$

(b)

Fig. 6: Model of the MPEG-4 simple profile decoder [22]. (a) Annotated introspection output of the ForSyDe-SystemC model. (b) The scenario table.

P-frame being decoded. The output is a frame constructed by shifting the blocks of the previous frame according to the motion vectors. Finally, the RC kernel combines the motion-compensated frame with the macro blocks produced by the IDCT kernel. This process yields a decoded frame containing $m \times n$ pixels.

Table 1 provides a comparison of different approaches for functional modeling of MPEG-4 using the SADF MoC. Specifically, the SADF models of ForSyDe-SystemC, as contributed in this work, are compared to the approaches presented by Bonna et al. [7]. ForSyDe-SADF integrates SADF models into the Haskell-based implementation of ForSyDe, offering a concise and abstract syntax for modeling. However, its shallow embedded implementation in the host language makes it suitable only for simulation and does not provide an easy path for using the model for analysis and synthesis. Additionally, its unfamiliar syntax and semantics for typical designers make its adoption in industrial setups challenging.

The approaches named Imperative High-Level Languages (IHLL) provide templates to implement the kernels and detectors as concurrent tasks communicating using FIFO-like buffers in Python and SystemC. As stated in earlier sections, these approaches are more suitable for code generation and cannot be considered as comprehensive system-level modeling frameworks for embedded and CPS design. However, the ForSyDe-SystemC-based approach for SADF modeling allows us to model the MPEG-4 decoder using ForSyDe abstractions while being based on the familiar and efficient SystemC/C++ language. In addition, the MPEG-4 ForSyDe-SystemC models are exported as an intermediate representation, providing a path for the integration of analysis and synthesis tools.

The performance of all approaches for decoding 200 64×64 frames with macroblocks of size 8×8 is compared and reported in terms of decoded frames per second. The numbers for the first three cases are extracted from [7], and our model is executed on a quad-core Core-i7 Linux laptop. As evident, the MPEG-4 example in our case has significantly better performance compared to other approaches. Unfortunately, as the code for the IHLL SystemC is not published, it is not possible to make a thorough analysis

Table 1: Comparing the proposed framework for SADF modeling to similar approaches. Performance is reported in terms of decoded frames per second for 64×64 frames with macroblocks of size 8×8.

Modeling Framework	ForSyDe-SADF	IHIL Python	IHIL SystemC	ForSyDe-SystemC
Formal Modeling Abstractions	Y	N	N	Y
Industry-Friendly	N	Y	Y	Y
Model Analysis & Synthesis	N	N	N	Y(via introspection)
Performance (FPS)	12.35	3.37	51.32	2797

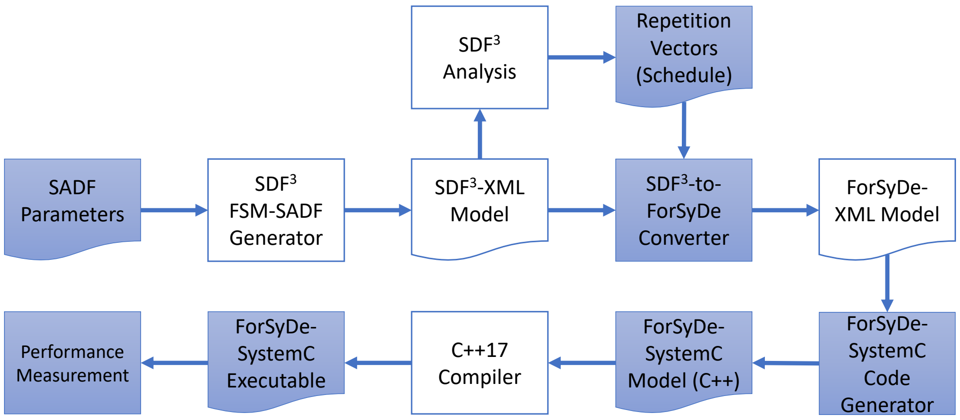

SDF³ Analysis
Repetition Vectors (Schedule)
SADF Parameters
SDF³ FSM-SADF Generator
SDF³-XML Model
SDF³-to-ForSyDe Converter
ForSyDe-XML Model
Performance Measurement
ForSyDe-SystemC Executable
C++17 Compiler
ForSyDe-SystemC Model (C++)
ForSyDe-SystemC Code Generator

Fig. 7: The developed tool flow used for synthetic benchmarks. The shaded boxes are developed in the context of this work.

of the root of such a performance gap. The possible causes are inefficient implementation and optimization of IHLL SystemC. The complete model is added to the ForSyDe-SystemC repository for interested authors for inspection.

4.2. Synthetic Benchmarks

The size and complexity of an SADF model can be influenced by various parameters. To assess the performance of our framework across a spectrum of valid SADF graphs with differing characteristics, we have devised a tool pipeline designed to generate synthetic models with adjustable parameters. The depicted tool flow, as illustrated in Fig. 7, has been implemented and is accessible in a dedicated code repository[2]. This tool pipeline serves as a valuable resource for systematically exploring the behavior and scalability of our framework under diverse conditions and model configurations.

The user input for generating synthetic SADF graphs involves specifying a list of parameters, including:

1. The number of kernel actors.
2. The average number of input and output channels (degree) for each actor.
3. The average execution time for each actor.
4. The number of graph scenarios.
5. The average size of tokens communicated through the channels.

The SDF^3 graph generator tool leverages an algorithm to produce valid, deadlock-free, FSM-based SADF. A slight modification has been made to the analysis tool of SDF^3, enabling the generation of schedules for each scenario in the form of repetition vectors.

To transform the generated graph into a ForSyDe model in XML format, a conversion tool has been developed. This format is an implementation of the ForSyDe intermediate representation [3]. This tool incorporates the addition of a detector and extraction of scenario tables, among other tasks. Subsequently, the ForSyDe-SystemC code generator takes the converted ForSyDe-XML and produces a simulation model in SystemC/C++. This step involves generating placeholder code for kernel functions, approximating the intended execution times using a busy loop.

Both the converter and the code generator are implemented in Python, and a top-level script orchestrates the execution of these steps in sequence. Ultimately, the compiled simulation models are executed, and the measured execution times are reported, facilitating the evaluation and analysis of the framework's performance under varying synthetic model conditions.

Fig. 8 provides a comprehensive overview of simulation performance, with each graph parameter undergoing variation for analysis. In cases where a parameter is not subjected to change, the default values for graph generation are as follows: 20 for the number of actors, an average degree of connectivity in the graph set to 2, an average intended execution time of actors at 20^{ms}, 2 scenarios, and token sizes of 2 words for communication within the channels. The experiments have been conducted on a system equipped with a 7^{th}-generation Core i7 Intel processor boasting 16^{GB} of RAM and running a Linux operating system. The models are compiled utilizing g++ 11.4 with optimization set to level 2. All of the raw input data and intermediate models are available in the repository for further investigation by the interested reader.

[2]`https://github.com/SBU-CPS-Lab/forsyde-systemc-sadf-experiments`

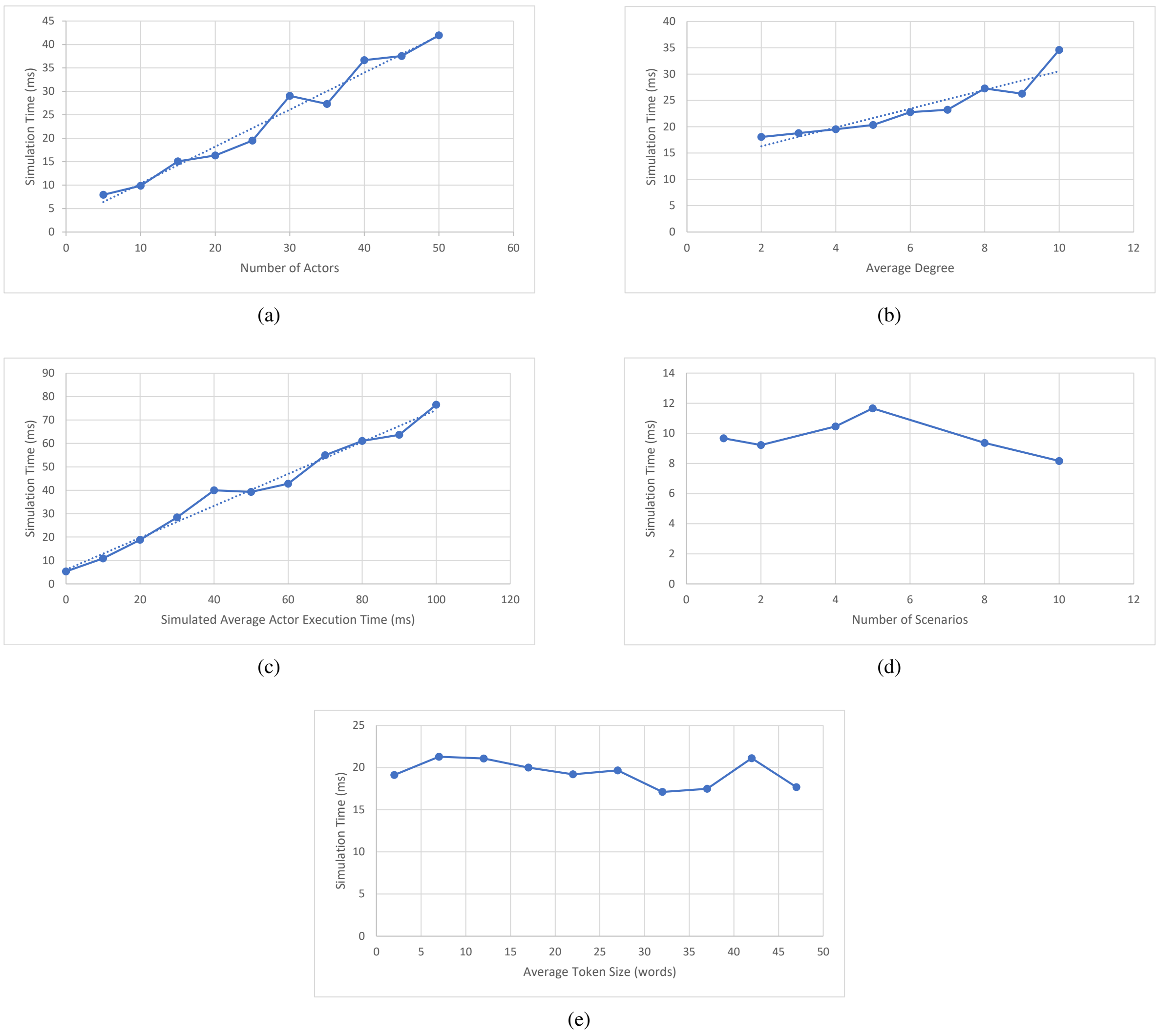


Fig. 8: The simulation time for 100 graph iterations with respect to (a) number of actors in the graph; (b) average degree of the graph; (c) average execution time of the actors; (d) number of scenarios; and (e) communicated token sizes. Dashed lines represent linear interpolation of the values.

Observing the results, it is evident that simulation time exhibits a linear increase as the number of actors (Fig. 8a), the degree of graph connectivity (Fig. 8b), and the intended average execution time of actors (Fig. 8c) increase. Conversely, simulation times remain relatively stable even with variations in the number of scenarios (Fig. 8d) and token sizes (Fig. 8e). These results effectively demonstrate the scalability of our modeling framework across the analyzed application scales, considering the chosen granularity of average actor execution times.

5. Conclusion

Effective modeling abstractions and tools play a pivotal role in assisting designers in managing the intricacies inherent in dynamic and intelligent embedded and cyber-physical systems (CPSs). In this context, we have introduced a modeling framework founded upon the scenario-aware dataflow (SADF) model of computation (MoC). This framework adeptly captures the inherent dynamism within CPSs through a controlled representation of scenarios.

In our contribution, we have provided a rigorous denotational-style semantics for modeling within the context of the **For**mal **Sy**stem **De**sign (ForSyDe) framework. Additionally, we have presented an operational-style semantics tailored for practical implementation within industrial languages. The incorporation of the SADF MoC into the ForSyDe-SystemC modeling framework is achieved through the utilization of contemporary features of the C++ language. This integration inherits several advantageous properties of the framework, including type- and size-safety, model introspection, parallel simulation capabilities, and seamless foreign model integration.

To illustrate the effectiveness and versatility of the framework, we have presented demonstrational examples encompassing an encoder/decoder system and the simple profile of the MPEG-4 algorithm, serving as case studies. These exemplars showcase the correctness and practicality of the framework in real-world scenarios. Furthermore, we have established a tool flow for the automatic generation of synthetic benchmarks. These benchmarks are instrumental in scrutinizing and substantiating the scalability of ForSyDe-SystemC SADF models, providing valuable insights into their performance under diverse conditions.

6. Acknowledgment

This publication was partially supported by grant No. RD-51-9911-0028 from the R&D Center of Mobile Telecommunication Company of Iran (MCI) for advancing information and communications technologies.

Appendix A. Acronyms

CPS	cyber-physical system
CT	continuous-time
DDE	distributed discrete-event
DE	discrete-event
ESL	electronic system-level
ForSyDe	**For**mal **Sy**stem **De**sign
FSM	finite-state machine
KPN	Kahn process network
MI	MoC interface
MoC	model of computation
SADF	scenario-aware dataflow
SDF	synchronous dataflow
SoC	system on chip
SY	synchronous
UT	untimed

References

[1] Kenji Asano, Junji Kitamichi, and Kenichi Kuroda. Dynamic Module Library for System Level Modeling and Simulation of Dynamically Reconfigurable Systems. *Journal of Computers*, 3(2):55–62, February 2008.

[2] IEEE Standards Association et al. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2011.

[3] Seyed-Hosein Attarzadeh-Niaki and Ingo Sander. An extensible modeling methodology for embedded and cyber-physical system design. *Simulation*, 92(8):771–794, August 2016.

[4] Seyed-Hosein Attarzadeh-Niaki and Ingo Sander. Heterogeneous co-simulation for embedded and cyber-physical systems design. *SIMULATION*, 96(9):753–765, 2020.

[5] Seyed-Hosein Attarzadeh-Niaki, Ingo Sander, and Mohammad Ahmadi. An automated parallel simulation flow for cyber-physical system design. *Integration*, 77:48–58, 2021.

[6] K. Bellman, C. Landauer, N. Dutt, L. Esterle, A. Herkersdorf, A. Jantsch, N. TaheriNejad, P. R. Lewis, M. Platzner, and K. Tammemäe. Self-aware cyber-physical systems. *ACM Trans. Cyber-Phys. Syst.*, 4(4), jun 2020.

[7] Ricardo Bonna, Denis S. Loubach, George Ungureanu, and Ingo Sander. Modeling and simulation of dynamic applications using scenario-aware dataflow. *ACM Trans. Des. Autom. Electron. Syst.*, 24(5), aug 2019.

[8] Francky Catthoor, Twan Basten, Nikolaos Zompakis, Marc Geilen, and Per Gunnar Kjeldsberg. *System-scenario-based design principles and applications*, volume 1. Springer, 2020.

[9] Gabriel C. Duarte and Denis S. Loubach. An analysis on formal models of computation for the avionics systems domain. In *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, pages 1–9, 2023.

[10] Fernando Herrera and Eugenio Villar. A framework for heterogeneous specification and design of electronic embedded systems in SystemC. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):22:1–22:31, May 2008.

[11] Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74(471-475):15–28, 1974.

[12] G. Martin, B. Bailey, A. Piziali, M. Burton, J. Greenbaum, K. Hashmi, A. Haverinen, L. Lavagno, M. Meredith, B. Murray, et al. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Elsevier Science, 2007.

[13] Svetlana Minakova, Dolly Sapra, Todor Stefanov, and Andy D. Pimentel. Scenario based run-time switching for adaptive CNN-based applications at the edge. *ACM Trans. Embed. Comput. Syst.*, 21(2), feb 2022.

[14] Thomas Martyn Parks. *Bounded scheduling of process networks*. PhD thesis, University of California, Berkeley, USA, 1995.

[15] Andreas Raabe, Philipp A. Hartmann, and Joachim K. Anlauf. ReChannel: Describing and Simulating Reconfigurable Hardware in SystemC. *ACM Trans. Des. Autom. Electron. Syst.*, 13(1), feb 2008.

[16] Ingo Sander, Axel Jantsch, and Seyed-Hosein Attarzadeh-Niaki. *ForSyDe: System Design Using a Functional Language and Models of Computation*, pages 99–140. Springer Netherlands, Dordrecht, 2017.

[17] S. Stuijk, M.C.W. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

[18] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 404–411, 2011.

[19] Andrew Sutton and Richard Smith. Fold expressions (Document number: N4295), November 2014.

[20] Elena Suvorova, Nadezhda Matveeva, Alexey Rabin, and Valentin Rozanov. System level modeling of dynamic reconfigurable system-on-chip. In *2015 17th Conference of Open Innovations Association (FRUCT)*, pages 222–229, 2015.

[21] Bart D. Theelen, Ed F. Deprettere, and Shuvra S. Bhattacharyya. *Dynamic Dataflow Graphs*, pages 1173–1210. Springer International Publishing, Cham, 2019.

[22] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, pages 185–194, 2006.

Seyed-Hosein Attarzadeh-Niaki is an assistant professor in computer systems architecture at Shahid Beheshti University GC (SBU), Tehran, Iran. His research interests include modeling and design methodologies for smart and safe realtime embedded and cyber-physical systems.

Mohammad Vazirpanah is currently a research fellow in the Department of Computer Science at the University of Salerno, Italy. His research interests include Embedded and Cyber-Physical Systems (CPS), as well as High-Performance Computing (HPC).