Domain-specific and General Syntax and Semantics in the Talkamatic Dialogue Manager

Staffan Larsson • Alexander Berman

Abstract We present a design philosophy for dialogue system development, where domain-specific domain knowledge is clearly separated from the logic for generic dialogue capabilities. We hope that this provides a useful illustration of how one may approach the division of labour between general and domain-specific syntax, semantics and pragmatics.

Keywords dialogue · dialogue systems · syntax · semantics · dialogue management · grammar

S. Larsson, University of Gothenburg, `sl@ling.gu.se`
A. Berman, Talkamatic AB, `alex@talkamatic.se`

In Christopher Piñón (ed.), Empirical Issues in Syntax and Semantics 11, 91–110. Paris: CSSP.
`http://www.cssp.cnrs.fr/eiss11/`

1 Introduction

This paper outlines a format, currently under development, for specifying *Dialogue Domain Descriptions* (DDD) for a domain-independent dialogue system, the Talkamatic Dialogue Manager (Larsson et al. 2011a,b). One of the central principles underlying the design of TDM is the separation of domain-specific knowledge from general dialogue capabilities. We hope that this provides a useful illustration of how one may approach the division of labour between general and domain-specific syntax, semantics and pragmatics in a dialogue system.

These practical considerations may also be of interest from a more theoretical perspective. One may conjecture that general principles that provide the basis for a useful dialogue systems design also says something about the nature of the human ability to participate in natural language dialogue. Exactly how to pin down the relation between dialogue systems and human linguistic competence, however, is a delicate matter. It is important to be aware that the usefulness of some design principles may be due to the needs of dialogue application designer in being able to quickly construct (and debug) dialogue system applications, something which has no counterpart with regard to human linguistic competence.

2 The Talkamatic Dialogue Manager

The Talkamatic Dialogue Manager (TDM) is a commercial platform for building spoken dialogue systems. It is a reimplementation and development of the GoDiS/IBiS (Gothenburg Dialogue System/Issue-Based Information System) system described in Larsson 2002, developed using the information state update approach to dialogue management (Traum & Larsson 2003), which takes considerable input from the KoS (conversation-oriented semantics) framework (Ginzburg 2012).

TDM consists of the following runtime subcomponents:

- Frontend: mainly consisting of an automatic speech recogniser and a text-to-speech synthesiser.
- Backend: consisting of a dialogue move engine, natural-language interpreter, etc.
- Session Manager: providing each frontend with access to a backend, and routing communication between frontend and backend.

TDM also contains design-time subcomponents constituting an SDK (software development kit) for developing DDDs. Dialogue domains consist of the following parts:

- An ontology defining concepts, entities and actions that the user and the system may reference in questions, answers and requests.
- Domain knowledge in the form of dialogue plans (and related notions), describing how actions are carried out and how questions are answered. Plans also describe what information is needed in order to carry out the actions or to answer the questions.
- A language model or grammar, describing words and utterances used by the user and system. In other words, the language model defines syntax rules and mappings between linguistic surface forms and semantic entities.
- A service interface describing how services that the domain depends on are accessed and used, for example web APIs or functionality

hosted natively on the user's device.

3 Dialogue Design

TDM is the result of an effort to build a dialogue manager on sound engineering principles, exploiting knowledge from research about human dialogue. The following principles have guided TDM design:

- Apply general solutions to general problems
- Don't mix different kinds of knowledge

These principles have led to an architecture where knowledge about the domain (e.g., telephony or navigation) is separated from general knowledge about dialogue. This means that app developers can focus on defining domain-specific knowledge, such as information about concepts and which words are used to talk about the concepts. General dialogue capabilities such as asking questions, giving answers and providing feedback are built into the dialogue manager and do not need to be provided by app developers. This facilitates building apps since general dialogue strategies need not be reinvented each time a new dialogue is built. Thus, the developer can focus on app-specific development.

For example, one can consider a simple app enabling the user to make phone calls. The developer specifies that calling contacts is an action which requires the system to know who to call. The developer also specifies that the system asks about this information with the sentence "Who do you want to call?" Based on this domain knowledge, TDM will choose to ask the question "Who do you want to call?" whenever the information is required. It may also choose to repeat the question when motivated, or to refrain from asking the question if the answer has been provided without prompting the user. In other words, the overall logic governing the dialogue is contained within TDM, while domain-specific knowledge, such as dependencies between various kinds of information in the form of dialogue plans, are kept in the DDDs.

The same principle of division of labour holds for the surface forms of user and system utterances. General forms for dialogue moves are specified in a domain-independent grammar, which is then fleshed out by a domain-specific grammar which supplies the surface forms associated with domain-specific entities, predicates and actions.

When an app developer builds a new app, there is no need to extend or modify the dialogue manager as such – only domain-specific knowledge needs to be supplied. The idea is that since dialogue design is built into the TDM, it should be easy to produce usable, well-designed dialogue interfaces. Natural and flexible dialogue flow is a built-in feature which comes for free when specifying the dialogue plans. The built-in feedback model ensures that user and system are on the same track. A rich contextual model is available for intelligent interpretation of speech recognition results, as well as providing information for disambiguation of unclear utterances. Additionally, language models (grammars) are described at a high level of abstraction in a simple format, which makes it easier for non-linguists to build and localise apps.

However, it is important to note that TDM cannot participate in arbitrarily complex dialogue. Roughly, TDM covers dialogues requiring the system to provide some information to a user or to perform some action, and to do this, the system needs to collect certain bits of information from the user. Information search can be incremental in that a range of options is manipulated in successive steps until one option is chosen by the user. For example, if the user asks the system to play a radio program, the system will ask the user to specify parameters such as genre (music, news, sports, etc.) and channel (in Swedish radio: P1, P2, P3, local channels, etc.). Whenever the user specifies or modifies a parameter, the range of options listed changes. When the user finally selects one of the options, the program starts playing. During the interaction, the user can behave fairly freely, and for example switch to other conversational topics and return to the radio program topic without the system losing context. However, TDM is currently *not* designed to handle, for example, complex planning tasks where several plans to achieve a goal are compared and argued for and against. Nor can it handle purely socially oriented dialogue that has no concrete task other than maintaining social relations.

TDM's built-in dialogue design enables more complex interactions than most other dialogue managers on the market, while keeping a fairly simple dialogue design format. The subsequent subsections describe some of the general dialogue features in more detail.

3.1 Flexible Dialogue Flow

Given a simple dialogue plan specifying a default dialogue flow which achieves a given goal, TDM manages a wide variety of dialogue flows to achieve the goal. If the user simply responds to system questions, the dialogue will follow the default flow. But if the user chooses to give more or other information than requested, or takes initiative to talk about something else, TDM adapts to this. The user may even just provide some information which is relevant to him or her at the moment, and TDM will either figure out what the user wants to do, or ask a clarification question to move the dialogue along. It is also possible to revise answers without having to restart the dialogue.

Some aspects of flexible dialogue flow are present in systems like Siri and Google Now, and some are supported by the VoiceXML standard. However, compared to most other systems TDM offers a relatively wide and complete range of flexible dialogue behaviours. To take one example, if the user changes the subject while talking to Siri, the previous topic will be forgotten by the system. In contrast, once the embedded topic has been finished, TDM will switch back to the previous topic and signal this explicitly. This means that if the user asks, for example, about the weather while checking bus routes to a specific destination, TDM provides the weather information and then returns to the previous activity by saying "Returning to selecting a route." The surface form for such dialogue moves, which indicate *sequencing*, are defined in the domain-independent grammar component.

3.2 Feedback

TDM features a fairly elaborate feedback model to cope with communication problems. Feedback (positive and negative) and clarification questions are given on several levels. For example, assuming that a user said "Anna" to a telephony app, the system could give feedback regarding perception ("I didn't hear anything from you," "I heard you say Anna, is that correct?"), semantics ("I don't understand," "OK, Anna"), intentions ("OK, you said Anna. Do you want to make a call?") and acceptance/rejection ("I don't have a phone number for Anna"). All feedback utterances are defined in a domain-independent grammar component.

Pinpointing communication errors and clarifying potential misunderstandings means better chance of dialogue success. However, excessive feedback may lead to inefficient dialogues and dissatisfied users, which is why speech recognition and interpretation leads to various contextual factors. These contextual factors in turn may be helpful in perceiving and understanding user utterances.

4 Dialogue Domain Descriptions

This section gives a brief overview of the XML format for DDDs. The format supports ontology, plans, grammar and interaction tests. The last part (service interface) will be addressed in future work. Note that this section is not intended as a manual for building DDDs, and only provides a partial description of the DDD XML format. Code excerpts are taken from the Talkamatic GitHub repository, where the complete example DDD can be found.[1]

4.1 Ontology and Semantics

Ontology works as a TDM's table of all entities and actions that a specific application talks about. The following kinds of entities are defined by the ontology, also described by the example in figure 1.

- *Sorts* (general or domain-specific) ontological categories which are used to enforce sortal constraints on semantic representations, and to guide interpretation and generation.
- *Individuals* which include all single entities that the app can talk about (e.g., contacts). Individuals can be declared explicitly in the ontology. For example, in the phone domain an individual `Anna` can be declared to be of the sort `contact`. Alternatively, a sort can be declared as `dynamic`, which means individuals of that sort are created dynamically in runtime by consulting the service interface.
- *Predicates* are used for representing propositions and questions (represented in XML using only predicates). Each (one-place) predicate declares the required sort of its argument (or value). For example, the argument of the `contact_to_call` predicate can only be a `contact`, thus `Anna` will be a valid argument.

[1]See the site `https://github.com/Talkamatic/dialogue-domain-descriptions/tree/master/android/android`.

```
<ontology name="PhoneOntology">

  <action name="call"/>

  <sort name="contact" dynamic="true"/>
  <sort name="phone_number" dynamic="true"/>

  <predicate name="phone_number_of_contact" sort="phone_number"/>
  <predicate name="selected_contact_to_call" sort="contact"/>
  <predicate name="selected_contact_of_phone_number" sort="contact"/>

</ontology>
```

Figure 1 Ontology for the phone domain

- *Actions* that TDM can be requested to carry out, typically by calling the service interface.

The elements defined in the ontology are used in domain-specific semantic representations in TDM. The account in Larsson 2002 uses a very simple representation of propositions loosely based on predicate logic (without quantification). This is extended this with lambda-abstraction of propositions and a question operator "?" which can be thought of as a function from a (possibly lambda-abstracted) proposition to a question. Furthermore, Larsson (2002) uses a (domain-independent) semantic category to account for the content of short answers (e.g., "yes" or "Paris"). This representation is also the basis for the semantic representations currently used in TDM.

Propositions correspond roughly to basic formulae of predicate logic consisting of an 0-ary or 1-ary predicate together with constants representing its arguments, for example **return-trip** (0-ary predicate) and **dest-city(paris)** (using an 1-ary predicate).

- $Expr$: Proposition if
  - $Expr$: Pred_0 or
  - $Expr = pred_1(arg)$, where arg : Ind and $pred_1$: Pred_1 or
  - $Expr = \neg P$, where P : Proposition

In a dialogue system operating in a domain of limited size, it is often not necessary to keep a full semantic representation of utterances. For example, a user utterance of “I want to go to Paris” could be represented semantically as, for example, **want(user, go-to(user, paris))** or **want(u, go-to(u,p)) & city(p) & name(p, paris) & user(u)**. TDM uses a reduced semantic representation with a coarser, domain-dependent level of granularity; for example, the above example will be rendered as **dest-city(paris)**. This reduced representation reflects the level of semantic granularity inherent in the underlying domain task. As an example, in a travel agency domain there is no point in representing the fact that it is the user (or customer) rather than the system (or clerk) who is going to Paris; it is implicitly assumed that this is always the case.

As a consequence of using reduced semantics, it will be useful to allow 0-ary predicates, for example **return-trip** meaning “the user wants a return ticket”. Furthermore, so far we have not found reason to move beyond unary (1-place) predicates in TDM. We conjecture that this is due to the structure of the kind of dialogue that TDM can currently engage in, where propositions are essentially equivalent to feature-value pairs. An interesting question is how far one can get with one-place predicates, and when this breaks down. One hypothesis is that binary predicates will be needed as soon as there is a need to talk about several entitities of the same kind (flights, for example), which have different properties (e.g., travel time, number of stops, price, etc.). This happens, for example, in negotiative dialogue of the sort described in Larsson 2002.

The advantage of the semantic representation used in TDM is that the specification of domain-specific semantics becomes simpler, and that unnecessary “semantic clutter” is avoided. However, it does have limited expressive power and would need to be extended to deal with more complex genres of dialogue requiring a more fine-grained semantics, for example by adding binary and perhaps n-ary ($n > 2$) predicates.

Three sorts of questions are treated by TDM: *y/n*-questions, *wh*-questions, and alternative questions.

- $Expr$: Question if $Expr$: YNQ or $Expr$: WHQ or $Expr$: ALTQ
- $?P$: YNQ if P : Proposition
- $\mathbf{?}x.pred_1(x)$: WHQ if x : Var and $pred_1$: Pred$_1$
- $\mathbf{\{}?ynq_1, \ldots, ?ynq_n\mathbf{\}}$: ALTQ if ynq_i : YNQ for all i such that $1 \leq i \leq n$

In TDM semantics, *y/n*-questions correspond to propositions preceded by a question mark, for example **?dest-city(london)** ("Do you want to go to London?"). *Wh*-questions correspond to lambda-abstracts of propositions, with the lambda replaced by a question mark, for example **?*x*.dest-city(*x*)** ("Where do you want to go?"), and alternative questions are sets of *y/n*-questions, for example **{?dest-city(london), ?dest-city(paris)}** ("Do you want to go to London or do you want to go to Paris?"). Here, TDM semantics goes beyond standard predicate logic. Note, by the way, that we do not provide a model theoretic semantics for this notation. While this would be fairly straightforward (possibly with some minor complications related to the semantics of questions), we see no clear role for such a semantics in a dialogue system, except possibly as a tool for ensuring consistency and orderliness. The use of the term "semantics" for these representations is motivated, rather, from their role in providing a structure for the domain which is used for mediating between natural language utterances (from both user and system) and the underlying service interface.

Ginzburg uses the term "short answers" for phrasal utterances in dialogue such as "Paris" as an answer to "Where do you want to travel?" in a travel agency setting. These are standardly referred to as *elliptical* utterances. Ginzburg argues that (syntactic) ellipsis, as it appears in short answers, is best viewed as a semantic phenomenon with certain syntactic presuppositions. That is, the syntax provides conditions on what counts as a short answer but the processing of short answers is an issue for semantics.

We follow this in seeing short answers from a semantic point of view. What this means, in effect, is that we are not interested in syntactic ellipsis, but rather in semantic underspecification of a certain kind. Furthermore, the semantics used by the system is domain-dependent and thus what we are really interested in is semantic underspecification *with regard to the domain/activity*. On this account, an utterance is semantically underspecified iff it does not determine a unique and complete proposition in the given activity. Of course, this means that whether an utterance is regarded as underspecified or not depends on the granularity of propositional content, and what types of entities are interesting in a certain activity. For example, given the type of simple semantics that we are using on our sample travel agency domain, "to Paris" is not semantically elliptical, since it determines the complete proposition **dest-city(paris)**. However, "to Paris" would be semantically underspecified in an activity where it could also be taken to mean for example "You should go to Paris."

- *Expr* : ShortAns if
  - *Expr* = **yes** or
  - *Expr* = **no** or
  - *Expr* : Ind or
  - *Expr* = $\neg arg$ where *arg* : Ind

In general, semantic objects of type ShortAns can be seen as underspecified propositions. In TDM, we only deal with individual constants (i.e., members of Ind), and answers to *y/n*-questions, i.e., **yes** and **no**. Individual constants can be combined with *wh*-questions to form propositions, and **yes** and **no** can be combined with *y/n*-questions.

Note that we allow expressions of the form ¬arg where *arg*:Ind as short answers. This is used for representing the semantics of phrases like "not Paris." In a more developed semantic representation these expressions could be replaced by a type-raised expression, for example $\lambda P.\neg P(arg)$.

Questions and answers can be combined to form propositions, as shown in table 1. The special case for *wh*-questions is similar to functional application, as when the question **?**x**.dest-city(**x**)** is combined with **paris** to form **dest-city(paris)**. Questions can also be combined with propositions, yielding the same propositions as result provided the question and the propositions have the same predicate and that the proposition is sortally correct. It is also possible to combine *y/n*-questions and alternative questions with answers to form propositions. In general, we say that a question q and an answer a *combine* to form a proposition p. Related definitions of answers being *relevant* to and *resolving* questions are given in Larsson 2002.

4.2 Dialogue Plans

Plans include information about how the dialogue with the user should progress. Figure 2 shows an example. TDM requires a top plan (`action = "top"`) declaring what the system should at the outset of each interaction. In general, plans are identified by their goals, that is, things that the plan shall have done by its completion. There are two types of goals in TDM:

Question	Answer	Proposition
$?x.pred_1(x)$	a or $pred_1(a)$ $\neg a$ or $\neg pred_1(a)$	$pred_1(a)$ $\neg pred_1(a)$
$?P$	**yes** or P **no** or $\neg P$	P $\neg P$
$\{?P_1, ?P_2, \ldots, ?P_n\}$	$P_i, (1 \leq i \leq n)$ $\neg P_i, (1 \leq i \leq n)$	P_i $\neg P_i$

Table 1 Combining questions and answers into propositions

resolving a question, and performing an action. These are represented as follows (the corresponding XML representations can be seen in figure 2):

- **resolve(**q**)** where q:Question
- **perform(**α**)** where α:Action

A plan tag includes a goal and all the steps that are needed to be done to accomplish the goal. Such a step can be the `findout(`q`)` item which tells TDM that the question q needs to be resolved. For example, within the `call` goal, the `findout` statement instructs TDM to resolve a wh-question formed by the `selected_contact_to_call` predicate). A `dev_perform` item signifies that TDM has to execute an action external to the dialogue, for example making a call, sending an SMS or updating a database. The execution of external actions is specified in the service interface. Similarly, `dev_query` is like `dev_perform` except that it specifies a question, prompting TDM to await an answer to the question to be returned from the service interface.[2]

4.3 Grammar

TDM takes hybrid template/grammar approach to natural language generation and interpretation, where grammatical knowledge is used to minimize the work involved in developing and localising an application to a new language. Domain-specific linguistic knowledge, defined by the app

[2]The TDM service interface definition is currently being converted into XML format, and we will not describe it further here. Suffice to say that the service interface needs to define all the queries and actions (defined using `dev_query` and `dev_perform`) that are included in the dialogue plans, as well as some related knowledge.

```
<domain name="PhoneDomain" is_super_domain="true">

  <goal type="perform" action="top">
    <plan>
      <forget_all/>
      <findout type="goal"/>
    </plan>
  </goal>

  <goal type="perform" action="call">
    <plan>
      <findout type="wh_question" predicate="selected_contact_to_call"/>
      <dev_perform action="Call" device="AndroidDevice" postconfirm="true"/>
    </plan>
    <postcond><device_activity_terminated action="Call"/></postcond>
  </goal>

  <goal type="resolve" question_type="wh_question" predicate="phone_number_of_contact">
    <plan>
      <findout type="wh_question" predicate="selected_contact_of_phone_number"/>
      <dev_query device="AndroidDevice" type="wh_question"
                 predicate="phone_number_of_contact"/>
    </plan>
  </goal>

</domain>
```

Figure 2 Domain knowledge for phone domain (excerpt)

developer, is kept separate from other domain knowledge and from general linguistic knowledge built into TDM.

The grammar specifies a language model which consists of mappings between linguistic surface forms (primarily text strings) and semantic entities relating to the ontology, such as individuals, actions and questions. A single grammar specifies both user and system utterances, thus promoting consistency between what the system can say and what it can understand.

The grammar used by a TDM application is a combination of a domain-specific grammar (such as that shown in figure 3) and a general TDM grammar, which specifies the general form of the main TDM dialogue moves (`ask`, `answer`, `request`, `confirm` and `greet`) as well as for feedback moves. This means that the domain-specific grammar can be kept to a minimum.

The first entry in figure 3 (`action name="call"`) expresses that the action **call** can be referred with a verb phrase containing the verb *call*. It also contains a lexicon describing the grammar of call in English. We only

```
<grammar>

  <action name="call">
    <verb-phrase>
      <verb ref="call"/>
    </verb-phrase>
  </action>

  <lexicon>
    <verb id="call">
      <infinitive>call</infinitive>
    </verb>
  </lexicon>

  <request action="call">
    <utterance>
      <one-of>
        <item>make a call</item>
        <item>call <individual sort="contact"/></item>
      </one-of>
    </utterance>
  </request>

  <question speaker="system" predicate="selected_contact_to_call" type="wh_question">
    <utterance>who do you want to call</utterance>
  </question>

  <predicate name="phone_number_of_contact">
    <noun-phrase>
      <noun ref="number"/>
    </noun-phrase>
  </predicate>

  <question speaker="user" predicate="phone_number_of_contact">
    <utterance>
      <one-of>
        <item>tell me a phone number</item>
        <item>what is <individual sort="contact"/>'s number</item>
        <item>tell me <individual sort="contact"/>'s number</item>
      </one-of>
    </utterance>
  </question>

  <answer speaker="system" predicate="phone_number_of_contact">
    <utterance>
      <individual predicate="selected_contact_of_phone_number"/> has number
        <individual predicate="phone_number_of_contact"/>
    </utterance>
  </answer>

</grammar>
```

Figure 3 Grammar for the phone domain (excerpt)

need to specify the infinitive form for the verb. The other forms, such as imperative and present progressive, are derived automatically from the general grammar resource for English.

The domain-general grammar also states that actions can be requested by using the imperative form that refers to the action, in this case “Call.” When the system asks what the user wants to do, it may use the infinitive form, as in “Do you want to call?” The present progressive can be used by system confirmations, for example “Calling” or “Calling Anna.”

Since these basic forms may not always be sufficient, additional forms can be declared in the domain-specific grammar. Multiple alternative forms can be provided using the `<one-of>` tag which encloses alternatives as `<item>`s. See the third entry in figure 3 for an example.

As can be seen by the second `item` in the third entry in figure 3, TDM grammar entries may use placeholders for individuals, represented by the `<individual>` tag. This tag specifies a predicate, which in generation is instantiated with the surface form of the individual that the predicate holds of. In interpretation, a slot is similarly expected to be instantiated with the surface form of the individual that the predicate holds of. The rest of the grammar excerpt in figure 3 declares how the system is to ask questions about who to call; how the user may ask for the phone number of a contact; and how the system may answer questions about phone numbers of a given contact.

Given a grammar such as that shown in figure 3 (extended with some additional surface forms for other actions), TDM can generate and understand a range of utterances combining elements from a domain-specific grammar (in dark green) and the domain-independent resource grammar provided by TDM (in blue):

- System alternative question: Do you want to make a call or get the number of a contact?
- System wh-question: Who do you want to call?
- System report: Calling; Calling Anna
- System feedback: I heard you say Anna, is that correct?; OK, Anna
- System topic management: Returning to calling
- User request: I want to make a call; Call Anna, I want to call Anna; Would you please call Anna

- User answers: Anna
- User feedback: Pardon?, Please repeat

Above, we have only shown excerpts from the English grammar. Complete English, French and Dutch grammars for the example domain are available from GitHub.[3]

5 Domain-independent Knowledge in TDM

Apart from the rules and algorithms governing dialogue management, which are general within the confines of the kind of action- and issue-oriented dialogue that TDM was designed for, domain-independent knowledge in TDM includes the following:

- The types of dialogue moves that speakers can perform, and the kinds of semantic entities they take as arguments (e.g., `ask` moves take questions)
- Information state update rules and algorithms governing dialogue management, including rules connecting dialogue moves to information state updates
- The format for sentence-level semantic entities such as propositions and questions, and their relation to the domain-specific predicates, entities and actions
- General and abstract semantic relations between questions and answers, such as whether an answer is *relevant* to, *resolves*, or *combines with* a question, defined in terms of semantics, and used to define update rules
- General surface forms and patterns which are used together with domain-specific grammars for parsing and generating utterances, thus connecting them to the TDM dialogue moves

In the information-state approach, the precise semantics of a dialogue move type is determined by the update rules which are used to integrate moves of that type into the information state. This means that all occurrences of a move type are integrated by the same set of rules. The update rules (and associated algorithms) used in the GoDiS/IBiS system,

[3] See the site `https://github.com/Talkamatic/dialogue-domain-descriptions/tree/master/android/android/grammar`.

and forming the starting point for the rules used in TDM, are descibed in Larsson 2002.

While dialogue move types are often defined in terms of sentence mood, speaker intentions, and/or discourse relations (Core & Allen 1997), we opt for a different solution. In our approach, the type of move realized by an utterance is determined by the relation between the content of the utterance, and the activity in which the utterance occurs. For example, if an utterance provides information which is relevant to a question in the domain, it is regarded as realizing an answer move (regardless of whether the question has been asked).

The following dialogue moves are used in TDM:

- ask(q), where q : Question
- answer(a), where a : ShortAns or a : Proposition
- request(α), where α : Action
- report(α, σ), where α : Action and σ : Status is the status of the action (one of **started**, **ended**, and **failed**)
- greet
- quit

In inquiry-oriented dialogue, the central dialogue moves concern raising and addressing issues. This is done by the ask and answer moves, respectively. For action-oriented dialogue, the request and report moves are added to enable requesting and reporting on the status of actions. The greet and quit moves are used in the beginning and end of dialogues to greet the user and indicate that the dialogue is over, respectively.

6 Semantic Coordination in Dialogue Systems

Cooper & Ranta (2008) propose a shift in perspective from the view of natural languages as formal languages to natural languages as a collection of resources for constructing local languages for use in particular situations. They point to a research programme investigating how such resources play a role in linguistic innovation by agents constructing situation-specific local languages and how they can be made dynamic, modified by the linguistic agent's exposure to innovative linguistic data. This is related to a prominent problem in current dialogue systems, namely, the fact that users are constrained to a static pre-programmed language – what Brennan (1998) refers to as the *vocabulary problem in spoken dialogue systems*.

Present-day dialogue systems require users to talk in ways foreseen by programmers. This makes systems less useful and may lead to increased cognitive load on user, making systems potentially dangerous to use, for example while driving. When exposed to unexpected formulations, language understanding in a dialogue system will break down. By contrast, when exposed to unexpected formulations, people are capable of *semantic coordination* (Larsson 2015), either by (silently) figuring out (based on linguistic and contextual clues) a plausible meaning and updating their own take on how language is used in the current context, or by interactive clarification and meaning negotiation.

Eventually, we will want to enable dialogue systems to handle semantic coordination, which requires the ability to adapt old meanings and learn new ones, and clarify and negotiate meanings in metalinguistic dialogue. The kind of semantics used in present-day dialogue systems capture only a fraction of the natural language meanings of the words in the grammar. A simple addition would be to allow adding new ways of referring to known individuals, predicates, etc. However, semantic coordination will be more useful when meaning representations have more structure and where more reasoning is performed.

In this context, a possible conjecture with respect to learning vs. programming of domain-dependent and domain-independent knowledge about syntax, semantics and pragmatics could be that only domain-specific knowledge need to be learnable, whereas domain-general knowledge can be preprogrammed. (Pre-programmed pragmatics will include strategies and dialogue acts for engaging in semantic coordination.) The intuition behind this conjecture is that while language is continually adapted by speakers to specific domains, the general linguistic resources that underpin this adaptation change at a pace that, for the purposes of dialogue systems development, can be handled on an engineering level without excessive cost. Further support can perhaps be found in the observation regarding human speakers is that while we tend to have no problem adapting our language to new domains and new dialogue partners, we frequently resist (and even protest) changes to our shared general vocabulary and grammar.

7 Multilinguality and Domain-specific Grammar

Regarding the division of labour at the level of syntax and semantics, general forms for dialogue moves are specified in a domain-independent grammar. This grammar is defined using GF (Grammatical Framework) Resource Grammar Library (Ranta 2004). The general grammar is then fleshed out by the domain-specific grammar (written in XML), which supplies the surface forms associated with domain-specific entities, predicates and actions. The XML format allows taking advantage of GF resource grammars without knowing GF.

A major benefit of GF is that it provides resource grammars for a large number of languages, which simplifies localization of dialogue system applications to new languages. An interesting question arises here with respect to how language-dependent the semantics implemented in the domain and grammar is. It is well-known that languages differ with respect to their semantic categories, but arguably many of these differences are at the level of language in general rather than at the level of specific domains. When building dialogue system applications and porting them to new languages, it is often implicitly assumed that activities and domains are invariant across languages. Insofar as this is true, it may be that differences between languages at the general (domain-independent) level are more or less cancelled out in the process of adapting the language to the domain (either by design or through interaction).

Still, it may be that different languages will be differently equipped to handle certain domains, insofar as semantic distinctions in each domain derive from general distinctions in the language. If this is true, this means that the process of achieving a domain language for a domain may differ between languages, and may be easier for some domain + language pairs than others. At the present time, this is just a speculation, but if (when) future dialogue systems become able to interactively coordinate on new meanings and learn from experience how to talk about new activities, it will become a testable hypothesis.

8 Conclusions

We have illustrated a design philosophy for dialogue system development, where domain-specific domain knowledge is clearly separated from the logic for generic dialogue capabilities. We hope that this provides a useful illustration of how one may approach the division of labour between general and domain-specific syntax, semantics and pragmatics in a dialogue system. We also briefly discussed issues of multilinguality and the possibility of dialogue systems learning (rather than being programmed), and coordinating with users on, domain-specific meanings.

Acknowledgments We are very grateful to the anonymous reviewer and to the editor for providing useful and insightful comments and suggestions. This work was supported by the Centre for Language Technology (CLT) and the Centre for Linguistic Theory and Studies in Probability (CLASP) at Gothenburg University, and partially financed by the European Commission under the FP7-ICT-Project ALFRED (grant agreement no. 611218).

References

Brennan, Susan E. 1998. The vocabulary problem in spoken language systems. In Susann LuperFoy (ed.), *Automated spoken dialog systems*, Cambridge, MA: The MIT Press. `http://www.psychology.sunysb.edu/sbrennan-/papers/luperfoy.pdf`.

Cooper, Robin & Aarne Ranta. 2008. Natural languages as collections of resources. In Robin Cooper & Ruth Kempson (eds.), *Language in flux: Relating dialogue coordination to language variation, change and evolution*, 109–120. London: College Publications.

Core, Mark G. & James F. Allen. 1997. Coding dialogues with the DAMSL annotation scheme. In David Traum (ed.), *Working Notes: AAAI Fall Symposium on Communicative Action in Humans and Machines*, 28–35. Menlo Park, CA: American Association for Artificial Intelligence.

Ginzburg, Jonathan. 2012. *The interactive stance*. Oxford University Press.

Larsson, Staffan. 2002. *Issue-based dialogue management*. Göteborg, Sweden: Göteborg University dissertation.

Larsson, Staffan. 2015. Formal semantics for perceptual classification. *Journal of Logic and Computation* 25(2). 335–369.

Larsson, Staffan, Alexander Berman & Jessica Villing. 2011a. Adding a speech cursor to a multimodal dialogue system. In *INTERSPEECH 2011, 12th Annual Conference of the International Speech Communication Association, Florence, Italy, 2011*, 3319–3320.

Larsson, Staffan, Alexander Berman & Jessica Villing. 2011b. Multimodal menu-based dialogue with speech cursor in Dico ii+. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Lan-*

guage Technologies: Systems Demonstrations HLT ’11, 92–96. Stroudsburg, PA: Association for Computational Linguistics.

Ranta, Aarne. 2004. Grammatical framework: A type-theoretical grammar formalism. *The Journal of Functional Programming* 14(2). 145–189.

Traum, David & Staffan Larsson. 2003. The information state approach to dialogue management. In Jan van Kuppevelt & Ronnie W. Smith (eds.), *Current and new directions in discourse and dialogue*, 325–353. Kluwer Academic Publishers.