A Software Engineering Framework for Switched Fuzzy Systems

David Harel, Assaf Marron, Amir Nissim
Weizmann Institute of Science
Rehovot, Israel
Email: firstname.lastname@weizmann.ac.il

Gera Weiss
Ben-Gurion University
Be'er Sheva, Israel
Email: geraw@cs.bgu.ac.il

***Abstract*—We propose a framework for the development of switched fuzzy systems, in which the discrete characteristics of the mode-switching logic are implemented using the paradigm of *behavioral programming*: they are coded as independent behavior threads and are interwoven at runtime. We demonstrate how such mode switching enables the simplification of fuzzy rules, and reduces their total number, as well as the number of rules evaluated in a computation cycle. The ability of the behavioral programming approach to describe independent simultaneous aspects of behavior in a modular and incremental manner, which aligns with how people often specify requirements, is shown to complement the intuitive nature of fuzzy logic. Our approach is backed by a Java package that provides an initial infrastructure for implementations.**

I. Introduction

Hybrid controllers embody control strategies that employ a combination of discrete and continuous rules in order to benefit from the advantages of both worlds (see e.g., [2, 3, 5, 10, 27, 36]). Here, we are interested in the particular case of switched fuzzy systems (see [29] for a survey), where the switching of operation mode, or sets of fuzzy rules, enables the simplification of the rules, a reduction in the total number of rules, and a reduction in the number of rules evaluated during a computation cycle.

The paper is centered around a relatively new approach to incremental and modular development of discrete systems, called *behavioral programming*, in which simultaneous aspects of system behavior are coded as independent behavior threads to be interwoven at runtime. We suggest that this approach is useful as a tool for the natural development of control systems. Specifically, fuzzy set theory [42], which uses natural language quantifications like "hot" or "dangerous" in programming [43], has been successfully applied to a wide range of control applications (see, e.g., [12] for a survey). We propose to combine the fuzzy and behavioral approaches to programming with two goals in mind, First, we wish to make it possible to program discrete multi-step scenarios using fuzzy quantification or classification for inputs and outputs. Second, we wish to simplify the development of switched fuzzy systems when the switching logic can be structured as a composition of multiple independent scenarios.

The rest of the paper is organized as follows. Section II presents the proposed integration method. Section III shows how fuzziness can enhance discrete behavioral programs, and how behavioral programming can be used to streamline the development of switched fuzzy systems. Section IV contains a brief description of a Java package that supports the proposed design pattern, and in Section V we discuss related work and conclusions.

II. A computational model for interfacing discrete-behavioral and continuous control

In this section we briefly introduce behavioral programming and describe the formal model that we propose to use in integrating, or interfacing, discrete behavioral components with continuous fuzzy ones.

A. Behavioral programming

The main goal of behavioral programming is to enable program construction by interweaving at runtime behavior specifications that were coded independently of each other, where separate requirements are captured in separate modules. First introduced through the visual scenario-based language of *live sequence charts* (LSC) [9, 17], basic concepts of behavioral programming were then shown to be language independent, and are now integrated into general purpose programming languages such as Java [20] (using the BPJ package [19]) and Erlang [39]. Additional implementations include SBT [25] and in the PiCos environment [33]. The approach is supported by the PlayGo IDE [16], which integrates the visual LSC language with behavioral programming in Java, and by a prototype model-checking tool [15], which verifies behavioral Java programs directly without first translating them into a model-checker-specific language. In [21] an architecture for scalable behavioral control application is proposed, and is illustrated by stabilizing a quad-rotor aircraft (in a MATLAB simulator) behaviorally. The main ideas behind behavioral programming, as well as several additional pieces of relevant research, are summarized in [18].

We use transition systems as a mathematical abstraction of the discrete part of the switched controller. Recall that a labeled transition system is a quadruple $\langle S, E, \rightarrow, init \rangle$, where S is a set of states, E is a set of events, $\rightarrow$ is a function from $S \times E$ to S, and $init \in S$ is the initial state. The runs of such a transition system are sequences of the form $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_i} s_i \cdots$, where $s_0 = init$, and for

all $i = 1, 2, \cdots$, $s_i \in S$, $e_i \in E$, and $\rightarrow(s_{i-1}, e_i) = s_i$ (commonly written as $s_{i-1} \xrightarrow{e_i} s_i$).

As defined in [20], each behavior thread is modeled as a transition system, in which states are associated with event sets, thus:

Definition 1 (behavior thread [20]). A *behavior thread* (abbr. *b-thread*) is a tuple $\langle S, E, \rightarrow, init, R, B\rangle$, where $\langle S, E, \rightarrow, init\rangle$ forms a deterministic total labeled transition system, $R\colon S \rightarrow 2^E$ is a function that associates each state with the set of events *requested* by the b-thread when in that state, and $B\colon S \rightarrow 2^E$ is a function that associates each state with the set of events *blocked* by the b-thread when in that state.

The set of all possible collective, interlaced runs of a set of behaviors threads is formalized as a composition operator:

Definition 2 (runs of a set of b-threads [20]). We define the *runs of a set of b-threads* $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i\rangle\}_{i=1}^n$ as the runs of the labeled transition system $\langle S, E, \rightarrow, init\rangle$, where $S = S_1 \times \cdots \times S_n$, $E = \bigcup_{i=1}^n E_i$, $init = \langle init_1, \ldots, init_n\rangle$, and $\rightarrow$ includes a transition $\langle s_1, \ldots, s_n\rangle \xrightarrow{e} \langle s'_1, \ldots, s'_n\rangle$ if and only if

$$\underbrace{e \in \bigcup_{i=1}^{n} R_i(s_i)}_{e \text{ is requested}} \quad \bigwedge \quad \underbrace{e \notin \bigcup_{i=1}^{n} B_i(s_i)}_{e \text{ is not blocked}}. \qquad (1)$$

and

$$\bigwedge_{i=1}^{n} \Big(\underbrace{(e \in E_i \implies s_i \xrightarrow{e}_i s'_i)}_{\text{affected b-threads move}} \wedge \underbrace{(e \notin E_i \implies s_i = s'_i)}_{\text{unaffected b-threads don't move}} \Big) \qquad (2)$$

Based on these definitions, when we say that a b-thread *requests*, *blocks*, or *waits for* an event, the informal intention is as follows:

Requesting an event: requesting, or proposing, that the event be considered for triggering/execution. The request may or may not be satisfied.

Blocking an event: forbidding the triggering/execution of an event.

Waiting for an event: waiting for the triggering/execution of an event. A b-thread is considered to be waiting for an event (a.k.a. watching out for it) in a given state, when there is an outgoing transition from that state labeled with the event. The triggered event may have been requested by this b-thread or by some other b-thread.

Note that these definitions are an abstraction. In practice, we propose that b-threads use the full power of the language (Java, in this case) to encode the logic succinctly and use the BPJ library (a) to indicate when the program is "in a state" of the transition system and (b) to assign requested, blocked and waited-for events to each state.

The non-determinism in Definition 2 allows for more than one run of a set of b-threads. To enable deterministic execution, we add a priority scheme, in which the b-threads and the events are linearly ordered — inducing an order on the runs too. The BPJ library executes a set of b-threads by choosing, in each state of the composite system, the first event that is requested and is not blocked. Thus, BPJ always chooses and executes the first run in the set, subject to the above induced order. More generally:

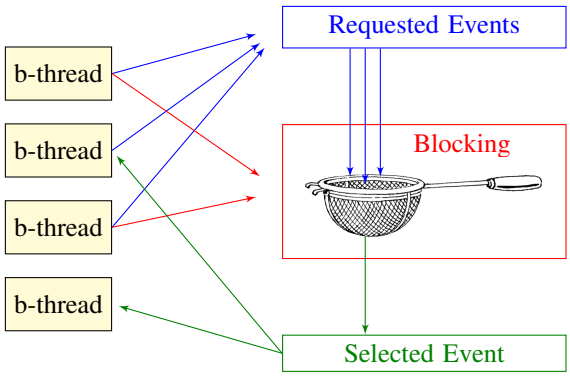


Fig. 1. Collective execution of behavior threads using an enhanced publish/-subscribe protocol: (a) all b-threads place their "bids", specifying requested events and blocked events; (b) a synchronization mechanism chooses an event that is requested but is not blocked; (c) b-threads waiting for the event are notified; (d) the notified b-threads progress to their next states, where they can place new bids.

Definition 3 (deterministic discrete behavioral execution mechanism). For a given set of b-threads let T be the transition system defined in Definition 2. A deterministic discrete behavioral execution mechanism for T is an event selection function $f\colon S \rightarrow E$ such that for each $s \in S$ there exists a transition $s \xrightarrow{f(s)} s'$ of T (where S and E are as in Definition 2).

Figure 1 illustrates such an execution mechanism. Its single run is as in Definition 2, with the added requirement that in each state the event selected is the one specified by f.

Deterministic execution mechanisms are implemented in the BPJ package for usage by Java programs, as well as in the scenario-based language of live sequence charts (LSC). In LSC, scenarios are specified visually, using multi-modal charts that depict the partial order and flow of events along *lifelines* associated with the participating objects (see [9, 17]). In the Java implementation of behavioral programming which is used in the present paper, behavioral programs are written in Java and hence, in addition to the behavioral aspects, they can implement rich logic and use external packages [19, 20].

Priority-based selection is just one implementation of deterministic event selection. One can introduce application-specific intelligence, or use various forms of look-ahead, as is done in the LSC-based techniques of *smart play-out* [14] and *planned play-out* [22]. In the water-tap example in Section III we discuss a generalization of the event selection logic, where the selection among possible candidate events is programmed using fuzziness.

As shown in [20], requesting, waiting for and blocking events, are basic programming idioms that enable the kind of behavioral modularity and incremental development that are at

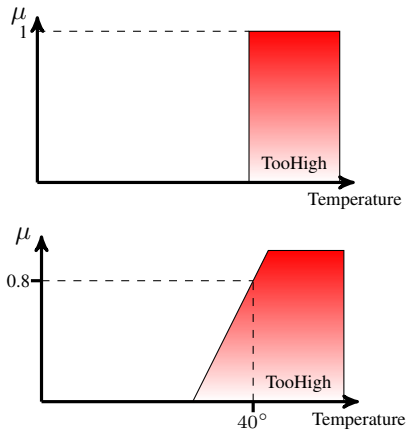


Fig. 2. High temperature as a crisp (top) and fuzzy (bottom) sets. μ denotes membership (indicator) functions.

the heart of the idea of behavioral programming (and which are present in a somewhat different fashion also in LSC). Significantly, they appear to align with the way people tend to specify system requirements.

To illustrate the usage of these idioms in programming, consider a b-thread that increases water flow in a hot water tap by requesting five times the event of turning the tap anticlockwise (`AddHot`). Another b-thread performs a similar action on the cold water tap (`AddCold`). To increase the water flow in both taps at the same time, as may be desired for keeping the temperature stable, one may activate the above b-threads alongside a third one, which forces the interleaving of events in the two scenarios. The third b-thread, for example, can be coded as "`repeatedly: {block AddCold until AddHot; block AddHot until AddCold}`". This pseudo code maps to a transition system with two states q_1, q_2 where $R(q_1) = R(q_2) = \emptyset$, $B(q_1) = \{\text{AddCold}\}, B(q_2) = \{\text{AddHot}\}$, $q_1 \xrightarrow{\text{AddHot}} q_2$, and $q_2 \xrightarrow{\text{AddCold}} q_1$.

B. Fuzzy control

Fuzzy set theory and linguistic variables are mathematical formalisms for dealing with the imprecision and vagueness of natural language notions often used in human reasoning.

A fuzzy set [42] $A \subseteq X$ is defined by a membership function $\mu_A : X \to [0, 1]$ which assigns each element a *degree of membership* in the set. Figure 2 shows the definition of 'too high' temperature as a classic and fuzzy set.

Using fuzzy sets, whose membership criteria is *gradual*, linguistic notions can be defined more naturally than with crisp (non-fuzzy) sets.

A linguistic variable [43] is a variable whose values are fuzzy sets. Figure 3 shows an example of 'temperature' defined as a linguistic variable whose values may vary between *too low*, *pleasant*, and *too high*. The classification of a given crisp actual temperature (e.g. 25 degrees) as, say, *too low* is called fuzzification, and is based on the membership values of the crisp temperature in the three fuzzy sets. Conversely, a fuzzy value can be defuzzified into an actual crisp number.

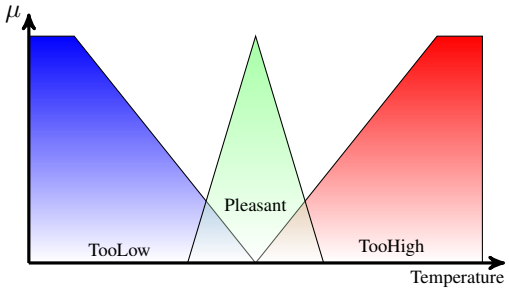


Fig. 3. Temperature as a linguistic variable with *TooLow*, *TooHigh* and *Pleasant* as values defined by fuzzy sets.

Linguistic variables are used to formulate fuzzy control rules, such as "if the temperature is too low then add hot water".

Fuzzy control utilizes a *fuzzy inference system* (FIS) leveraging control rules specified in natural terms. Crisp input signals (e.g. actual temperature in degrees) can be fuzzified (e.g. interpreted as *too hot*) and evaluated by each rule; the generated outputs are then aggregated and the result is defuzzified, yielding a crisp output.

C. Interfacing behavioral programming and continuous control

Underlying our work is the issue of using behavioral programming in hybrid designs that include a discrete behavioral component. Following the lines of other work on switched fuzzy systems, our approach is based on attaching such a discrete system to a control system, where continuous signals u and y have their usual roles as in classical control, with an added input signal σ that switches the mode of the controller, and an output signal ω that provides discrete classification of the continuous signals.

Definition 4 (switchable control mechanism). A switchable control mechanism is modeled by equations of the form:

$$\begin{cases} u(t) = g(\sigma\big([0,t)\big), y\big([0,t)\big), t) \\ \omega(t) = h(\sigma\big([0,t)\big), y\big([0,t)\big), t) \end{cases}$$

where $y\colon [0,\infty) \to \mathbb{R}^m$ is the input signal to the controller, $u\colon [0,\infty) \to \mathbb{R}^l$ is the output signal that the controller generates, $\sigma\colon [0,\infty) \to \Sigma$ is the switching signal fed to the controller from the discrete logic, and $\omega\colon [0,\infty) \to \Omega$ is a classification that the continuous controller feeds back to the discrete logic. The notations $y\big([0,t)\big)$ and $\sigma\big([0,t)\big)$ denote the history of the signals from time 0 to t (not including time t). The symbols Σ and Ω denote finite sets from which the discrete inputs and outputs are selected, respectively; the numbers $l, m \in \mathbb{N}$ are the dimensions of the respective continuous input and output signals; and g, h are functions that map histories of the input signals to, respectively, the continuous and discrete output signals.

We propose that the signal σ be provided by the discrete behavioral-programming component, which, in turn, uses the signal ω as an input:

Definition 5 (behavioral control interface). For a discrete behavioral execution mechanism and a switchable control mechanism, a behavioral control interface is a pair of functions: $f_{d2c} : E \rightarrow \Sigma$, which maps events of the behavioral system to discrete (switching) inputs of the control mechanism, and a $f_{c2d} : \Omega \rightarrow E$, which maps changes in the discrete outputs of the control mechanism (signal classifications) to events of the behavioral system.

Specifically, we propose to combine a behavioral program with a fuzzy controller, in a way depicted structurally in Figure 4: the controller reads the continuous inputs and generates the continuous outputs; some of the events of the behavioral program component drive the switching in the continuous component by changing the fuzzy inference rules and the input/output filters (fed as the discrete input σ); some of the linguistic variables generated by the fuzzy inference system serve as the signal ω, which is translated into events that are fed into the behavioral component.

III. Examples

A. Example 1: Adding fuzzy control to behavioral programs

Before examining behaviorally controlled switching, we first focus on enhancing behavioral control systems with fuzzy semantics.

Consider a control system for adjusting the water temperature and flow of a water tap, which mixes the flow from two taps — one for hot water and one for cold. Opening or closing one of the taps affects the flow and the temperature: For example, turning the hot water tap further in the open direction, when both taps are already open, increases the flow as well as the temperature. The purpose of the application is to adjust water temperature and flow by turning hot and cold water taps until the temperature and flow are sufficiently "pleasant". The program consists of the following b-threads (shown in pseudo code):

AdjustHighTemp When the temperature is `tooHigh`, request the events `AddCold` and `ReduceHot`, and block the events `ReduceCold` and `AddHot`.

AdjustLowTemp When the temperature is `tooLow`, request the events `ReduceCold` and `AddHot`, and block the events `AddCold` and `ReduceHot`.

AdjustHighFlow When the flow is `tooHigh`, request the events `ReduceCold` and `ReduceHot`, and block the events `AddCold` and `AddHot`.

AdjustLowFlow: When the flow is `tooLow`, request the events `AddCold` and `AddHot`, and block the events `ReduceCold` and `ReduceHot`.

IndicateWhenPleasant: When both the flow and the temperature are `pleasant`, show indicator.

These b-threads are interwoven and run simultaneously, watching for changes in fuzzy linguistic variables. Their coordination (as described in Definitions 2 and 3), results in a cohesive integrated system behavior. For example, if the water is too hot and the flow is too low, `AdjustLowFlow` will take care of increasing the flow, while `AdjustHighTemp` will independently take care of reducing the temperature. The behavioral programming event selection mechanism will then select only the `AddCold` event, which is the only event that is requested and not blocked.

This basic example shows only short b-threads. In general, b-threads can leverage the full power of the language — in our case here, Java — to go through several phases, in each of which they request and block different sets of events.

We now provide more details. Our example starts with a non-fuzzy implementation of the specification using the BPJ package and the following events (and b-threads):

External events, such as `FlowTooHigh` or `TempTooLow`, are driven by a b-thread that repeatedly checks the flow and temperature and generates the events based on their current values and crisp numeric thresholds (e.g., the external event `TempTooHigh` is generated only when the current temperature exceeds 40 degrees).

Tap events, such as `AddHot` or `ReduceCold`, change the flow by fixed amounts (e.g., closing a tap always results in turning it five percent of its capacity). These events are requested by "user" b-threads that wait for and react to external events. An environment b-thread waits for these tap events and actuates the taps (or a simulator thereof). Additional b-threads simulate the maximal and minimal limits of the tap positions.

System Events, such as `Update` and `Stop`, are used for controlling the overall execution.

In way of adding fuzziness to the above program, the first step is to associate natural language terms, such as *Too High* or *Pleasant*, with linguistic variables (instead of crisp values per membership functions as in Figure 3).

The membership values of these linguistic terms may now be used by the environment b-thread for generating external events, instead of by using numeric thresholds. In our case, the system chooses the linguistic term with the highest membership value as the event to be requested; i.e., when the current temperature is more "too high" than it is "pleasant" or "too low", the event of `TempTooHigh` will be requested. We call events generated in this way *linguistic events*.

The b-threads can also access the linguistic variables directly for arbitrary processing. Thus, for example, the value of a variable such as the `flowQuality` or `tempQuality` is the linguistic term with the highest membership value, and a termination condition is coded using natural language terms and the fuzzy-based assessment directly

```
do{ ... }
  until (flowQuality == Pleasant
      && tempQuality == Pleasant);
```

In addition to the above, b-threads can obtain from linguistic events the associated crisp values of the measured quantity and its μ membership value, and use these to select among alternative actions or to perform fuzzy inference in the behavioral program. For example, the μ value can be used in decisions (e.g., when the temperature "is more pleasant than it is too

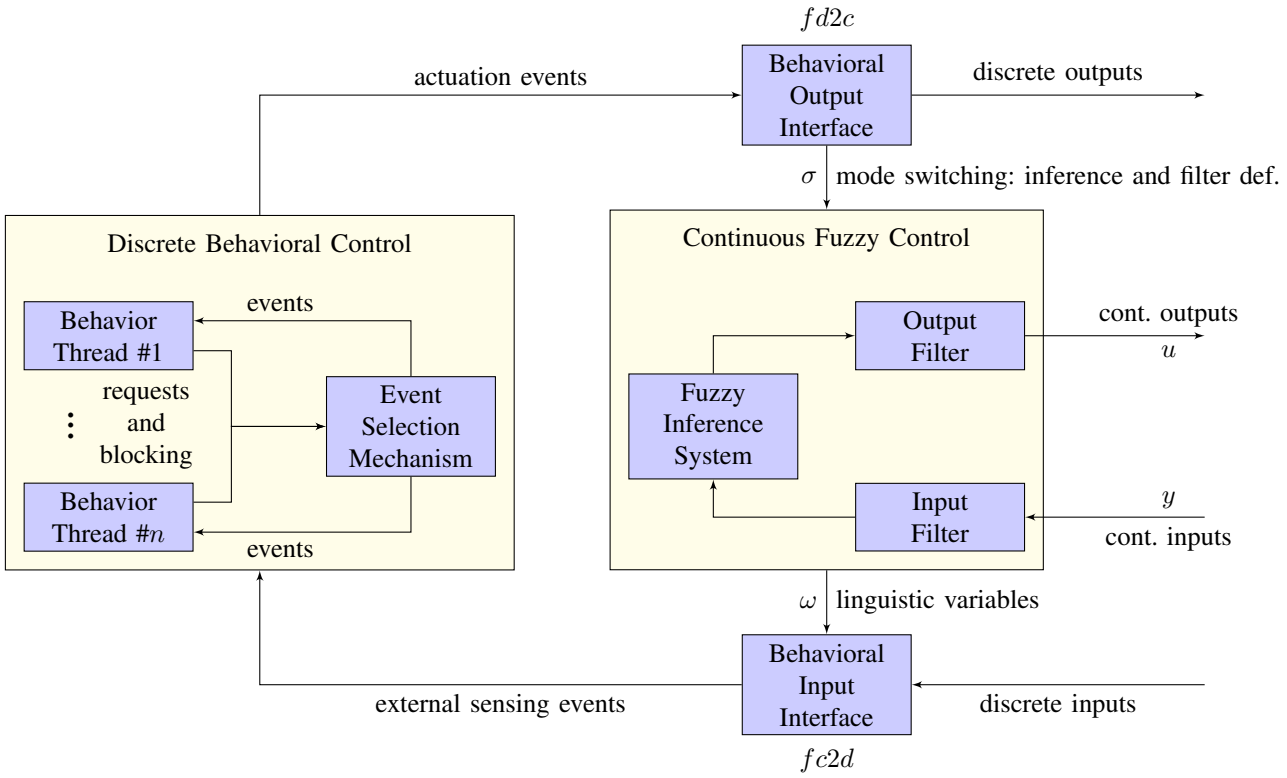


Fig. 4. An architecture for a behaviorally-controlled switched fuzzy system. Behavior threads issue event-request and event-block commands to an event selection mechanism that publishes allowed events. A behavioral output interface watches out for actuation events directed at the controlled system, and either switches the mode of a fuzzy controller or actuates devices directly. Inputs from the controlled system are processed directly as crisp signals, or through the fuzzy control system, or as linguistic variables that are used for generating external events to which behavior threads can react.

low"), or to refine the outputs of the behavioral program by associating the tap actuation events with an amount that relies on fuzzy computations to specify exactly how much a tap should be opened or closed.

B. Example 2: Enhancing fuzzy rules with behavioral control

We now focus on the mode-switching section of the model. We show that one can use behavioral programming to simplify the programming of switched fuzzy controllers, aligning the process better with how people often think about behavior, when the control of a physical process can be broken into distinct operation modes. Our exposition is carried out by modifying a benchmark example, that of a steam generation plant, which is part of a publicly available demo of the *fuzzyTECH* fuzzy control software package by INFORM Corporation [23].

In this example, a tank is filled with water, and a circulation pump is started in order to fill the water pipes, following which the fire is started. As the water pipes get hot and steam is generated, pressure builds up. When the pressure is sufficient the steam turbines are started.

The fuzzy rules of the original system in the *fuzzyTECH* solution are grouped into three blocks:

`RB3` (34 rules): controls the water level in the tank. For example,

```
1. IF Status IS TankFill AND WaterLevel IS
    AboveNormal
   THEN DrainValve IS Closed, FeedValve IS
      Closed
...
20. IF Status IS BuildPressure AND WaterLevel
    IS Low
   THEN DrainValve IS Medium, FeedValve IS
      MinFlow
...
```

`RB4` (10 rules): controls operation mode transitions. For example,

```
...
 4. IF FireOn IS False AND PumpOn IS False
   AND WaterLevel IS CritFull
   THEN Status IS StartPump
...
```

`RB5` (6 rules): controls pump and fire actuation. For example,

```
...
 2. IF Status IS StartPump
   THEN StartPump IS True AND StartFire IS
      False
...
```

The variables of the *fuzzyTECH* system are divided into two groups: *input / output* variables and variables for *operation-mode control*.

The input variables are linguistic variables describing physical properties (e.g., `WaterLevel` is `BelowNormal`), and/or binary input signals indicating the actuation of devices (such as `PumpOn`). Similarly, the output variables are linguistic variables describing quantitative parameters for device actuation (e.g., `FeedValve` should be `OpenMedium`) and/or binary signals driving ON/OFF signals for device actuation (such as `StartPump`).

In contrast, `Status`, although highly visible in the rules, is an internal variable. It is a *discrete* variable whose values determine the operation mode (abbr. opmode) of the system. It may be associated with specific stages or steps in the process; e.g., `TankFill`, `StartPump`, `BoilerFill`, or `Operating`.[1] As stated in the *fuzzyTECH* user manual [23]:

> *The linguistic variable* `Status` [...] *is an intermediate linguistic variable that does not have membership functions associated with its terms. The variable Status is only used within the fuzzy logic system as an input to the other two fuzzy rule blocks.*

Here are the main issues that we focus on when introducing behavioral programming into the design of this control system:

Intuitive modularity: Even though the expected changes in opmode do follow some form of scenario, such behavior is not always directly visible from the rules. Moreover, using behavioral capabilities like blocking events allows the description of a complex scenario to be decomposed into multiple relatively independent behaviors as perceived by humans.

Efficiency: All three rule blocks are active while this fuzzy system is active, and all are evaluated with every computation cycle, even though in each process-phase only a subset of the rules can be triggered.

Complexity: Although `Status` is a simple discrete variable, its maintenance requires ten rules. These rules are recomputed with every system computation step, regardless of whether an event that may cause an operation-mode change has occurred. Additionally, the appearance of `Status` as an input variable in all of rules renders the entire application more complex.

So, how can behavioral programming techniques be applied in the programming of switched fuzzy control to enhance the rules-only solution?

Our approach is based on using behavioral events for marking changes in the linguistic variables that describe physical properties, and for indicating changes in the operation mode. The mode changes are carried out by translating them into changes in the definition of the fuzzy system; and these, in turn, are carried out by loading new rules, changing input/output definitions, or even switching to a totally different control system.

We introduce and then use a package called BFuz, which provides an interface for loading a fuzzy system from a specification file, changing its active rule blocks and input/output definitions, and generating linguistic events that reflect monitored physical properties (see Section IV).

a) Creating behavioral linguistic events: The connection between the physical properties of the continuous part of the system and the event-based discrete behavioral part is established by creating events that correspond to changes in the *linguistic values* of the variables in the fuzzy system. As the relevant physical properties of the system are monitored, these events, which we term *[behavioral] linguistic events*, are generated as follows.

A separate event class is associated with each linguistic term (e.g., *Water Level Below Normal*) of each linguistic variable (in this case, *Water Level*). An ongoing process repeatedly obtains measurements of the set of relevant physical properties of the system, evaluates the membership functions for the linguistic variables, and generates linguistic events accordingly.[2]

b) Creating behavioral opmode events: The changes in operation mode are managed as follows. Variables that participate in the fuzzy rules but reflect operation modes and not physical properties, are replaced by events called *[behavioral] operation-mode events* (abbr. *opmode events*). In our example, the `Status` variable is implemented using opmode events. A subclass of this event class is created for each possible mode; i.e., for each possible value of the variable. Then, the rules that change the values of the operation mode variables are replaced with b-threads that watch out for a corresponding behavioral linguistic event and request a corresponding opmode event. In this example, when the linguistic event `WaterLevelCritFull` occurs, the opmode event `StartPump` is requested. The resulting b-thread that manages the opmode transitions is depicted in Figure 5.

When the state of the system is actually composed of the states of multiple opmode variables, the developer may choose either to designate a separate explicit opmode value for each combination or to handle the combinations implicitly in the logic of the b-threads that track the opmode variables.

c) Reorganizing the rule blocks: The rules of the fuzzy system that depend on opmode variables such as `Status` are reorganized in blocks, where all rules in a given block are required to have the same value of the opmode variable. Thus, for each opmode there is a rule block, all of whose rules are relevant to it. In our implementation, the 34 rules that resided in one rule block (`RB3`; see Figure 6) were divided into six (smaller) rule-blocks. For example, the rule

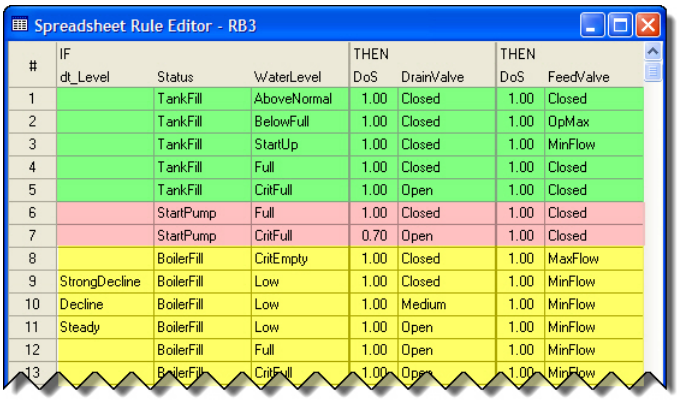
```
IF Status IS TankFill AND WaterLevel IS
    AboveNormal
THEN DrainValve IS Closed, FeedValve IS Closed
```

[1] Although some of the mode names in this example may sound like commands, they actually describe the mode associated with executing the command. For example, when the `Status` variable is assigned the status/opmode value `StartPump`, the rules issue a command by setting the output variable `StartPump` to `True`.

[2] In the current BPJ implementation, the external linguistic events are not different from the internal events. They are generated with the assumption that no b-thread blocks such events. In the future, we might enhance BPJ to allow events that represent external events, and which cannot be blocked.

```
...
// TankFill is the initial operation mode
Request(InitialOpmode);

// move to the next opmode
WaitFor(WaterLevelFull or WaterLevelCritFull);
Request(StartPump);

WaitFor(PumpOn);
Request(BoilerFill);

WaitFor(WaterLevelFireStart);
Request(StartFire);

WaitFor(FireOn);
Request(BuildPressure);

WaitFor(PressureOperating);
Request(Operating);
...
```

Fig. 5. Controlling operation mode changes: pseudo-code of the main behavior thread.



#	IF			THEN		THEN	
	dt_Level	Status	WaterLevel	DoS	DrainValve	DoS	FeedValve
1		TankFill	AboveNormal	1.00	Closed	1.00	Closed
2		TankFill	BelowFull	1.00	Closed	1.00	OpMax
3		TankFill	StartUp	1.00	Closed	1.00	MinFlow
4		TankFill	Full	1.00	Closed	1.00	Closed
5		TankFill	CritFull	1.00	Open	1.00	Closed
6		StartPump	Full	1.00	Closed	1.00	Closed
7		StartPump	CritFull	0.70	Open	1.00	Closed
8		BoilerFill	CritEmpty	1.00	Closed	1.00	MaxFlow
9	StrongDecline	BoilerFill	Low	1.00	Closed	1.00	MinFlow
10	Decline	BoilerFill	Low	1.00	Medium	1.00	MinFlow
11	Steady	BoilerFill	Low	1.00	Open	1.00	MinFlow
12		BoilerFill	Full	1.00	Open	1.00	MinFlow
13		BoilerFill	CritFull	1.00	Open	1.00	MinFlow

Fig. 6. Rule Block RB3

resides only in the block for the opmode `TankFill`. Changes in operation modes are indicated by opmode events that can be watched out for by b-threads, and which can be optionally handled by (a) *loading* new rule blocks into the running fuzzy system, and/or (b) *executing* crisp device actuation.

d) Rule simplification: In the opmode-specific rule blocks, rules may be simplified. Depending on the use of the opmode variable, once its value is known the condition may be eliminated. For example, the aforementioned rule related to an above-normal water level can be replaced by the simpler one:

```
IF WaterLevel IS AboveNormal
THEN DrainValve IS Closed, FeedValve IS Closed
```

e) Rule elimination: Rules that use only discrete input and output variables can be completely replaced by appropriate behavioral program parts. For example, the rules of `RB5` that initiate actuation of the circulation pump and the fire, based on the non-fuzzy `Status` variable, can be replaced by a b-thread that waits for the system to enter the `StartPump` (resp., `StartFire`) opmode and after a short delay requests the corresponding `PumpOn` (resp., `FireOn`) event. When the behavioral event occurs, the b-thread performs device-specific actuation, validates the results and reports them. Thus the rule

```
IF Status IS StartFire
THEN StartFire IS True, StartPump IS True
```

can be eliminated. In our solution, the entire rule block `RB5` is eliminated in this manner and is replaced by a single actuation b-thread:

```
WaitFor(StartPump);
startPump();
Request(PumpOn);

WaitFor(StartFire);
startFire();
Request(FireOn);
```

Completing the process described in the previous subsections results in a system whose architecture fits the structure shown in Figure 4. The following basic advantages in efficiency and maintainability are derived from our use of scenarios and behavioral event-driven control.

First, rule blocks `RB4` and `RB5` are eliminated, so that their rules are not evaluated with every system computation cycle. Second, rule block `RB3` is split into six smaller blocks, thus reducing the number of fuzzy control rules that are evaluated with every cycle and simplifying them. The following table summarizes some of the transformation results:

	fuzzyTECH	BFuz
Fuzzy rules	50	34
Linguistic variables	10	5
Rule blocks	3	6
Modes	1	6
Min. rules per cycle	50	2
Max. rules per cycle	50	8
Max. rules in a block	34	8
Max. variables in `if` clause (excluding eliminated blocks)	3	2
Mode-specific rule blocks	1	6
Opmode switching rule blocks	1	0
Direct actuation rule blocks	1	0

Introducing behavioral programming principles into "classical" control systems has other benefits as well. First, as shown in the steam generation example, mode transitions are coded in a scenario (see Figure 5), in a way that expresses the flow of time explicitly rather than implicitly. Second, the full power of the programming language can be leveraged in making decisions about mode transitions. This allows for more intricate computations involving membership and crisp values, reliance on external data and conditions, and possibly also efficiency gains by reducing the frequency with which the

opmode transition is considered.[3]

Finally, behavioral programming allows blocking undesired behaviors without the need for direct communication with the software components that may request them. For example, it is easy to add to our example a safety condition through a b-thread that blocks the actuation of the fire when the water pump is not on. Such a new scenario does not need to modify existing b-threads or rules and does not have to change when new b-threads or rules that drive fire actuation are added. This can also be achieved by a small modification of the existing actuation b-thread mentioned above, by replacing `WaitFor( StartPump)` with:

```
WaitFor(StartPump) and Block(StartFire).
```

IV. BFuz: A fuzzy package for Java

The BFuz package that we constructed to support the approach described in this paper, allows one to develop behavioral-fuzzy systems in Java via BPJ. The source code for the implementation, as well as a recorded demo of an execution, are available online [19].

The BFuz package provides an interface that allows the programmer to connect to and control the continuous rule-based fuzzy component of the model from the event-based component, and can be viewed as an extension of a fuzzy inference system (FIS). We assume that a FIS specification is given as a *fuzzy control programming* file (`.fcl`), which is a standard published by the International Electrotechnical Commission (IEC), and that the fuzzy components of the system are developed with `jFuzzyLogic` [30], an open source fuzzy logic package for Java.

The main purpose of the interface is to facilitate translation of crisp sensor values into behavioral events and vice versa, and ongoing replacement of active rule blocks that autonomously control a fuzzy system.

V. Related work and discussion

Using multiple controllers and programming a computer to switch between them is common practice in control engineering. One example involves systems with selectors, which have been used for constraint control [6]. Systems with gain scheduling constitute another example [26]. Some examples of hybrid systems for control are described in [8] and [7], where many different controllers are used and their coordination is dealt with by a specially constructed language. Other proposals, such as the controller discussed in [4], use a hierarchical structure, where a collection of controllers is driven by an expert system. Malmborg's thesis [28] contains additional references to related hybrid models.

Frameworks that integrate fuzzy logic [24] and discrete event controllers are described in [31, 40]. The idea of mixing, or switching between, control signals for hybrid systems that are represented as local models with local controllers is very much in line with what is done in the research on fuzzy control. Some hybrid control schemes can be viewed as fuzzy controllers [34]. In [1], Altamiranda et al propose to use a fuzzy inference system to generate events for a discrete event controller. In, e.g., [11, 32, 44, 45], logic is used to switch between fuzzy logic and a PID controller. Takagi-Sugeno fuzzy systems [35] are described by fuzzy IF-THEN rules that locally represent linear input-output relations; i.e., they model a fuzzy inference system that switches the modes of a linear controller. Similarly, Ferreira and Krogh [13] apply neural networks to switching control strategies. A survey of switched fuzzy systems is presented in [29].

Other related work is the research on fuzzy state machines [37, 38, 41]. Motivated by that work, we examined the possibility of allowing scenarios whose state is fuzzy. We chose, however, not to allow this as a programming construct at this point, because such constructs may be too hard to use and comprehend. Still, we consider this issue open for future research.

Another possible direction for future research involves enhancing the existing modalities with fuzziness. The specification of what must, may or must not occur can also be associated with fuzzy natural language terms indicating the strength of the behavioral command. For example, "block event e_1 and try to avoid event e_2". The event selection mechanism can then consider these fuzzy values for making, or optimizing, its decision.

Another interesting research direction is to extend the model-checking capabilities of the BPmc tool [15] to behaviorally-controlled switched fuzzy systems.

In summary, we have shown that combining fuzziness and behavioral programming provides synergy that can make the development both of discrete behavioral controllers and of switched fuzzy systems more natural and intuitive.

Acknowledgment

We would like to thank Moshe Vardi for his valuable suggestions, and Yossi Weiss for reviewing earlier versions and providing fresh insights. The research of the first three authors was supported in part by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant to DH from the European Research Council (ERC) under the European Community's FP7 Programme. The research of the fourth author was supported in part by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University and a research grant (IRG) under the European Community's FP7 Programme.

References

[1] E. Altamiranda, H. Torres, E. Colina, and E. Chaćon. Supervisory control design based on hybrid systems and fuzzy events detection. application to an oxichlorination reactor. *ISA Trans*, 41(4):485–99, 2002.

[3] In a different behavioral implementation we could do without the b-thread in Figure 5. The opmode transition rules in `RB4` could possibly be replaced by mode-exit rules placed inside the opmode specific rule blocks. However, the result would still require an internal `status` variable and would not express the flow of time explicitly.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[3] P. Antsaklis and X. Koutsoukos. On hybrid control of complex systems: A survey. *ISIS*, 97:017, 1997.

[4] P. Antsaklis, J. Stiver, and M. Lemmon. Hybrid system modeling and autonomous control systems. *Hybrid Systems*, pages 366–392, 1993.

[5] Z. Artstein. Examples of stabilization with hybrid feedback. *Hybrid Systems III*, pages 173–185, 1996.

[6] K. Åström and T. Hägglund. PID controllers: theory, design, and tuning. *Instrument Society of America*, 67, 1995.

[7] R. Brockett. Hybrid Models for Motion Control Systems. *Essays on Control: Perspectives in the Theory and its Applications*, page 29, 1993.

[8] R. Brooks. Elephants don't play chess. *Robotics and autonomous systems*, 6(1-2):3–15, 1990.

[9] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1), 2001.

[10] M. Egerstedt. Behavior based robotics using hybrid automata. *Hybrid Systems: Computation and Control*, pages 103–116, 2000.

[11] I. Erenoglu, I. Eksin, E. Yesil, and M. Guzelkaya. An intelligent hybrid fuzzy PID controller. *Proceeding of the 20th European Conference on Modeling and Simulation*, pages 1–5, 2006.

[12] G. Feng. A survey on analysis and design of model-based fuzzy control systems. *IEEE Trans. on Fuzzy Systems*, 14(5):676–697, 2006.

[13] E. Ferreira and B. Krogh. Switching controllers based on neural network estimates of stability regions and controller performance. *Hybrid Systems: Computation and Control*, pages 126–142, 1998.

[14] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *FMCAD*, pages 378–398. Springer, 2002.

[15] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-checking behavioral programs. In *EMSOFT*, 2011.

[16] D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: towards a comprehensive tool for scenario based programming. In *ASE*, 2010.

[17] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[18] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Communications of the ACM*. To appear.

[19] D. Harel, A. Marron, and G. Weiss. The BPJ Library. www.cs.bgu.ac.il/~geraw.

[20] D. Harel, A. Marron, and G. Weiss. Programming coordinated scenarios in Java. In *24th European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[21] D. Harel, A. Marron, G. Weiss, and G. Wiener. Behavioral programming, decentralized control, and multiple time scales. *AGERE ("Agents and Actors Reloaded")*, 2011.

[22] D. Harel and I. Segall. Planned and traversable play-out: A flexible method for executing scenario-based programs. *TACAS*, pages 485–499, 2007.

[23] INFORM Gmbh. *fuzzyTECH* Software Package www.inform-ac.com/fuzzytech.htm.

[24] G. Klir and B. Yuan. *Fuzzy sets and fuzzy logic: theory and applications*. Prentice Hall, 1995.

[25] H. Kugler, C. Plock, and A. Roberts. Synthesizing biological theories. In *CAV*, 2011.

[26] D. Leith and W. Leithead. Survey of gain-scheduling analysis and design. *International Journal of Control*, 73(11):1001–1025, 2000.

[27] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-time, theory in practice: REX Workshop, Mook, the Netherlands, June 3-7, 1991: proceedings*, page 447. Springer, 1992.

[28] J. Malmborg. *Analysis and Design of Hybrid Control Systems*. PhD thesis, Lund Institute of Technology, Sweden, 1998.

[29] V. Ojleska and G. Stojanovski. Switched fuzzy systems: Overview and perspectives. In *9th International PhD Workshop on Systems and Control: Young Generation Viewpoint, Slovenia*, volume 1, 2008.

[30] P.Cingolani. jFuzzyLogic Open Source Project http://jfuzzylogic.sourceforge.net.

[31] P. Ramadge and W. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1):81–98, 1989.

[32] M. Rashid and A. Wali. Fuzzy-PID hybrid controller for point-to-point (PTP) positioning system. *American Journal of Scientific Research*, 9:72–80, 2010.

[33] B. Shimony, I. Nikolaidis, P. Gburzynski, and E. Stroulia. On coordination tools in the PicOS tuples system. *SESENA*, 2011.

[34] M. Sugeno and T. Takagi. Multi-dimensional fuzzy reasoning. *Fuzzy Sets and Systems*, 9(1-3):313–325, 1983.

[35] K. Tanaka, T. Ikeda, and H. Wang. Fuzzy regulators and fuzzy observers: relaxed stability conditions and LMI-based designs. *IEEE Trans. on Fuzzy Systems*, 6(2):250–265, 1998.

[36] C. Tomlin, J. Lygeros, and S. Shankar Sastry. A game theoretic approach to controller design for hybrid systems. *Proceedings of the IEEE*, 88(7):949 –970, jul. 2000.

[37] H. Wang and D. Qiu. Computing with words via turing machines: a formal approach. *IEEE Trans. on Fuzzy Systems*, 11(6), 2003.

[38] W. Wee and K. Fu. A formulation of fuzzy automata and its application as a model of learning systems. *IEEE Trans. on Systems Science and Cybernetics*, 5(3):215–223, 1969.

[39] G. Wiener, G. Weiss, and A. Marron. Coordinating and visualizing independent behaviors in Erlang. In *9th ACM SIGPLAN Erlang Workshop*, 2010.

[40] W. Wonham. On the control of discrete-event systems. *Three decades of mathematical system theory*, pages 542–562. Springer, 1989.

[41] M. Ying. A formal model of computing with words. *IEEE Trans. on Fuzzy Systems*, 10(5):640–652, 2002.

[42] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

[43] L. A. Zadeh. Outline of new approach to the analysis of complex systems and decision processes. *IEEE Trans. Systems, Man, and Cybernet.*, SMC-3:28–44, 1973.

[44] M. Zerikat and S. Chekroun. Design and implementation of a hybrid fuzzy controller for a high-performance induction motor. *World Academy of Science, Engineering and Technology*, 26:263–269, 2007.

[45] Y. Zhang. Fuzzy-PID hybrid control for temperature of melted aluminum in atomization furnace. In *ISDA '06*, volume 1, pages 332 –335, Oct. 2006.