Formalizing Functional Flow Block Diagrams Using Process Algebra and Metamodels

Allan I. McInnes, *Member, IEEE*, Brandon K. Eames, *Member, IEEE*, and Russell Grover

***Abstract*—Functional flow block diagrams (FFBDs) are a traditional tool of systems engineering and remain popular in some systems engineering domains. However, their lack of formal definition makes FFBDs imprecise and impossible to rigorously analyze. The inability to analyze FFBDs may allow specification errors to remain undetected until well into the system design process or, worse, until the system is operational. To help address these problems, we have developed a precise formal syntax and semantics for FFBDs, based on the application of metamodels and the process algebra Communicating Sequential Processes (CSP). FFBDs constructed within our formalized framework are precisely defined and amenable to analyses of properties, such as safety, progress, and conformance to required scenarios. We demonstrate some of the analyses made possible by our formalization in a simple case study of system specification and show how our formalization can be used to detect and correct subtle system errors during the specification phase.**

***Index Terms*—Modeling, process algebra, systems engineering, visual languages.**

I. Introduction

FUNCTIONAL flow block diagrams (FFBDs) or functional flow diagrams are a popular notation used by systems engineers to visually represent control flow and, in some cases, data flow [1]–[5]. They have been used by a wide range of organizations since their introduction in the 1950s and make an appearance in most systems engineering textbooks. Although other visual representations of system behavior have since been developed, FFBDs continue to be a popular systems engineering tool, particularly among U.S. Federal Government agencies [5]–[8]. Indeed, their long and close association with the practice of systems engineering resulted in the inclusion of an FFBD variant in the recently released Object Management Group SysML v1.1 standard [9].

Despite its longevity and popularity, the FFBD notation is not without problems. Different authors use different conventions, which are rarely explicitly presented, making the behavior defined by an FFBD potentially ambiguous. Furthermore, since the meaning or semantics of FFBDs are typically only informally defined, the interpretation of a given FFBD is imprecise at best. As a result, rigorous analysis of FFBDs is difficult.

Tools for creating executable specifications of system behavior based on FFBDs, such as RDD-100 [10] and CORE [11], provide a significant improvement over manually drawn FFBDs. The use of a tool to support design specification ensures that all diagrams developed for the design share a common syntax and semantics. The common semantics implies that, when a collection of models is composed or executed, the same translation and simulation algorithms will be applied to all of the diagrams, providing a consistent set of results.

Unfortunately, the internal consistency provided by a given tool does not imply an unambiguous interpretation of system behavior, particularly when considering cross-tool compatibility. Harel and Rumpe [12] asserted that, without a well-defined mapping to a semantic domain, the syntax of a language is worthless, due to the possibility of ambiguous interpretation. However, all too commonly, the semantics of a graphical language is defined informally, embodied in a tool's translators, simulators, and interpreters. The meaning of the diagrams is typically conveyed to users through informal documentation. The lack of formality leads to subtle inconsistencies between competing tools or, worse, between different diagrams represented in the same tool. For example, von der Beek [13] documented the issue of a splintering of semantics for the Statecharts visual language. In an effort to alleviate some of the confusion surrounding the semantics of Statecharts, Harel and Namaad [14] published an unambiguous execution semantics for Statecharts.

Just as the definition of a clear execution semantics for Statecharts has helped to make the language more precise and consistent, a formalization of FFBD semantics, based on a well-defined abstract syntax, can address difficulties with ambiguity, analysis, and semantic comparisons of FFBDs by providing a precise definition of the structure and meaning of diagrams expressed in FFBD notation. The contribution to the formalization of FFBDs offered in this paper is twofold. First, we propose a *formal syntax* for FFBDs, including an abstract syntax and well-formedness rules, which are defined using the Generic Modeling Environment (GME) [15], [16] metamodeling language. The metamodel provides an unambiguous definition of the elements from which an FFBD may be constructed and the way in which those elements can be combined. It also forms the basis of a graphical specification tool for constructing FFBDs and automatically translating those FFBDs into one or more semantic models. Second, we propose a *formal semantics* for FFBDs, which is defined in terms of the process algebra CSP.

Manuscript received October 24, 2007; revised September 6, 2008 and September 11, 2009; accepted December 6, 2009. Date of publication June 14, 2010; date of current version November 10, 2010. The work of A. I. McInnes was supported in part by the Space Dynamics Laboratory (SDL) through an SDL "Tomorrow" Ph.D. Fellowship. This paper was recommended by Associate Editor R. Qiu.

A. I. McInnes was with the Department of Electrical and Computer Engineering, Utah State University, Logan, UT 84322 USA. He is now with the Department of Electrical and Computer Engineering, University of Canterbury, Christchurch 8140, New Zealand (e-mail: allan.mcinnes@acm.org).

B. K. Eames and R. Grover are with the Department of Electrical and Computer Engineering, Utah State University, Logan, UT 84322 USA (e-mail: beames@engineering.usu.edu).

Digital Object Identifier 10.1109/TSMCA.2010.2048749

To the best of our knowledge, this is the first formal semantics that covers both the structure and behavior of FFBDs. It builds upon existing research into process algebraic theories of concurrency to provide a precise mathematical description of one interpretation of FFBD semantics. The CSP-based semantics that we propose provides a target semantic model for testing the graphical specification tool and a baseline against which to compare other interpretations of FFBD semantics. FFBDs expressed in our semantic model are precise, unambiguous, and amenable to exhaustive automated analysis through model checking.

This paper begins by briefly reviewing the informal syntax and semantics of FFBDs, the language definition techniques provided by GME, and the basic theory of CSP. We then introduce our FFBD syntax model, which incorporates both an FFBD control-flow view and a simple view of data flow between the functions defined by the FFBD. In Section IV, we describe our interpretation of FFBDs in CSP and discuss the rationale for our chosen mapping from FFBDs to CSP. In Section V, we present a small case study involving the classic "traffic light" problem, which we use to illustrate both the use of our GME-based FFBD tool and the kinds of analysis that are enabled by our mapping from FFBDs to a formal process algebraic language. Finally, in Sections VI and VII, we discuss related work and present our conclusions.

II. Background

In this section, we present a brief overview of FFBD notation, GME, and CSP.

A. FFBDs

FFBDs provide an intuitive graphical method of representing the behavior of complex hierarchical systems. Various descriptions have been given of FFBDs (e.g., [1], [2], and [17]); lacking a definitive specification of FFBD syntax, we provide a brief review of commonly used constructs.

An FFBD is composed of labeled function blocks, such as those shown in Fig. 1, which may represent leaf-level functions or nested FFBDs. Function blocks are connected by unidirectional arrows that define the order of function execution. FFBDs are conventionally drawn to flow left to right and top to bottom. Most authors use some variation on this FFBD notation.

The notation in Fig. 1 provides several different control-flow structures from which to construct a behavior specification. The *sequence* structure simply captures the order in which two functions execute. The *choice* structure allows the definition of a choice between two different execution paths. Which path is selected for execution depends on some logical condition, which is not shown in the diagram. The "Go" path from the choice block is executed if the condition is met; otherwise, the "NoGo" path is executed. The *parallel* construct represents two or more functions that can execute concurrently. The parallel paths are enclosed in a pair of AND nodes, marking the beginning and ending of parallel execution. When the system may execute one of several possible paths, the *alternative* construct is used. Each possible execution path is connected to a pair of

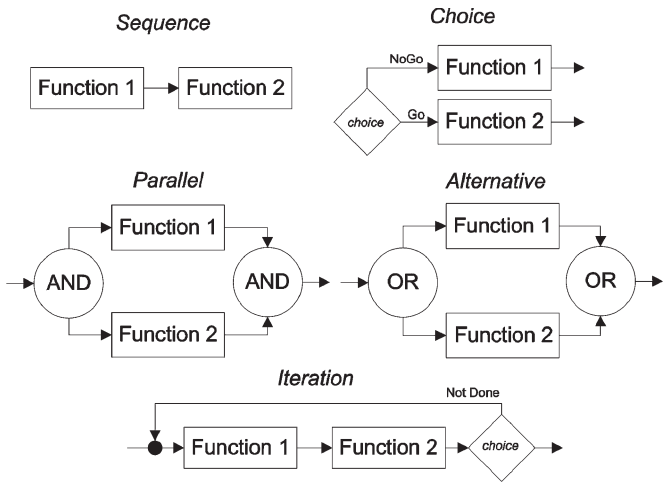

Fig. 1. FFBD notation.

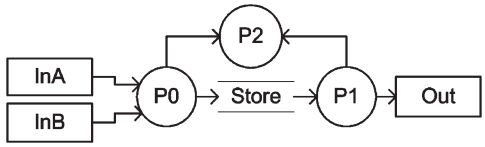

Fig. 2. Data flow diagram.

OR nodes, marking the beginning and ending of the alternate path selection. The construct does not show the path selection criteria but simply indicates that one of the represented paths will be executed. The *iteration* structure models repeating behavior. It consists of a *choice* function that repeats a functional path until a condition is met, after which execution follows the path leading out of the iteration.

The basic FFBD control structures can be combined and nested to produce a complete FFBD description of the system control flow. In addition, FFBDs can be layered in a hierarchy of diagrams that provide progressively more detailed descriptions of individual system behaviors as the reader moves down the hierarchy, making the system and each of the functions easier to understand. Examples of hierarchical FFBD diagrams are presented in Section V with the case study, e.g., Figs. 13 and 14.

FFBDs are fundamentally a construct for representing control flow: they describe what functions occur, in what order those functions can occur, and what alternative paths of execution are available. By themselves, FFBDs do not capture the flow of data between operations in the system. A complete description of system behavior requires information about data flow and control flow. Some authors directly annotate FFBDs with arrows indicating the flow of data between function blocks, forming enhanced FFBDs (EFFBDs) or behavior diagrams, whereas others maintain information about data flow in separate data-flow diagrams (DFDs) [3], [17]. DFDs depict the data flow in a system as a directed graph of the connections between data-transforming processes and data storage elements. Fig. 2 is a simple example of a DFD, in which InA, InB, and Out are external interfaces; P0, P1, and P2 are processes; *Store* is a store that holds persistent data; and the arrows show the flow of data.

B. GME

The GME [15], [16] is a metaprogrammable toolkit designed to support the use and development of domain-specific visual modeling languages. A domain-specific modeling language is a graphical notation that leverages visual constructs from a particular engineering discipline. The syntax rules of a domain-specific language, including the set of available parts and part types, the permissible relations between part types, and a set of well-formedness rules captured as constraints, are encoded in a configuration file called a *paradigm*, which can be loaded into the GME. The GME will then support the syntax of the specified language, enforcing the rules of that language and informing the modeler on the occurrence of violations.

GME supports the creation of paradigms through the definition of a *metamodel*, which is itself a graphical model capturing the rules for creating diagrams specific to a particular domain. The metamodel specifies the types of parts available for composition, how those parts can be interconnected, and what attributes can be associated with the parts. Metamodels are created using the MetaGME paradigm, which provides a variant of Unified Modeling Language (UML) class diagrams as a means for modeling a modeling language and supports the specification of well-formedness rules as constraints using the Object Constraint Language [18]. A translator tool included with GME converts the metamodel into a paradigm, which can then be loaded into GME to support the creation of domain-specific models.

While GME allows users to develop modeling languages and domain-specific models, these models serve little purpose beyond documentation without the ability to perform translations on the user-specified diagrams. A model *interpreter* is a program that extends GME with the ability to perform domain-specific translations or analyses on a model. Depending on the domain, an interpreter can be created to generate code implementations of a modeled system, translate a specification into a data format acceptable by an analysis tool, or perform introspective analysis on and modifications to the model itself. GME supports a variety of programming language bindings for interpreters, including Java, Python, and C++. The software development kit for GME offers a high-level application programming interface (API) for traversing and manipulating models, hiding the low-level details of managing a graphical user interface and the model database.

C. Communicating Sequential Processes (CSP)

CSP is a mathematical theory of concurrency and communication. In contrast to other well-known formalisms, such as Z or B, CSP is specifically designed for modeling concurrent systems. In CSP, the behavior of a system is represented as a process that defines the observable sequences of interactions between the system and its environment. Commercial tools, such as FDR2 [19], allow process expressions to be automatically checked for properties, such as deadlock freedom, and compliance with system specifications also expressed in CSP. Roscoe's *The Theory and Practice of Concurrency* [20] is a comprehensive introduction to modern CSP. For the purposes of this paper, we need only be concerned with a subset of the CSP operators, which we briefly review.

1) Process Expressions: Process expressions are built from *events*, which are abstractions of real-world interactions between systems. The *alphabet* of a process $\mathtt{P}$ is the set of all events in which $\mathtt{P}$ can engage. Events may be atomic names or compound objects formed by combining names with a dot, such as $\mathtt{out.0}$. If $\mathtt{c}$ is declared as a channel that communicates objects of type $\mathtt{T}$, $\mathtt{channel\ c : T}$, then the set $\{|\mathtt{c}|\}$ is a set of compound events $\{\mathtt{c.t}|\mathtt{t} \in \mathtt{T}\}$.

The simplest CSP *process* is $\mathtt{STOP}$, which never engages in any event. Successful termination is represented by the $\mathtt{SKIP}$ process. More complex processes are defined by combining processes and events using operators that express sequences of events, alternative execution paths, sequential and parallel execution, and communication between processes.

Simple sequences of events are defined using the event prefix operator $\rightarrow$. For example, the recursive process $\mathtt{P} = \mathtt{a} \rightarrow \mathtt{b} \rightarrow \mathtt{P}$ engages in the event $\mathtt{a}$, then $\mathtt{b}$, and then repeats. Process definitions may be parameterized by events, channels, data values, or function definitions, allowing reuse of the behavior definition in different contexts. Thus, for example, the parameterized process expression $\mathtt{P(x)} = \mathtt{x} \rightarrow \mathtt{P}$ may be used to create the process $\mathtt{Q} = \mathtt{P(a)}$, which will generate an infinite sequence of $\mathtt{a}$ events.

Alternative execution paths can be defined using several different operators. The choice process $\mathtt{P}[]\mathtt{Q}$ may behave either as $\mathtt{P}$ or as $\mathtt{Q}$, whereas the more general indexed choice $[]\mathtt{i} : \mathtt{I} \ @\ \mathtt{P(i)}$ may behave as any one of the processes $\mathtt{P(i)}$, for $\mathtt{i} \in \mathtt{I}$. The nondeterministic choice $\mathtt{P}|\sim|\mathtt{Q}$ offers at least one of the two processes $\mathtt{P}$ and $\mathtt{Q}$ as a possible behavior. For each of the choice operators, only those execution paths that begin with an event in which the environment is also prepared to engage can proceed.

Processes may execute sequentially or in parallel. The *sequential* composition $\mathtt{P;Q}$ behaves as $\mathtt{P}$ until $\mathtt{P}$ terminates in a $\mathtt{SKIP}$, after which the composite process behaves as $\mathtt{Q}$. The *interleaved parallel* composition $\mathtt{P}|||\mathtt{Q}$ independently executes the processes $\mathtt{P}$ and $\mathtt{Q}$ at the same time. The *interface parallel* composition $\mathtt{P}[|\mathtt{X}|]\mathtt{Q}$ executes $\mathtt{P}$ and $\mathtt{Q}$ in parallel but requires the simultaneous participation of both processes to engage in any event in the interface set $\mathtt{X}$. Similarly, the *alphabetized parallel* composition $\mathtt{P}[\mathtt{A}\|\mathtt{B}]\mathtt{Q}$ requires $\mathtt{P}$ and $\mathtt{Q}$ to synchronize on any event in the set $\mathtt{A} \cap \mathtt{B}$, whereas the more general *indexed parallel* composition $\|\mathtt{i} : \mathtt{I} \ @\ [\mathtt{A(i)}]\mathtt{P(i)}$ constrains each $\mathtt{P(i)}$ to only perform events in the corresponding interface set $\mathtt{A(i)}$ and requires all processes that have events in common to synchronize on those events.

Communication can take place between parallel processes that synchronize on a common channel. The parallel composition $\mathtt{P}[|\{|\mathtt{c}|\}|]\mathtt{Q}$ requires $\mathtt{P}$ and $\mathtt{Q}$ to synchronize on all events on the channel $\mathtt{c}$, allowing values to flow between the two processes. Communication events are expressed by the notations $\mathtt{c?x}$ (input) and $\mathtt{c!x}$ (output). Thus, the process $\mathtt{P} = \mathtt{c?x} \rightarrow \mathtt{c!x} \rightarrow \mathtt{P}$ first receives an input message from $\mathtt{Q}$, storing the received value into the variable $\mathtt{x}$. The output event $\mathtt{c!x}$ then transmits the previously received value back to $\mathtt{Q}$, after which $\mathtt{P}$ is again ready to receive a new input.

2) Process Refinement and Equivalence: CSP theory allows different process expressions to be compared by examining the observed behavior of the processes within the context of a *denotational model.* The denotational model defines the kind of information included in an observation and therefore also defines the kind of distinctions that can be drawn between processes. The most commonly used denotational models are given as follows:

1) *traces* model, in which the observed behavior is a set of traces that record all possible sequences of interaction between the process and its environment;
2) *stable failures* model, which, in addition to recording traces, also records the events that a process can refuse to perform at each step of its execution;
3) *failures/divergences* model, which extends the stable failures model by distinguishing between processes that have stopped completely and those that have simply stopped interacting with the environment.

Analysis can be carried out by examining refinement relationships between processes within the context of a given denotational model. The *refinement* relation, which is denoted by $[=$, indicates that the behavior of one process is somehow "contained" within the behavior of another process. For example, refinement in the traces model is defined as [20]

$$(\texttt{P1}\ [\texttt{T} = \texttt{P2}) \Leftrightarrow (\texttt{traces(P2)} \subseteq \texttt{traces(P1)})\,.$$

That is, process P2 is a trace refinement of process P1 if and only if the traces of P2 are a subset of the traces of P1. If $\texttt{P}[\texttt{T} = \texttt{Q}$ and $\texttt{Q}[\texttt{T} = \texttt{P}$, then $\texttt{traces(P)} = \texttt{traces(Q)}$, and processes P and Q are said to be *trace equivalent*. Similar refinement and equivalence relations exist for the stable failures and failures/divergence models. Checking of refinement relations can be used to establish whether a process possesses certain behavioral properties or to determine whether a process model of a system implementation refines a model of the system specification. The FDR2 refinement checker is a commercially available tool for automatically checking such refinement relations.

III. Metamodeling the FFBD Syntax

In this section, we use a *metamodel* to define a formal syntax for a visual modeling language that integrates FFBDs and DFDs. Using the GME, a modeling tool has been developed that offers the ability to quickly compose mixed FFBD/data-flow models of a system, coupled with a model interpreter that translates the models into CSP for evaluation and analysis. The metamodel defines an abstract syntax for representing FFBDs. GME supports metamodeling using a variant of UML class diagrams and also facilitates the specification of a concrete syntax for the language through the association of visualizations, such as icons and arrows, with entities in the abstract syntax. Since the metamodel only captures the abstract syntax, concrete syntax, and well-formedness rules for the modeling language, it does not tie the modeling language to a specific semantic domain. In particular, the metamodel itself is not specific to CSP but rather it offers a framework upon which a mapping to a semantic domain can be developed. Jointly with the metamodel, we have developed such a mapping to a CSP-based semantic specification of FFBDs. Due to the separation between the semantic mapping realized through model interpreters and the abstract syntax captured by the metamodel, the approach adopted is amenable to the creation of alternate backend evaluation tools based on other formalisms, e.g., Promela [21] or Petri nets [22].

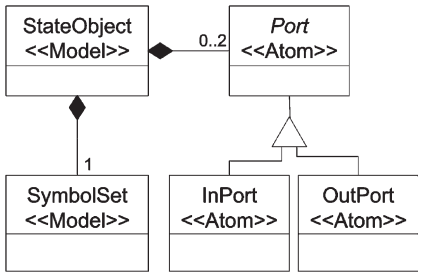

Fig. 3. Metamodel of *StateObjects*.

The metamodel offers the ability to create two separate views of a system: a control-flow view using FFBDs and a data-flow view using DFDs. The choice of separate views was based on the goals of scalable model development, separation of concerns, and flexibility. An alternative approach is to use a metamodel based on EFFBDs, which merge data-flow and control-flow views in a single diagram. However, a merged view can lead to an overwhelming number of modeling objects at each level of the hierarchy.

The discussion of the metamodel focuses on the abstract syntax of the modeling language, deferring most of the examples of concrete syntax to the "traffic light controller" case study presented in Section V.

A. Symbols and StateObjects

The foundation of the visual language is an abstraction called *Symbol*. A *Symbol* can be used to model an event, a piece of data, a control flag, a signal, or any other type of information significant to the system behavior. Each *Symbol* is assigned a name. Related *Symbols* are collected into a set, which is referred to as a *SymbolSet*. *Symbols* form the basis of interaction between the constructs of the visual language.

System behavior is often state dependent. System state may be recorded by one component of the system and used by a different component. A *StateObject*, which is modeled in Fig. 3, represents a variable that can be assigned a value from a fixed set, as defined by the contained *SymbolSet*. For example, in the design of a traffic light controller, a *StateObject* could be used to hold the current state of the traffic light, and its associated *SymbolSet* would contain three Symbols modeling the three possible light states: `Red`, `Yellow`, and `Green`. A *StateObject* is accessed via a *Port*, through which a *Symbol* is written (an *InPort*) or read (an *OutPort*).

B. Metamodel for FFBDs

The abstract syntax for an FFBD must account not only for the function blocks but also for the other graphical elements that form the basic FFBD constructs shown in Fig. 1. The

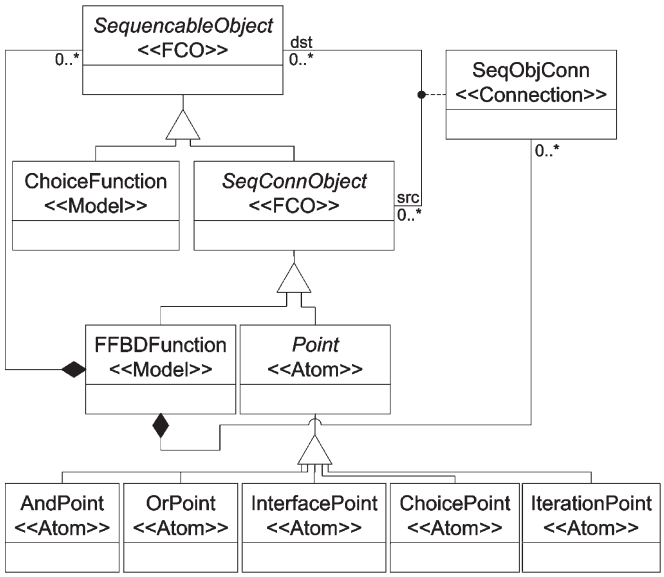

Fig. 4. Metamodel of FFBD functions and block constructs.

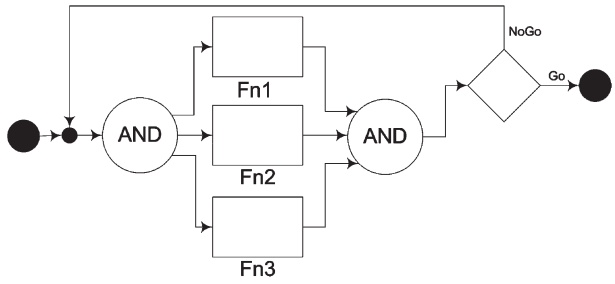

Fig. 5. Sample FFBD diagram, conforming to the metamodel in Fig. 4.

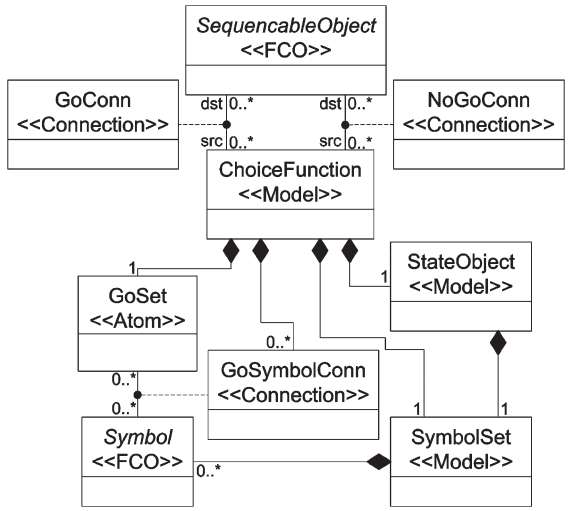

Fig. 6. Metamodel of *ChoiceFunction*.

metamodel for FFBDs is shown in Fig. 4. The basic building block of an FFBD is the *FFBDFunction*. Functions can reside at the leaf level in the hierarchy or can be defined in terms of other FFBD constructs. FFBDs use sequencing to capture the order in which operations occur. In Fig. 1, sequencing is shown as a solid directional connection between functions. In the metamodel in Fig. 4, the *SeqObjConn* class represents such connections. The *SequencableObject* abstract class represents an object that can be connected in a sequence. *ChoiceFunctions* model Boolean conditions used in defining Choice and Iteration constructs. The *SeqConnObject* class represents an object that can be the source of a sequencing connection. Since functions can be composed in sequence, *FFBDFunction* inherits from the *SeqConnObject* class. Objects of type *Point* can also be composed in sequences. *Point* objects are used to formulate the FFBD constructs (Parallel, Alternation, Iteration, Choice) shown in Fig. 1. The metamodel supports several types of *Point* objects, each of which is employed in different contexts. *AndPoint*s mark the start and end of an AND construct. Similarly, *OrPoints* mark the start and end of an OR construct. *InterfacePoints* mark where behavior starts and ends within a function. For the Iteration and Choice constructs, separate paths converge at a *Point* object: a *ChoicePoint* represents the location where paths corresponding to separate outcomes of a Choice construct merge, whereas the *IterationPoint* represents the junction of the entry and loop paths in the Iteration construct.

Fig. 5 shows a sample FFBD that conforms to the FFBD metamodel. The large dots on the left and right represent *InterfacePoints*. *AndPoints* are denoted by circles enclosing the term AND. The two *AndPoints* enclose a Parallel construct, indicating that the functions `Fn1`, `Fn2`, and `Fn3` all execute concurrently. The diamond represents a *ChoiceFunction* and forms part of an Iteration construct.

Iteration and Choice both imply conditional evaluation. The *ChoiceFunction* is used to model the evaluation of a Boolean condition. Its metamodel is shown in Fig. 6. A *ChoiceFunction* can be the source of two types of connections: *GoConnection* and *NoGoConnection*. If the *ChoiceFunction* evaluates to "true," execution proceeds by following the *GoConn* connection; if not, the *NoGoConn* is traversed. For example, Fig. 5 shows an *Iteration* construct, embedded in which is a Parallel construct. The *ChoiceFunction* diamond represents the Boolean choice of whether to terminate the iteration. The *NoGo* path of type *NoGoConn* leads from the *ChoiceFunction* to the *IterationPoint* just to the left of the initial *AndPoint* and models the feedback path of the Iteration.

Internal to the *ChoiceFunction* are a *GoSet* object and a *StateObject*, as defined in Fig. 6. Different *Symbols* from the *StateObject*'s *SymbolSet* can be associated with the *GoSet* through the *GoSymbolConn*. Evaluation of a *ChoiceFunction* involves retrieval of the current *Symbol* stored in the *StateObject*, followed by determination of whether that *Symbol* is connected to the *GoSet* object. If such a connection exists, the *ChoiceFunction* evaluates to "true."

C. Modeling Data Flow

The metamodel also provides for the specification of data-flow relationships between functions. From a data-flow perspective, a function represents a stateless mapping of inputs to outputs [17]. Fig. 7 shows the metamodel for specifying data-flow behavior. A data value is represented as a *Symbol*. A *DataflowFunction* contains *Port*s, each of which is associated with a *SymbolSet*. The *SymbolSet* represents the set of values that can be communicated via that *Port*. The input/output

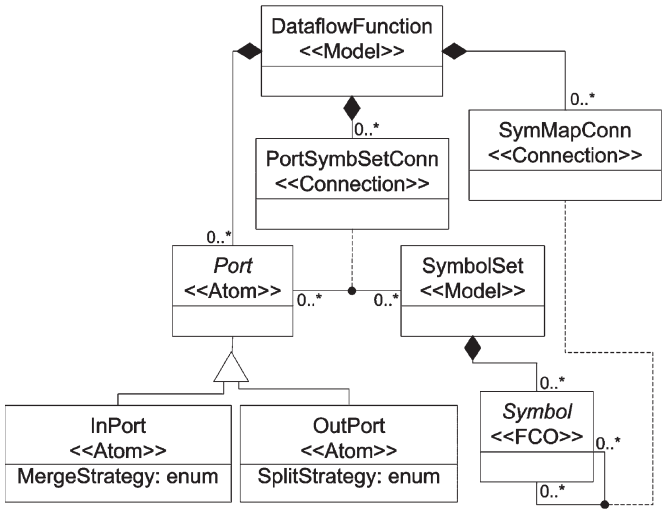

Fig. 7. Metamodel of the data-flow function.

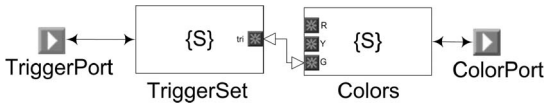

Fig. 8. *DataflowFunction* model of a function that maps occurrences of a trigger *Symbol* to a green traffic light state.

mapping is specified by connecting *Symbols* in the *InPort*'s *SymbolSet* to *Symbol*s in the *OutPort*'s *SymbolSet* via the *SymMapConn*. The MergeStrategy and SplitStrategy attributes on the *InPort* and *OutPort* aid in the modeling of functions containing more than one input or output *Port*.

A sample *DataflowFunction* model named `ChangeLight ToGreen` is shown in Fig. 8. On receipt of a trigger signal, the function outputs the Symbol `Green`. This function could be used in a traffic light controller to describe changing light state. The `TriggerPort` object on the left is of type *InPort* and is associated with the *SymbolSet* `TriggerSet`. `ColorPort`, on the right, is of type *OutPort* and is associated with the *SymbolSet* `Colors`. The connection between the Symbols `trigger` in `TriggerSet` and `Green` in `Colors` is of type *SymMapConn* and dictates the mapping defined by the function: from Symbol `trigger` to Symbol `Green`.

Fig. 9 shows the rules for composing a system-level DFD based on *SharedState* objects, *DataflowFunctions*, and *ExternalChannels*. A root-level *DataflowFunction* defines the system DFD. A *SharedState* object represents a reference to a *StateObject* and models a data store or state-bearing data-flow connector. An *ExternalChannel* models a data source or sink that interfaces with the model's environment, e.g., a sensor, actuator, or other external device. An *ExternalChannel* contains a single *InPort* or *OutPort*, depending on the sense of the *ExternalChannel*. The *ExtChanRef* allows an *ExternalChannel* to be used in multiple locations in the model, regardless of hierarchical decomposition. All of the objects in a DFD can be interconnected via their ports, using the *PortToPortConn*. Fig. 10 shows a sample DFD composed of two functions `Timer` and `ChangeLightToGreen` (see Fig. 8). The DFD indicates that the system receives a clock signal from an external source and forwards the signal to the `Timer` function. The

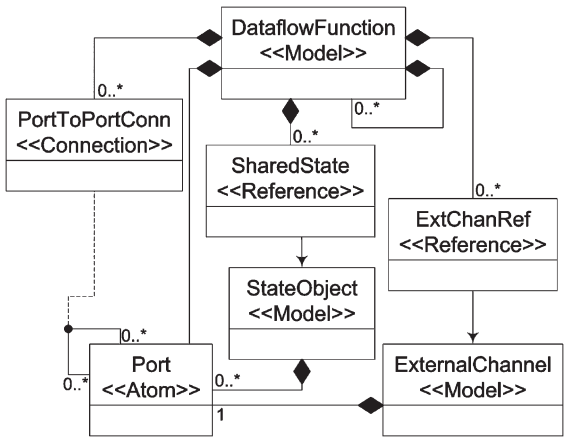

Fig. 9. Metamodel showing *DataflowFunction* composition and interaction.

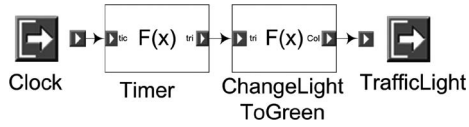

Fig. 10. Sample DFD with two functions and two external channels.

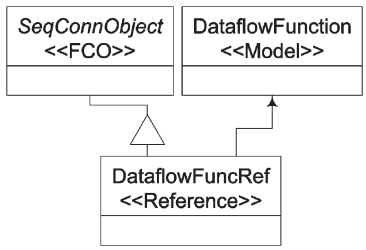

Fig. 11. Metamodel illustrating the relationship between DFDs and FFBDs.

`Timer` function emits a trigger symbol, which is consumed by `ChangeLightToGreen`. That function then emits a `Color`, which is sent to the `TrafficLight` external channel.

D. Integrating Data Flow With FFBDs

The metamodel allows users to integrate the data-flow view with the FFBD view using the constructs defined in Fig. 11. *DataflowFuncRef* is defined as a reference to a *DataflowFunction* object and inherits from the *SeqConnObject* class defined in Fig. 4. The inheritance relationship allows *DataflowFuncRef* objects to be involved in sequences along with other FFBD constructs. Thus, references to *DataflowFunctions* can act as leaf-level functions in an FFBD. Semantically, the composition allows the control flow to constrain the execution of the data-flow functions, permitting them to fire only when the corresponding FFBD dictates. As an example of syntax, Fn1 defined in the Parallel construct in Fig. 5 could be specified as a reference to the `ChangeLightToGreen` function, which is defined in Fig. 8. Inside GME, the user can double click on a leaf-level function (e.g., Fn1 in Fig. 5) in the FFBD view to navigate to the composed DFD (e.g., Fig. 10), which shows the corresponding function in its data-flow context. Double clicking on the data-flow function yields the internal data-flow mapping specification (e.g., Fig. 8).

E. Interpretation: From Models to CSP

The composition of visual models to represent a system may have value in and of itself as an exercise in abstraction and understanding, as well as for documentation purposes. However, visual modeling can also become the basis of an effective analysis tool when the information captured in the diagrams can be extracted and automatically mapped into some useful format. We have developed a model interpreter that generates a CSP representation of a modeled system that can be verified using a model checking tool. The model interpreter was written in C++ using the Universal Data Model [23] API and library. The interpreter traverses a given model, following the structure specified in the metamodel. It first maps the model into a set of internal data structures modeling the FFBD/data-flow constructs, decoupling the CSP generation from the metamodel. The CSP generator is implemented as a traversal over these data structures, utilizing the visitor pattern [24]. The output of the interpreter is a text file containing the CSP equivalent of the user-specified model, based on the CSP constructs described in Section IV.

IV. Representing FFBDs in CSP

A number of different formalisms could be used to define a formal semantic model for FFBDs, each providing access to different kinds of analysis. We have elected to use CSP as our initial target formalism for several reasons. First, CSP is an established formalism for modeling concurrent systems [25]. It is capable of representing the concurrent behavior found in most FFBDs and DFDs and supports analysis of the sequencing, safety, and progress properties that are fundamental to determining the correctness of the behavior specified by an FFBD. Second, CSP allows us to build process models that have a hierarchical structure (see [19, Sec. 2.1]), paralleling the FFBDs upon which the models are based. Third, a wide array of tools are available for analyzing CSP process models, including the mature and commercially available FDR2 refinement checker [19], [26], which has been used on industrial projects by companies such as DaimlerChrysler Aerospace [27], QinetiQ [28], Praxis High-Integrity Systems [29], and IBM [30]. Academic tools targeting CSP are also available, such as the ProB model checker [31] and the PAT toolsuite [32], both of which can perform model checking of CSP processes for conformance to behavior specifications expressed in linear temporal logic. Support for proof-based analysis of CSP process models is available in Isabelle/HOL [33], which embeds CSP semantics into a theorem prover. The combination of concurrency, hierarchical modeling, and diverse tool support, along with its breadth of exposure in academia and industry, makes CSP a good starting point for experimenting with and specifying FFBD semantic models.

Because there is no single definition of the precise meaning of FFBD notations, we had to select an interpretation for each FFBD construct to create a mapping from FFBD notation to CSP. The resulting formal semantic model is intended to be consistent with the informal descriptions of FFBD behavior found in widely used references such as DSMC Systems Engineering Fundamentals [4], the NASA Systems Engineering Handbook [5], and the FAA System Engineering Manual [6]. Other interpretations of FFBD semantics are of course possible, resulting in FFBDs that define different behaviors than those constructed using the model defined here. The potential for differing interpretations of FFBD notations is one motivation for having a formal definition of the semantics associated with an FFBD. The formal semantic model defined in this paper provides a reference point from which other semantic models can be developed.

The basic building blocks of the FFBD semantic model are categorized as follows:

1) *function blocks*, which describe input/output transformations;
2) *data-flow connections*, which model the flow of data between functions;
3) *constraint processes*, which define the sequencing and control flow of functions and are used to represent the fundamental FFBD constructs shown in Fig. 1.

In the rest of this section, we use both informal descriptions and CSP to describe each building block and then discuss how they can be composed to form a system behavior model.

A. Function Blocks

The metamodel described in the previous section syntactically represents data-flow functions as stateless mappings that transform inputs to outputs. This representation of functions is consistent with the way that functions are typically described in systems engineering textbooks [1], [17]. However, most authors fail to explicitly define the behavior of leaf-level function blocks, leaving it ambiguous whether a block must complete the transformation of an input into its corresponding output before accepting the next input. Such ambiguities must be resolved if the overall behavior of a system is to be simulated or analyzed. In the formal semantic model, leaf-level function-block behavior is made explicit by combining a definition of behavior with a representation of the input/output mapping associated with the function block.

Mathematically, a function input/output mapping can be defined either extensionally, as sets of pairs of input and output values, or intensionally, as a rule defining the relationship between inputs and outputs. CSP supports a rich language of functions and sets that is capable of expressing mappings defined extensionally by explicit enumeration or intensionally as algebraic or algorithmic rules, or predicates on sets of input/output pairs. However, the visual language defined by the metamodel presently only supports extensional definition of mappings as enumerations of input/output pairs. Although the restriction to extensional definitions limits the complexity of the functions that can be defined, these limitations are not inherent in the CSP semantic model. A future extension to the metamodel and visual language could be constructed to allow other forms of function definition.

Both intensional and extensional definitions of functions define how inputs are mapped to outputs, but neither describes the sequencing of successive inputs and outputs. To define this sequencing, we associate a behavior with the input/output

mapping by encapsulating the definition of the mapping inside a CSP process description, effectively "lifting" the function into the behavior domain. The "lifted function" is a parameterized process that alternates between receiving inputs and producing corresponding outputs, i.e.,

$$\begin{aligned}&\texttt{LiftF}(\texttt{in},\ \texttt{out},\ \texttt{rdy},\ \texttt{fin},\ \texttt{f}) = \\ &\quad \texttt{rdy} \rightarrow \texttt{in?x} \rightarrow \texttt{out!f(x)} \rightarrow \texttt{fin} \rightarrow \\ &\quad\quad \texttt{LiftF}(\texttt{in},\ \texttt{out},\ \texttt{rdy},\ \texttt{fin},\ \texttt{f})\end{aligned}$$

where the `in` parameter represents the input channel, and the `out` parameter represents the output channel. The mapping from inputs to outputs is represented by a function parameter `f`. The event parameters `rdy` and `fin` are used in conjunction with the constraint processes described in Section IV-C to model control flow: other processes may synchronize on these events to enable execution of the function block (`rdy`) or to determine when execution of the function block has completed (`fin`). The `LiftF` process thus defines the data-flow and control-flow behavior of a leaf-level function block, corresponding to a single leaf-level box in an FFBD (see Fig. 1). Because `LiftF` is a CSP process, it can be combined in well-defined ways with process expressions representing other aspects of an FFBD.

As an example of using the `LiftF` process, consider the `ChangeLightToGreen` function block shown in Fig. 10. This might be modeled by the process expression

$$\begin{aligned}&\texttt{ChangeLightToGreen} = \\ &\quad \texttt{LiftF}\,(\texttt{chgLt.tri},\ \texttt{chgLt.col},\ \texttt{chgLt.rdy}, \\ &\quad\quad \texttt{chgLt.fin},\ \{(\texttt{trigger},\ \texttt{green})\})\end{aligned}$$

where `chgLt.tri` and `chgLt.col` represent the input and output ports, and the function is a single two-tuple representing the `ChangeLightGreen` function mapping shown in Fig. 8. The process `ChangeLightToGreen` signals that it is ready on `chgLt.rdy`, receives an input symbol on the `chgLt.tri` channel, uses the mapping $(\texttt{trigger}, \texttt{green})$ to convert the input to an output on `chgLt.col`, and finally signals completion on `chgLt.fin`. It will then repeat this behavior.

The `LiftF` process models a function block that accepts a single input and produces a single output. Multi-input–multi-output functions can be modeled by composing a `LiftF` that transforms N-tuples with additional processes that combine separate inputs into N-tuples or break N-tuples into separate outputs. We refer to these additional processes as *merge* and *split* strategies. A merge strategy specifies whether inputs are received in parallel or in sequence. A split strategy provides the same kind of specification for outputs. For example, to model a two-input function, such as the function `P0` in Fig. 2, we might combine a sequential merge that collects a pair of inputs into a two-tuple with a `LiftF` process that maps two-tuples to some output. The merge strategy is defined as

$$\begin{aligned}&\texttt{SeqIn2}(\texttt{rdy},\ \texttt{in1},\ \texttt{in2},\ \texttt{inLF}) = \\ &\quad \texttt{rdy} \rightarrow \texttt{in1?x} \rightarrow \texttt{in2?y} \rightarrow \texttt{inLF!}(\texttt{x},\texttt{y}) \rightarrow \\ &\quad\quad \texttt{SeqIn2}(\texttt{rdy},\ \texttt{in1},\ \texttt{in2},\ \texttt{inLF}).\end{aligned}$$

The two-tuple generated by the merging of inputs from `in1` and `in2` is passed to the `LiftF` for transformation into an output. Note that the merge strategy is designed to synchronize with the `LiftF` process on the `rdy` event, thereby ensuring that the overall behavior of the composite `SeqIn2`/`LiftF` function block adheres to the expected alternation of consuming inputs and producing outputs.

B. Data-Flow Connections

The data-flow portion of the metamodel provides for two different kinds of connections between functions: *PortToPortConn*s represent the direct transmission of *Symbols* between functions, and *StateObject*s allow functions to be decoupled from each other by providing a storage medium or buffer for asynchronously transmitted *Symbols*.

Within the CSP semantic model, the direct connections represented by *PortToPortConn*s are modeled as a parallel composition that synchronizes the output channel of one function block with the input channel of another function block. For example, to model the connection between the data-flow functions `Timer` and `ChangeLightToGreen` shown in Fig. 10, we use the parallel composition

$$\begin{aligned}&\texttt{Timer}\,[\,[\texttt{timer.tri} \leftarrow \texttt{trig}]\,] \\ &[\,|\,\{\,|\texttt{trig}|\,\}\,|\,] \\ &\texttt{ChangeLightToGreen}\,[\,[\texttt{chgLt.tri} \leftarrow \texttt{trig}]\,].\end{aligned}$$

Here, we have used the renaming operator $[[\texttt{a} \leftarrow \texttt{b}]]$ to give the linked input and output channels the common name `trig` and then synchronized the two function-block processes on that common channel.

The data-flow connections represented by *StateObject*s support asynchronous communications by holding a function output value, allowing another function to later read the value when it is ready to process an input. In CSP, mutable value storage is usually modeled by a process that includes the current value as one of the process parameters and that responds to events that modify the stored value by recursively invoking the process with the updated value. At present, our semantic model includes only one kind of *StateObject*, i.e., the `Var` process, which models a one-element data store that retains the most recent value passed to it. The `Var` process is parameterized by a state value `val` and three channels.

1) `set`, through which state value updates are received;
2) `get`, through which the current state value may be read;
3) `trans`, through which notification of changes (transitions) in the state value is signaled to those processes that synchronize on `trans`.

The definition of the `Var` process is

$$\begin{aligned}&\texttt{Var}(\texttt{set},\texttt{get},\texttt{trans},\texttt{val}) = \\ &\quad \texttt{get!val} \rightarrow \texttt{Var}(\texttt{set},\texttt{get},\texttt{trans},\texttt{val}) \\ &\quad [\,] \\ &\quad \texttt{set?v} \rightarrow \\ &\quad\quad \texttt{if v} \,!= \texttt{val} \\ &\quad\quad \texttt{then trans!v} \rightarrow \texttt{Var}(\texttt{set},\texttt{get},\texttt{trans},\texttt{v}) \\ &\quad\quad \texttt{else Var}(\texttt{set},\texttt{get},\texttt{trans},\texttt{val}).\end{aligned}$$

Models of other kinds of state-bearing data-flow connections, including key/value stores such as databases, and buffering

schemes such as queues can be defined using an approach similar to that used in defining the Var process. These can be added to the semantic model as the metamodel is extended with additional types of data-flow connections.

C. Functional-Flow Constraints

The fundamental purpose of FFBDs is to define the order in which different system functions are executed and the decision logic used to select an execution path when several possibilities are available. Within the context of our CSP model of FFBDs, the prescription of functional flow can be viewed as a set of constraints on the sequencing of lifted-function processes representing leaf-level function blocks. The sequencing constraints are imposed on the lifted functions by constructing a constraint process, which defines the permissible function execution sequences in terms of lifted function rdy and fin events, and using a CSP parallel operator to compose the constraint process with the lifted functions to be constrained. Although it is possible to directly construct a single constraint process representing a given FFBD, such a construction is likely to be difficult for any nontrivial diagram. Fortunately, the basic elements of FFBDs are readily described in terms of composable CSP processes, making it possible to construct a constraint-process representation of an FFBD in a compositional manner.

The fundamental primitive from which FFBDs are constructed is the leaf-level function block, which denotes a single execution cycle of the function represented by the block. The *FFBDblock* constraint process is parameterized by the rdy and fin channels of the lifted function that represents the function block being constrained, i.e.,

$$\mathtt{FFBDblock}(\mathtt{rdy}, \mathtt{fin}) = \mathtt{rdy} \rightarrow \mathtt{fin} \rightarrow \mathtt{SKIP}.$$

Each of the FFBD compositional elements provides a way to compose primitive function blocks or blocks that represent lower level FFBDs to form a more complex functional flow. To allow for this hierarchical nesting of FFBD elements within our CSP model, the CSP representation of each compositional element combines two or more *FFBD processes* to produce a new *FFBD process*, where an *FFBD process* is defined as being either a leaf-level FFBDblock process or a composite process formed using an FFBDseq, FFBDand, FFBDor, FFBDchoice, FFBDmulti, or FFBDiteration compositional element.

We now define process models for each of the compositional elements.

Sequencing in the FFBD represents temporal ordering of function blocks, which we model in CSP as a sequential composition of the FFBD1 and FFBD2 processes, i.e.,

$$\mathtt{FFBDseq}(\mathtt{FFBD1}, \mathtt{FFBD2}) = \mathtt{FFBD1}; \mathtt{FFBD2}.$$

The FFBD AND element defines concurrent behavior. We model the AND element as an interleaving. The FFBD process, which is drawn from FFBDset, is executed in parallel with the other members of FFBDset, i.e.,

$$\mathtt{FFBDand}(\mathtt{FFBDset}) = ||| \, \mathtt{FFBD} : \mathtt{FFBDset} \, @ \, \mathtt{FFBD}.$$

The FFBD OR element indicates that each of the branches emanating from the "OR" bubble is a valid alternative behavior. The method of selecting an alternative is left unspecified by most authors. In the absence of specific information about how to choose a particular branch, we treat resolution of the choice as nondeterministic. We therefore model the OR element as a generalized nondeterministic choice over a set FFBDset. One FFBD process drawn from FFBDset is nondeterministically selected for execution, i.e.,

$$\mathtt{FFBDor}(\mathtt{FFBDset}) = | \sim | \, \mathtt{FFBD} : \mathtt{FFBDset} \, @ \, \mathtt{FFBD}.$$

The FFBD choice element provides the ability to specify a deterministic choice of alternative behaviors. The usual binary choice specifies which of two branches may be taken, based on the outcome of some test. Following the assumptions of the FFBD metamodel, the test is performed on the value currently held in a data store. If the value held in the store is a member of a subset of values defined as the GoSet, then the Go branch is taken. If not, then the alternative branch, i.e., the NoGo path, is taken. We model the FFBD choice element as a process that reads a state value and, based on the value read, selects for execution either the Go or NoGo branch

$$\mathtt{FFBDchoice}(\mathtt{test}, \mathtt{GoSet}, \mathtt{Go}, \mathtt{NoGo}) = \mathtt{test?x} \rightarrow \mathtt{if\ x} \in \mathtt{GoSet\ then\ Go\ else\ NoGo}.$$

To allow the specification of deterministic choices over more than two behaviors, the semantic model includes a multiway choice element.[1] Multiway choice operates on a similar principle to binary choice, selecting a branch for execution based on the value of a data store. The choices are specified as a set Branches of two-tuples, where the first element of each two-tuple is a set of values that should result in the selection of a branch, and the second element is the corresponding FFBD process. Multiway choice is defined as a generalized external choice over the possible branches

$$\mathtt{FFBDmulti}(\mathtt{test}, \mathtt{Branches}) = [\,] \, (\mathtt{GoSet}, \mathtt{Go}) : \mathtt{Branches} \, @ \, \mathtt{test?x} : \mathtt{GoSet} \rightarrow \mathtt{Go}.$$

The FFBD iteration element provides looping behavior that terminates when a test is passed. We model iteration as a recursive invocation of FFBDseq and FFBDchoice processes in which one of the branches of the FFBDchoice is successful termination of the iteration process

$$\begin{array}{l} \mathtt{FFBDiteration}(\mathtt{test}, \mathtt{StopSet}, \mathtt{FFBD}) = \\ \quad \mathtt{let} \\ \quad\quad \mathtt{Loop} = \\ \quad\quad\quad \mathtt{FFBDseq}\,(\mathtt{FFBD}, \\ \quad\quad\quad\quad \mathtt{FFBDchoice}(\mathtt{test}, \mathtt{StopSet}, \mathtt{SKIP}, \mathtt{Loop})) \\ \quad \mathtt{within\ Loop}. \end{array}$$

[1] Multiway choice is not one of the usual basic FFBD constructs and is not presently included in the FFBD abstract syntax. It is included here for completeness.

The aforementioned Loop process uses a composition of FFBD processes to define an internal sequential behavior. More generally, the constraint process corresponding to an arbitrary FFBD may be created by composing the FFBD processes previously defined. For example, a simple FFBD that is graphically identical to that given in Fig. 5 and that repeatedly executes functions F1, F2 and F3 in parallel until state X contains a number between 3 and 5 can be represented by the process

$$\begin{aligned}&\texttt{SimpleFFBD} = \\ &\quad \texttt{FFBDiteration}\,(\texttt{X}, \{3,4,5\}, \\ &\qquad \texttt{FFBDand}\,(\{\texttt{FFBDblock}(\texttt{F1.rdy},\ \texttt{F1.fin}), \\ &\qquad\qquad \texttt{FFBDblock}(\texttt{F2.rdy},\ \texttt{F2.fin}), \\ &\qquad\qquad \texttt{FFBDblock}(\texttt{F3.rdy},\ \texttt{F3.fin})\}))\,.\end{aligned}$$

D. Composite System Behavior

In terms of the CSP processes previously defined, the data-flow view of the metamodel can be thought of as a composition of lifted functions and assignable states. Similarly, the FFBD view can be seen as a composition of lifted functions and an FFBD process that constrains the sequencing of those functions. Using parallel composition, the FFBD and data-flow views can be brought together to form a composite picture of the overall behavior the system, i.e.,

$$\begin{aligned}&\texttt{SystemBehavior} = \\ &\quad (\,(\texttt{LF1}\ [\,|\,\texttt{Con1}\,|\,]\ldots[\,|\,\texttt{ConN}\,|\,]\ \texttt{LFN}) \\ &\quad\ [\,|\,\texttt{IFfd}\,|\,] \\ &\quad\ (\texttt{Store1}\,|\,|\,|\ldots|\,|\,|\,\texttt{StoreN})\,) \\ &\quad\ [\,|\,\texttt{IFff}\,|\,] \\ &\quad\ \texttt{FFBD}\end{aligned}$$

where LF1 . . . LFN are lifted functions, Con1 . . . ConN represent *PortToPortConn*s between functions, Store1 . . . StoreN are data stores (state-bearing data-flow connections), FFBD is an FFBD process constraining the functions LiftF1 . . . LiftFN, IFfd is the set defining the interface between the data stores and the lifted functions, and IFff is the set defining the interface between the FFBD and the lifted functions. Note that the data stores are all independent of each other and execute concurrently, whereas the lifted functions may interact with each other either directly via synchronization or indirectly via the data stores. In practice, wiring all of the processes together into a full system model often requires some use of techniques such as channel renaming, to ensure that processes communicating with shared resources do not interfere with each other. However, the essential structure of a system behavior process description is that of the parallel composition previously shown.

V. Case Study

In this section, we present a brief case study developed to highlight the features and capabilities of the modeling tool and CSP-based semantics. The case study examines a previously published problem from systems engineering literature: the traffic light problem [34]. While the problem does not rise to the level of a real-world systems engineering problem, it offers sufficient complexity to demonstrate the capabilities of the approach.

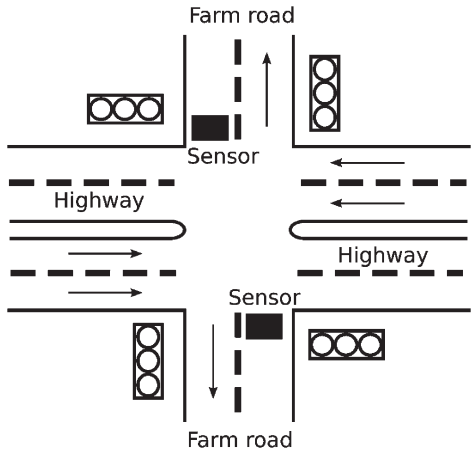

Fig. 12. Traffic light problem.

A. Modeling the System

The traffic light problem involves specifying a control system for a set of traffic lights located at the intersection of a busy highway and an infrequently used farm road (Fig. 12). Sensors are placed on the farm road to detect the presence of vehicles attempting to cross the highway. The control system must manage the traffic lights such that vehicles on the highway are allowed through the intersection without interruption, except on the occasion when a farm vehicle wishes to cross the highway. The problem statement specifies that the system should have the following behavior.

1) When a vehicle is detected on the farm road, the highway light should turn yellow. Then, after a short time interval (STI), the highway light should turn red, and the farm-road light should turn green.
2) The farm-road light should remain green until the vehicle clears a sensor or a long time interval (LTI) passes, whichever comes first. Then, the farm-road light should turn yellow and hold that state for an STI, after which the farm-road light should turn red and the highway light should return to green.
3) Once the farm-road light is red, it cannot turn green again until at least one LTI has passed.

Several solutions to the traffic light problem have been offered and analyzed, using various tools and analysis approaches. We have adapted the FFBD models from the *Solution via Functional Decomposition* section of Bahill's "Design Methods Comparison" (DMC) project [34].

The overall behavior of the traffic light system is defined by the top-level FFBD shown in Fig. 13. The system first executes two initialization functions: 1) INIT, which sets the highway light to green and 2) ResetLTI, which resets the LTI counter.

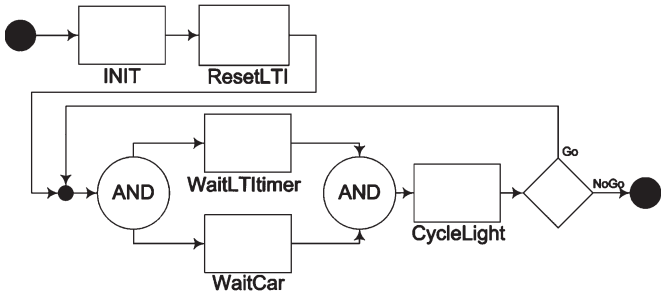

Fig. 13. Top-level FFBD for the traffic light problem.

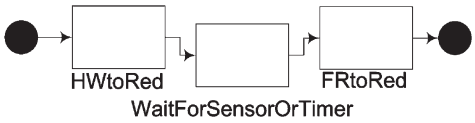

Fig. 14. `CycleLight` FFBD definition.

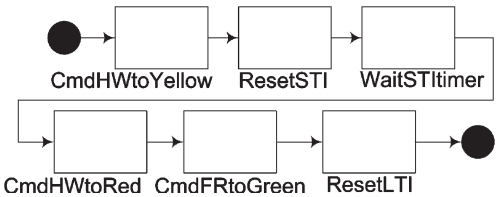

Fig. 15. `HWtoRed` FFBD definition.

The system then enters a loop in which it waits until a change in the light color is required and then cycles the traffic light colors. As shown in Fig. 13, a cycle is triggered when both a vehicle is detected by the farm-road sensor and at least one LTI has passed since the farm-road light was last green.

The traffic light system is modeled hierarchically. The two `WaitX` function blocks model functions that wait for an event from the corresponding sensor or timer, and then report a sampled value. Their definitions are omitted due to space considerations. The core traffic light behavior, which is defined in the `CycleLight` function block (Fig. 14), involves three sequential steps. The `HWtoRed` block, as shown in Fig. 15, changes the highway light to yellow, waits for an STI, then changes the highway light to red and the farm-road light to green, and starts the LTI timer. The `FRtoRed` block specifies a similar behavior but switches the farm-road light to red. Between the two light-change steps, the `WaitForSensorOrTimer` function prevents the farm-road light from changing back to red until either the vehicle sensor indicates the farm-road vehicle has cleared the intersection or the LTI timer has expired, whichever occurs first. As shown in Fig. 16, the wait behavior involves waiting simultaneously for both the road sensor and the LTI timer, followed by a check of the values acquired to see if the conditions for terminating the wait loop have been satisfied.[2]

[2]This specification differs slightly from the FFBD presented in the DMC paper. The DMC FFBD uses an OR block to combine two wait functions, implicitly executing both and terminating the OR when one of the functions terminates. Such an interpretation of OR composition is not consistent with that used by other authors nor with our semantic model (which selects one branch for execution). Consequently, we used a different model for the wait termination conditions.

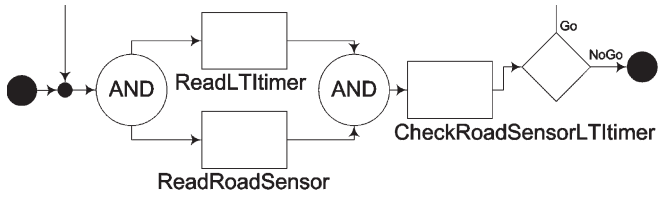

Fig. 16. `WaitForSensorOrTimer` FFBD definition.

Each block in Fig. 16 represents a leaf-level block in the FFBD hierarchy. In the preceding discussion of functional flow, it is assumed that such leaf-level functions have the ability to exchange data. For example, the `CheckRoadSensorLTItimer` function examines values that `ReadRoadSensor` acquires from the road sensor. The FFBD control-flow view of the system does not depict these data dependence relationships. Interfunction relationships are instead captured in the data-flow view of the system, as shown in Fig. 17.

Within the DFD, functions are represented by boxes labeled $\mathbf{F}(\mathbf{x})$, and data stores are represented by pairs of horizontal lines. The large arrows to the far right and left of the diagram represent channels to external entities. The input `PowerOn` models a flag indicating system startup, whereas `RoadSensor` represents inputs from the vehicle sensors on the farm-road, and the `timerLTI` and `timerSTI` inputs provide timer status in the form of a simple expired/not-expired flag. The `trigger` input channel provides a signal that models the activation of functions that produce outputs that do not depend on any input data flow. On the output side, `HWlight` and `FRlight` represent channels through which commands are issued to the traffic light hardware, whereas `toSTItimer` and `toLTItimer` are the channels through which timer reset signals are issued.

The DFD shows several different functions responsible for commanding changes to each of the two traffic lights; however, the functional flow defined in the FFBD ensures that only one of these functions per traffic light is active at any time. Each function maps the inputs it receives from external inputs or from internal data stores onto a traffic light color command. For example, the `CmdHWtoRed` function maps the current status of the STI timer, which is used to define the interval over which lights are to be yellow, onto a light color that is either yellow or red. The input/output mapping used to define the `CmdHWtoRed` function is depicted in Fig. 18. The function is defined as a mapping from `TimerSignals` *Symbols*, which represent the currents timer status, to `LightColor` *Symbols*, which represent traffic light color commands. The diagram specifies that, if the `TimerSignal` received via the `STIStatus` port indicates that the timer has not expired (`tNotUp`), then `yellow` is output through the `HWLightColor` port. Similarly, if the timer has expired (`tUp`), then a `red` color command is output. All leaf-level functions in the FFBD view of the system are defined in the data-flow view in a similar fashion.

The system model defined by the FFBD and data-flow views was automatically translated into CSP using the model interpreter. The resulting CSP model is approximately 300 lines of generated code and is thus too large to present here. A sample of the generated CSP, showing just the top-level process

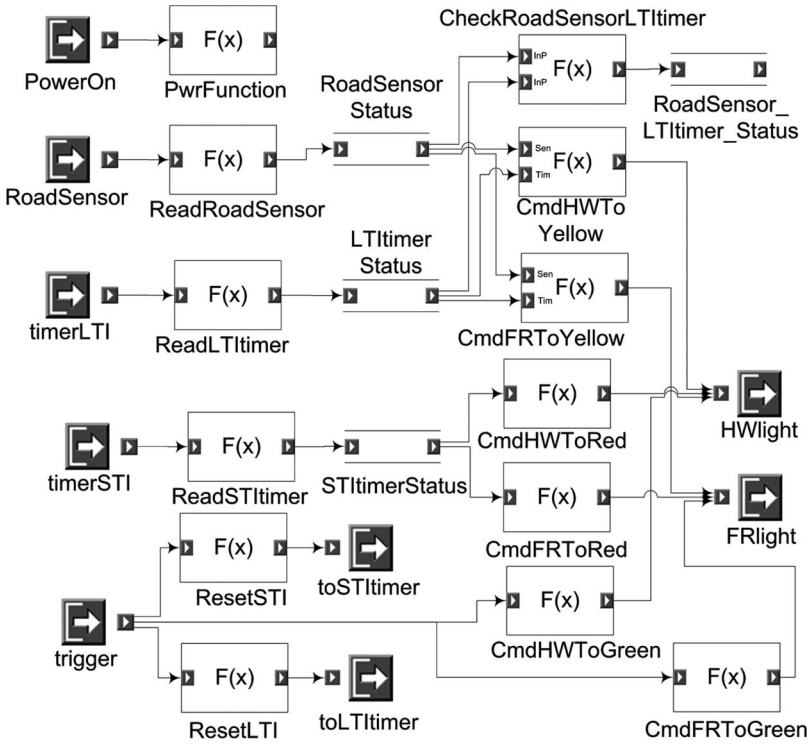

Fig. 17. Data-flow view of traffic light controller.

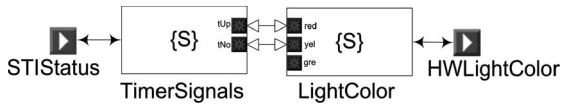

Fig. 18. Specification of a function mapping timer signals to light colors.

definitions and eliding details of the renaming used to wire together the DFD, appears as follows:

```
trafficLight_dfd =
    ( ( (| | (Alpha, Proc) : trafficLight_dfd_set @
                  [Alpha] Proc) [[ ⟨renaming⟩ ]] )
  [| IFfd |] dfd_StateObjects)

trafficLight_FFBD =
    FFBDseq (blk_PwrFunction,
     FFBDseq (blk_ResetLTI,
      FFBDiteration (
          sense_LightStatus_FFBD, {err},
          FFBDseq (FFBDand ({blk_WaitForLTITimer,
                             blk_WaitForCar}),
                   blk_ExecuteLightCycle) ) ) )

trafficLight = (trafficLight_dfd
                    [| IFff |] trafficLight_FFBD).
```

The preceding discussion has shown how the traffic light problem may be modeled using FFBDs and DFDs created with a tool implementing the formal syntax proposed in this paper. A CSP system model that conforms to the formal semantics described in Section IV and corresponds to the traffic light FFBDs and DFDs was generated using a model interpreter. In the next section, we examine the use of model checking to analyze the properties of the system model.

B. Analyzing the System

We performed several different analyses on the generated CSP of the traffic light model using the FDR2 refinement checker, which is a mature commercial tool for CSP analysis. Analyses performed using FDR2 are specified by defining *assertions* about the behavior that a model should exhibit, which are typically expressed as a refinement of a more abstract process. If an assertion is found to be false, FDR2 generates a *counterexample* that shows the sequence of events which led to an incorrect behavior.

While FDR2 does not limit the number of assertion checks that can be performed on a model, we performed only four illustrative checks on the traffic light model, which are given as follows:

1) check for consistency between the functional flow and data-flow views of the system;
2) check that the modeled design will never enter a state in which both lights are green at the same time;
3) check that, given certain assumptions about the environment with which the design interacts, the lights will actually switch;
4) check that the modeled design is consistent with the original problem statement.

We now describe each check in more detail, and give the corresponding CSP used to specify each assertion.

The consistency check between functional flow and data flow is intended to ensure that there are no incompatibilities between the sequencing of different functions and the interactions between those functions. The check is expressed as an assertion that the traffic light model does not deadlock (i.e., never reaches a state in which no progress is possible), written as

$$\texttt{assert trafficLight} : [\texttt{deadlock free [FD]}] .$$

The deadlock check revealed an incompatibility between the functional-flow and data-flow views in an early iteration of the model; the data-flow view specified that the CmdHWtoRed function should send an output to the CmdFRtoGreen function, but the functional flow was such that CmdFRtoGreen could not accept an input until CmdHWtoRed had completed, which could not happen until CmdHWtoRed had completed its output to CmdFRtoGreen. The counterexample information provided by FDR2 allowed the source of the incompatibility to be rapidly diagnosed and the model to be revised accordingly.

The second check, which is a safety check, is expressed as an assertion that the traffic light model is a trace refinement of the process Safety, which starts with the highway light green as specified in the problem statement and excludes traces containing the obviously unsafe situation in which both traffic lights are green simultaneously, i.e.,

$$\begin{aligned}
&\texttt{Safety} = \\
&\quad \texttt{let} \\
&\quad \texttt{LightGY(light)} = \\
&\qquad (\texttt{light?c} : \{\texttt{green, yellow}\} \rightarrow \\
&\qquad\quad \texttt{LightGY(light)}\ ) \\
&\qquad [\,]\ (\texttt{light.red} \rightarrow \texttt{LightsNotGreen}) \\
&\quad \texttt{LightsNotGreen} = \\
&\qquad [\,]\ \texttt{light} : \{\texttt{HWlight, FRlight}\}\ @ \\
&\qquad\quad \texttt{light?c} \rightarrow \texttt{if c} == \texttt{green} \\
&\qquad\qquad\qquad \texttt{then LightGY(light)} \\
&\qquad\qquad\qquad \texttt{else LightsNotGreen} \\
&\quad \texttt{within LightGY(HWlight)} \\
&\texttt{assert Safety } [\texttt{T} = \\
&\quad \texttt{trafficLight} \setminus \texttt{diff(Events,} \\
&\qquad\qquad \{\,|\texttt{HWlight, FRlight}|\,\}\ )
\end{aligned}$$

where $\setminus$ denotes "hiding" of all events in the set following the hiding operator, diff represents a set difference, and Events is the set of all events. The model passes this check, indicating that $\texttt{traces(System)} \subseteq \texttt{traces(Safety)}$ and that the modeled design does not enter a state in which both lights are green in any possible execution sequence.

Safety verification alone does not guarantee that the system will actually function as desired since one possible way to avoid producing unsafe traces is to avoid switching the lights. Therefore, our third check is that the system does not remain stuck in a single light state but instead always makes some kind of progress. The specification that the lights will always be switched is formulated in assumption-commitment style [35]. The commitment is that some light-color transition must always occur, whereas the assumptions involve the behavior of the environment in which the traffic light system will operate. The commitment is expressed as a recursive nondeterministic choice over light-transition events, i.e.,

$$\begin{aligned}
&\texttt{DF(A)} = |\sim|\ \texttt{a} : \texttt{A}\ @\ \texttt{a} \rightarrow \texttt{DF(A)} \\
&\texttt{Commitment} = \texttt{DF}\,(\{\,|\texttt{HWlight, FRlight}|\,\})\,.
\end{aligned}$$

The use of the nondeterministic choice operator means that a process failures-divergence refining the commitment process may only be capable of performing a subset of the light-switching events in any given state but can never reach a state in which no further light-switching events will occur.

The assumptions are expressed as processes that encode the assumed behavior of the environment of the traffic light system, i.e., the LTI and STI timers, and the vehicles on the farm road. For example, the system clearly cannot meet the commitment if there are never any cars on the farm road and thus never a reason to switch the lights. The process

$$\begin{aligned}
&\texttt{CarModel(v, Init)} = \texttt{Cars(v, Init, Init)} \\
&\texttt{Cars(v, Init, N)} = \\
&\quad \texttt{if N} > 0 \\
&\quad \texttt{then}\ (\ (\texttt{v.car} \rightarrow \texttt{v.noCar} \rightarrow \texttt{CarModel(v, Init)}\ ) \\
&\qquad\quad |\sim|\ (\texttt{v.noCar} \rightarrow \texttt{Cars(v, Init, N} - 1)\ )\ ) \\
&\quad \texttt{else v.car} \rightarrow \texttt{CarModel(v, Init)}
\end{aligned}$$

encodes the assumption that there are at least some cars on the farm road by specifying that there can be no more than N consecutive negative reads from the car sensor. Similar processes encode assumptions about the consecutive number of times that the timer can signal that it has not yet expired and the response of the timers to reset signals.

The assumption/commitment assertion used to confirm light activity is

$$\begin{aligned}
&\texttt{assert Commitment } [\texttt{FD} = \\
&\quad (\texttt{trafficLight} \\
&\qquad [\,|\,\{\,|\texttt{RoadSensor, timerLTI, toLTItimer,} \\
&\qquad\quad \texttt{timerSTI, toSTItimer}|\,\}\,|\,] \\
&\qquad (\texttt{CarModel(RoadSensor, 200)} \\
&\qquad |||\ \texttt{Timer(timerLTI, toLTItimer, 50)} \\
&\qquad |||\ \texttt{Timer(timerSTI, toSTItimer, 10)}\ )\ ) \\
&\qquad \setminus \texttt{diff}\,(\texttt{Events}, \{\,|\texttt{HWlight, FRlight}|\,\})\,.
\end{aligned}$$

Checking this assertion against an early iteration of the model uncovered a decision-logic error that had slipped through previous manual reviews of the design and could, under a rare combination of conditions, result in the system getting stuck in an infinite loop that left the farm-road light green. Again, the counterexample trace provided by FDR2 enabled quick diagnosis and resolution of the problem.

The final analysis to confirm that the modeled design is consistent with the problem statement is formulated as a check for scenario feasibility. The process

$$\begin{aligned}
&\texttt{NominalScenario} = \\
&\quad (\texttt{timerLTI.tUp} \rightarrow \texttt{RoadSensor.car} \rightarrow \\
&\quad \texttt{HWlight.yellow} \rightarrow \texttt{timerSTI.tUp} \rightarrow \\
&\quad \texttt{HWlight.red} \rightarrow \texttt{FRlight.green} \rightarrow \\
&\quad\quad (\,(\texttt{RoadSensor.noCar} \rightarrow \\
&\quad\quad\quad \texttt{timerLTI.tNotUp} \rightarrow \texttt{SKIP}) \\
&\quad\quad [\,]\ (\texttt{RoadSensor.car} \rightarrow \\
&\quad\quad\quad\quad \texttt{timerLTI.tUp} \rightarrow \texttt{SKIP})\,) \\
&\quad \texttt{FRlight.yellow} \rightarrow \texttt{timerSTI.tUp} \rightarrow \\
&\quad \texttt{FRlight.red} \rightarrow \texttt{HWlight.green} \rightarrow \\
&\quad \texttt{timerLTI.tUp} \rightarrow \texttt{SKIP})
\end{aligned}$$

expresses in CSP the system behavior informally described in the problem statement. Verification that the traffic light system model is capable of successfully completing this scenario is checked by asserting that a `success` event signaling completion of the nominal scenario must occur, i.e.,

$$\begin{aligned}
&\texttt{assert}\ (\texttt{success} \rightarrow \texttt{STOP})\ [\texttt{FD} = \\
&\quad (\texttt{trafficLight} \\
&\quad\quad [\,|\,\{\,|\texttt{RoadSensor}, \texttt{timerLTI}, \\
&\quad\quad\quad\quad \texttt{timerSTI}, \texttt{HWlight}, \texttt{FRlight}|\,\}\,|\,] \\
&\quad\quad \texttt{NominalScenario}\ \texttt{success} \rightarrow \texttt{STOP}) \\
&\quad\quad\quad\quad\quad\quad \setminus\ \texttt{diff}\,(\texttt{Events},\ \{\texttt{success}\}\,)\,.
\end{aligned}$$

Running this analysis on an early iteration of the model revealed a minor inconsistency with the original problem statement, in which the system described by the FFBD incorrectly used the STI timer, instead of the LTI timer, during one of the waiting periods. Once the error was corrected, a repetition of the analysis confirmed that the FFBD produces the behavior described by the problem statement.

Checking all four assertions took approximately 45 s on a 2.4-GHz laptop. The most complex of the assertions, i.e., the assumption/commitment specification, involved checking over 1 500 000 possible system states, which is a feat obviously beyond the capabilities of manual review techniques. As this case study demonstrates, even fairly simple system designs can contain errors that escape manual review.

VI. Related Work

Modeling and formal analysis have been utilized in a variety of contexts (e.g., [36]–[40]). In this section, we briefly review related work on FFBD syntax and semantics, formalization of graphical notations with CSP, and definition of domain-specific modeling languages with GME.

A. FFBD Syntax and Semantics

Both CORE [11] and RDD-100 [10] provide executable FFBD variants that can be used to specify and understand system behavior. RDD-100 allows users to define the duration of each function execution and to derive overall system execution durations. CORE also adds facilities for specifying the probability of selecting different branches of an FFBD and for estimating resource usage during function execution. However, unlike the work presented in this paper, the underlying execution semantics of the FFBDs does not appear to be publicly available, and execution of FFBD models in both tools appears to be restricted to simulating individual traces, rather than performing the kind of comprehensive analyses possible with exhaustive state-space exploration.

B. Formalizing Graphical Notations Using CSP

A number of researchers have investigated the use of CSP to formalize various graphical notations. For example, Ng and Butler [41], Fischer *et al.* [42], and Benghazi *et al.* [43] used CSP to provide a formal semantics for several kinds of UML diagrams. Allen [44] developed the Wright architecture description language as a CSP-based formalization of the informal component-connectivity diagrams typically used to describe software architectures and applied FDR2 to evaluate the interface compatibility of connected components. Wong and Gibbons explored the use of CSP to formalize and analyze work-flow patterns [45] and the Business Process Modeling Notation [46], both of which bear some similarities to FFBDs. Roscoe and Wu [47] developed a detailed CSP semantics for the Statecharts language, permitting Statecharts to be analyzed using FDR2.

C. Domain-Specific Modeling Languages

Several projects that make use of GME to create domain-specific modeling languages have been undertaken. Bapty *et al.* [48] developed a GME-based design tool for supporting the development of adaptive computing systems, where applications are captured as a coarse-grained DFD. The Milan project [49], [50] provides a modeling tool for electronic system design that integrates simulations at multiple levels of granularity. Gray *et al.* [51] developed a suite of tools based on GME to support model transformation between modeling languages using an embedded constraint language. Karsai *et al.* [52] provided a discussion of model-integrated computing and discussed projects that have been completed using GME.

VII. Conclusion

FFBDs are an informal graphical notation for modeling system behavior and a popular tool for system functional analysis and decomposition [5]. Although FFBDs have proven to be a useful tool for system design, their informality makes them difficult to rigorously analyze. By instead using a formalized FFBD notation to define system models, systems engineers can obtain rapid feedback during the model development process and quickly uncover subtle conceptual and design errors. To that end, we have developed a graphical specification tool that permits the capture of precise descriptions of system behavior using FFBDs and DFDs. The abstract syntax of the modeling

language has been formalized as a metamodel, capturing the well-formedness rules of the language. We have implemented a C++-based model interpreter program to automatically translate system specifications captured using our tool into a target semantic model that precisely defines the behavior described by the graphical model.

Our current target semantic model is a CSP-based formal semantics for FFBDs that was developed to enable experimentation with analyses of FFBDs. The CSP realization of FFBDs and DFDs provides a precise and unambiguous definition of the behavior implied by a diagrammatic specification and allows the user to employ existing theory and tools to exhaustively verify a system specification. The CSP semantic model permits analyses of potential deadlock scenarios and verification of safety and progress properties.

As a case study in FFBD formalization, we have used our tool to model the traffic light problem. The resulting model provides precise specification of the functional flow and data flow of the traffic light system in a hierarchical set of mutually consistent diagrams. We have applied the FDR2 refinement checker to the CSP generated by our tool and uncovered several subtle errors in early revisions of the FFBD that had escaped manual reviews of the diagrams.

Both the metamodel frontend and the CSP backend are highly extensible. The GME-based metamodel can be extended to support other notations beyond FFBDs, whereas the model interpreter can be retargeted to CSP semantic models that capture different interpretations of FFBD behavior. Alternatively, a semantic model based on some other formalism could be incorporated into the tool to provide access to analyses not available with CSP. Doing so would require developing a semantic model for FFBDs in terms of the target formalism, just as we constructed our current CSP semantic model. Existing work on translating CSP to other formalisms (e.g., [53]) may make this process somewhat easier, although directly developing a definition of FFBD semantics in the target formalism could result in a less complex semantic model than that generated by translation from CSP. For those formalisms that support specification of priorities, probabilities, temporal constraints, or other properties not captured in standard FFBD notation, it would also be necessary to extend the metamodel with notations suitable for specifying such properties.

The primary contribution of this work is the definition of a formal semantic model for FFBDs. In its current form, the tool requires a certain amount of expertise in CSP to formulate appropriate checks against the FFBD models. It also offers little support for modifying a system model based on feedback obtained through model checking. Users are responsible for interpreting the counterexamples supplied by the model checker in order to pinpoint those constructs in the system model that are responsible for invalid behaviors. In the future, we plan to extend the graphical tool to support the specification of assertions and constraints, which can be checked in the verification process, as well as to support the graphical presentation of counterexamples. We also plan to examine the integration of this tool into a broader tool flow to support end-to-end system level specification of spacecraft systems. That effort will build upon previous work on verifying spacecraft designs against FFBD specifications via CSP refinement checks [54] and using CSP-based formalizations of graphical notations to combine FFBDs with other kinds of notation in well-defined ways [55]. Further interesting research directions involve the coupling of formal methods at the system level, with system synthesis and generative tools to compose provably correct system implementations.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers of this paper for their valuable feedback.

REFERENCES

[1] B. S. Blanchard and W. J. Fabrycky, *Systems Engineering and Analysis*. Upper Saddle River, NJ: Prentice-Hall, 1998.
[2] *INCOSE Systems Engineering Handbook*, Int. Council Syst. Eng., San Diego, CA, 2000.
[3] J. Long, *Relationships Between Common Graphical Representations in Systems Engineering*. Vienna, VA: Vitech Corp. 2002.
[4] *Systems Engineering Fundamentals*, Defense Acquisition Univ., Fort Belvoir, VA, 2001.
[5] S. J. Kapurch and N. E. Rainwater, *NASA Systems Engineering Handbook: NASA SP-2007-6105 Rev 1*. Washington, DC: Nat. Aeronautics Space Admin., Dec. 2007.
[6] *F.A.A. National Airspace System—System Engineering Manual 3.1*, Federal Aviation Admin. Systems Eng. Safety Office, Washington, DC, Oct. 2006.
[7] D. Davis, *SMC Systems Engineering Primer and Handbook*, 3rd ed. Los Angeles, CA: U.S.A.F. Space Missile Syst. Center, 2005.
[8] C. R. Siel, Jr., *Naval 'System of Systems" Systems Engineering Guidebook*. Washington, DC: Office Assistant Secretary Navy for Res., Develop., Acquisition, Nov. 2006.
[9] *OMG Systems Modeling Language (OMG SysML) v1.1*, Object Manage. Group, Needham, MA, Nov. 2008.
[10] K. Jackson, "System animation using RDD-100," in *Proc. IEE Colloq. Methods Techn. Real-Time Syst. Develop.*, London, U.K., Nov. 1994, pp. 4/1–4/5.
[11] *CORE System Definition Guide*, Vitech Corporation, Vienna, VA, 2004.
[12] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of 'semantics'?" *Computer*, vol. 37, no. 10, pp. 64–72, Oct. 2004.
[13] M. von der Beek, "A comparison of statechart variants," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*. New York: Springer-Verlag, 1994, pp. 128–148.
[14] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, pp. 293–333, Oct. 1996.
[15] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, pp. 44–51, Nov. 2001.
[16] *GME User's Manual*, Inst. Softw. Integr. Syst., Vanderbilt Univ., Nashville, TN, Mar. 2004.
[17] D. W. Oliver, T. P. Kelliher, and J. G. J. Keegan, *Engineering Complex Systems With Models and Objects*. New York: McGraw-Hill, 1997.
[18] *Object Constraint Language Specification*, Object Manage. Group, Needham, MA, Ver. 1.1, Sep. 1997.
[19] *Failures-Divergence Refinement: FDR2 User Manual*, Formal Syst. (Europe) Ltd., Oxford, U.K., 2005.
[20] A. W. Roscoe, *The Theory and Practice of Concurrency*. Englewood Cliffs, NJ: Prentice-Hall, 1998.
[21] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Boston, MA: Addison-Wesley, 2004.
[22] C. A. Petri, "Kommunikation mit Automaten," Ph.D. dissertation, Univ. Bonn, Germany, 1962.
[23] E. Magyari, A. Bakay, A. Lang, T. Paka, A. Vizhanyo, A. Agrawal, and G. Karsai, "UDM: An infrastructure for implementing domain-specific modeling languages," in *Proc. 3rd OOPSLA Workshop Domain Specific Modeling*, Anaheim, CA, Oct. 2003.
[24] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1994.

[25] A. E. Abdallah, C. B. Jones, and J. W. Sanders, Eds., *Communicating Sequential Processes: The First 25 Years*. Berlin, Germany: Springer-Verlag, 2005.
[26] M. Goldsmith and I. Zakiuddin, "Critical systems validation and verification with CSP and FDR," in *Proc. Int. Workshop Current Trends Appl. Formal Methods (FM-Trends)*, Boppard, Germany, Oct. 1998, pp. 243–250.
[27] H. Shi, J. Peleska, and M. Kouvaras, "Combining methods for the analysis of a fault-tolerant system," in *Proc. Pacific Rim Int. Symp. Dependable Comput.*, Dec. 1999, pp. 135–142.
[28] S. Creese, "Industrial strength CSP: Opportunities and challenges in model-checking," in *Communicating Sequential Processes: The First 25 Years*. Berlin, Germany: Springer-Verlag, 2005.
[29] A. Hall, "What does industry need from formal specification techniques?," in *Proc. 2nd IEEE Workshop Ind. Strength Formal Specification Techn.*, Oct. 1998, pp. 2–7.
[30] J. Lawrence, "Practical application of CSP and FDR to software design," in *Communicating Sequential Processes: The First 25 Years*. Berlin, Germany: Springer-Verlag, 2005.
[31] M. Leuschel and D. Plagge, *Seven at One Stroke: LTL Model Checking for High-Level Specifications in B, Z, CSP, and More*. Düsseldorf, Germany: Inst. fur Informatik, Heinrich-Heine-Univ., 2007.
[32] J. Sun, Y. Liu, and J. S. Dong, "Model checking CSP revisited: Introducing a process analysis toolkit," in *Proc. 3rd ISoLA*, Oct. 2008, pp. 307–322.
[33] H. Tej and B. Wolff, "A corrected failure-divergence model for CSP in Isabelle/HOL," in *Proc. 4th Int. Symp. Formal Methods Europe*, Graz, Austria, Sep. 1997, pp. 318–337.
[34] A. T. Bahill, M. Alford, K. Bharathan, J. Clymer, D. L. Dean, J. Duke, G. Hill, E. LaBudde, E. Taipale, and A. W. Wymore, "The design methods comparison project," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 28, no. 1, pp. 80–103, Feb. 1998.
[35] W. Simmonds and T. Hawkins, "A CSP framework for analysing fault-tolerant distributed systems," in *Future of Reliable Wireless and Ad Hoc Networks of Roaming Devices (FORWARD) Deliverable D9*, Jun. 2004.
[36] J. Wang, D. Rosca, W. Tempfenhart, A. Milewski, and M. Stoute, "Dynamic workflow modeling and analysis in incident command systems," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 38, no. 5, pp. 1041–1055, Sep. 2008.
[37] J. J. P. Lu and M. Tsai, "Formal modeling and analysis of a secure mobile-agent system," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 38, no. 1, pp. 180–196, Jan. 2008.
[38] H. Wang and Q. Zeng, "Modeling and analysis for workflow constrained by resources and nondetermined time: An approach based on Petri nets," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 38, no. 4, pp. 802–817, Jul. 2008.
[39] Y. Y. Du, C. J. Jiang, and M. C. Zhou, "Modeling and analysis of real-time cooperative systems using Petri nets," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 37, no. 5, pp. 643–654, Sep. 2007.
[40] V. R. L. Shen and T. T.-Y. Juang, "Verification of knowledge-based systems using predicate/transition nets," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 38, no. 1, pp. 78–87, Jan. 2008.
[41] M. Y. Ng and M. Butler, "Towards formalizing UML state diagrams in CSP," in *Proc. 1st IEEE Int. Conf. Softw. Eng. Formal Methods*, Sep. 2003, pp. 138–147.
[42] K. Fischer, E.-R. Olderog, and H. Wehrheim, "A CSP view on UML-RT structure diagrams," in *Fundamental Approaches to Software Engineering*. New York: Springer-Verlag, 2001, pp. 91–108.
[43] K. Benghazi, M. I. Capel Tuñón, J. A. Holgado Terriza, and L. E. Mendoza Morales, "A methodological approach to the formal specification of real-time systems by transformation of UML-RT design models," *Sci. Comput. Program.*, vol. 65, no. 1, pp. 41–56, Mar. 2007.
[44] R. J. Allen, "A formal approach to software architecture," Ph.D. dissertation, School Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, 1997.
[45] P. Y. H. Wong, "Towards a unified model for workflow processes," in *Proc. 1st SOSoRNet Workshop*, Manchester, U.K., Jun. 2006.
[46] P. Y. H. Wong and J. Gibbons, "A process semantics for BPMN," in *Proc. 10th Int. Conf. Formal Eng. Methods*, vol. 5256, *Lecture Notes in Computer Science*, 2008, pp. 355–374.
[47] A. W. Roscoe and Z. Wu, "Verifying statemate statecharts using CSP and FDR," in *Proc. 8th Int. Conf. Formal Eng. Methods*, vol. 4260, *Lecture Notes in Computer Science*, 2006, pp. 324–341.
[48] T. Bapty, S. Neema, J. Scott, J. Sztipanovits, and S. Asaad, "Model-integrated tools for the design of dynamically reconfigurable systems," *VLSI Des.*, vol. 10, no. 3, pp. 281–306, 2000.
[49] A. Bakshi, V. K. Prasanna, and A. Ledeczi, "MILAN: A model based integrated simulation framework for design of embedded systems," *ACM SIGPLAN Notices*, vol. 36, no. 8, pp. 82–87, Aug. 2001.
[50] A. Ledeczi, J. Davis, S. Neema, and A. Agrawal, "Modeling methodology for integrated simulation of embedded systems," *ACM Trans. Model. Comput. Simul.*, vol. 13, no. 1, pp. 82–103, Jan. 2003.
[51] J. Gray, Y. Lin, and J. Zhang, "Automating change evolution in model-driven engineering," *Computer*, vol. 39, no. 2, pp. 51–58, Feb. 2006.
[52] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003.
[53] K. M. Kavi, F. T. Sheldon, and G. Shirazi, "Reliability analysis of CSP specifications using Petri nets and Markov processes," in *Proc. 28th Hawaii Int. Conf. Syst. Sci.*, Jan. 1995, pp. 516–524.
[54] A. I. McInnes, "A formal approach to specifying and verifying spacecraft behavior," Ph.D. dissertation, Utah State Univ., Logan, UT, 2007.
[55] B. Eames, A. I. McInnes, J. E. Crace, and J. M. Graham, "A model-based design tool for systems-level spacecraft design," in *Proc. 20th AIAA/USU Conf. Small Satellites*, Logan, UT, 2006.

Allan I. McInnes (M'96) received the B.E. degree (Hons.1) in electrical and electronic engineering from the University of Canterbury, Christchurch, New Zealand, in 1997, the M.S. degree in engineering from Purdue University, West Lafayette, IN, in 2000, and the Ph.D. degree in electrical and computer engineering from Utah State University, Logan, in 2007.

He is currently a Lecturer of electrical and computer engineering with the Department of Electrical and Computer Engineering, University of Canterbury, Christchurch, New Zealand. He has been a Senior Embedded Systems Engineer with Syncroness, Inc., and a Member of Technical Staff with The Aerospace Corporation. His current research interests are formal methods, networked embedded systems, and heterogeneous system simulation.

Brandon K. Eames (M'98) received the B.S. degree in computer engineering from Utah State University, Logan, in 1999 and the M.S. and Ph.D. degrees in electrical engineering from Vanderbilt University, Nashville, TN, in 2001 and 2005, respectively.

He is currently an Adjunct Faculty in electrical and computer engineering with the Department of Electrical and Computer Engineering, Utah State University, where he has been an Assistant Professor of electrical and computer engineering. His research interests include design space exploration, embedded software, model integrated computing, computer architecture, and constraint satisfaction.

Dr. Eames is a member of the IEEE Computer Society and ACM SIGBED.

Russell Grover received the B.S. degree in computer engineering, in 2006, from Utah State University, Logan, where he is currently working toward the M.S. degree in computer engineering, specializing in embedded systems and software modeling, in the Department of Electrical and Computer Engineering.

He has been the software lead for a student-led satellite project and is currently an Embedded Software Engineer. His research interests include modeling tools for software verification.