Platon: Dialog Management and Rapid Prototyping for Multilingual Multi-User Dialog Systems

Martin Gropp, Anna Schmidt, Thomas Kleinbauer, Dietrich Klakow

Spoken Language Systems Group
Saarland University
Saarbrücken, Germany
⟨firstname⟩.⟨lastname⟩@lsv.uni-saarland.de

Abstract. We introduce Platon, a domain-specific language for authoring dialog systems based on Groovy, a dynamic programming language for the Java Virtual Machine (JVM). It is a fully-featured tool for dialog management that is also particularly suitable for, but not limited to, rapid prototyping making it possible to create a basic multilingual dialog system with minimal overhead and then gradually extend it to a complete system. It supports multilinguality, multiple users in a single session, and has built-in support for interacting with objects in the dialog environment. It is possible to integrate external components for natural language understanding and generation, while Platon can itself be integrated even in non-JVM projects or run in a stand-alone debugging tool for testing. In this paper we describe important elements of the language and present two scenarios Platon has been used in.

Index Terms: dialog framework, dialog systems, dialog management, multilingual, multi-user, rapid prototyping

1 Introduction

Platon is a domain-specific language for authoring dialog systems. It was designed with rapid prototyping in mind and has integrated facilities to interact with discourse and world objects. Platon is able to support multi-linguality and can handle multi-user scenarios. Its modular architecture allows building dialog systems that can either be used as a component in a more complex application, or function as a stand-alone system with built-in support for speech recognition and text synthesis.

Platon is designed for both technical and non-technical users. The language is designed for readability and maintainability, yet offers advanced users flexible extension options. Being based on Groovy, a dynamic language for the Java Virtual Machine (JVM), Platon dialogs can draw on the full set of features of an established powerful programming language, integrate existing Java classes, and utilize the vast amount of existing libraries for the JVM. This ability allows, for

example, to incorporate existing parsers, or to connect to external databases or services.
This article introduces the Platon language and exemplifies typical use cases.[1]
The main contributions of Platon to the landscape of already existing solutions to dialog management are:

- Accessibility: Easy to use script format for implementing dialog behavior
- Extensibility: Backed by a dynamic programming language (Groovy)
- Integration: Modular interaction with third-party components
- Flexibility: Not tied in with a specific dialog management model
- Speed: Rapid prototyping through compact and expressive syntax
- Availability: OS-independent open source package ready for download

2 Related Work

Platon is agnostic to the underlying dialog management *model*, hence we concentrate in this section on relevant alternative *systems*. (A general introduction to different models of dialog management is given by [2].) A number of approaches for implementing dialog managers have been proposed in the past, which can be classified according to the complexity of interactions they allow.

Declarative interpreted languages, such as e. g. AIML[2] or VoiceXML[3], can be appealing due to their ease of use, allowing non-expert users to model simple dialog behavior. When based on widely accepted standards, such as XML, the availability of sophisticated editing tools can facilitate speedy development. However, this often comes at the price of limited functionality. AIML, for instance, interprets user input with a simple pattern matching language, provides only rudimentary management of the dialog history through its `<that>` and `<topic>` tags, and is restricted to purely user-initiative dialog behavior. Similarly, VoiceXML has strengths in form-filling applications but has been criticized for its inflexible handling of dialog initiative and shallow dialog model [3, 4].

NPCEditor [5] is a tool that allows to model question/answer pairs and uses information retrieval techniques to process user input that matches only partially. Some more elaborate tools, such as e. g. RavenClaw [6], aim for higher flexibility and task independence. To this end, RavenClaw employs a two-tiered architecture that separates domain and dialog management aspects, and has been successfully used in a number of applications. However, the system expects dialogs to be modeled as a hierarchical plan which may be too rigid a constraint for dialogs that are not purely task-driven.

In general, the more opinionated a framework or tool is with respect to expressing possible interactions, the more it runs the risk of hampering what was not anticipated by the framework's developers. This is one criticism that

[1] A complete documentation is available as a separate technical report [1] with details about the language definition and the implementation of Platon.
[2] `http://www.alicebot.org/aiml.html`
[3] `http://www.w3.org/TR/voicexml21`

motivated the development of the DIPPER [7] architecture as an alternative to TrindiKit [8]. Both systems are inherently based on the Information State Update approach [9] which, however, restricts their use to situations where this model is applicable.

In comparison, Platon unites the strengths of the above systems while avoiding some of their shortcomings. Basic keyword-based dialog behavior can be easily implemented, similar to simple scripting languages, but based on very flexible pattern matching capabilities. In addition, Platon also contains a sophisticated task model based on agents to allow modeling more complex dialogs. Moreover, being tightly coupled with Groovy/Java, Platon makes it especially easy to extend the off-the-shelf functionality, e. g. to integrate external resources. These points will be highlighted in more detail in the following sections.

3 Elementary Features

3.1 Platon for Rapid Prototyping

Platon can be used to realize a basic chatbot-like dialog system as a standalone application, as a first impression of a planned bigger system, or for early integration with other components.

A developer can directly define and test reactions to a limited number of textual inputs using simple matching rules, such as the one in figure 1. These rules can, as in this example, use regular expressions, but Platon also supports more complex input analyses. A number of predefined functions can be used in the reactions to an input match, for example for language output, for waiting for user responses, or for interacting with the outside world (see below).

```
input(~/\bhello\b/) {
  tell all, "Hello World!";
}
```

Fig. 1. Monolingual *Hello World* script.

3.2 Multilingual Multi-User Dialog Systems

Especially for prototypes and dialog systems without dedicated NLU and NLG components, Platon scripts can (optionally) provide internationalization support. Figure 2 shows a complete "Hello World" example for English and German: both input matching and reactions are realized bilingually.

```
input(
  en: ~/\bhello\b/,
  de: ~/\bhallo\b/
) {
  tell all, [
    en: "Hello World!",
    de: "Hallo Welt!"
  ];
}
```

Fig. 2. Multilingual *Hello World* script

Platon supports dialog situations with more than one user. To keep the complexity as low as possible and to make it easy to use the correct resources for users with different languages, a new dialog engine instance is created for each user at first. Interaction between these instances is achieved either by using shared variables or by sending and receiving arbitrary messages. Reacting to such messages works much like receiving text input from users.

4 Complex Platon Systems

Although Platon works well for rapid prototyping, it was built with more complex scenarios in mind. In a more complex system, it is intended to take the role of the dialog manager that interacts with external NLU and NLG components.

4.1 Dialog Management: Task Decomposition and Agents

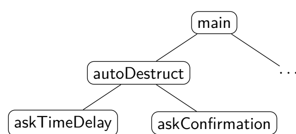

Fig. 3. Part of a task tree

Dialog management in Platon is based on a concept of hierarchical task decomposition similar to that of RavenClaw [6], breaking a complex scenario up into manageable parts. For example, the tree in figure 3 is a small excerpt of the task tree for a computer game set on a space station. Here, the station has an auto-destruct system that can be activated by the player. For this, a time delay needs to be specified and the player has to confirm that he/she is really sure about giving the command.

In a dialog script these sub-tasks appear as *agents*, each of which contains a set of rules for input handling and for other reactions, and may keep a local state. Figure 4 shows the script part of the example. The `autoDestruct` agent has two variables, `delay` and `confirmed`, that represent the state of the agent. The following `enter` block is executed when the agent is activated and every time a sub-agent completes (technically: whenever the agent becomes the top element on the stack; see below). In the example, `enter` checks which information is still needed and activates other agents accordingly, or, if all necessary information is available, exits.

Although Platon can provide basic language understanding tasks as described above, a more complex dialog system typically integrates a separate NLU module that can provide a comprehensive analysis of the user input. Platon's JVM foundation makes the integration of many existing parsers, taggers, dialog act classifiers, etc. straight-forward. Moreover, if necessary, the dialog manager can provide access to certain context information, e. g. about the active agents, the dialog history, or entities in the environment, which can, for example, be used for the context-aware disambiguation of the input.

Platon is able to integrate such a broad range of external NLU modules because it does not impose any restrictions on the kinds of input from such modules. In particular, it does not expect a specific kind of semantic representation, dialog act scheme, domain ontology, etc. Platon can operate with any user-defined input type. For instance, an application can use a set of different classes as in the example of figure 5 where the NLU module uses the class `TimeDelay` for utterances specifying time delays, or opt for a different representation, such as simple strings, if that is considered more suitable.

4.2 Processing Input

Active agents are organized in a stack. When an agent calls another agent, e. g. `askTimeDelay()` in figure 4, the new agent is pushed on the stack, and stays

```
agent autoDestruct {
  def delay = null;
  def confirmed = false;

  enter {
    if (delay == null) {
      askTimeDelay();
    } else if (!confirmed) {
      askConfirmation();
    } else {
      exit();
    }
  }
  ...
  agent askTimeDelay { ... }
  agent askConfirmation { ... }
}
```

Fig. 4. Outline of the `autoDestruct` agent

```
input(TimeDelay) {
  input ->
  delay = input;
  askConfirmation();
}

agent askTimeDelay {
  input(String) {
    input ->
    delay = new TimeDelay(input);
    exit();
  }
}

...
```

Fig. 5. Input statements from the `autoDestruct` agent

there until it exits[4]. In the example of figure 6, the agent `autoDestruct` has called `askTimeDelay`, which has consequently been put at the top of the stack. Every time the dialog manager has to determine the system's reaction to an event (e. g. user input), it starts with the agent at the top of the stack and then proceeds downwards until an agent accepts the event. Optionally, an agent can delegate events to another (possibly inactive) agent, either on a case-by-case basis, or as a regular part of its own event processing procedure. This feature makes it easy to integrate agents for common tasks without adding complexity to the general stack-based processing scheme. We are currently working on a standard library of common agents (e. g. for repetitions or confirmations).

Examining the agent stack in figure 6, we see that `autoDestruct` (from figure 4) has already called `askTimeDelay`, which is now on top of the stack. Its only input statement accepts `String` objects, but not objects of type `TimeDelay`. These are matched in the second agent, `autoDestruct`. This means that objects of type `TimeDelay` will be handled even if the `askTimeDelay` agent is not active: as long as `autoDestruct` is somewhere on the stack, `TimeDelay` objects can be interpreted as the delay for the self-destruct sequence.

Assuming a user input of "*set the time delay to five minutes*" this string would first be passed to the NLU which recognizes it as a time delay specification and returns a `TimeDelay` object storing the duration. The `input` statement in the

[4] Since all agents on the stack are active and can manipulate the stack, the call semantics are actually more complex than for example with regular functions. By default, agent changes are handled as if the agent executing the operations were on top of the stack, removing other agents covering the caller. This leads to the behavior expected for a regular function call. If required, this "stack cutting" mechanism can be disabled for each call. See [1] for details.

`askTimeDelay` agent only matches objects of type `String`, hence we proceed down the stack and find the next agent, `autoDestruct`. Its first `input` rule accepts the `TimeDelay` object and calls the next agent, which is pushed on top of the stack replacing `askTimeDelay`.

4.3 Situated Interaction

Platon was built to interact with objects in the dialog environment, to affect this "world" using voice input, and to react to changes. Platon systems can connect to an external server to exchange information about world objects, either using a direct Java-compatible interface or via an RPC protocol based on Apache Thrift[5]. Such a world object server must implement one function to allow the manipulation of object attributes, plus an additional two if atomic transactions are required. On the other side of the interface, Platon implements functions to receive notifications about added, deleted, and modified objects from the world server, which are transparently cached, and supports transactions as well. From the perspective of a dialog designer, this complexity is completely invisible. Platon provides the statements `objectAdded`, `objectDeleted`, and `objectModified` to react to changes in the world state, which support complex selectors to decide whether or not a given change in an object is relevant.

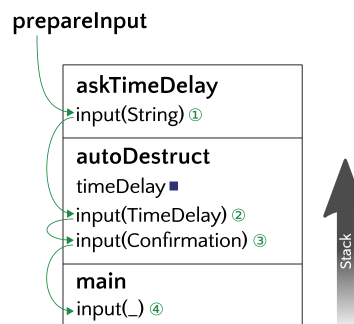

Fig. 6. Three active agents on a stack

5 Sample Applications

Platon was originally developed in the context of an interactive multi-user game focusing on collaboration between players speaking different languages. The dialog system plays a central role in this game, acting as the on-board computer controlling a space station in an emergency situation. The players cannot communicate with each other directly. Instead, they interact with the game *environment* using their voice, and external changes to the environment may be communicated via voice output in addition to the graphical user interface and sound effects. Consequently, in addition to being the interface to the space station, the dialog system becomes a mediator between the players when they have to collaborate in order to achieve common goals. This kind of setup requires a flexible dialog system framework which supports (a) multiple users (b) speaking different languages and (c) which is able to interface well with the game world as well as (d) with the other software components. Platon's design meets all of these requirements. Its rapid prototyping capabilities proved to be a crucial feature for integrating the individual parts of the game as early as possible, including external ASR and TTS and world server components. Once the early

[5] `https://thrift.apache.org`

prototype stages had been established, Platon allowed a seamless progression to a more feature-rich dialog system. Figure 7 shows an example dialog from this game scenario.

S: Alert! Oxygen levels are decreasing rapidly in Module 0. Activate oxygen pump to prevent suffocation of crew members.
P: Then activate this pump?
S: Error 201. Can't turn on oxygen pump. Energy missing. Deactivate systems to have at least 20% energy.
P: Okay, which systems can we switch off?
S: Systems currently online: ...

Fig. 7. Excerpt of a dialog between system (S) and player (P) from the beginning of the adventure game.

To demonstrate Platon's suitability for other domains, we built a second dialog system for a home automation scenario. Here, we control a virtual apartment with a number of devices including lights, heating, door locks, etc. The user can query and manipulate the status of each of these devices. This system does not rely on an external NLU. Instead, the necessary functions for basic reference resolution and keyword spotting were implemented directly in Groovy. Platon's built-in object interaction support proved especially useful here, allowing us to easily react to opening doors or finished washing machines, etc. With custom classes and methods for the world objects it was possible to perform most environment manipulations in a single line of code.

6 Integration and Tools

Platon comes with command line and graphical tools to run and test dialog scripts. Both support input and output of written text, the GUI also has built-in support for speech synthesis[6] and speech recognition[7] and can automatically test a dialog system with prefabricated bulk input.

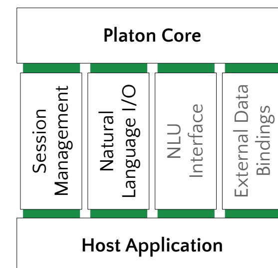

Fig. 8. Platon Interfaces (gray: optional)

To run a Platon dialog system outside this tool, a host application needs to manage sessions and take care of handling input and output, as illustrated in figure 8. The figure also includes the optional interfaces for natural language understanding and for interacting with world objects, as described in subsections 4.2 and 4.3. In addition to the direct Java-compatible interfaces, Platon provides additional Apache Thrift RPC interfaces to maximize the compatibility

[6] MaryTTS: http://mary.dfki.de/
[7] Sphinx: http://cmusphinx.sourceforge.net/

with non-JVM applications. When it is ready, a Platon application can be deployed as a single *jar* file including all dialog scripts.

7 Conclusions

We described Platon, a domain-specific language for dialog systems. Its focus ranges from rapid prototyping to the realization of fully-fledged dialog systems. Sophisticated input processing is implemented through a hierarchical task decomposition model based on agents for individual sub-tasks. Platon is agnostic toward the choice of underlying dialog management model as well as to the (semantic or dialog act) representation of system inputs and outputs. As it is based on Groovy, dialog scripts have ready access to third-party software written for the Java Virtual Machine. With two example systems, we further demonstrated how a Platon-based dialog system can interact with an application environment.

Platon is available under the Apache License on `https://github.com/uds-lsv/`.

8 Acknowledgments

The research presented in this paper has been funded by the Eureka project number E!7152. `https://www.lsv.uni-saarland.de/index.php?id=71`

References

1. Gropp, M.: Platon. Technical Report LSV TR 2015-002 (2015)
2. McTear, M.F.: Spoken dialogue technology: Enabling the conversational user interface. ACM Computing Surveys **34**(1) (March 2002) 90–169
3. Fabbrizio, G.D., Lewis, C.: Florence: a dialogue manager framework for spoken dialogue systems. In: Proceedings of Interspeech 2004, Jeju Island, Korea (2004) 3065–3068
4. Nyberg, E., Mitamura, T., Hataoka, N.: Dialogxml: extending voicexml for dynamic dialog management. In: Proceedings of the second international conference on Human Language Technology Research. (2002) 298–302
5. Leuski, A., Traum, D.: NPCEditor: Creating virtual human dialogue using information retrieval techniques. AI Magazine **32**(2) (2011) 42–56
6. Bohus, D., Rudnicky, A.I.: The RavenClaw dialog management framework: Architecture and systems. Computer Speech & Language **23**(3) (July 2009) 332–361
7. Bos, J., Klein, E., Lemon, O., Oka, T.: DIPPER: Description and formalisation of an information-state update dialogue system architecture. In: Proceedings of the 4th SIGdial Workshop on Discourse and Dialogue. (2003) 115–124
8. Larsson, S., Traum, D.: Information state and dialogue management in the trindi dialogue move engine toolkit. Natural Language Engineering **5**(3–4) (2000) 323–340
9. Traum, D.R., Larsson, S.: The information state approach to dialogue management. In: Current and new directions in discourse and dialogue. Springer, Netherlands (2003) 325–353