Reactive Chatbot Programming

Guillaume Baudart
IBM Research
USA

Martin Hirzel
IBM Research
USA

Louis Mandel
IBM Research
USA

Avraham Shinnar
IBM Research
USA

Jérôme Siméon
Clause Inc.
USA

Abstract

Chatbots are reactive applications with a conversational interface. They are usually implemented as compositions of client-side components and cloud-hosted services, including artificial-intelligence technology. Unfortunately, programming such reactive multi-tier applications with traditional programming languages is cumbersome. This paper introduces `wcs-ocaml`, a new multi-tier chatbot generator library designed for use with the reactive language ReactiveML. The paper explains our library with small didactic examples throughout, and closes with a larger case-study of a chatbot for authoring event-processing rules.

CCS Concepts • **Software and its engineering** → *Concurrent programming languages*; • **Human-centered computing** → *Natural language interfaces*;

Keywords Chatbot, reactive programming, synchronous programming

ACM Reference Format:
Guillaume Baudart, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2018. Reactive Chatbot Programming. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '18), November 4, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281278.3281282

1 Introduction

A *chatbot* is a virtual conversational agent that communicates with users through natural-language dialog. Companies often find it beneficial to offer services to their customers or employees via chatbots, because they support diverse delivery channels (web pages, phone, messaging systems) and versatile applications (question answering, form filling).

REBLS '18, November 4, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6070-8/18/11…$15.00
https://doi.org/10.1145/3281278.3281282

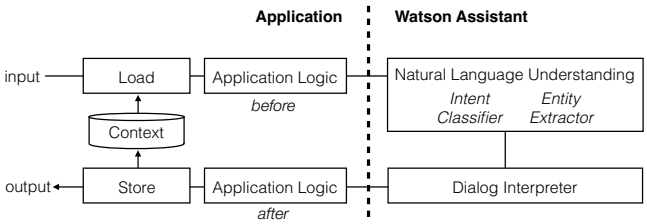


Figure 1. Chatbot Architecture.

Unfortunately, implementing good conversational agents requires a lot of programming effort, because they are typically multi-tier, repetitive, reactive, and multi-component.

Chatbots often use cloud-hosted runtimes to make them easier to scale and operate [20]. Figure 1 shows a typical architecture. The cloud tier uses a stateless runtime (here Watson Assistant [12]) for natural-language understanding and dialog interpretation. The application tier maintains the conversation state, interfaces with the delivery channel, and implements additional logic, such as calling a weather service or querying a database. Unfortunately, this *multi-tier* architecture often requires chatbot designers to keep separate programs written in multiple domain specific languages in-sync by hand. To ease this process, we propose `wcs-ocaml`, a library to program multi-tier chatbots in a single programming language. Our library generates the code for the runtime tier; this also provides an opportunity to generate *repetitive* code as needed.

The appeal of conversational agents is their ability to offer the services of a company to users via diverse delivery channels. That makes them naturally *reactive*: chatbots are stateful applications that interact asynchronously with external services and external delivery channels. Using a reactive language simplifies chatbot development; this paper demonstrates that with ReactiveML [17].

Like with any other software, the code base and the development team for a chatbot can grow large, so modularity is paramount. Chatbot composition often involves features such as dispatch among composable skills, dialog preemption or digression, and mixed-initiative dialog. It is not trivial to find suitable abstractions for these without a reactive language. This paper showcases how to use features of a reactive language for modular chatbot composition.

This paper makes the following contributions:

- `wcs-ocaml`, a strongly typed OCaml library for generating multi-tier chatbots (Section 2).
- Leveraging a general-purpose reactive programming language for chatbots orchestration (Section 3).
- RuleBot, a case-study using chatbot generation and reactive orchestration, that is itself used to author event processing rules (Section 4).

Overall, this paper suggests that chatbots are a fitting application of and for reactive and event-based technologies.

2 Generating Multi-tier Chatbots

This section shows how to generate a multi-tier chatbot following an approach inspired by multi-tier programming of web applications [6, 22]. Developers can thus program both Watson Assistant and the application in a single language using `wcs-ocaml`.[1] As a running example, we generate *KnockKnockBot*, a simple chatbot that tells the following joke:

1 *Bot: Knock Knock*
2 *User: Who's there?*
3 *Bot: Broken Pencil*
4 *User: Broken Pencil who?*
5 *Bot: Never mind it's pointless...*

2.1 Programming Watson Assistant

In Watson Assistant, a program is called a *workspace* and is represented by a JSON object containing definitions of *intents*, *entities*, and the *dialog* (the right side of Figure 1).

Intents. Intents are the desired user actions. For instance, the user may ask who is knocking at the door. In Watson Assistant, intents are defined by a label and a set of examples that are used to train the intent classifier.

```
let who_intent =
  Wcs.intent "Who"
    ~examples: [ "Who's there?";
                 "Who is there?";
                 "Who are you?"; ] ()
```

The intent classifier is a machine learning model that is able to recognize similar utterances [26]. For instance the utterance: *Who's that stumbling around in the dark?* will be successfully classified as a `who_intent`.

Entities. Entities are the objects the user is referring to. In our example, we define an entity for the name of the main character *Broken Pencil*. Entities are defined by a name and a list of synonyms (or, a regular expression).

```
let char_entity =
  Wcs.entity "Characters"
    ~values: [ "Broken Pencil",
               ["Damaged Pen"; "Fractured Pencil"] ] ()
```

[1] Available at https://ibm.github.io/wcs-ocaml/.
WCS refers to the former name Watson Conversation Service.

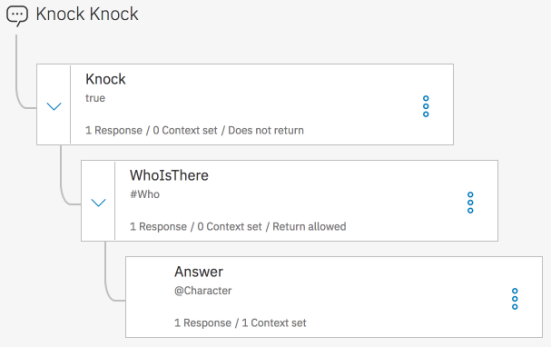


Figure 2. Watson Assistant User Interface.

The entity extractor detects an entity if one of the synonyms appear in the utterance (or if the regular expression matches).

Dialog. The dialog is specified by a finite state machine that reacts to the intents and entities or context information set by the application. For instance, the control logic of our joke is a three-states automaton: 1) `knock` starts the joke, 2) `whoisthere` answers the question with the name of the main character, 3) `answer` concludes with the punchline.

```
let knockknock who_intent char_entity answer =
  let knock =
    Wcs.dialog_node ("Knock")
      ~conditions_spel: (Spel.bool true)
      ~text: "Knock knock" () in
  let whoisthere =
    Wcs.dialog_node ("WhoIsThere")
      ~conditions_spel: (Spel.intent who_intent)
      ~text: (entity_value char_entity)
      ~parent: knock () in
  let answer =
    Wcs.dialog_node ("Answer")
      ~conditions_spel: (Spel.entity char_entity ())
      ~text: answer
      ~parent: whoisthere
      ~context: (`Assoc ["return", `Bool true]) () in
  [ knock; whoisthere; answer ]
```

The states of the automaton, called *dialog nodes*, are defined using the `Wcs.dialog_node` function. This builder function takes the name of the node and a set of options: the condition to enter the node (`~condition_spel`) expressed in SpEL [23], the response text (`~text`), and the parent node in the automaton (`~parent`). A node can also set information in the context object (`~context`). This object stores the state of the system, and can be used to pass information between conversation turns. In our example, the node `answer` sets `return` to true in the context, signaling the end of the joke.

In `wcs-ocaml`, SpEL expressions can be defined with a SpEL AST, or directly in the SpEL syntax used by Watson Assistant. For instance, the condition of the node `answer` can be rewritten `~condition:"@Characters"` which evaluates to true if the `Characters` entity is identified in the current

utterance. However, only the first approach returns an error at compile time if the `char_entity` entity is not defined.

Notice also that we are using OCaml name binding and type checking to link elements. For instance, the `~parent` argument is a dialog node instead of just a string as in the JSON format of a node.

Workspace. We now have all the elements of a workspace that can be run with Watson Assistant: a list of intents, a list of entities, and a list of dialog nodes.

```
let ws_knockknock =
  Wcs.workspace "Knock Knock"
    ~intents: [ who_intent ]
    ~entities: [ char_entity ]
    ~dialog_nodes: (knockknock who_intent char_entity
                    "Never mind it's pointless") ()
```

The library can be used to programmatically manipulate and check workspaces. `wcs-ocaml` can also generate the corresponding JSON object, and the developer can load it into Watson Assistant using the REST API or the user interface. The workspace can then be used by an application or visualized in the Watson Assistant editor (Figure 2).

2.2 Programming the Application

The workspace is only one tier of the complete chatbot. As shown in Figure 1, Watson Assistant also requires an application tier. Therefore, `wcs-ocaml` also provides support to deploy workspaces and to invoke them.

Workspace invocation. The REST call `message` sends a request containing the user utterance and the current *context* to Watson Assistant and returns the corresponding response. The context contains all the information required to resume the conversation (e.g., the current dialog node), and additional data that can be used to communicate between the application and the chatbot. For instance, the node `answer` defines a context variable `return` to notify the application of the end of the joke (Section 2.1).

Using the `message` function, we can define an `interpret` function that executes a conversation turn with a workspace `id` and extracts the value of the context field `return`. The first argument of `message` denotes the service credentials `creds`.

```
let interpret creds id req =
  let resp = Wcs_call.message creds id req in
  let ctx, ret = Json.take resp.msg_rsp_context "return" in
  { resp with msg_rsp_context = ctx }, ret
```

The context is an arbitrary JSON object that can be defined in OCaml with the Yojson library. There are thus only limited type guaranties on the context variables used to communicate between the application and the workspace. However, when both the application and the workspace are programmed in OCaml it is possible to share data structures. In our example, we can define a new type `return` (defined by `return = { return: bool; }`) and use JSON serialization in node `answer` and deserialization in the `interpret` function to impose stronger type guaranties.

A complete chatbot. The `interpret` function only executes one conversation turn. We can now define a complete chatbot that loops over the `interpret` function while maintaining the context between conversation turns.

```
let exec creds id =
  let rec loop ctx =
    let txt = input_line stdin in
    let req =
      Wcs.message_request ~text:txt ~context:ctx ()
    in
    let rsp, return = interpret creds id req in
    print_output rsp;
    begin match return with
    | Some v -> v
    | None -> loop rsp.msg_rsp_context
    end
  in
  loop Json.null
```

Deploying a chatbot. The `wcs-ocaml` library provides utility functions to deploy workspaces in Watson Assistant. These make it possible to integrate workspaces deployment in the application to avoid discrepancies between the two parts of the chatbot and use a version control system to manage the complete code. Programmers can thus frequently update and re-deploy the application to integrate user feedback and continuously improve the system [25].

3 Reactive Orchestration

Section 2 showed how to generate workspaces and invoke them in a simple chatbot application. This section demonstrates how to leverage reactive programming to build advanced chatbots by composing elementary ones. All the orchestration techniques presented here are generic and can be applied to any chatbot application.

3.1 ReactiveML

To illustrate our approach, we use ReactiveML [17], a reactive extension of OCaml. ReactiveML adds to OCaml the notion of *processes* that can be composed in parallel and communicate through signals. The language also provides some high-level reactive control structures like suspension and preemption (see [17] for more details on the semantics of the language and its implementation).

As a first example, we can lift the `exec` function of Section 2.2 to a ReactiveML process.

```
let process exec creds id input output =
  let rec process loop_ ctx =
    await input([txt]) in
    let req = message_request txt ctx in
    let rsp, return = interpret creds id req in
    emit output rsp;
```

```
    begin match return with
    | Some v -> v
    | None -> run loop_ rsp.msg_rsp_context
    end
  in
  run loop_ Json.null
```

Compared to the exec function presented in Section 2.2, the exec process takes two additional arguments input and output. These signals are the communication channels used to interact with the chatbot.

The process waits for a new user utterance to trigger a reaction using the await construct. It calls the cloud service using message_request and decodes the response with interpret.[2] After the call to interpret, instead of directly printing the output text on the standard output, the response is emitted on the output signal using the emit construct. Then, depending on the value of the return variable, the code returns the corresponding value or launches the recursive call with the run construct.

Signals input and output can be used to delegate the interface with the user to dedicated processes which ease the composition with other chatbots. For examples, we can build a simple chatbot by running in parallel exec with a process io that fetches user utterances (e.g., from the standard input) and prints the chatbot responses (e.g., on the standard output).

```
let process simple_bot creds id =
  signal bot_output, user_input in
  run io bot_output user_input ||
  run exec creds id user_input bot_output
```

The parallel composition is realized with the || operator. The two processes communicate through the signals bot_output and user_input, introduced with the signal/in construct.

3.2 Chatbot State Machines

We now show how to use the control structures provided by ReactiveML to orchestrate the execution of chatbots.

Restart. First, consider a chatbot with a multi-modal interface with a reset button to restart the conversation from the beginning. This behavior can be implemented as follows:

```
let rec process exec_restart creds id input output restart =
  do
    run exec creds id input output
  until restart ->
    run exec_restart creds id input output restart
  done
```

The preemption is realized with the do/until construct. The execution of the exec process is interrupted when the signal restart is emitted, causing a recursive call to exec_restart.

[2] interpret could be called asynchronously using Rml_async.

Timeout. Consider now a security mechanism that checks if the user is still active. The controller is a two-state automaton. The first state runs the main chatbot. If the user remains silent for 10 seconds, the controller enters a second mode, where a second chatbot asks if the conversation should continue. The corresponding ReactiveML code is:

```
let process exec_timeout creds id confirm input output =
  signal timeout, suspend_resume, kill in
  do
    control
      run exec creds id input output
    with suspend_resume done
    ||
    run periodic 10 timeout input
    ||
    loop
      await timeout;
      emit suspend_resume;
      let continue = run confirm input output in
      if continue then emit suspend_resume
      else emit kill
    end
  until kill done
```

The exec_timeout process comprises three parts executed in parallel. 1) The main chatbot is launched inside a control construct. Its execution suspends when suspend_resume is emitted and resumes at the next emission of the signal. 2) In parallel, the process periodic is a countdown that emits the signal timeout after 10 seconds of inactivity. The countdown is reset at each new user input. 3) In parallel, whenever signal timeout is observed, the controller emits signal suspend_resume to suspend the execution of the main chatbot and launch the process confirm. If the user wants to continue the conversation (confirm returns true) the controller emits suspend_resume again to resume the execution of the main chatbot. Otherwise it emits signal kill, triggering the do/until construct enclosing the three parts, thus terminating the process.

Note that exec_timeout is a higher-order process. It takes the process confirm as argument. Process confirm can be implemented with a Watson Assistant workspace executed by exec or by an arbitrary ReactiveML process.

3.3 Chatbot Dispatch

In the previous example the switch between the two chatbots is triggered by an external event (timeout or termination). A common pattern is to do the switch based on the conversation flow (e.g., invoking the joke skill of a personal assistant).

Turn dispatch. A first solution is to add a dedicated dispatcher chatbot whose role is to explicitely call other chatbots. Such a dispatcher can be programmed in a generic way as follows:

```
let process exec_dispatch creds
    dispatch_id trigger_id_list input output =
  let trigger_id_input_list =
    List.map
      (fun (trigger, id) -> (trigger, (id, new_signal())))
      trigger_id_list
  in
  Rml_list.par_iter
    (proc (_, (id, local_input)) ->
      run exec creds id local_input output)
    trigger_id_input_list
  ||
  loop
    let trigger = run exec creds dispatch_id input output in
    let _, local_input =
      List.assoc trigger trigger_id_input_list
    in
    emit local_input (last ?input)
  end
```

Besides the service credentials and the `input` and `output` signals, the `exec_dispatch` process takes as arguments the id of the dispatcher chatbot `dispatch_id`, and `trigger_id_list`, a list of pairs (trigger, chatbot id). Whenever the dispatcher terminates, its return value is matched against the list of triggers to launch the corresponding chatbot. This architecture supports a multi-turn conversation with the dispatcher to trigger a chatbot, but the user must go through the dispatcher for each conversation turn with the chatbot.

The first part of the process `exec_dispatch` turns the list `trigger_id_list` into an association list indexed by the triggers `trigger_id_input_list`, where chatbot ids are paired with a unique local input signal (`new_signal()`). The second part launches in parallel one `exec` process for each chatbot using its local input signal. All the chatbots thus publish their responses on the same `output` signal, but await inputs on their own local signal. The last part executes the dispatcher to get a `trigger` value. Based on this value, it retrieves the corresponding local input signal in `trigger_id_input_list` and emits the last received input on this signal, thus triggering the corresponding chatbot.

Note that the dispatcher maintains the states of all the chatbots in parallel. The user can thus suspend and resume multiple conversations at will.

Session dispatch. Instead of addressing the dispatcher before each conversation turn, the next example uses the dispatcher to start multi-turn conversations with the chatbots. A first conversation determines which skill to trigger for the next session. In addition, a supervisor provides high-level controls during a session. At any time during the conversation, the user can express generic commands such as *stop* or *restart*.

The process `session_dispatch` is a two-state automaton implemented with two mutually recursive processes. State `dispatch_state` determines the chatbot for the next session. Afterwards state `control_state` launches a supervised conversation with the corresponding chatbot.

```
let process session_dispatch creds
  dispatch_id controller_id trigger_id_list input output =

  let rec process dispatch_state =
    let trigger = run exec creds dispatch_id input output in
    let _, id = List.assoc trigger trigger_id_list in
    run control_state id

  and process control_state id =
    signal kill, restart in
    signal cmds, local_input in
    do
      run exec creds controller_id input cmds ||
      run emit_commands cmds input kill restart local_input ||
      run exec_restart creds id local_input output restart;
      emit kill
    until kill -> run dispatch_state
    done

  in
  run dispatch_state
```

State `control_state` launches in parallel: 1) one `exec` process for the controller chatbot which reads the main `input` signal and emits commands on the local signal `cmds`. 2) A process `emit_commands` that controls the emission of signals `kill` and `restart` based on the commands emitted by the controller. If no command is received, `emit_commands` simply pipes the current input in `local_input`. 3) A process `exec_restart` (Section 3.2) controlled by the `restart` signal for the session chatbot, which reads the `local_input` signal and publishes on the global `output`. These three processes are executed inside a `do`/`until` construct controlled by `kill`. The controller can thus terminate the conversation at any moment to go back to `dispatch_state`.

The process `emit_commands` emits signals corresponding to the commands detected on signal `cmds`.

```
type command = Stop | Reset | Continue

let process emit_commands cmds input kill
    restart local_input =
  loop
    await input(txt, ctx) in
    match get_command ctx with
    | Stop -> emit kill
    | Continue -> emit local_input (txt, ctx)
    | Reset -> emit restart
  end
```

This process is another illustration of how to share data structures between the application and the Watson Assistant workspace (see Section 2.2). The function `get_command` marshals the JSON object returned by the controller workspace into the OCaml `command` type. The function that generates the workspace does the opposite operation, unmarshalling a

```
1 when a transaction occurs, called 'the transaction'
2 if
3  the country code of this transaction is not "US"
4  and the amount of this transaction is more than 1000
5 then
6  emit a new authorization response where
7   the account is 'the account',
8   the message is "R01: Out of country transaction greater than 1000",
9   the transaction is this transaction;
```

(a) Event rule

```
1 an account is a business entity identified by an id.
2 an account has a status (an account status).
3 an account status can be one of: Excellent, Good, Fair, Poor, Risky.
4
5 a transaction is a business event time–stamped by a date.
6 a transaction is related to an account.
7 a transaction has an amount (a number).
8 a transaction has a country code.
9
10 an authorization response is a business event time–stamped by a date.
11 an authorization response is related to an account.
12 an authorization response has a message.
13 an authorization response has a transaction.
```

(b) Data model

Figure 3. ODM credit card example [11].

command into a JSON object. We can use this type to discriminate between the possible values returned by `get_command`. If the command is `Stop` or `Reset` we emit the corresponding signals. If the command is Continue, which is returned by `get_command` when the user does not trigger the controller, we simply pipe the current input in `local_input`.

4 Case Study: RuleBot

As a case study, we now present *RuleBot*, a chatbot for authoring *business rules* [5].[3] We target the event-processing engine ODM Insights (*Operational Decision Manager: Decision Server Insights*) in which programs are expressed in a controlled natural language called BERL (Business Event Rule Language). The design of RuleBot was challenging for two main reasons. 1) The dialog is mixed-initiative, that is, both the chatbot and the user can lead the conversation. This desirable trait for conversational agents benefits from advanced composition techniques. 2) The dialog builds a recursive data structure, the AST of an event processing rule, which heavily relies on data sharing between the service and the application.

4.1 ODM Insights

Figure 3a shows an example of a BERL program for handling credit card transactions and authorization responses taken directly from the tutorial of the official product documentation [11]. The keywords **when**, **if**, and **then** introduce the event, condition, and actions clauses of the rule, respectively.

```
rule    ::= when event (if cond)? then actions
event   ::= (a | an) EVENT occurs, (called VAR)?
cond    ::= expr (and expr)*
actions ::= (action ;)+
action  ::= print expr | emit expr
          | define VAR as expr
          | set FIELD of VAR to expr

expr ::= VAR | expr BOP expr | not expr | lit
       | this EVENT | the FIELD of expr
       | the (total | average) FIELD of (all ?) expr
       | a new CONCEPT where (the FIELD is expr ,)+
lit  ::= INT | REAL | STRING | BOOL | ENUM | DATE | DURATION
BOP  ::= and | or | is | is more than | is not | ...
```

Figure 4. Grammar of a subset of BERL.

BERL is a controlled natural language [14]. While users can intuitively read programs like the one presented in Figure 3a, programmers must follow the grammar of the language. Figure 4 shows a representative subset of the BERL grammar. A *rule* consists of an *event* (when-clause), a *condition* (optional `if`-clause), and *actions* (`then`-clause). There are four different kinds of actions: print a message, emit an event, define a variable, or set a field. A large portion of the grammar (everything after *expr*) is devoted to the syntax of expressions that resembles expression languages found in many conventional programming languages but with a syntax reminiscent of natural language.

The concepts, fields, and events used in the rules (e.g., account, transaction, and authorization response) are defined in a separate Business Model Definition (BMD) file. For instance, Figure 3b shows the BMD file used for the credit card transactions rules. This data model is also described in a controlled natural language.

4.2 RuleBot Overview

RuleBot provides a conversational interface with the goal to help non-programmer business users to author rules using the information provided in the BMD file. Presenting the authoring experience as a conversation enables RuleBot to guide and assist the human who does not need to learn how to program in the BERL language. For example, Lines 6-7 of Figure 3a can be generated with the following conversation:

```
1 Bot: What kind of action would you like to add?
2 User: Emit a new authorization response?
3 Bot: Ok, I'm adding a new emit action.
4 Bot: What is the account of the authorization response?
5 User: The value of the variable "the account".
6 Bot: What is the message of the authorization response?
```

[3] https://github.com/IBM/wcs-ocaml/tree/master/examples/rulebot

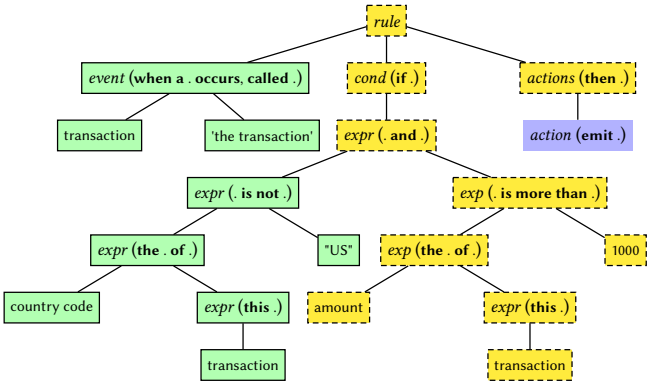


Figure 5. Partial tree under construction.

The bot uses the structure of a rule and the data model (e.g., the required field of an authorization defined in Line 11 of Figure 3b) to guide the dialog.

Through the flow of the conversation, the dialog controller incrementally builds the abstract syntax tree (AST) of a BERL rule. The central data structure in RuleBot is thus the *partial tree*, that is, a BERL AST under construction. This is a key data structure because both the flow of the conversation itself and its outcome are driven by the shape of BERL rules. Figure 5 shows the partial tree state at Line 3 of the sample conversation. Note that it is a partial tree since the emitAct node is still missing required children.

4.3 Building the Rule

The partial tree associates a state to each node (represented with different colors and styling in Figures 5): undefined (blue, no border), filled (yellow, dashed border), or accepted (green, solid-line border). The state is used to guide the conversation. A node in the undefined state is still missing children. The conversation should thus go towards adding these AST elements. For example, the partial tree of Figure 5 is leading to the generation of Line 4 of the sample conversation. When all its required children are filled, a node switches to the filled state. However, since the bot (and the user) can make mistakes, it is important to be able to modify a node. A node in the filled state can thus still be edited.

The user can accept or reject parts of the AST, in which case they must be filled again. The conversation can terminate only when all the nodes are in the accepted state.

We introduce separate data structures to represent tree transformations called *instructions*. Instructions constitute a protocol by which components of the dialog controller communicate with each other, as shown in Section 4.5. There are two kinds of instructions: *replace*, which swaps out one subtree for a new one, and *accept*, which changes the state of a subtree to be accepted.

4.4 Natural Language Understanding

RuleBot handles user input that can combine natural language text and BERL expressions. At each step of the conversation, the user can input BERL fragments to short-circuit RuleBot understanding. This ability provides for a smooth transition between free natural language and controlled natural language. BERL expressions are quoted in markdown style with back-ticks so the parser can identify them. The natural language part is processed using the entity extractor and intent classifier of Watson Assistant. The quoted expressions are parsed using the BERL grammar of Figure 4 to produce an AST fragment.

Intents are denoted with a hash-sigil (#). RuleBot has intents for BERL grammar non-terminals, such as `#printAct`, `#emitAct`, etc. These drive dialog choices. In addition, RuleBot has intents for mixed-initiative dialog, such as `#fill`, `#help`, and `#reset`. Intents are detected via classifiers trained from example utterances.

Entities are denoted with an at-sigil (@). RuleBot uses the following three kinds of entities:

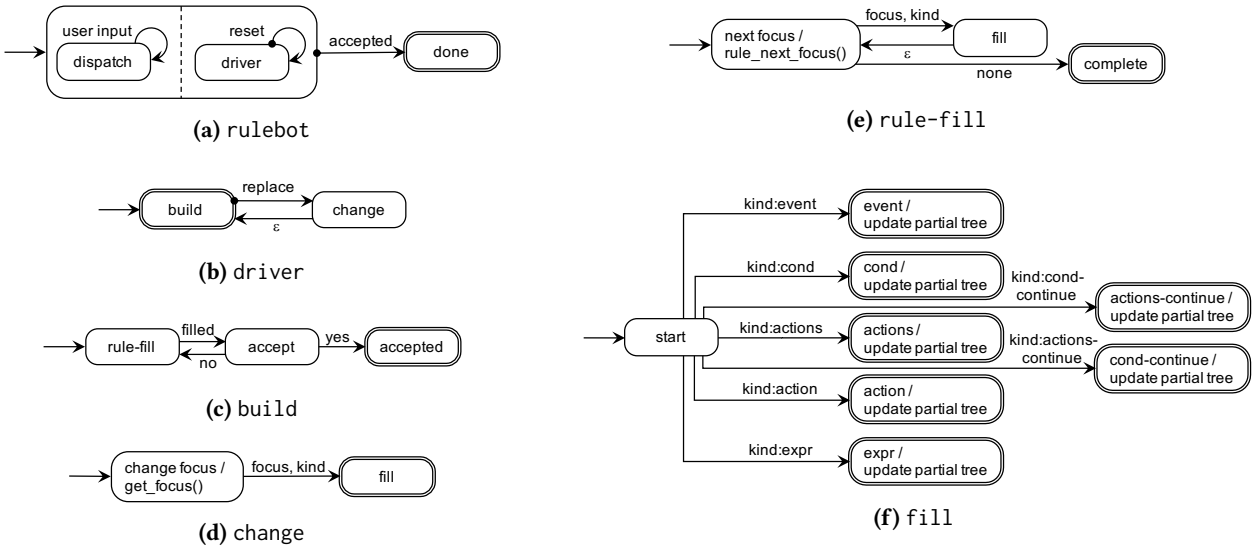
- Built-in entities that are predefined in Watson Assistant, e.g., `@number` to extract numbers from user input.
- General-purpose entities, e.g., `@yes` and `@no` for yes/no answers from user input.
- Application-specific entities whose definition is based on the BMD file. Those correspond to schema or data model information and are specific to a given application domain. These entities are prepared by parsing the BMD file.

4.5 RuleBot Control

The dialog controller drives the conversation until the rule is ready to be deployed. We implemented RuleBot using ReactiveML for the orchestration part and Watson assistant for the conversation part. To present the whole application within a single formalism, we use hierarchical state machines [9], which are commonly used as a graphical representation of reactive programs. Figure 6 shows the ReactiveML controllers and Figure 7 shows the Watson Assistant workspaces. Workspaces are invoked in particular states of the controllers. The finite-state approach is a least common denominator of sorts for specifying conversational agents [18]. The user interface of Watson Assistant visualizes dialog logic as finite state machines (c.f. Figure 2), but where Figure 7 renders transition triggers on arrows, Watson Assistant renders transition triggers on nodes. We introduce hierarchical state machine concepts as they come up.

To enable natural interaction with the user, one key feature of RuleBot is mixed-initiative dialogs: the conversation flow can be directed by the application (filling AST nodes one after the other), but the human is also able to pick the part of the rule they want to work on (user initiative).

Process `driver` (Figure 6b) drives the construction of the AST based on both user instructions and its own control logic.



(a) `rulebot` **(b)** `driver` **(c)** `build` **(d)** change **(e)** `rule-fill` **(f)** `fill`

Figure 6. RuleBot controllers

States are shown as rectangles with rounded corners labeled by their name, and transitions are shown as arrows labeled by their trigger. The start state, here `build`, has an incoming arrow that does not originate from another node. Final states, here also `build`, have a double border. The state machine is hierarchical in the dual sense that first, individual states in Figure 6b can contain state machines of their own (e.g., `build` is presented in Figure 6c), and second, the state machine itself constitutes a state in another one (`driver` in Figure 6a). In the start state `build` the chatbot keeps the initiative of the conversation. At any time during the conversation, the user can grab the initiative by requesting a change, wich preempts the execution of `build`. Preemption transitions are indicated with a bullet. The controller then enters state change. When the execution of change terminates, the controller enters state `build` without further input (ε transition) and takes the control back to complete the rule. The ReactiveML code corresponding to Figure 6b is:

```
let rec process driver =
  do run build
  until replace(focus, txt) ->
    run change focus txt;
    run driver
  done
```

The `replace` signal carries a pair `(focus, text)`, so users can specify the focus for the next update. For space reasons, we only present the graphical representation of the code.

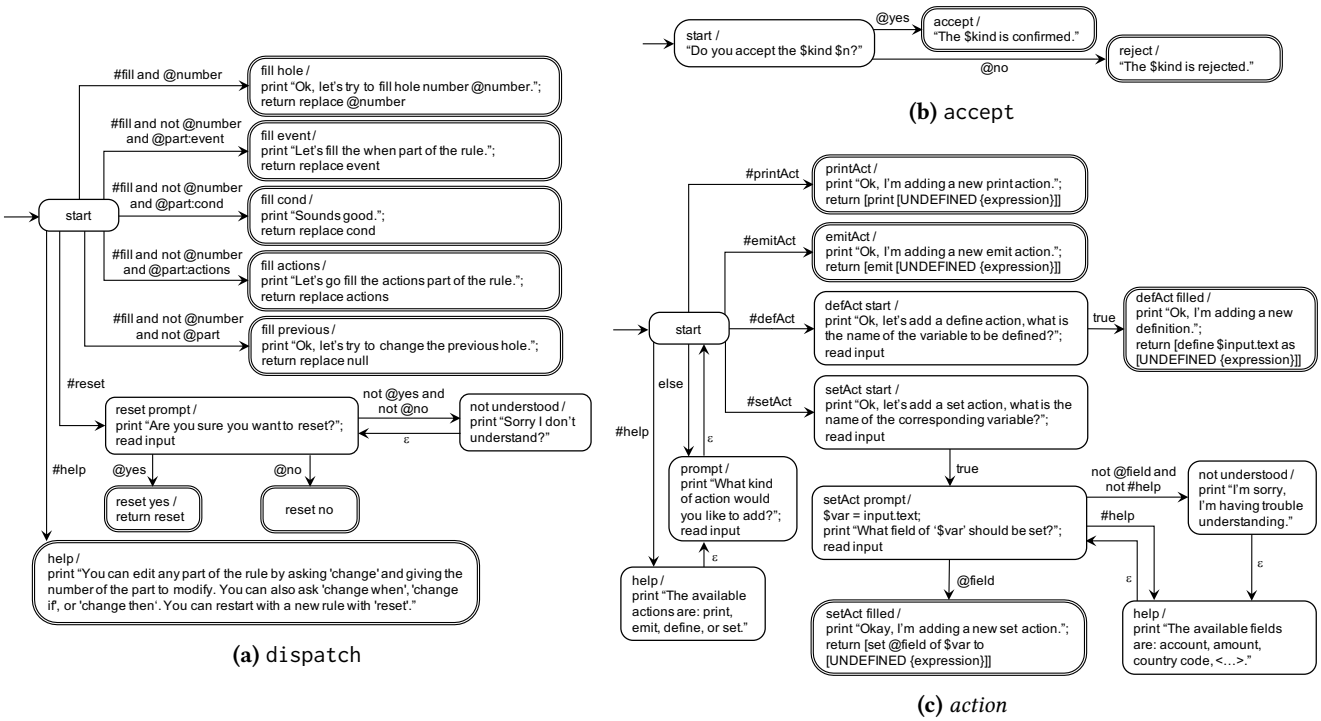
Process `build` (Figure 6c) coordinates the conversation flow to fill all the nodes in the AST and then ask for confirmation that the rule is correct and ready for deployment. When the execution of `rule-fill` terminates, RuleBot asks the user for confirmation using the `accept` workspace of Figure 7b.

Process change (Figure 6d) handles user modification requests. The label of the `get-focus` state is written in the form `state-name / action`, meaning that the action gets executed upon entering the state, in the fashion of a Moore Machine. In this case, the action consists of calling a function, `get_focus()`, to select the AST node to focus on based on the user input. AST nodes are uniquely identified by a pair (focus, kind), where focus is an integer and kind denotes the syntactical category (e.g., *event*, *cond*, *actions* in Figure 4).

Process `rule-fill` (Figure 6e) is a loop that first selects the next AST node to fill (`next-focus` state) and then updates the rule (`fill` state). This is the counterpart of change when RuleBot takes control of the conversation. Here the node selection (function `rule_next_focus()`) is based on the current partial tree and the previous node that has been filled. We chose a heuristic to select the first undefined or rejected node that followed the previously filled one in the pre-order of the AST. If there is no node to fill behind the one which has been just filled, `rule_next_focus()` searches a node to fill from the beginning of the rule.

Process `fill` (Figure 6f) drives the conversation to the sub-dialog that produces an AST of the expected kind. Depending upon the selected node, RuleBot must construct different kinds of sub-AST to complete the partial tree. The kinds correspond to syntactic classes of the grammar. Each state of `fill` triggers a workspace to builds an AST of a different kind, and then uses a replace instruction to update the rule. For space reasons, we only present one of the workspace for constructing AST subtrees.

Workspace *action* (Figure 7c) builds an *action* subtree. It uses four intents for the four action types in the BERL grammar (**print**, **emit**, **define**, and **set**). The states `not understood` and `help` display some messages and loop back to `prompt`



(a) `dispatch` **(b)** `accept` **(c)** *action*

Figure 7. RuleBot workspaces

without further input (ε transitions). Transitions labeled `true` succeed for any user input: the input serves as a field or variable name and is not subjected to natural-language understanding. This workspace also illustrates a design decision we made: RuleBot does not allow defining a variable or setting a field without immediately giving the name of the variable or field, because separating that step would be confusing.

Process `rulebot` (Figure 6a), the top-level controller of the complete chatbot, executes the `driver` process in parallel with the `dispatch` workspace to gather user instructions. Parallel composition is indicated by a dashed line. This process is an advanced version of `session_dispatch` presented in Section 3.3. A `reset` instruction preempts and restarts the execution of `driver`, and the execution of the two parallel processes ends when the user validates the rule (`accepted`).

Workspace `dispatch` (Figure 7a) analyzes each user input and can trigger events that change the conversation flow. It detects the intent `#fill` along with the AST node to modify. The AST node can be specified by an entity `@number` or by the name of the part of the rule. The workspace can also generate a `reset` event after confirmation by the user. Finally, it also handles top-level help.

5 Related Work

A popular survey on chatbot technologies identifies three main approaches to chatbot programming and orchestration: finite-state based, frame-based, and agent-based [18]. The finite-state based approach is the most low-level, giving the programmer fine-grained control but also forcing them to specify a lot by hand. The frame-based approach is more high-level, letting the programmer specify slots to be filled in, but ceding control over which slots are filled in when to the system [3]. The agent-based approach is even higher-level by leaving even more of the dialog control to be planned and optimized by the system, and has been the subject of much research over the years [18]. The VoiceXML standard for chatbots supported both finite-state and frame-based approach as well as subdialogs [16], but did not adopt the agent-based approach. Similarly, the recent surge in commercial platforms for authoring and hosting chatbots has been adopting finite-state and frame-based approaches but not agent-based approaches [20], perhaps to retain more control over dialog flow. Most commercial chatbot platforms are programmable via libraries in general-purpose programming languages such as JavaScript, causing control to be rather low-level. In contrast, our work uses a reactive language, a more natural fit for the kind of control needed in a chatbot. Watson Assistant targets low-code users by offering graphical authoring [12]. Our work complements this with an alternative code-based authoring approach. The RuleBot case study in Section 4 drives the dialog from a grammar, following a vision we sketched in earlier work [10]. For chatbot orchestration, consumer platforms often dispatch among libraries of skills implemented as third-party subdialogs [1]. In contrast, since our work focuses more on enterprise platforms, it provides more expressive composition.

This paper shows that using a reactive language, namely ReactiveML, simplifies the programming of advanced chatbot composition. ReactiveML is based on the reactive synchronous model [4], providing a well-defined semantics of concurrency and expressive control structures. Dataflow languages [8, 21] are reactive languages with explicit information flow that could be useful to thread chatbot responses to compose multiple skills. Orchestration languages like Flapjax [19], ORC [13], and Elm [7] might also be good candidates to program chatbots compositions; this paper uses a reactive synchronous language, and leaves applying those languages on chatbots to future work.

Our case study uses a chatbot as an authoring interface for event-processing rules. There has been some earlier work in using controlled natural language [14] to author event-processing rules [15]. In fact, the Operational Decision Manager also adopts this approach [11]. In contrast, since chatbots have a natural-language understanding module, our work does not require the natural language to be "controlled". There has been substantial work on natural language interfaces to databases [2], but that focuses on batch queries, not event-processing rules. Besides natural language, another approach to making event-processing rules easier to author by low-code, non-technical users is spreadsheets [24].

6 Conclusion

This paper demonstrates that leveraging existing programming languages can help to program advanced chatbot applications. We present `wcs-ocaml`, a lightweight embedding of the Watson Assistant API in OCaml. Based on this library, we show how to program advanced chatbot orchestration in a general purpose reactive language, ReactiveML. We can then program generic chatbot combinators as well as ad-hoc components. To make these ideas more concrete, we describe RuleBot, a conversational interface for authoring event processing rules programmed in ReactiveML with `wcs-ocaml`.

The RuleBot case study showcases two main advantages of our approach. First, in RuleBot, the application and Watson Assistant frequently exchange complex data structures. Using a single strongly typed programming language to program both tiers of the application ensures the coherence of the two tiers. Second, the RuleBot architecture relies on parallel and hierarchical compositions of multiple chatbots that are typically difficult to program from scratch but can be easily expressed in ReactiveML.

References

[1] Amazon 2016. *Alexa Skills Kit.* Amazon. https://developer.amazon.com/alexa-skills-kit (Retrieved 08/2018).

[2] I. Androutsopoulos, G. Ritchie, and P. Thanisch. 1995. Natural Language Interfaces to Databases – An Introduction. *Natural Language Engineering* 1, 1 (1995).

[3] D. Bobrow, R. Kaplan, M. Kay, D. Norman, H. Thompson, and T. Winograd. 1977. GUS, a frame-driven dialog system. *Artificial Intelligence* 8, 2 (1977), 155–173.

[4] F. Boussinot. 2010. *Safe Reactive Programming: The FunLoft Language.* Lambert Academic Publishing.

[5] Business Rules Group 2003. *The Business Rules Manifesto: The Principles of Rule Independence, Version 2.0.* Business Rules Group. http://www.businessrulesgroup.org/brmanifesto.php (Retrieved 08/2018).

[6] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. 2006. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects (FMCO).* 266–296.

[7] E. Czaplicki and S. Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Conference on Programming Language Design and Implementation (PLDI).* 411–422.

[8] C. Elliott and P. Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming (ICFP).* 263–273.

[9] D. Harel. 1987. StateCharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (1987), 231–274.

[10] Martin Hirzel, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Mandana Vaziri. 2017. I Can Parse You: Grammars for Dialogs. In *Summit oN Advances in Programming Languages (SNAPL).* 6:1–6:15.

[11] IBM 2014. *Operational Decision Manager Tutorial: Getting started with Decision Server Insights.* IBM. https://www.ibm.com/support/knowledgecenter/SSQP76_8.9.2/com.ibm.odm.itoa.tutorial/topics/itoa_tut_intro.html (Retrieved 08/2018).

[12] IBM 2018. *Overview of Watson Assistant.* IBM. https://console.bluemix.net/docs/services/conversation/index.html (Retrieved 08/2018).

[13] D. Kitchin, A. Quark, W. Cook, and J. Misra. 2009. The Orc Programming Language. In *Proceedings of FMOODS/FORTE 2009.* 1–25.

[14] T. Kuhn. 2014. A Survey and Classification of Controlled Natural Languages. *Computational Linguistics* 40, 1 (2014), 121–170.

[15] M. Linehan, S. Dehors, E. Rabinovich, and F. Fournier. 2011. Controlled English Language for Production and Event Processing Rules. In *Conference on Distributed Event-Based Systems (DEBS).* 149–158.

[16] B. Lucas. 2000. VoiceXML for Web-based Distributed Conversational Applications. *Commun. ACM* 43, 9 (2000), 53–57.

[17] L. Mandel, C. Pasteur, and M. Pouzet. 2015. ReactiveML, Ten Years Later. In *Symposium on Principles and Practice of Declarative Programming (PPDP).* 6–17.

[18] M. McTear. 2002. Spoken dialogue technology: Enabling the conversational interface. *Comput. Surveys* 34, 1 (2002), 90–169.

[19] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[20] A. Patil, K. Marimuthu, R. Niranchana, et al. 2017. Comparative study of cloud platforms to develop a Chatbot. *International Journal of Engineering & Technology* 6, 3 (2017), 57–61.

[21] M. Pouzet. 2006. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI.

[22] M. Serrano and G. Berry. 2012. Multitier Programming in Hop. *Commun. ACM* 55, 8 (Aug. 2012), 53–59.

[23] Spring. 2018. Spring Expression Language (SpEL). https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#expressions (Retrieved 08/2018).

[24] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. 2014. Stream Processing with a Spreadsheet. In *European Conference on Object-Oriented Programming (ECOOP).* 360–384.

[25] J. Williams, N. Niraula, P. Dasigi, A. Lakshmiratan, C. Suarez, M. Reddy, and G. Zweig. 2015. Rapidly Scaling Dialog Systems with Interactive Learning. In *International Workshop on Spoken Dialog Systems (IWSDS).*

[26] Mo Yu, Xiaoxiao Guo, Jinfeng Yi, Shiyu Chang, Saloni Potdar, Yu Cheng, Gerald Tesauro, Haoyu Wang, and Bowen Zhou. 2018. Diverse Few-Shot Text Classification with Multiple Metrics. *arXiv preprint arXiv:1805.07513* (2018).