Model-based Cosimulation for Industrial Wireless Networks

Jing Geng*, Honglei Li*, Yanzhou Liu*, Yongkang Liu†,
Mohamed Kashef†, Richard Candell†, Shuvra S. Bhattacharyya*‡,
* Department of Electrical and Computer Engineering, University of Maryland, College Park, USA
† Intelligent Systems Division, National Institute of Standards and Technology, USA
‡ Department of Pervasive Computing, Tampere University of Technology, Finland
Email: {jgeng, honglei, yzliu}@umd.edu, {yongkang.liu, mohamed.hany, richard.candell}@nist.gov, ssb@umd.edu

***Abstract*—Wireless communications technology has the potential to provide major benefits in lowering the cost and increasing the efficiency of factory automation (FA) systems. However, design of FA systems that employ wireless networks involves stringent constraints on real-time performance and reliability, and requires the assessment of and experimentation with complex interactions among process control, factory topology construction (layout and connectivity of subsystems, such as machines, rails, etc.), and wireless communication. In this paper, we introduce a novel simulation framework to support such assessment and experimentation in the design of next-generation FA systems. Our simulation framework employs model-based design principles to enhance design reliability, and enable systematic and efficient integration of control, topology, and network modeling aspects. We demonstrate the utility of our framework through a case study that involves topology design and scalability analysis for a large class of FA systems. Our results demonstrate the ability of the proposed framework to provide insights on complex design trade-offs, while the underlying model-based features enhance efficient and reliable system-level integration.**

I. Introduction

Wireless communications technology has the potential to provide major benefits in lowering the cost and increasing the efficiency of factory automation (FA) systems. In particular, wireless communication significantly reduces costs for network deployment, and allows for integration of sensing devices and associated monitoring capabilities in parts of the system that are difficult or impossible to connect through wired sensing installations (e.g., see [1]).

However, to realize the full potential of wireless communications integration with FA, significant challenges must be overcome in ensuring the reliability, real-time operational speed, and energy efficiency of industrial wireless networks. This is a major challenge due to the conflicting relationships among these metrics, and the complexity of the system-level design space, which includes key parameters at the layers of communication protocols, FA system topologies (placement and connectivity of units), and the embedded processing platforms (hardware and software) within the network nodes.

To enable efficient experimentation, validation, and optimization processes in the context of this complex design space, effective simulation methodologies are of critical importance. In this paper, we introduce a novel simulation framework to address the challenge of simulating complex FA systems that are integrated with wireless communication techniques, networks, and connections. Our framework is novel in its model-based approach, where different models are employed to efficiently and flexibly integrate the design perspectives of FA topologies (system-level structure in the form of networks of machinery and control units), FA nodes (processing within individual nodes of a networked FA system), and wireless communications (protocol functionality and channel characteristics). Model-based design is an important paradigm for cyber-physical systems in which systems are specified through components that interact through formal models of computation [2].

In particular, we apply the *dataflow* model of computation in new ways to formally capture key aspects of FA system behavior and integrate these with discrete event models of computation that are used in state-of-the-art simulators for communication networks. Our application of dataflow principles enables dataflow-based analysis of FA process flows (system-level analysis) and signal and information processing that takes places within individual FA nodes, while providing efficient mechanisms for leveraging and interfacing to arbitrary discrete-event-based network simulation tools.

Our simulation framework is built on top of the *lightweight dataflow environment* (*LIDE*), which is a design tool for dataflow-based design and implementation of signal and information processing systems [3], [4]. Distinguishing features of LIDE are that it is minimally intrusive on existing design processes, and it is easily adaptable to different platform- or simulation-oriented programming languages, such as C, C++, Verilog, VHDL, and MATLAB. LIDE achieves these features through its formulation in terms of a compact set of application programming interfaces (APIs) for constructing software or hardware components that communicate through dataflow interfaces, and for building up graphs as connections of such dataflow-based components. In our simulator prototype, we specifically employ *LIDE-C*, which provides C-language implementations of the LIDE APIs. However, the simulator can be retargeted readily to other languages by exploiting the retargetability of LIDE described above.

In our simulation framework for FA systems, we incorporate

978-1-5386-1066-4/18/$31.00 ©2018 IEEE

extensions to LIDE for managing time because the dataflow model of computation, which LIDE is based on, is an untimed model with no built-in concept of time or time stamps. In dataflow, functional components (*actors*) synchronize with one another based only on the availability of data. The "time-extended" version of LIDE that we employ is referred to as τLIDE, where τ represents the incorporated notion of time.

Based on its foundations in τLIDE, we refer to our new factory simulation framework as τLIDE–factorysim, which we abbreviate as TLFS (Tau Lide Factory Sim). The TLFS framework can be viewed as a design tool that applies timed dataflow concepts in novel ways to enable model-based cosimulation of factory process flows together with discrete-event simulation of communication networks that link physically-separated subsystems within the factories.

Rather than re-invent functionality for simulating communication networks, TLFS defines a structured process for integrating third-party network simulation tools with the dataflow-based factory simulation capabilities within TLFS. Thus, TLFS is agnostic to any specific network simulation tool, and can be adapted in a modular fashion to work with different network simulators depending on the designer's preference. In our first-version prototype of TLFS, we apply NS-3 [5] as the network simulation tool that is incorporated into the framework. However, the underlying architecture of TLFS is not dependent on NS-3, and can be adapted to work with other network simulation tools, as described above.

The remainder of this paper is organized as follows. Section II provides background on dataflow modeling that the later sections in the paper depend on. Section III discusses related work and positions the contribution of this paper in the context of other simulation techniques and tools. Section IV presents the architecture of the TLFS simulation framework. Section V presents an example to concretely demonstrate the process of modeling factory process flows in TLFS using its' underlying dataflow capabilities. Section VI presents experiments that demonstrate the utility of TLFS in assessing design trade-offs across different combinations of factory process flow and networking configurations. Section VII summarizes the contributions of the paper and points to useful directions for further investigation.

II. Background

Dataflow is widely used as a programming model in many areas of embedded and cyber-physical systems, especially in areas that involve significant emphasis on digital communication, signal and information processing, or both. Model-based design processes in terms of dataflow graphs provide useful features such as efficient retargetability to different implementation languages and platforms; support for deterministic, concurrent programming; and support for a wide variety of analysis and optimization techniques (e.g., see [6], [7]).

A variety of specific forms of dataflow has been developed and continues to evolve. These different forms of dataflow provide different trade-offs between generality (the class of applications that can be expressed) and *analysis potential* — that is, the tractability of analysis and optimization problems or the availability of algorithms that address such problems. For background on dataflow models and various specialized forms of dataflow in embedded and cyber-physical systems, we refer the reader to [7].

The specific form of dataflow used in TLFS, inherited from the modeling approach in LIDE, is referred to as *core functional dataflow* (*CFDF*) [8]. CFDF is a general (Turing complete) form of dataflow that can be used to model a wide variety of applications. It is possible to apply more restricted models in TLFS that trade-off some of the generality of CFDF to enable more powerful analysis and optimization techniques. Such exploration is a useful direction for future work that builds naturally on the developments of this paper.

In the remainder of this section, we provide a brief introduction to CFDF semantics. This background is useful in understanding the methodology employed in TLFS for modeling FA systems.

In CFDF, as in other dataflow programming variants, the application system is represented as a directed graph in which vertices (*actors*) correspond to functional modules, and each edge e corresponds to a first-in, first-out (FIFO) buffer. This buffer represents the storage of data as it passes from the source actor of e to the sink actor of e. Each unit of data that passes along an edge is referred to as a *token*. Tokens can be of arbitrary data types, such as integers, floating point numbers, or objects of any complexity.

In a simulation model of a factory system, each token that passes through a given edge can represent some physical entity (such as a part) that moves from one factory subsystem (e.g., a machine) to another subsystem. Alternatively, a token can represent some form of information, such as a command that is sent from a factory controller module to a machine or rail that is controlled by the module.

In dataflow, the execution of an actor is decomposed into discrete units of execution called *firings* of the actor. On each firing, an actor consumes zero or more tokens from each of its input edges and produces zero or more tokens onto each output edge. The numbers of tokens that are produced and consumed in this way are referred to as the *production and consumption rates* that are associated with the firings.

An actor A in CFDF consists of a set $\mu(A)$ of one or more *modes*, where each $m \in \mu(A)$ corresponds to a distinct state in which the actor can fire. In other words, each firing f of the actor is a associated with a specific mode in $\mu(A)$. As a side effect, each firing determines the *next mode* $n \in \mu(A)$ of the actor, which is the mode that will be associated with the next firing of the actor. An initial next mode is assigned to each actor as part of the initialization of a CFDF graph.

In a given CFDF mode, the production or consumption rate on each incident edge must be constant — that is, all firings in the same mode must have identical production (consumption) rates on a given output (input) edge of the actor. This is a key "design rule" of CFDF. However, the rates can vary between distinct modes. A CFDF actor can fire whenever the number of tokens buffered on each input edge is at least equal to the consumption rate of the edge for the actor's next mode.

As a simple example of a CFDF actor, consider the integer-typed *switch* actor that is illustrated in Figure 1. This actor has two input ports x_d (data) and x_c (control), and two output ports y_f and y_t. The ports x_d, y_f, and y_t have integer token type, while the token type of x_c is Boolean. The actor operates by copying each ith token consumed from x_d to y_f or y_t depending on whether the ith value consumed from x_c is `false` or `true`, respectively.

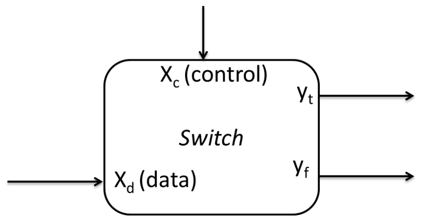

Fig. 1. A switch actor used to illustrate CFDF modeling.

This functionality can be implemented as a CFDF actor with three modes $\mu = \{m_c, m_t, m_f\}$. The subscripts here stand for "control", "true", and "false", respectively. In m_c, the actor consumes the next token ("control token") from x_c, and sets the actor's next mode to m_f or m_t depending on whether the value consumed is `false` or `true`, respectively. In m_f (m_t), the actor outputs a copy of the most-recently consumed control token onto y_f (y_t) and unconditionally sets the actor's next mode (back) to m_c.

One can easily verify, based on this description, that the number of tokens consumed and produced on each actor port is constant for each given mode. This is the basic "dataflow requirement" when decomposing an actor into modes for CFDF modeling. For example, the production rates associated with ports x_c, x_d, y_t, y_f in mode m_c are $1, 0, 0, 0$, respectively.

For more detailed background on dataflow modeling that is oriented to embedded and cyber-physical systems, we refer the reader to [6], [7]. For further details on CFDF, we refer the reader to [8]. For foundational concepts related to the dataflow paradigm, we refer the reader to [9], [10].

III. Related Work

A variety of approaches have been developed over the years that are relevant to the effective simulation of networked factory automation systems.

For example, Neema et al. have developed a model-based platform that integrates heterogeneous simulation tools for cyber-physical systems [11]. Düngen et al. demonstrate a simulation approach for parallel sequence spread spectrum signals in the context of industrial wireless communication environments [12]. Liu et al. present a simulation framework that integrates process control system modeling and wireless network modeling into a unified discrete-event simulator [13]. Bause et al. present a tool that integrates the ProC/B toolset for process chain modeling with the OMNeT++ tool for simulating communication networks [14]. Won et al. present a tool that integrates the targeted dataflow interchange format, a dataflow tool for embedded signal processing, with the NS-2 environment for communication network simulation [15].

The work presented in this paper differs from the related work in its emphasis on dataflow-based modeling of factory process flows, and its general approach for integrating dataflow-based process flow models with arbitrary discrete-event-based tools for communication network simulation.

For concreteness, we demonstrate our models and methods by integrating a specific dataflow-based design tool, LIDE, with a specific network simulation tool, NS-3. However, the architecture of TLFS is formulated carefully in terms of abstract dataflow principles and model-based interfacing principles between dataflow and network modeling subsystems. This use of abstraction promotes the adaptability of TLFS to incorporate other network simulation tools as well as other tools for dataflow-based design. This flexibility is important given that a variety of different network simulation tools exist with different combinations of features (e.g., see [16], [17]).

In this way, our contribution is complementary to related work such as the works discussed previously in this section. For example, the network simulation techniques applied in the works by Liu [13] and Düngen [12] can be interfaced with dataflow-based factory models using TLFS, and similarly, the techniques for embedded software synthesis used in the work of Won [15] can be applied to networked factory automation systems. Such extensions of TLFS represent a useful direction for future work.

IV. Simulation Approach

In this section, we present our simulation framework, called TLFS, for FA systems that are integrated with wireless communication networks. We describe the architecture of TLFS, which is designed to be integrated with arbitrary simulators that are specialized for communication network simulation (CNS). In Section VI, we present experiments using this architecture with the integration of a specific CNS tool.

Figure 2 illustrates the architecture of TLFS. The blue-, green-, and red-colored parts of the diagram (the top 2, middle 2, and bottom 2 blocks, respectively) correspond to parts that pertain to factory process flow simulation, interfacing (cosimulation) between the factory and network simulation subsystems, and communication network simulation.

In TLFS, a factory system is modeled as a dataflow graph in which vertices (actors) represent distinct physical or computational components within the system, and edges represent the flow of information or physical entities (e.g., parts for products that are being manufactured). Examples of TLFS actors include actors for representing individual machines within a factory, individual rails that connect different machines, and machine controllers that send signals to machines to instruct the machines to perform specific functions (such as loading parts on rails when specific conditions are met). More detailed actor examples will be presented in Section V.

As illustrated in Figure 2, cosimulation is enabled by the TLFS coordination module (or just *coordination module* in

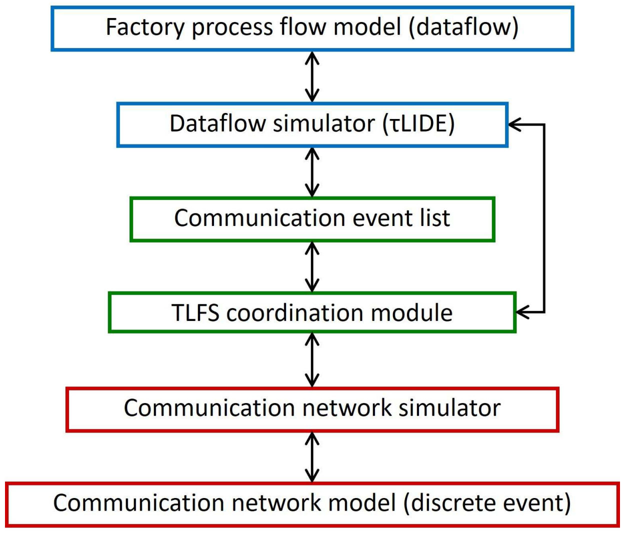
Factory process flow model (dataflow)
Dataflow simulator (τLIDE)
Communication event list
TLFS coordination module
Communication network simulator
Communication network model (discrete event)

Fig. 2. Cosimulation architecture based on TLFS.

abbreviated form), which transfers relevant simulation events between the factory and network simulation subsystems. There are two types of events that trigger event transfer through the coordination module: *dataflow send events*, which are generated in the factory simulation (dataflow simulator), and *network receive events*, which are generated in the CNS. These events are communicated between simulation subsystems using a dedicated event list, which is represented by the block in Figure 2 labeled “communication event list”, together with the coordination module, which acts as a gateway to achieve proper synchronization of communication events. More details on the process of managing dataflow send events and network receive events are described in Section IV-C.

A. Coordination Module

Figure 3 shows a pseudocode sketch of the coordination module. Here, `ntime` represents the value of simulated time in the CNS (*network time*), and `nresult` is a simple data structure that is used to communicate selected status information from the CNS tool to the coordination module. In particular, `nresult.result_time` gives the current value of network time, and `nresult.receive_events` provides the list of network receive events that have been generated in the CNS tool during the most recent call to the CNS tool. The function `save_receive_events` removes events from `nresult.receive_events`, and generates corresponding events in τLIDE that trigger processing of the received data (see Figure 3). The dataflow simulator has access to the communication event list directly through special *communication interface actors* within the factory process flow model. Communication interface actors are discussed further in Section IV-C.

The `next_lide_event_time` function is a method of τLIDE that returns the time of the earliest pending event within the event list of the dataflow (factory) simulation. The `simulate_network` function serves as a “wrapper” for

```
while ((ntime < time_limit) and (!deadlock)) {
    simulate_network(next_lide_event_time(),
            nresult);
    if (!is_empty(nresult.receive_events)) {
       save_receive_events(nresult.receive_events);
    }
    ntime = nresult.time;
    deadlock = nresult.deadlock;
    if (!deadlock) {
        simulate_lide(ntime, lide_result);
        transfer_comm_events(comm_event_list);
    }
} /* while */
```

Fig. 3. A pseudocode sketch of the block in Figure 2 that is labeled TLFS coordination module.

whatever CNS tool is being used in the given application of TLFS. This function uses application programming interfaces of the CNS tool to simulate the FA system’s communication network until a new receive event is generated or the `next_lide_event_time` is reached, whichever happens first. If (a) the return value from `next_lide_event_time` is infinite, which means that there are no pending dataflow events, and (b) there are no more pending communication events in NS-3, then `simulate_network` stops with `network_result.deadlock = true`. The simulation (`while` loop in Figure 3) stops when this deadlock condition is reached or when the predefined simulation time limit `time_limit` is reached.

B. Latency Modeling

The passage of time in the factory process flow model incorporates latencies that are associated with the actors in the model. These latencies are specified as part of the modeling process. Additionally, the physical placement of factory components (machines, controllers, etc.) with respect to the enclosing factory environment is annotated as attributes of actors that represent the components in the dataflow model. This placement information is passed to the CNS tool when the cosimulation is initialized.

Similarly, the CNS tool is responsible for simulating latencies associated with sending and receiving data across the communication network. In general, CNS tools take into account channel characteristics, network traffic conditions, and transmitter-receiver separation (distance) in determining these latencies. The interoperability of TLFS with arbitrary CNS tools allows designers to leverage the various communication latency modeling approaches of available tools.

Each dataflow actor A in a TLFS factory process flow model has an associated execution time estimation function, which is denoted by θ_A or by θ if A is understood from context. The arguments to the function include any parameters and state variables of A.

When a new firing of A becomes enabled at some simulated time t, TLFS calls the θ_A function to determine the amount of simulated time T_f that will be expended by the firing. Here, by an *enabled* firing, we mean a firing for which there is sufficient data on the input edges of the associated actor A, as defined by the consumption rate specification for the next mode of A.

Upon determining the value T_f, TLFS schedules a *firing completion* event to be processed by the simulator at time $(t+T_f)$. The firing completion event triggers the execution of a τLIDE function that carries out a single firing of A, which in turn updates the input and output edges (FIFOs) of A based on the token consumption and token production that occurs as part of the firing.

Just as τLIDE is used to manage simulated time delays associated with actor firings (execution of factory subsystems), the CNS tool is responsible for managing the simulated delays associated with sending and receiving communication packets through the wireless communication network. The TLFS coordination module (see Figure 2) is responsible for synchronizing the advancement of time across the two cooperating simulators for factory dataflow and network communication, respectively.

C. Communication Interface Actors

For dataflow modeling in TLFS, we introduce a special type of actor called a communication interface actor, as mentioned in Section IV-A. Communication interface actors are used in factory process flow models to represent functionality for sending or receiving data across wireless communication channels. These actors are used to provide modular interfaces between the dataflow subsystem within a TLFS simulation model and the TLFS coordination module (see Figure 2), which is used to provide time synchronization and information transfer between the dataflow simulator and the CNS tool. Communication across these interfaces goes through the dataflow simulator and communication event list.

In TLFS, we use different types of communication interface actors for sending and receiving data across the wireless network. These are referred to, respectively, as *send interface actors* (*SIAs*) and *receive interface actors* (*RIAs*). When an SIA fires (executes) in the dataflow simulation, it injects one or more events into the communication event list. This list is used, as described in Section IV-A, by the TLFS coordination module to transfer events between the dataflow simulator and the CNS tool.

Similarly, events associated with the reception of data in the CNS tool (network receive events) trigger the firing of RIAs in the dataflow simulation. This triggering is enabled again by the coordination module, which injects an event into the τLIDE event list corresponding to each network receive event that it detects. This process of injecting reception-related events into τLIDE is represented by the function call labeled `save_receive_events` in Figure 3. More specifically, an RIA firing completion event is scheduled in τLIDE for each packet reception. The time of the event is the receive time as determined by the CNS tool.

Note that there need not be a one-to-one correspondence between the SIAs and RIAs in an TLFS model. The routing of packets between SIAs and RIAs is achieved through information in the packets (as they are assembled in the dataflow simulation), and the communication protocols that are being used (as they are simulated in the CNS tool).

V. Factory System Modeling

In this section, we concretely demonstrate the process of modeling factory process flows in TLFS using the underlying dataflow capabilities. Once modeled within the dataflow framework of τLIDE, the factory models can be integrated systematically with arbitrary CNS tools using the TLFS framework defined in Section IV.

A. Factory Process Flow Model

Figure 4 illustrates a simple factory system that we use to demonstrate the dataflow-based modeling process in TLFS. The factory processes parts that are generated by a Parts Generator subsystem. Each part is processed by a pipeline that consists of three distinct machines. A given machine could, for example, be responsible for adding some specific feature to the part. The third machine outputs the fully-processed parts, which are subsequently collected for storage or further processing by a subsystem called the Parts Sink. In this and related examples of this paper, we are not concerned with the details of what the Parts Sink does; we consider storage within the Parts Sink to represent the last stage of processing within the FA system (pipeline) that is being investigated.

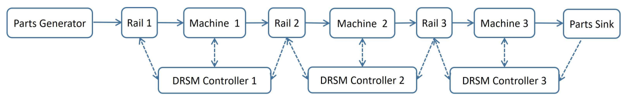

Fig. 4. A simple factory system that we use to illustrate the dataflow-based modeling process in TLFS.

Adjacent machines and rails in Figure 4 are controlled by controller subsystems, which we refer to as *machine controllers*. Each of the lower three blocks in the figure represents a *Dual-Rail, Single Machine* (*DRSM*) controller, which is a machine controller that is designed to interface with two rails and a single machine. For example, the DRSM controller for Machine 1, Rail 1 and Rail 2 sends a command to Machine 1 to output the next part onto Rail 2, and similarly, it sends a command to Rail 1 when it is time for Machine 1 to take in the next part to process.

When a machine or rail completes a command that is sent from an associated machine controller, it sends an acknowledgment message back to that controller indicating the completion of the command operation. Additionally, state changes within the machines and rails may trigger notification messages to inform the corresponding machine controllers. The communication between machines and rails and their associated machine controllers is assumed to be carried out using wireless connections, which are shown as the dashed, bidirectional edges in Figure 4.

Figure 5 illustrates a TLFS-based dataflow model that can be used to model the factory system of Figure 4. This model is an actual test case for TLFS, and can be used by TLFS to simulate the functionality and performance of the factory system together with specific communication protocols that are used to implement the wireless networking, as well as specific models of the wireless channels within the factory environment.

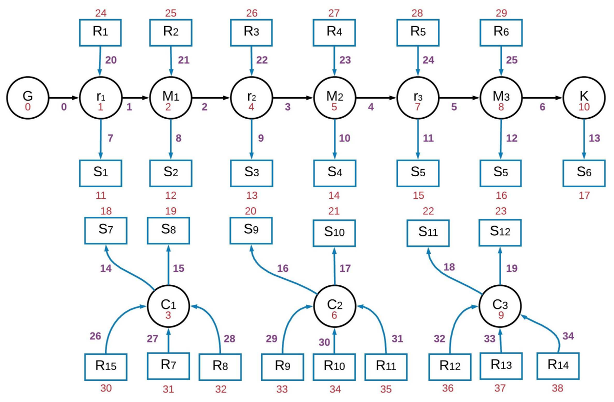

Fig. 5. A dataflow model for the factory system of Figure 4.

In the dataflow graph of Figure 5, the actors M_1, M_2, M_3 and r_1, r_2, r_3 model the machines and rails, respectively, of the factory pipeline that is depicted in Figure 4. Similarly, the actors C_1, C_2, C_3 represent the three machine controllers that provide commands to different subsets of machines and rails. The actors G and K are used to model the Parts Generator and Parts Sink.

The rail r_1 is a special rail that incorporates storage for multiple parts that are generated by the Parts Generator. If this storage fills up (i.e., if parts are being generated faster than they are processed in the pipeline), then the Parts Generator stops generating new parts until there is space in r_1 to accommodate them. This functionality ensures that parts do not keep accumulating at the output of the Parts Generator if they cannot be processed by the machines fast enough.

The numbers next to or inside the actors and next to the edges in Figure 5 are used as unique labels (indices) to help identify the different graph components.

The actors in Figure 5 that are labeled $S_1, S_2, \ldots S_{12}$ are SIAs. As described in Section IV-C, SIAs and their counterparts called RIAs are special actors that model wireless communication interfaces for transmitting and receiving data across the FA network. Actors $R_1, R_2, \ldots, R_{14}$ are RIAs. The RIAs and SIAs are actors that encapsulate the functionality for interfacing the dataflow-based FA process flow model to the event-based model that is used in the CNS tool. More details about SIAs and RIAs are discussed in Section IV-C.

B. Actor Modeling Example

Actors that model individual factory subsystems, such as machines, rails, and the Parts Generator, are specified in TLFS using CFDF semantics, which were introduced in Section II. In CFDF, a given actor can be viewed as a state machine where the current mode of the actor defines the present state, and state transitions occur between firings based on the determination of the actor's next mode as part of each firing. A graphical, finite state machine representation of a CFDF actor is therefore referred to as a *mode transition graph*. In a mode transition graph, vertices represents actor modes for a given actor, and each edge (m_a, m_b) indicates that it is possible for the actor to determine its next mode as m_b when it is executing in mode m_a.

To illustrate actor modeling in TLFS, we discuss the modeling of machine controllers in our example factory process flow model of Figure 5. Figure 6 illustrates the interface (input and output ports) for each of the machine controller actors C_1, C_2 and C_3 in Figure 5. These actors are among those that have the most complex mode transition graphs in our factory system example, and help to concretely illustrate the modeling of factory subsystems in TLFS. Actors C_1, C_2 and C_3 represent DRSM controllers.

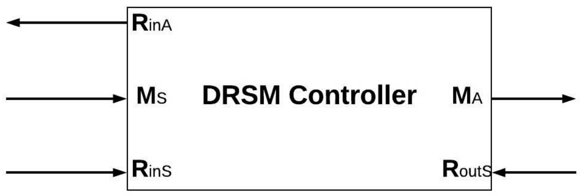

Fig. 6. Interface of a DRSM controller actor.

A DRSM controller is associated with a specific machine M in the factory system, an input rail R_{in} of the machine, and an output rail R_{out} of the machine (or to some subsystem that has a similar input interface as a rail). The DRSM controller communicates through a wireless channel to monitor updates to the state of M, R_{in}, and R_{out} that are detected by sensors on these three subsystem. The DRSM controller also sends actuation commands to M, and R_{in} through the wireless channel.

In our factory model, the DRSM C_3 interfaces to the Parts Sink actor in place of an output rail. However, the input interface of the Parts Sink is modeled in a similar fashion as that of a rail. Thus, a DRSM is used to model the controller for Machine M_3 even though the output of this machine is connected to the Parts Sink rather than to a rail.

The input ports M_S, R_{inS}, R_{outS}, shown in Figure 6 represent ports for receiving monitoring messages from the relevant sensors of M, R_{in}, and R_{out}, respectively. Similarly, the output ports M_A and R_{inA} represent ports for sending actuation messages to M and R_{in}. In the notation associated with these ports, R, M, s, and a stand respectively for rail, machine, sensing, and actuation. Note that in this factory system example, the actuation associated with R_{out} (the transfer of parts from the rail to the next factory pipeline unit) is assumed to be performed by a different controller that is also associated with R_{out}.

The approach described here for DRSM controllers is just one possible way of modeling interfaces in TLFS between machine controllers and their associated rails and machines; many other interfacing models are possible given the flexibility of the underlying dataflow model of computation.

C. Mode Transition Graph

In Section V-B, we discussed the mode transition graph as a useful representation of actor functionality. Figure 7 illustrates the mode transition graph for a DRSM controller actor.

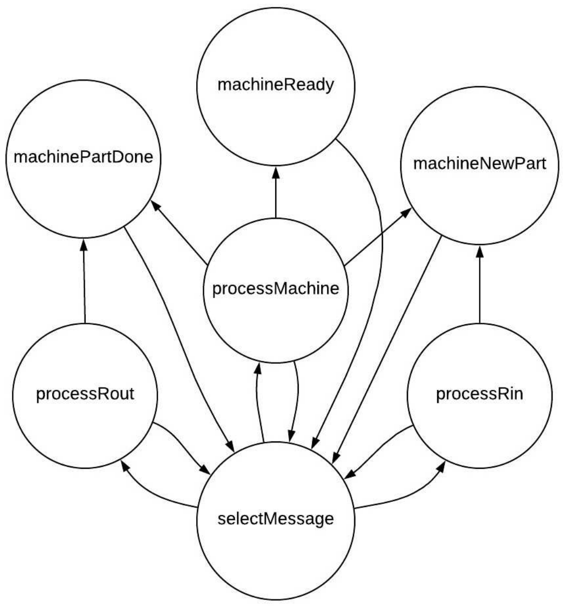

Fig. 7. Mode transition graph for a DRSM controller actor.

As a DRSM controller actor D operates, it keeps track of (as part of its internal state) the status of the associated factory subsystems M, R_{in} and R_{out} through messages it receives through the M_S, R_{inS}, and R_{outS} ports of D, respectively. Note that dataflow actors for embedded applications are typically allowed to maintain state. Such state can be represented formally within a dataflow model as a self loop edge — that is, an edge whose source and sink actors are identical. To avoid clutter in diagrams of dataflow graphs, we often omit the self-loop edges.

The status information that a DRSM keeps track of includes the state of the machine M and the number of parts that reside at any given time on the rails R_{in} and R_{out}. This control information is used to help ensure the constraints that at any given time there should be at most one part in the machine, and that the number of parts on a rail at any given time should not exceed its capacity. The capacity of a rail (the maximum number of parts that can reside on it at a given time) is an actor parameter in the dataflow model. In our experiments, we use a capacity of 10 for rail r_1 in Figure 5, which connects to the output of the parts generator, and we use a capacity of 1 for each of the other rails.

The DRSM operates through sequences of *message processing cycles*, where each cycle involves a trajectory through multiple actor modes. Each cycle begins in the selectMessage mode, where the next input message to process is selected. This selection is made based on the unprocessed messages that are buffered at the actor inputs, and some priority scheme for selecting messages if messages are available on multiple inputs. If no messages are available at the inputs, then the actor waits in the selectMessage mode until a message becomes available. The TLFS framework can be used to prototype and experiment with different priority schemes. In our experiments, we assumed the simple priority scheme in which messages are selected in decreasing priority order from the inputs R_{outS}, R_{inS}, and M_S, respectively.

The processRout, processMachine, and processRin modes shown in Figure 7 are used to perform initial processing of messages received from actor inputs R_{inS}, R_{outS}, and M_S, respectively. These actors use the newly received sensor data and the current state information saved within the DRSM to determine whether a new command message needs to be sent to R_{in}, R_{out} or M. If a new command is not required to be sent at this time, the DRSM completes the current message processing cycle by returning to the selectMessage mode, where the next cycle will subsequently begin.

Otherwise — if the DRSM determines in the processRout, processMachine, or processRin mode that a command message needs to be sent — then it transitions to Mode machineReady, machineNewPart, or machinePartDone depending on the type of message that needs to be sent. Mode machineReady represents the situation where a part has entered the machine, and the machine is waiting for a command to start performing its processing on the part. Similarly, Mode machineNewPart is entered when the machine is ready to receive a new part for processing from its input rail, and a part is available to be ingested from the rail. Mode machinePartDone represents the situation where the machine has finished its processing of a part that is still inside the machine, and there is a vacancy on the output rail so that the part can now be transferred to the rail.

All of the modes in Figure 7 are based on standard CFDF semantics (see Section II), except for the selectMessage mode, which is called a *transition-only mode*. This type of mode can be used to model nondeterminate behaviors in LIDE. In our case, nondeterminism arises because of the unpredictable order in which messages in general arrive at wireless communications interfaces of network nodes. Support for transition-only modes is a new feature that we have added to LIDE as part of developing TLFS.

In its transition-only mode, the machine controller actor does not consume any input tokens. Instead, it checks the populations of the input edges, and determines the next mode as a function of these populations. The machine controller actors in our factory process flow model are the only actors that have transition-only modes, and each of these actors has exactly one such mode. Thus, the overall factory system model is technically based mostly on CFDF — all actors are CFDF, except the three machine controllers, which are equipped with transition-only modes, as described above.

The use of transition-only modes allows factory system models in TLFS to incorporate non-determinism at the communication network level together with deterministic, CFDF-based modeling of signal and information processing that is

TABLE I
DATAFLOW TABLE FOR A DRSM CONTROLLER ACTOR.

Mode	R_{inS}	M_S	R_{outS}	M_A	R_{inA}
selectMessage	0	0	0	0	0
processRin	-1	0	0	0	0
processMachine	0	-1	0	0	0
processRout	0	0	-1	0	0
machineReady	0	0	0	0	1
machinePartDone	0	0	0	1	0
machineNewPart	0	0	0	1	0

embedded within individual factory subsystems. This multi-layer modeling is achieved all within a unified dataflow framework along with model-based interfacing to CNS tools. Experimentation with complex signal and information processing functionality in TLFS — for example, by integrating machine learning subsystems into factory workflow models — is a useful direction for future work.

D. Dataflow Table

In addition to the mode transition graph, another data structure that is useful for representing the behavior of CFDF actors is the *dataflow table* [4]. The dataflow table for a CFDF actor A is a matrix D whose rows are in one-to-one correspondence with the mode set $\mu(A)$ and whose columns are in one-to-one correspondence with the ports of A. for each port p of A and each mode $m \in \mu(A)$, the matrix element $D[m, p]$ gives the net change in the token population on the edge connected to p that results from firing A in Mode m. If p is connected to a self-loop edge or if m does not produce or consume data on p during m, then $D[m, p] = 0$. Otherwise, $D[m, p]$ gives the number of tokens produced on p during m if p is an output port of A, and $D[m, p]$ gives the negative of the number of tokens consumed from p during m if p is an input port of A.

Table I shows the dataflow table for a DRSM controller actor. By definition, all of the entries for the selectMessage mode are zero in the table because this is a transition-only mode. As shown in the table, all of the other modes perform dataflow (consume or produce tokens) on at least one actor port. The dataflow table is useful, for example, as a data structure for model-based testing. Using the dataflow table, one can generate tests for validating that the implementation of an actor is consistent with the interface behavior defined by the actor model.

E. Summary

In this section, we have presented concretely, through an example, the modeling of a factory process workflow in TLFS. We have focused in the example on modeling the interfacing between different subsystems in a factory model. Of course, this is just one example of a factory model in TLFS, and many other kinds of process flow models can be developed and experimented with using the tool.

While models of the type demonstrated in this section are relatively abstract, they can be useful for fast design space exploration across large design spaces involving communication protocols, factory topologies, and distributed control policies. Such early stage design space exploration can be used to inform later stage simulation experiments that investigate specific combinations of protocols, topologies, and policies in more depth. Experimenting with TLFS for such detailed simulation experiments, which incorporate lower levels of process flow design abstraction, is a useful direction for future work.

VI. EXPERIMENTS

In this section, we present experiments to demonstrate the utility of TLFS. First, we use TLFS to study the effect of different wireless communication protocols on communication system performance for a given factory model. Then we demonstrate how communication performance changes as the complexity of the factory system increases, and also as the distances between network nodes in the factory increase. These experiments are conducted with NS-3 as the CNS tool.

The experiments presented in this section are representative of the kinds of studies that can be performed with TLFS. They are by no means intended to be comprehensive. Indeed, the flexible architecture of TLFS, which facilitates integration with arbitrary CNS tools, enables a wide variety of different kinds of trade-off studies, and investigations into the influence of parameters and other design decisions on factory system performance. Development of more extensive case studies in simulation and design using TLFS is an interesting direction for future work.

A. Simulation Parameters

Each factory model simulated in our experiments consists of one or more pipelines of the form illustrated in Figure 4 and Figure 5. Each factory model has two size-related parameters — the number of pipelines N_p, and the number of machines per pipeline N_m. Thus, the example shown in Figure 4 and Figure 5 corresponds to $N_p = 1$ and $N_m = 3$.

Table II summarizes the other key simulation parameters used in our experiments. A given data point in the experiments is derived by executing a simulation with the same settings N_s times, and averaging the results over the N_s executions. Each such simulation involves N_j generated parts for each Parts Generator actor in the factory dataflow graph. The simulation completes when all of the generated parts are fully processed in their respective pipelines. Since there is one Parts Generator actor per pipeline, this means that each simulation involves processing a total of $(N_p \times N_j)$ parts.

The values of t_m and t_r give, respectively, the estimated execution time values used in the simulation models for a machine to process a part, and for a rail r to move a part from one end of r to the other end. Similarly, t_i is the estimated time required to generate a new part after the previous part has been generated. The values of t_m, t_r, and t_i are used in the execution time estimation functions (θs) for the relevant actors (see Section IV-B).

TABLE II
SIMULATION PARAMETERS.

Parts Generated Per Pipeline N_j	100
Number of Simulation Iterations N_s	10
Machine Processing Time t_m	10 sec
Rail Transfer Time t_r	4 sec
Part Generation Interval t_i	10 sec
Channel Frequency	2.4 GHz
Large Scale Path Loss Model	Log-distance
Decay Exponent α	3
Distance Reference d_0	1 m
Loss at Reference L_0	46.6777 dB

The parameters α, d_0, and L_0 in Table II are related to the simulation of propagation path loss. In our simulations, we apply features in NS-3 for using the log-distance path loss model to estimate signal loss in communication channels. The log-distance model is often used to estimate path loss within buildings. In this model, the power loss at the receiver side when transmitting over a distance d is calculated by

$$L = L_0 + 10\alpha log_{10}(\frac{d}{d_0}) + Z, \quad (1)$$

where L_0 is the path loss at the reference distance, d_0 is the reference distance, α is the decay exponent, and Z is the log-normal shadowing.

B. Experiments with Different Protocols

We first use TLFS to study the average communication delay T_c for a fixed factory size, and the variation in T_c for different communication protocols — in particular, for different variants of IEEE 802.11. Using TLFS, we measure T_c as the average time difference between the time when a communication packet P is successfully received (through an RIA), and the time when P was transmitted (through an SIA). This average is taken over all packet communications within a given simulation.

In this experiment, we use a factory model with a single pipeline that contains 3 machines — that is, $N_p = 1, N_m = 3$.

Figure 8 shows a box plot representation of how T_c was found to vary across four different protocols — IEEE 802.11xx, for $\text{xx} \in \{\text{ac}, \text{b}, \text{g}, \text{n}\}$. Significant performance variation is shown between the best-performing protocol in this context (IEEE 802.11g) and the worst-performing one (IEEE 802.11b). One reason for the relatively low performance of IEEE 802.11b may be its low maximum data rate. Compared with other protocols, which can achieve 54 Mbps speed, the maximum speed of IEEE 802.11b is only 11 Mbps.

Overall, the results in Figure 8 show a clear advantage of IEEE 802.11g in terms of T_c for the factory model studied in this experiment.

C. Scalability Experiments

Next, we study how communication performance changes as we increase the factory size (N_p, N_m) and the distance between factory subsystems. Here we study five different factory dataflow graphs, denoted (a) through (e),

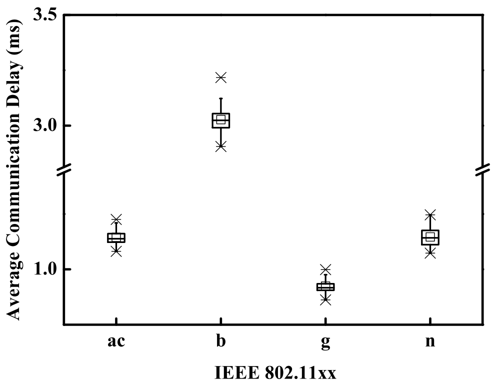

Fig. 8. A box plot representation of the variation in average communication delay across different communication protocols.

with sizes that are defined, respectively, as $(N_p, N_m) = (1,3), (1,4), (1,5), (4,4), (5,5)$. The total number of network nodes in the CNS tool modeling subsystem are, respectively, 11, 14, 17, 56, and 85. Each rail, machine, and machine controller corresponds to a distinct network node. Additionally, each Parts Generator and Parts Sink is modeled as a separate network node.

We define a *distance parameter* d associated with the simulations in this experiment. The units of this parameter are meters. For each factory pipeline, the spacing between adjacent network nodes is set to d. Additionally, for factory models that consist of multiple pipelines, the successive pipelines are spaced apart by distance d. Note that for the experiments reported in Section VI-B, we used a constant distance value of $d = 10$.

Figure 9 and Figure 10 show the variation in average communication delay and packet retransmission rate, respectively, for the five different factory sizes defined above, and for different settings of d for each factory size. In each of these two figures, each of the five plots corresponds to a distinct (N_p, N_m) pair, and each curve within a given plot corresponds to a distinct value of $d \in \{5, 10, 15, 20, 25\}$. The data is plotted for each of the four IEEE 802.11 variants discussed in Section VI-B. In Figure 10, the vertical axis represents the fraction of packet transmissions that have to be repeated due to errors in the original transmission. For example, a value of 0.5 means that 50% of the packets have to be retransmitted. In addition to increasing communication delays, packet retransmissions result in energy consumption overhead due to the increased operational load placed on the communication transceivers in the system.

The results in Figure 9 and Figure 10 show the general trends that one would expect of increasing communication delay and packet retransmission rate with increases in the distance parameter value d, and increases in the factory size. The results also provide insight into how different IEEE 802.11 protocol variants perform for the different factory size/distance combinations that are evaluated. The simulations carried out

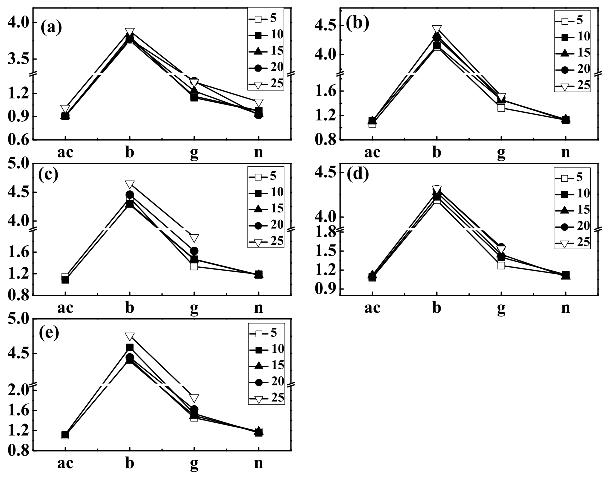
Fig. 9. Variation in average communication delay for different factory sizes, distance parameter settings, and IEEE 802.11 variants.

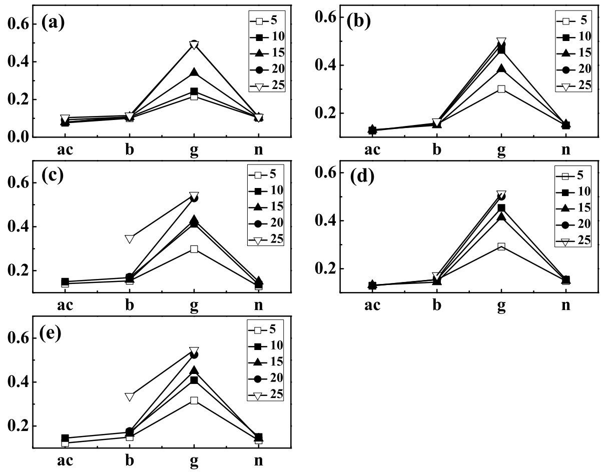
Fig. 10. Variation in packet retransmission rate for different factory sizes, distance parameter settings, and IEEE 802.11 variants.

using TLFS provide a quantitative assessment of all of these trends, and the associated factory system design trade-offs. The results also help to validate the capabilities of TLFS and demonstrate these capabilities with further concreteness.

VII. Conclusions

In this paper, we have introduced a simulation framework called TLFS (Tau Lide Factory Sim) to address the challenge of simulating complex factory automation systems that are integrated with wireless communication. TLFS incorporates novel techniques, based on dataflow concepts, for representing factory process flows, and for systematically integrating dataflow-based process flow simulations with discrete event simulations of communication network functionality. Useful directions for future work include automatically generating the corresponding communication network models from dataflow graph representations of networked factory process flows, and experimenting with more detailed and specialized factory process flows using TLFS.

Disclaimer

Certain commercial equipment, instruments, materials, software or systems are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

References

[1] A. A. K. S., K. Ovsthus, and L. M. Kristensen, "An industrial perspective on wireless sensor networks — a survey of requirements, protocols, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1391–1412, 2014.

[2] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*, 2011, http://LeeSeshia.org, ISBN 978-0-557-70857-4.

[3] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya, "A lightweight dataflow approach for design and implementation of SDR systems," in *Proceedings of the Wireless Innovation Conference and Product Exposition*, Washington DC, USA, November 2010, pp. 640–645.

[4] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya, "The DSPCAD framework for modeling and synthesis of signal processing systems," in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds. Springer, 2017, pp. 1–35.

[5] *ns–3 Tutorial, Release ns–3.25*, ns–3 Project, 2016.

[6] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, pp. 773–799, May 1995.

[7] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, 2nd ed. Springer, 2013.

[8] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya, "Heterogeneous design in functional DIF," in *Transactions on High-Performance Embedded Architectures and Compilers IV*, ser. Lecture Notes in Computer Science, P. Stenström, Ed. Springer Berlin / Heidelberg, 2011, vol. 6760, pp. 391–408. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24568-8_20

[9] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, 1974.

[10] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, ser. Lecture Notes in Computer Science, B. Robinet, Ed. Springer Berlin Heidelberg, 1974, vol. 19, pp. 362–376.

[11] H. Neema *et al.*, "Model-based integration platform for FMI co-simulation and heterogeneous simulations of cyber-physical systems," in *Proceedings of the International Modelica Conference*, 2014, pp. 235–245.

[12] M. Düngen, F. Hofmann, and H. Schulze, "Bit error rate simulation studies for PSSS with multi-user detection for industrial multipath-fading environments," in *IEEE International Workshop on Factory Communication Systems*, 2017, pp. 1–6.

[13] Y. Liu, R. Candell, K. Lee, and N. Moayeri, "A simulation framework for industrial wireless networks and process control systems," in *IEEE World Conference on Factory Communication Systems*, 2016, pp. 1–11.

[14] F. Bause, P. Buchholz, J. Kriege, and S. Vastag, "A simulation environment for hierarchical process chains based on OMNeT++," *Simulation*, vol. 86, no. 5–6, pp. 291–309, 2010.

[15] S. Won, C. Shen, and S. S. Bhattacharyya, "NT-SIM: A co-simulator for networked signal processing applications," in *Proceedings of the European Signal Processing Conference*, Bucharest, Romania, August 2012, pp. 1094–1098.

[16] A. Khan, S. M. Bilal, and M. Othman, "A performance comparison of open source network simulators for wireless networks," in *Proceedings of the IEEE International Conference on Control System, Computing and Engineering*, 2012, pp. 34–38.

[17] M. Korkalainen, M. Sallinen, N. Kärkkäinen, and P. Tukeva, "Survey of wireless sensor networks simulation tools for demanding applications," in *Proceedings of the International Conference on Networking and Services*, 2009, pp. 102–106.