Cognitive Systems Research 92 (2025) 101378

Contents lists available at ScienceDirect

Cognitive Systems Research

journal homepage: www.elsevier.com/locate/cogsys

Towards knowledge autonomy in the Companion cognitive architecture☆

Constantine Nakos *, Kenneth D. Forbus

Qualitative Reasoning Group, Northwestern University, 2233 Tech Drive, Evanston, IL 60208, USA

ARTICLE INFO

Action editor: Serge Thill

Keywords:
Companion cognitive architecture
cognitive architecture
knowledge management
knowledge representation
knowledge base
provenance

ABSTRACT

One of the fundamental aspects of cognitive architectures is their ability to encode and manipulate knowledge. Without a consistent, well-designed, and scalable knowledge management scheme, an architecture will be unable to move past toy problems and tackle the broader problems of cognition. Moreover, it will not be able to reach a state of *knowledge autonomy*, in which the architecture has the tools it needs to acquire and maintain knowledge on its own. In this paper, we document some of the challenges we have faced in developing the knowledge stack for the Companion cognitive architecture and discuss the tools, representations, and practices we have developed to overcome them. We also lay out a series of next steps that will allow Companions to play a greater role in managing their own knowledge, an important part of knowledge autonomy. It is our hope that these observations will prove useful to other cognitive architecture developers facing similar challenges.

1. Introduction

Cognitive architectures are an infrastructure for intelligent systems, specifying those aspects of cognition that remain fixed over time and across domains (Laird et al., 2017; Langley et al., 2009). In order for a cognitive architecture to produce intelligent behavior, it requires knowledge to fuel its reasoning and guide its behavior. Complex tasks such as question answering (Crouse, 2021), moral reasoning (Dehghani, 2009; Olson & Forbus, 2023), visual understanding (Chen, 2023), tutoring students (Anderson & Gluck, 2013), simulating patients for medical training (McShane & Nirenburg, 2021), learning new games (Hinrichs & Forbus, 2014; Kirk & Laird, 2014), or playing old ones (Hancock, Forbus, & Hinrichs, 2020; Hancock & Forbus, 2021; Hinrichs & Forbus, 2011) underscore the need for rich knowledge across a wide variety of domains. The specific format of the knowledge a cognitive architecture uses will depend on its design, its core claims, and the way it is deployed. These constraints will also shape best practices for authoring, storing, and accessing the knowledge for a particular architecture, and these practices will evolve over time as the architecture scales up, changes, and develops new capabilities.

The Companion cognitive architecture (Forbus & Hinrichs, 2017) is undergoing just such a transition. Recent work on knowledge extraction (Ribeiro, 2023) has laid the groundwork for automatically acquiring world knowledge that can be reused across domains, while the ongoing SocialBot project, an online expansion of the information kiosk described by Wilson et al. (2019), has required new knowledge management capabilities to support Companions in their role as social agents. For the Companion architecture to reach the next stage of its development, it will need a new set of tools and practices for knowledge management, enabling the agent itself to play a larger role in its own learning.

With new developments on the horizon, we take this opportunity to document the Companion knowledge stack[1] as it currently stands, centered around the challenges we have faced and how we have addressed them. We discuss three major challenges: 1) *knowledge representation*, the design problem of encoding knowledge in a format amenable to reasoning, 2) *knowledge access*, the practical problem of efficiently making the knowledge available to a running Companion, and 3) *knowledge management*, the problem of handling knowledge and saving it for future use, in particular the tools and practices a Companion needs to handle knowledge on its own.

Our solutions to these challenges lay a strong foundation for reaching the ultimate goal of *knowledge autonomy*, in which the Companion is responsible for acquiring and maintaining its own knowledge. Prior Companions research has focused on learning mechanisms (e.g., Kandaswamy & Forbus, 2012), an important component of autonomy, but

☆ This article is part of a special issue entitled: 'RACS' published in Cognitive Systems Research.

* Corresponding author.

E-mail addresses: cnakos@u.northwestern.edu (C. Nakos), forbus@northwestern.edu (K.D. Forbus).

[1] We use "stack" here in the sense of a software stack, i.e. software components arranged in layers. The Companion knowledge stack is the set of modules and abstractions which enable a Companion to manipulate knowledge.

https://doi.org/10.1016/j.cogsys.2025.101378
Received 28 November 2024; Received in revised form 30 April 2025; Accepted 2 July 2025
Available online 5 July 2025

comparatively little attention has been paid to the question of what should be done with the knowledge once it is learned. Now that there are several promising ways for a Companion to autonomously acquire knowledge, we explore how the Companion knowledge stack supports this endeavor and what extensions are necessary to reach true autonomy.

While our discussion is grounded in Companions, we expect that similar challenges will arise in any cognitive architecture that attempts to reason and learn at scale. As the architecture grows robust enough to tackle real-world problems, it will need new ways to manage the large amounts of knowledge involved. Thus, our goal in this paper is not just to checkpoint the Companion knowledge stack, but to share lessons learned in its development and foster a discussion on how to equip cognitive architectures for the complexities of managing their own knowledge and learning.

The remainder of the paper is structured as follows. We begin with an overview of the Companion cognitive architecture, its design principles, and two applications that have driven recent changes to its knowledge stack. Next, we discuss the stack itself, reviewing its established capabilities and presenting its new additions. We ground our discussion in three challenges—knowledge representation, access, and management—that the stack has grown to address. We then discuss the next steps we plan to take towards knowledge autonomy in the Companion cognitive architecture. We conclude with an overview of related work and some closing thoughts, including our suggestions for other cognitive architecture developers.

2. Companions

The main purpose of the Companion cognitive architecture (Forbus & Hinrichs, 2017) is to learn how to build *software social organisms*—software systems that can learn, reason, and interact with people the way they do—as a step towards achieving human-level AI (Forbus, 2016). In support of this, Companions are equipped with a wide range of capabilities. Input modalities such as sketching and language allow the Companion to encode the kinds of inputs people use. Hierarchical task networks allow a Companion to plan out its behaviors, including complex reasoning about which actions to take. A robust analogy stack supports mapping (Forbus et al., 2016), retrieval (Forbus, Gentner, & Law, 1995), and generalization (Kandaswamy & Forbus, 2012).

Every Companion consists of several agents operating in parallel, each one responsible for handling different tasks. For example, the Interaction Manager can parse a question asked by the user, then pass the query off to the Session Reasoner to determine the answer. Other agents handle capabilities like sketch understanding (Forbus et al., 2011) or strategic reasoning (Hancock & Forbus, 2021). This architecture provides coarse-grained parallelism and allows a Companion's capabilities to be expanded as needed by creating new types of agents.

Fig. 1 shows an example Companion session for the SocialBot domain, described in detail below. The session consists of a set of agents running on multiple nodes in a cluster. Each agent has its own responsibilities, and they communicate via network connections to coordinate tasks and share knowledge.

A typical Companion agent is shown in Fig. 2. Each agent has its own instance of the FIRE reasoning engine (Forbus, Hinrichs, de Kleer, & Usher, 2010). FIRE gives the agent access to the analogy stack, logical inference, abduction, procedural attachments, and a working memory for storing reasoning results. But its most important duty is interfacing with the KB via the PlanB module (Section 4.1), allowing the agent to retrieve knowledge seamlessly as part of reasoning.

Companions depend heavily on conceptual knowledge. Rather than starting with an empty KB and trying to learn from scratch, Companions use NextKB (Section 3.3) as an approximation of the kind of conceptual knowledge a human has access to. While the coverage of NextKB is necessarily imperfect, it has been sufficient to support a variety of reasoning and learning tasks, including question answering (Crouse, 2021), legal reasoning (Blass, 2023), and learning by reading (Ribeiro, 2023).

Two particular applications illustrate how Companions' knowledge management needs have grown in recent years: learning by reading and the SocialBot. We discuss each in turn before shifting our attention to the Companion knowledge stack and the three broad challenges that have shaped its development.

2.1. Learning by reading

Ribeiro and Forbus (2021) show how a Companion can learn commonsense knowledge from text. They use analogical techniques to extract the knowledge and a BERT-based classifier to filter out erroneous facts (Devlin et al., 2019). These steps are complementary. The system first draws an analogy between a sentence and a training example to propose a candidate fact. Then the classifier, a BERT model finetuned on facts from the KB, judges whether the proposed fact is plausible. The combination of the two steps greatly increases the precision over analogical knowledge extraction alone while retaining the ability to extract structured facts across a wide range of relations. For example, the system can extract the fact `(properPhysicalPartTypes (MaleFn Deer) Antler)` from the sentence "Male deer have antlers." using a pattern it learned from the training sentence "Bees also have a stinger at the back of the abdomen." The system can learn to extract new relations from just a few examples. When tested on articles from Simple English Wikipedia,[2] it was able to extract 976 commonsense facts across 58 relation types from 2,679 articles with a precision of 71.4 % (Ribeiro, 2023).

Notably, the knowledge extracted this way is not specific to a particular task. It is general knowledge about the world. Ideally, it should be assimilated into NextKB to improve its coverage for future applications. This opens up a straightforward form of autonomy: Given a source of data (Simple English Wikipedia) and a method of acquiring knowledge from it (analogical knowledge extraction), a Companion should be able to go learn what it can from the data source and integrate its findings into NextKB for other Companions to benefit from. This same basic pattern applies across a wide variety of data sources (e.g., text corpora, simulated environments, or users) and knowledge acquisition mechanisms (e.g., learning by reading, experimentation, or asking users questions). Ribeiro (2023) provides a compelling example of a task that a Companion could pursue on its own to increase its knowledge, one of many such tasks that could give the Companion more autonomy and reduce its reliance on hand-crafted knowledge input by its developers.

However, actually integrating the learned knowledge into NextKB remains an open question. So far, most learning done by Companions has been in the context of specific experiments, resulting in few persistent changes to NextKB. To put this learned knowledge to use, the Companion needs to know how to organize it for reasoning with FIRE, how to save it as a persistent part of NextKB, and how to track its provenance so that appropriate weight can be given to different sources. These concerns have helped motivate the additions to the Companion knowledge stack discussed in Section 5, which focuses on giving the Companion the tools it needs to manage its knowledge.

2.2. SocialBot

The SocialBot is a Companion-powered bot[3] for the Microsoft Teams[4] platform, an expanded, online version of the multimodal information kiosk reported in Wilson et al. (2019). The SocialBot automatically scrapes the Northwestern University Department of Computer Science

[2] https://simple.wikipedia.org/wiki/Simple_English_Wikipedia.

[3] As a software system that interacts with online resources on its own, the SocialBot is a "softbot" in the sense of Etzioni & Weld (1994).

[4] https://teams.microsoft.com.

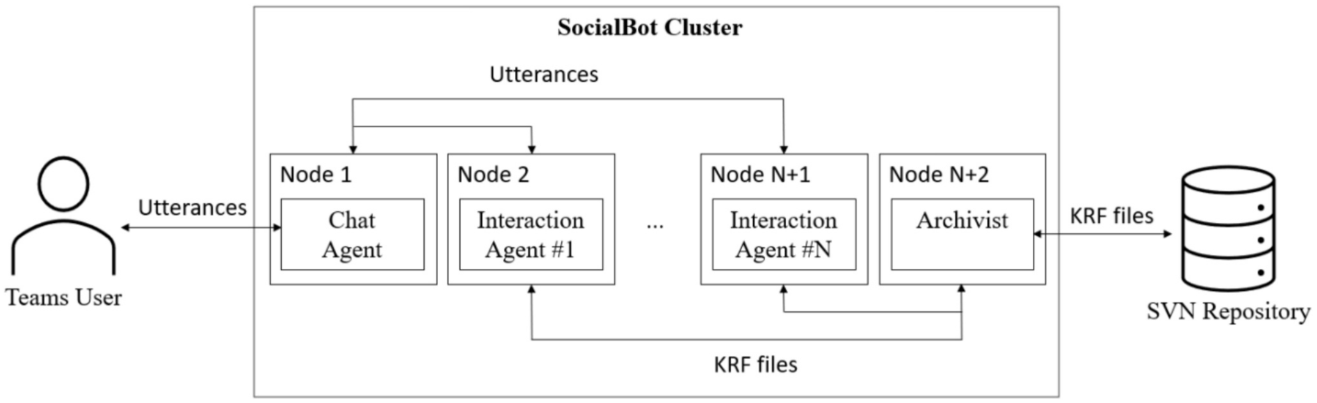

Fig. 1. Companion session for the SocialBot domain. Some agents omitted for clarity.

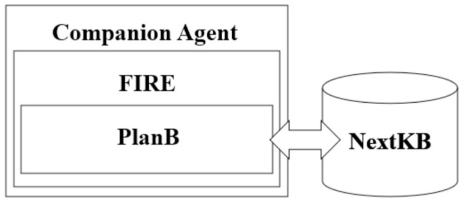

Fig. 2. A typical Companion agent.

website for information about courses, faculty, and events, then uses that information to answer user questions on Teams. Users can also share their food and drink preferences and their interest in computer science topics so the bot can potentially provide personalized event recommendations. Ongoing research will expand the range of feedback users can provide, improving the SocialBot's recommendations and filling in gaps in its knowledge.

The structure of a SocialBot session can be seen in Fig. 1. To support conversations with multiple users at once, the SocialBot is designed to run across multiple cluster nodes, allowing its agents to operate in parallel without undue contention for resources. The Chat Agent handles communication with the Teams platform, receiving messages sent by users and sending replies from the SocialBot. When a user starts a new conversation, the Chat Agent assigns that conversation to an available agent from the pool of Interaction Agents, variants of the Interaction Manager modified for use in parallel. Each Interaction Agent is responsible for tracking one conversation, interpreting the user's utterances, and responding appropriately via the Chat Agent. Having a pool of Interaction Agents allows the SocialBot to reason about how to answer one user's question without tying up the reasoner and KB for another conversation.

The SocialBot has stretched the Companion knowledge stack in two ways. First, its nightly scrapes of the Department of Computer Science website require tracking different versions of the same knowledge (e.g., updated event listings) to prevent stale data from appearing in its KB and thus in its answers to user questions. This is a case where the limited amount of autonomy we have given a Companion, the ability to scrape a website for information, has already raised questions of how to store the learned knowledge and how to deal with different versions of the knowledge over time. Our attempts to deal with these issues helped motivate the creation of the provenance cache, discussed in Sections 5.1 and 5.2.

Second, the SocialBot must manage the knowledge it learns from users. Currently, it only saves basic information about users, such as their food and drink preferences, interest in computer science topics, and permissions for sharing these preferences with other users (i.e. privacy preferences). However, we are in the process of extending the SocialBot's ability to learn more from its users, drawing on their knowledge about the world to expand its own. As the SocialBot's learning capabilities grow, it becomes crucial that it can organize, persistently store, and serve up the knowledge it learns.

The distributed architecture of the SocialBot causes a subtle issue here. Each Interaction Agent is located on a separate cluster node and thus has its own independent instance of the KB. As the SocialBot acquires new knowledge from users, that knowledge must be propagated from the agent that acquired it to the other agents, so that their KBs remain in sync.

The dual questions of what to do with learned knowledge and how to ensure agents remain in sync motivated the creation of the Archivist, shown in Fig. 1 and discussed in Section 5.3. The Archivist is a Companion agent dedicated to saving learned knowledge and serving it to any agents that need it. Rather than rely solely on their local KBs, Interaction Agents also query the Archivist for any knowledge updates they might need before conversing with a user. When an agent learns something new, it passes the knowledge to the Archivist for propagation to other agents. The Archivist also ensures that the learned knowledge is saved to a Subversion repository, where it can be incorporated into future KB builds.

The learning opportunities opened up by analogical knowledge extraction and the SocialBot have motivated recent extensions to the Companion knowledge stack, with the long-term goal of supporting knowledge autonomy. But to understand these extensions, we must first understand the stack itself. With that in mind, we turn our attention to the first challenge the Companion knowledge stack was created to address: knowledge representation.

3. Challenge #1: knowledge representation

The foundation of reasoning in any cognitive architecture is *knowledge representation.* How a cognitive architecture represents knowledge governs what kinds of knowledge it can encode and what operations it can perform over the knowledge it has. For example, an architecture that uses first-order logic may be able to reason efficiently about entities and relations, but it may struggle with statements about statements or higher-order quantification. Thus, the choice of knowledge representation plays a pivotal role in defining the capabilities of a cognitive architecture. In this section, we discuss knowledge representation concerns for the Companion cognitive architecture. We begin by reviewing how Companions represent knowledge, show how microtheories organize knowledge and constrain reasoning, and conclude with a look at NextKB, the knowledge base (KB) that serves as the foundation for a Companion's reasoning.

3.1. *Knowledge representation basics*

The Companion cognitive architecture follows in the footsteps of Cyc

(Lenat, 1995), a multi-decade effort to encode common-sense knowledge to support machine reasoning. Specifically, Companions use CycL, a language based on higher-order predicate calculus that represents assertions as Lisp-style *s*-expressions (Lenat & Guha, 1991). Properly formed assertions consist of a *predicate* and its arguments, each of which may be an atomic *constant* or *non-atomic term* denoting an entity or a nested assertion. For example, the English statement "Joe likes apples." might be represented as the CycL assertion `(likesType Joe (FruitFn AppleTree))`, where `likesType` is the predicate, `Joe` is the constant for Joe, and `(FruitFn AppleTree)` is the non-atomic term that represents the fruit (the *function* `FruitFn`) of an apple tree (`AppleTree`).

In this example, `(FruitFn AppleTree)` denotes the *collection* of all apples, whereas `Joe` denotes an *individual* entity. Individuals and collections form the backbone of the ontology. The two are linked through the `isa` predicate: `(isa Nero-TheCat Cat)` states that the individual `Nero-TheCat` is a member of the collection `Cat`. Collections are related to each other through the `genls` ("generalization") predicate: `(genls Cat Mammal)` states that all members of the collection `Cat` are members of the collection `Mammal`. The `genls` predicate is transitive, so if `(genls Mammal Animal)`, then all `Cat`s are `Animal`s. Higher-order collections support the categorization of collections themselves, such as `(isa Cat BiologicalSpecies)`.

CycL has proven to be an excellent fit for the kinds of reasoning a Companion does. It enables unification-based pattern-matching and retrieval (Section 4.3), it is well-suited to analogical reasoning (Forbus et al., 2016), and it is flexible enough to accommodate a wide variety of domains.

Knowledge is stored in KRF files (for "knowledge representation file"). Each file consists of a series of CycL assertions to load into a running KB, along with processing directives that affect how they are loaded: `in-microtheory`, discussed in Section 3.2, which assigns facts to a context, and `with-provenance`, discussed in Section 5.2, which declares the origins of facts. KRF files also support a handful of macros that do not conform to CycL syntax. These are useful for encoding groups of related facts in a compact way. For example, hierarchical task network methods, preconditions, and preference rules are bundled together with the `defPlan` macro. Macros are expanded into valid CycL assertions before being loaded into the KB.

3.2. Microtheories

Not all knowledge belongs together. Conflicting knowledge about the world can lead to logical contradictions. Reasoning problems in one domain may be derailed by knowledge intruding from another domain, such as the inclusion of fictional characters in a real-world planning task. More broadly, knowledge often must be maintained by humans and thus must be organized in a way humans find useful, especially when it comes to common ontologization tasks such as lookup, knowledge entry, and revision.

To that end, a Companion's knowledge is organized into *microtheories* (MTs) (Guha, 1992; Lenat, 1998), bundles of assertions that are logically consistent and belong in the same reasoning context. All KB queries and reasoning operations are *contextualized* with a microtheory that determines which assertions are visible to that operation. For example, two physics models can be stored in different microtheories and queried independently, even if they are incompatible with each other. Thus, facts that would otherwise contradict are insulated from each other, while related sets of facts are stored in reusable bundles.

Microtheories are connected using `genlMt` statements. The statement `(genlMt MT1 MT2)` means that `MT1` inherits the contents of `MT2`. Any query made in `MT1` will see the facts stored in both `MT1` and `MT2`. The `genlMt` relation is transitive, allowing microtheories to be arranged in a hierarchy. If `MT2` inherits facts from `MT3`, `MT1` will inherit those facts as well. The `genlMt` predicate is also monotonic. If a fact is true in `MT2`, it must also be true in `MT1` and cannot be overridden or retracted. Monotonicity greatly simplifies the problem of determining which facts are true in which contexts, but care must be taken to keep high-level microtheories clear of any facts that do not belong in every inheriting microtheory.

The primary way to specify the microtheory of an assertion is the `in-microtheory` directive in a KRF file. When a file is being loaded into the KB, `(in-microtheory FactsAboutJoeMt)` indicates that all subsequent facts should be stored in the microtheory `FactsAboutJoeMt`, up until the next `in-microtheory` statement. This groups the contents of each KRF file into one or more microtheories. The `ist-Information` predicate can be used to specify the microtheory of an individual fact, as in `(ist-Information FactsAboutJoeMt (likesType Joe (FruitFn AppleTree)))`.

Microtheories are invaluable from the perspective of both reasoning and ontologizing. In addition to keeping contradictory facts separate, microtheories greatly reduce the amount of work required for deduction. The reasoner does not have to try every rule in the KB, only the ones that are visible from the current reasoning context.

The compartmentalization of knowledge makes it easy to mix and match different types of knowledge. For example, a scratchpad microtheory might inherit rules from one MT and facts from another, letting the system reason about the combination of the two with no added effort. This scheme is used frequently by the SocialBot, where the microtheory for reasoning inherits from separate microtheories about events, people, and courses.

Microtheories also aid ontologization. Adding new knowledge to the KB typically means finding where in the `genlMt` hierarchy a new microtheory belongs. This also facilitates knowledge reuse. For example, the plans that govern the behavior of the SocialBot are partially inherited from the plans governing the physical information kiosk it is descended from (Wilson et al., 2019). Rather than rewrite the plans from scratch for the new online interface, we were able to reuse some of the existing plans with a new child microtheory and a `genlMt` statement. Both sets of plans coexist in the KB, and a Companion will run one or the other depending on which domain is selected when the session is started.

Finally, microtheories are central to knowledge autonomy. It is not enough to learn a new fact in isolation. The fact must also be stored in the appropriate microtheory so it can be used later in reasoning. Microtheories can help compartmentalize new knowledge until it has been thoroughly vetted, while the fact that they are first-class entities in the KB means that a Companion can reason about them just like any other concept, opening the door to sophisticated behavior when it comes to organizing, accessing, and verifying learned knowledge.

3.3. NextKB

With the knowledge representation format settled and microtheories in place to constrain reasoning and aid ontologization, the next question becomes what knowledge to represent. Companions uses NextKB,[5] a knowledge base derived from OpenCyc and extended with linguistic knowledge, qualitative representations, Companions rules and plans, and more. NextKB serves as a seed for a Companion's learning and reasoning, providing it with broad common-sense knowledge about the world. NextKB consists of over 700,000 facts spread across 1,300 microtheories and 1,000 KRF files. The ontology contains over 83,000 collections, 26,000 predicates, and 5,000 functions, with over 3,000 rules to support reasoning.

So far, the knowledge in NextKB comes from three sources: the original OpenCyc ontology, manual extensions written by Companion users, and extensions generated semi-automatically from existing resources such as FrameNet (Baker et al., 1998). All of these require some degree of hand curation.

We would like that to change. Learning by reading, the SocialBot,

[5] https://www.qrg.northwestern.edu/nextkb/index.html.

and other mechanisms for knowledge acquisition create the possibility for a Companion to extend NextKB autonomously. Coverage is an issue for any knowledge base, the well-known "knowledge acquisition bottleneck" (Olson & Rueter, 1987). Turning loose Companions to learn on their own would go a long way towards addressing the gaps in NextKB's coverage and improving its performance on future reasoning tasks.

4. Challenge #2: knowledge access

Having knowledge alone is not sufficient for reasoning. A cognitive architecture must also be able to *access* the knowledge when it is needed. We have already seen how microtheories help organize knowledge in the KB, ensuring that only the knowledge relevant to the current problem is seen by the reasoner. But implementing a KB capable of actually retrieving those facts is another challenge altogether. The size of NextKB, the need for real-time interaction in applications like the SocialBot, and the kinds of flexible retrieval needed for reasoning have all shaped the KB implementation for the Companion cognitive architecture. In this section, we explain the design of PlanB, FIRE's KB module, and some of the key features that have allowed it to meet Companions' reasoning needs.

4.1. PlanB

Companions access the knowledge in NextKB using PlanB, a knowledge base module written in Common Lisp. PlanB serves as an interface to an underlying database containing the contents of NextKB, as well as any additional KRF files the Companion has loaded or facts it has stored while running. PlanB is a component of the FIRE reasoning engine (Forbus, Hinrichs, de Kleer, & Usher, 2010). FIRE integrates facts retrieved from PlanB with other sources of knowledge, such as its working memory and its procedural attachments, to support reasoning. The details of PlanB's implementation are discussed in the following sections.

It is important to note that, while PlanB is part of a cognitive architecture, it does not embody any specific psychological claims. PlanB acts as a Companion's long-term memory, but it makes no attempt to model human memory except in the coarsest functional terms. Not every fact stored in the KB has a human analogue (i.e., some are only maintained for bookkeeping), and questions of representation are often settled on a practical basis, not a psychological one. We leave any psychological claims about memory to specific models implemented on top of PlanB, such as the MAC/FAC (Forbus et al., 1995) model of analogical retrieval.

4.2. AllegroCache

PlanB uses AllegroCache,[6] a fast and scalable database that natively supports Lisp data types. In addition to typical database functionality such as persistent storage and rollback, AllegroCache offers two features of interest: the ability to persistently store Lisp objects in the database without a separate serialization step, and key-value storage with persistent maps.

PlanB stores facts as persistent objects, with slots for the CycL expression, an integer ID, and the list of microtheories where the fact is believed. Any concept that is mentioned in a fact is also stored in the database as a *conceptual entity*, a persistent object with different slots depending on its type. For example, predicate objects have slots that keep track of their arity and commutativity, and microtheory objects cache their location in the `genlMt` hierarchy to support rapid inference. AllegroCache's indexing capabilities make it straightforward to look up a fact or entity by its CycL expression or internal ID.

[6] https://franz.com/products/allegrocache/.

The other major storage pattern used by PlanB is maps. Retrieving a persistent object from an AllegroCache database is slower than retrieving an entry from a persistent map, so PlanB uses maps whenever speed is essential. AllegroCache maps can take almost any Lisp data type as their keys, but PlanB primarily uses integers, often fact or entity IDs. AllegroCache also supports the efficient retrieval of ranges of keys. By carefully constructing keys to ensure related entries are contiguous, we can retrieve groups of entries in a single operation, as in the mentions map in the next section.

4.3. Unification-based retrieval

The main function of a knowledge base is to retrieve stored facts on demand. PlanB supports unification-based retrieval as its primary means of access. Given a query pattern with zero or more variables, PlanB will retrieve all facts which syntactically unify with that pattern. Queries can be contextualized with a microtheory, in which case only facts believed in that microtheory (either directly or by `genlMt` inheritance) will be returned.

Consider the KB shown in Fig. 3. Querying `(foo Bar)` in `MT1` will retrieve `(foo Bar)` but nothing else. The query is *ground* (i.e., it has no variables), so no other facts can unify with it. Querying `(foo ?x)` in `MT1`, where `?x` is a variable, will retrieve `(foo Bar)`, `(foo Baz)`, and `(foo Quux)`, the latter inherited from `MT2`. Lastly, querying `(foo ?x)` in `MT2` will only retrieve `(foo Quux)` because the other facts are only visible from `MT1`.

PlanB uses *coarse coding* (Forbus, Hinrichs, de Kleer, & Usher, 2010) to index its facts. Every fact is indexed according to the entities it mentions in a persistent AllegroCache map known as the *mentions map*. Mentions are stored as entity-location-fact triples, where the location is the position of the argument in which the entity appears.

Arguments can have nested structure, but only the top-level position matters for the purpose of calculating mentions (hence "coarse"). For example, the fact `(foo (TheList A B))` will have mentions for `foo` in position 0 and for `TheList`, `A`, and `B` in position 1. The fact `(foo (TheList B A))` will also have the same mentions, even though `B` and `A` have swapped places within the same position, and thus the expressions would not unify.

Mentions are compressed into integer keys using the entity and fact IDs, so that mentions for the same entity are stored adjacently. This makes it efficient to look up all mentions of an entity, or all mentions of the entity in a particular position, by passing the appropriate lower and upper bounds to AllegroCache. The ability to retrieve all facts that mention an entity is useful for automatic case construction to support analogical reasoning and learning, among other things.

During retrieval, PlanB breaks the query pattern down into mentions the same way it does to index facts. Variables are ignored. For each entity mention in the query, PlanB uses the mentions map to retrieve a *bucket* of facts that mention that entity in the same location. Because a fact will only unify with a pattern if the fact contains all of its mentions (i.e., there are no constants in the query that do not also appear in the fact), PlanB takes the intersection of these buckets to find a set of candidate facts that could unify with the query. It then filters out the candidates that do not unify with the pattern or which are not believed in the target microtheory, producing the final set of retrieved facts.

Coarse coding is an example of how choices regarding knowledge representation, knowledge access, and reasoning interact. Unlike the binary relations used in some knowledge representation languages, CycL

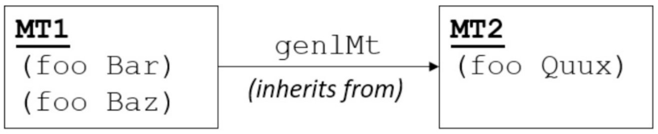

Fig. 3. Sample KB with microtheories.

predicates can take an arbitrary number of positional arguments, although few use more than seven. CycL's support for nested statements and non-atomic terms further complicates fact storage and retrieval. In exchange for a flexible representation scheme that can support FIRE's reasoning, we must accept the added complexity of PlanB's mentions map. The reward is that a Companion agent is free to work at the level of facts and rules, without worrying about how exactly the KB stores and retrieves them.

4.4. Special facts and indexed facts

Although coarse coding and the mentions map provide a flexible and reasonably efficient method for retrieving facts from the KB, there are situations when speed outweighs generality. For such situations, PlanB supports the definition of *special facts*, facts which are stored and retrieved using custom handlers rather than the mentions map. Special facts are used when efficiency is paramount, or when the nature of the fact lends itself to a specific type of storage. For example, `genlMt` statements are stored in a persistent TMS that allows efficient computation of microtheory inheritance, a vital part of reasoning that would otherwise cause a significant speed bottleneck (Forbus, Hinrichs, de Kleer, & Usher, 2010). Other special facts include fact probability and provenance (see Section 5), bookkeeping for analogical retrieval, and templates for converting KB concepts to natural language.

Special facts are retrieved much faster than regular facts because they can take advantage of custom lookup strategies, but the need for specialized code for each type of fact makes them harder to maintain than regular facts. Since special facts bypass the mentions map, they can only be accessed using query patterns that are explicitly allowed by the custom handlers. Queries that would traverse the underlying data structures in awkward or inefficient ways are not supported.

As a compromise between speed and flexibility, PlanB supports *indexed facts*, regular facts that have one of their arguments designated as a key. When PlanB receives a query whose predicate is indexed, if the argument corresponding to that predicate's key is ground (i.e., it does not contain a variable), it short-circuits the usual retrieval process and looks up the key directly in a special index. This is faster than regular retrieval but not as fast as a dedicated special fact handler. For queries where the key argument is not specified, indexed facts are retrieved the same way as regular facts, using the mentions map.

For example, the lexicon entries used by CNLU (Tomai & Forbus, 2009), Companions' semantic parser, are indexed by token (i.e., the basic unit of a linguistic utterance). This makes the common use case of mapping an input token to its lexicon entries highly efficient. However, this is not the only way that lexicon entries are accessed by CNLU. Occasionally, they must be retrieved based on other arguments, such as their root word or part of speech. Because they are stored as indexed facts, lexicon entries can be retrieved efficiently by specifying their token in the query pattern or at the same speed as regular facts when the token argument is a variable.

The ability to define special facts and indexed facts has softened the tradeoff between generality and efficiency in PlanB. Regular fact storage and retrieval are efficient enough for most types of reasoning, and they are robust enough that most Companion developers never need to peek behind the abstraction of unification-based retrieval. But when more speed is necessary (e.g., real-time natural language understanding for SocialBot conversations) or when the mentions map is a poor fit (e.g., fact probabilities and provenance, which would yield enormous buckets of mentions), we can optimize PlanB for specific predicates by storing them as special or indexed facts.

5. Challenge #3: knowledge management

With a reliable system for knowledge access in place, we turn our attention to the challenge of *knowledge management*. The knowledge management practices we have developed work well for hand-curated knowledge, where a human has the ultimate say over what gets checked into the NextKB repository, but they are not yet ready for a Companion to make changes on its own. As Companions gain new ways to learn autonomously, such as learning commonsense facts by reading or talking to users via the SocialBot, we must equip them with new tools to save the knowledge they learn and assimilate it into NextKB for use by other Companions.

This section outlines the recent developments in Companions' knowledge management capabilities. We begin by discussing the *epistemic layer*, new conceptual machinery that allows a Companion to track where the facts in its KB came from, and the *provenance cache*, the epistemic layer's implementation in PlanB. We then discuss the Archivist, a new Companion agent created for the SocialBot which supports knowledge updates between agents during a session and persists learned knowledge to a version control repository for future use.

There are many other aspects of knowledge management, such as establishing conventions for sorting learned knowledge into microtheories, developing tools to edit KRF files automatically, and defining procedures to assess the quality of learned knowledge. This section only documents the first steps we have taken towards giving Companions the tools they need for knowledge autonomy.

5.1. The epistemic layer

So far, the knowledge organization scheme we have discussed can be divided into two layers: the *physical layer* and the *logical layer*. The physical layer consists of the KRF files where knowledge is stored before being loaded into the KB, and its basic unit of organization is the file. For ease of organization, KRF files are arranged into hierarchical directories according to their content and purpose. The directory structure makes no difference once the KRF files have been loaded into the KB, but the KB build process uses it when determining which files to include. Specifically, *index files* contain pointers to the KRF files and subdirectories that should be included. Building a new KB consists of walking down the directory structure, reading the index files to generate a list of KRF files, and loading them into a fresh AllegroCache database using PlanB. This database is then uploaded for distribution to other developers and their Companions.

The primary way changes to the physical layer are propagated is through version control. We maintain a Subversion[7] repository that contains the latest code, KRF files, and KB build for the Companion architecture. Developers modify KRF files locally, then commit their changes to the repository for inclusion in the next build. Because the KRF file format is plaintext, with no more than a single fact per line, the standard diff-based merge tools work well on it. Developers can update their local copies of the KB in two ways: by pulling the latest KRF files from the repository and loading them into their KB, or by replacing the KB with the latest build for a fresh start.

This setup works well for the way NextKB has been developed so far: hand-curated knowledge maintained by a small set of developers, with only one KB configuration. However, as we gather knowledge from more sources, parts of this workflow will have to change. We discuss this issue further in Section 6.

Whereas the physical layer refers to knowledge stored in KRF files, the logical layer refers to knowledge as it exists in the KB. The basic unit of organization is the microtheory, which bundles knowledge into groups that are useful for reasoning. One of the main functions of the Companion knowledge stack is to support the logical layer as a clean abstraction. Reasoning in FIRE builds on primitive KB operations such as retrieving, storing, and forgetting facts, without regard for the underlying data structures in PlanB.

The logical layer is independent of the physical layer. One KRF file may contain facts from several different microtheories, each with its

[7] https://tortoisesvn.net/.

own `in-microtheory` statement, and the contents of one microtheory can be spread out across multiple files. This independence is very useful. Files can be merged, renamed, or split without changing the logical contents of the KB, and the reasoner only has to worry about facts and microtheories, not the files themselves. The only connections between the physical and logical layers are when KRF files are loaded into the KB or when facts from the KB are written out to a KRF file. The former converts knowledge from the KRF format to PlanB's internal data structures, so that the knowledge is available in the KB for the reasoner to use. The latter serializes knowledge in the KB for later use, as when the SocialBot writes out what it has learned during a conversation.

However, the independence of the logical and physical layers also causes a problem. The strict separation means that once a fact is stored in the KB, there is no way to tell where it came from. It is simply believed to be true in whatever microtheory it was stored in, with no record of why it is believed, which file or reasoning process stored it, or whether multiple sources have attested to it.

This poses a challenge for development. One of the most common patterns is iteratively testing and correcting KRF files, meaning repeatedly loading them into the KB. But without a record of where the facts in the KB came from, forgetting the old version of a file is cumbersome. The main options are to forget the microtheories contained in the file, which will cause collateral damage if other files use those microtheories, or to forget the old version of the file verbatim before making any modifications. Even the latter is not foolproof. Facts that appear in multiple KRF files will be clobbered if any one of the files is forgotten, leaving the state of the KB inconsistent with the KRF files that have been loaded into it.

The separation of layers also poses a challenge for reasoning. While most reasoning processes do not need to consider the source of the knowledge they use, there are cases when such metadata is useful, such as when assigning blame for an incorrect fact. As long as NextKB is hand-curated, this type of attribution does not matter much. While the version control logs can shed some light on which developer introduced a buggy fact, this is no more than a curiosity. The solution is always to go into the KRF file and fix the fact by hand. Effectively, the knowledge in NextKB comes from a single, authoritative source.

But as more of a Companion's knowledge comes from outside sources, it becomes important to track where the knowledge came from. A Companion should be able to distinguish between data scraped from the web, facts it was told by a user, and the hand-written background knowledge that forms the backbone of NextKB. A Companion should also be able to gauge how trustworthy its knowledge sources are and explain where its knowledge came from when the user asks.

These two challenges, the friction of updating KRF files and the problem of attribution for reasoning, motivated us to introduce a new layer of knowledge organization: the *epistemic layer*. Whereas the physical layer is organized by file and the logical layer is organized by microtheory, the epistemic layer is organized by *provenance*. Every time a set of facts is stored in the KB, they are tagged with a *provenance event*, metadata recording how the facts came to be known. Typical provenance events include loading a KRF file, scraping a webpage, or having a conversation with a user. More generally, any meaningful occurrence that stores facts in the KB can be a provenance event. Thus, a Companion not only knows facts but knows *why* it knows them.

One of the main purposes of the epistemic layer is to serve as an interface between the physical and logical layers. Every time a KRF file is loaded into the KB, a new provenance event is created, and the loaded facts are tagged with it. Every provenance event has a *source*, recording where the information came from, and a *timestamp*, recording when it was provided. For loading a KRF file, the source of the provenance event is the path of the file relative to base Companion directory, and the timestamp is the file modification date, represented in Lisp universal time.

Fig. 4 shows a simple example of how the epistemic layer helps with updating KRF files. Suppose File A and File B have been loaded into the KB. The facts from File A are tagged with provenance event `E1`, shown as dashed red arrows in the figure. This version of the file contains `(foo Bar)`, `(foo Baz)`, and `(foo Quux)`. The last of these, `(foo Quux)`, also appears in File B, so it is supported with a black arrow by provenance event `E0`, the record of loading File B into the KB. Microtheories and the timestamps of provenance events are omitted for simplicity.

Now suppose the user edits File A and reloads it. The new version of File A keeps `(foo Bar)`, deletes `(foo Baz)` and `(foo Quux)`, and adds `(foo Mumble)`. The updated facts are loaded into the KB tagged with `E2`, the provenance event that represents loading the new version of the file, indicated by the green arrows in the figure.

PlanB recognizes that `E2` represents an update to the file loaded in `E1`, so it retracts `E1` and all of its provenance. Then, for every fact that `E1` supported, PlanB checks whether the fact is still supported by some other provenance event. If it is, the fact is kept in the KB. If it is not, the fact is forgotten. Thus, `(foo Bar)` remains in the KB because it appears in the new version of File A, `(foo Quux)` remains because it is still supported by File B's event `E0`, even though it is no longer in File A, and `(foo Baz)` is forgotten because it no longer has any support. The KB has been successfully updated with the new version of File A without affecting facts that appear in File B.

It is worth noting that while the logical and epistemic layers are separate layers of *organization*, they coexist in the KB and can be reasoned about together. Provenance events are ordinary entities in the KB, complete with `isa` statements and other assertions. To access the provenance of a fact, the reasoner only needs to query `(provenance <fact> ?event)` and then do whatever it wants to with `?event`. The `provenance` predicate is treated as a special fact in PlanB, making it efficient enough for routine use.

While the logical and epistemic layers both reside in the KB, the distinction between them is important. The logical layer concerns the

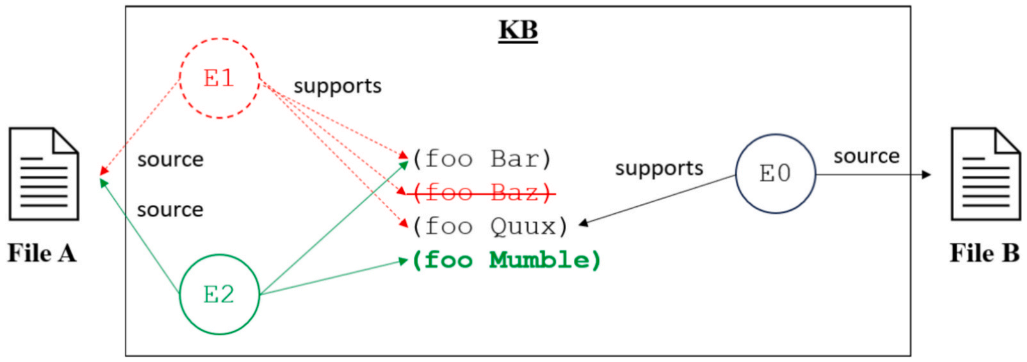

Fig. 4. Example of a KRF file update using the provenance cache. When a new version of File A is loaded into the KB, the facts are tagged with a new provenance event (E2) and the old provenance event (E1) is retracted. Any facts that are no longer supported by a provenance event will be forgotten. In this way, the KB can forget facts that were deleted from File A without forgetting facts that came from File B.

organization of knowledge for ordinary reasoning and centers around facts and microtheories. The epistemic layer acts as metadata for facts in the logical layer, preserving a pointer to the original KRF file for ease of development and tracking the source of each piece of knowledge for attribution.

To recap, the physical layer of knowledge consists of KRF files, outside of the KB. These files can be edited, distributed via version control, or loaded into a KB. The logical layer consists of facts organized into microtheories and serves as the backbone of Companion reasoning. Facts can be retrieved, stored, or forgotten, all in the context of a particular microtheory. The epistemic layer consists of provenance events that explain why facts are believed. Provenance events can be retracted, allowing the Companion to gracefully forget stale information, or retrieved to help with fact attribution.

In the next section, we discuss how the epistemic layer is used in practice.

5.2. The provenance cache

Provenance information is stored by PlanB in the *provenance cache*, a set of AllegroCache maps that provide a lightweight mapping from facts to provenance events and vice versa. The `provenance` predicate interfaces with the provenance cache via special fact handlers, while PlanB's storage and retrieval functions ensure the provenance cache is kept up to date as facts are added and removed from the KB.

The provenance cache can distinguish between the same fact being stored in different microtheories. For example, if `(foo Bar)` is believed in `MT1` and `MT2`, retrieving the provenance for `(foo Bar)` in `MT1` may yield a different provenance event than retrieving it in `MT2` (e.g., if the two versions of the fact came from different KRF files). Facts may also be supported by multiple provenance events, as seen in Fig. 4. This helps with KRF file updates and opens the door to a Companion accumulating redundant knowledge from multiple sources to gauge its reliability.

While provenance facts can be stored in the KB individually, the typical way to tag provenance is a `with-provenance` directive in a KRF file. Just as an `in-microtheory` statement specifies the microtheory for all facts that follow it, `with-provenance` specifies the provenance of all the subsequent facts in the file, up to the next `with-provenance` statement. For convenience, every KRF file is assumed to begin with an implicit `with-provenance` statement that has the file as its source and the file modification date as its timestamp. Thus, the default form of provenance we wish to track, the loading of KRF files, is handled automatically.

Fig. 5 shows an example of a `with-provenance` statement for a conversation with a SocialBot user. The two main arguments are**:source** and**:timestamp**, which specify the source and timestamp of the provenance event. Here the source is the user ID (truncated for length) and the timestamp is the time of the conversation. The optional**:entity** argument allows the provenance event to be passed in explicitly, rather than using a new entity automatically generated by PlanB. This is useful when there is more to say about the provenance event. In this case, the event is the user's conversation ("discourse") with the system. The `with-provenance` statement tags the facts that were learned from the conversation, while having an entity lets us store useful information about the conversation itself, such as what the user and the system said.

So far, we have discussed how provenance can help with updating KRF files and how it can be used to record where learned knowledge came from (e.g., a user conversation). We have not yet addressed the

```
(with-provenance
    :source (MSTeamsUserFn "<user_id>")
    :timestamp (UniversalTimeFn 3919440252)
    :entity Discourse-3919440252-8397)
```

Fig. 5. `with-provenance` statement for a conversation with a SocialBot user.

problem of how these two use cases interact. Each `with-provenance` statement supersedes the last, so once the KRF reader sees the statement for a conversation, all subsequent facts will only be tagged with that conversation, not with the KRF file loading event. This is sufficient for attribution, but it still leaves us the problem of tracking knowledge in the KB back to the KRF file it came from.

Our solution is to allow *meta-provenance*, that is, multiple layers of provenance. Conceptually, *meta*-provenance is like a chain of attribution: "Joe said that Tom said X." If any link in the chain is untrustworthy (Joe or Tom), we should retract our belief in X. For Companions, these chains tend to have at most two links, the first usually being a KRF file: "File A says that User B said X." In the KB, *meta*-provenance is represented as nested `provenance` statements. The procedure for retracting provenance events takes this nesting into account, so a fact will be considered for deletion if any provenance event in the chain is retracted. The implicit `with-provenance` statement at the beginning of a KRF file is tagged as *meta*-provenance, so the developer (or Companion) is free to use `with-provenance` statements throughout the file with impunity.

One last feature of the provenance cache is worth mentioning: Provenance events in KRF files can be marked as *updates*. When an update event is loaded into the KB, it signals that PlanB should retract all other provenance events with the same source and an earlier timestamp. The newest set of information from that source supersedes all others. For regular provenance events, this feature is not necessary. A user can have multiple conversations with a Companion, and none invalidates the others. But when the source of information is a webpage, each new scrape of the page should supersede the previous ones to avoid keeping stale information.

This is exactly the situation that arises with the SocialBot. The SocialBot performs nightly scrapes of the events webpage for the Northwestern University Department of Computer Science so it can answer questions about those events. The scraped information is written out to a KRF file and tagged with a `with-provenance` statement whose source is the week that block of information pertains to and whose timestamp is the time of the scrape. Flagging the event as an update means that the scraper only has to append the new scrape to the KRF file without worrying about deleting the old one. The previous scrape for that week of events will be retracted automatically when the new one is loaded.

Finally, provenance can be set programmatically, so that a set of facts stored by the Companion are tagged with a particular provenance event. By default, every Companion session is one such event, so that any facts that the Companion stores during that session can be exported to a KRF file for archival purposes or retracted if necessary. This is a step towards having a Companion manage its own knowledge. By keeping a record of what it has learned during a session, the Companion can decide what knowledge is worth keeping and what knowledge should be discarded. We plan to explore this possibility in future work.

5.3. The archivist

The SocialBot is one of the first instances where a Companion has been set to run and accumulate knowledge over an extended period of time. While the knowledge accumulated so far has been simple, mainly users' food, topic, and privacy preferences to personalize their recommendations and respect their desires regarding information sharing, having an online, persistent Companion opens the door to acquiring other kinds of knowledge from users. We are currently exploring options for knowledge capture games, such as learning moral norms (Olson & Forbus, 2021) or debugging the system's language understanding (Nakos, Kuthalam, & Forbus, 2022).

Even the limited forms of learning the SocialBot is already capable of have stretched our knowledge stack in interesting ways. The provenance cache was developed in part to track the sources of learned information in a reliable, systematic way. And, as discussed in Section 2.2, the needs of the SocialBot have motivated us to create the Archivist, a new Companion agent designed specifically for knowledge management.

The Archivist has two main responsibilities: to ensure that every agent in a session has access to up-to-date knowledge and to interface with our version control repository. The former prevents agents' KBs from getting too far out of sync when they are learning in parallel, while the latter ensures that learned knowledge is saved for future use.

To keep the other agents in sync, the Archivist maintains a canonical copy of every KRF file that agents in the session have been modifying. When an agent has finished making changes to a file, it sends a copy to the Archivist. Other agents can then query the Archivist to grab the latest copy of the file, giving them access to knowledge learned during the session without having to communicate directly with every other agent. As a SocialBot session can last for several weeks, the Archivist plays a valuable role in propagating knowledge between agents.

To handle knowledge persistence, the Archivist maintains a queue of version control operations that need to be performed when it has the bandwidth available. Rather than have each agent block as it commits its changes, the agent can simply send the file to the Archivist, which will commit it to the Subversion repository as soon as possible.

Thus, having a dedicated Archivist agent makes knowledge management simple for the learning agents. They are only responsible for acquiring knowledge, saving it to a KRF file along with its provenance, and sending the file to the Archivist when it is ready.

This still leaves a few open questions, such as how often to sync knowledge between agents or what to do about conflicting user testimony. For the SocialBot, the answers to these questions are straightforward. Interaction Agents sync their user models with the Archivist at the beginning and end of each conversation, and they periodically ask the Archivist for updates about users that other agents have been talking to. The only conflicting information that needs updating is when users update their preferences, which the SocialBot dialogue plans are equipped to handle.

As the types of knowledge that the SocialBot can learn grow more complicated, so too will the answers to these questions. In the next section, we discuss how the Companion knowledge stack will have to grow to support greater autonomy.

6. Towards knowledge autonomy

While developments like the provenance cache and the Archivist have expanded the knowledge management capabilities of Companions, we have only scratched the surface of what is possible. The ultimate goal is *knowledge autonomy*, a state where the Companion is capable of acquiring and organizing knowledge on its own. The SocialBot makes for a good starting point. It has limited opportunities to seek knowledge and a limited range of knowledge it can accept, but it contains the complete pipeline from knowledge source (user interaction) to knowledge storage (the repository). Expanding the types of knowledge a Companion can acquire and its autonomy in selecting learning goals will be an important step towards creating a full software social organism.

In the remainder of this section, we discuss the next steps we are considering towards this goal. We touch briefly on the challenge of self-directed learning, discuss how a Companion could assess the quality of learned knowledge, and lay out some extensions to the provenance cache that could open up new forms of introspection for Companions.

6.1. Self-directed learning

While prior work has explored how a Companion can direct its own learning (e.g. Hinrichs & Forbus, 2019), this capability has not been tested at scale. Ideally, a Companion should have a set of broad learning goals that drive its behavior. When not otherwise occupied, it should seek to satisfy its goals by reading text, asking a user questions, ruminating on previously acquired knowledge, or running games or simulations. Learning by reading (Ribeiro, 2023) is one particularly promising activity a Companion can work on on its own, especially if given large corpora to work through and ways to test what it has learned.

Fully supporting this behavior will require extending the Companion knowledge stack. The Archivist provides a basic mechanism for a Companion to save off what it has learned, but we have yet to establish conventions regarding where learned knowledge should be stored in the repository, where in the microtheory hierarchy it should be installed, and how it should interact with existing knowledge, which may itself be erroneous.

There is also the question of knowledge lifespan. Some learned facts are timeless ("Male deer have antlers."), others are transient ("The location of the talk is TBA."), and still others persist for a duration that's hard to anticipate ("The state of the art on this benchmark is X%."). The SocialBot has already introduced this question in a small way, as the nightly scraper updates its knowledge about events, but as we expand learning to more domains, modeling the lifespan of knowledge becomes more important.

Eventually, as we acquire more knowledge from more sources, we will need another layer of knowledge organization: the *distribution layer*, which would govern how KRF files are packaged into KBs for use by various applications. The current setup focuses on a single, general-purpose KB: NextKB. But the existence of SocialBot user models—kept separate from the core NextKB distribution, subject to their own privacy constraints, and applicable only to a single domain—opens the door to other specialized bodies of knowledge. Being able to tag groups of KRF files for inclusion in or exclusion from specific KB builds is a logical next step once a Companion can gather more knowledge on its own.

6.2. Knowledge integration & trust

One of the more interesting problems that arise as Companions move towards knowledge autonomy is figuring out how to integrate knowledge from multiple sources. Knowledge from an external source may not necessarily be accurate, and it may not be consistent with what the Companion has learned from other sources. To robustly learn from webpages, books, interactions with humans, and/or other outside sources of information, a Companion must be equipped with ways to compare, assess, and deploy the knowledge it has learned.

One important aspect of this problem is trust. Not all knowledge sources are trustworthy. Users may speak in ignorance or with intent to deceive, webpages vary wildly in terms of their veracity, and, in the extreme case, large language models generate plausible-sounding text with no specific grounding in reality. As such, a Companion needs to build up a model of how trustworthy its sources are so it can reconcile contradictions, proactively vet knowledge from suspect sources, and prioritize learning from more reliable ones.

The provenance cache provides a basis for this, tagging the source of each fact in the KB and making it available for introspection. When the Companion detects a suspect fact in the course of its reasoning, it can note the source that was to blame and update its trust model accordingly.

Dempster-Shafer Theory (Shafer, 1976) is one option for tracking reliability. The theory lays out how to combine claims from different sources, producing belief and plausibility estimates that provide lower and upper bounds on how probable the argument from evidence to conclusion is. Olson, Salas-Damian, and Forbus (2024) show how Dempster-Shafer Theory can be used to resolve competing testimony about norms. Extending this to general learned knowledge is an avenue of future work.

6.3. Extending the provenance cache

While the current iteration of the provenance cache has proven useful, there are several additions that will help support Companions' learning. First, the provenance cache should support events with hierarchical structure, such as scraping a single webpage as part of a larger scraping pass or running one trial within a larger experiment. Meta-provenance allows a crude form of this, but not well enough to handle structured, multi-part events gracefully, so at least some new machinery is required. Giving a Companion more information about the structure of its experiences will help it take control of its own learning.

Second, the provenance ontology should be expanded to track different types of provenance events and sources. The current ontology consists of a handful of concepts in NextKB, which has been sufficient so far but lacks the systematicity of an ontology like PROV (Moreau et al., 2015). As the scope of Companions' autonomous learning increases, keeping track of more information about knowledge sources would be helpful.

Finally, tracking KB deletions in the provenance cache would help with replicability across Companion sessions. Right now, the cache only tracks when facts are stored, not when they are forgotten. Tracking deletions would allow one Companion to reproduce the KB changes made by another and would provide an automatic record when erroneous facts are forgotten, opening up new patterns of learning and correction.

7. Related work

The Companion cognitive architecture is by no means the only one to face the challenges discussed in this paper. The developers of every cognitive architecture must make their own choices about knowledge representation, determine the right set of tradeoffs for efficient knowledge access, and, eventually, develop tools and practices for knowledge management. In this section, we compare the Companion knowledge stack with the knowledge solutions developed for Cyc and Soar, two mature architectures with similar knowledge needs. We also briefly review other work on tracking knowledge provenance, particularly for the Semantic Web.

7.1. Knowledge in Cyc

Cyc shares the Companion cognitive architecture's focus on conceptual knowledge, and due to the massive size of Cyc's KB, its need for knowledge management is even greater. As such, Cycorp has developed a robust knowledge entry pipeline for the Cyc KB (A. Sharma, personal communication, March 10, 2024; Siegel et al., 2004). Ontological engineers work locally through a UI, adding new facts to the KB and correcting old ones, and transcripts of their changes are sent to a central server for incorporation into the nightly KB build.

Notably, the Cyc build process is incremental, beginning with the most recent version of the KB and applying accumulated changes until the KB is up to date. That is, the locus of knowledge is the KB itself, rather than a collection of independent, declarative files. This design decision fits the incremental, transcript-driven workflow Cyc has developed, where a master copy of the KB is updated nightly. It also promotes the idea of the Cyc KB as a unified body of knowledge, rather than *ad hoc* snippets of knowledge for specific projects.

In exchange, changes to the data structures of the KB must be handled with care. Because the knowledge and its internal representations are so closely intertwined, any change must be applied to all relevant KB structures to bring them up to date (A. Sharma, personal communication, April 11, 2024). The nightly build process and the transcript server again show their worth when fixing errors in the KB: It is always possible to revert to a "last known good" build of the KB and update it from there.

In contrast to Cyc, NextKB is rebuilt from scratch using standalone KRF files. These files are considered the locus of knowledge for NextKB, rather than the KB build itself. KB builds act as checkpoints, a way of equipping a Companion with a recent, preloaded version of NextKB to use. Any changes to NextKB that have been made since the last KB build can be applied locally by pulling the latest KRF files from our version control repository and loading them into the KB. This achieves the same effect as patching the Cyc KB with transcript files, namely an up-to-date version of the KB, but KRF files contain knowledge directly, rather than logs of changes. For example, deleting a concept means removing the facts about it from the appropriate KRF files and reloading those files, rather than deleting it through a UI and logging the KB change in a transcript.

The decision to put KRF files first has an impact on how Companions are developed. It is very natural to experiment with KRF files locally before checking them into the repository. Large and situationally useful files like datasets can be stored in the repository but excluded from the default NextKB build, so that a Companion only has to load them when necessary. At the same time, the modularity of KRF files puts the burden on the knowledge engineer to make sure that new concepts are properly ontologized and that old knowledge is reused when appropriate. There is also currently no easy-to-use transcript mechanism. Changing NextKB requires modifying KRF files, rather than simply saving changes that were made to a running KB. The provenance cache is a step in this direction, but it will need to track deletions as well to support Cyc-style transcripts.

Finally, the Cyc KB tracks the provenance of its assertions, including the ontologist who added them, the date and time they were added, and the original source of the information. Records of deletions are not present in the KB but can be derived from transcript files. This is in line with the aims of the Companion provenance cache, and as we plan our next steps, we take inspiration from Cyc's ability to show the user the original sources of facts used during inference.

7.2. Knowledge in soar

The Soar cognitive architecture (Laird, 2012) provides an interesting contrast to the Companion knowledge stack. Where the Companion cognitive architecture prioritizes conceptual knowledge and reasoning, Soar prioritizes procedural knowledge and skill learning. This focus allows Soar to make concrete psychological claims about human performance and procedural learning while maintaining an architecture both general and performant enough for real-world applications.

The difference in emphasis leads to a variety of differences in knowledge representation and storage. Soar represents declarative knowledge with graph structures. Working memory consists of a single connected graph, while long-term declarative memory stores multiple graphs that can be retrieved into working memory based on cues. Soar's declarative memory is partitioned into semantic memory, which contains general knowledge about the world, and episodic memory, which records the agent's experiences, saved as snapshots of previous working memory states.[8]

In contrast, Companions treat CycL assertions as the basic unit of declarative knowledge, have a single long-term memory organized into microtheories, and lack a single, system-wide, automatic mechanism for episodic memory (cf. Forbus & Kuehne, 2007; Hancock, Forbus, & Hinrichs, 2020; Hancock & Forbus, 2021). Each domain has its own criteria for how episodes should be stored and its own mechanisms for employing that knowledge later (e.g., saving and generalizing combat outcomes in a strategy game to make better decisions in the future). Provenance tracking opens the door to more open-ended episodic memory by automatically organizing facts that are saved during a Companion session, but it remains to be seen whether a common pattern can meet the episodic memory needs of multiple domains.

Both Soar and Companions have mechanisms for retrieving long-term memory elements based on partial descriptions. In Soar, retrieval must be invoked deliberately by an operator as part of a problem-solving strategy, whereas Companions treat retrieval as an ubiquitous part of

[8] Soar also has procedural and perceptual memories, but they are not relevant to the current discussion.

planning and reasoning. Companions also support analogical retrieval (Forbus et al., 1995), where cases are retrieved based on their structural similarity to a probe.

7.3. Provenance

There is a large body of existing work on representing the provenance of information, especially for the Semantic Web (for reviews, see Herschel, Diestelkämper, & Ben Lahmar, 2017, Pérez, Rubio, & Sáenz-Adán, 2018, and Sikos & Philp, 2020). Languages such as PML (Pinheiro da Silva, McGuinness, & McCool, 2003) and PML 2 (McGuinness et al., 2007) can encode explanations for the conclusions produced by reasoning systems, including the provenance of the information used, while the PROV family of documents (Moreau et al., 2015) lays out a formal specification for encoding different types of provenance.

These endeavors are more ambitious than the Companion provenance cache, and they serve a different purpose. Unlike applications that use the Semantic Web, Companions perform most of their reasoning locally using knowledge aggregated at a central repository. Provenance matters for managing knowledge within the KB, tracking the sources of learned knowledge, and assessing their reliability, but it plays much less of a role than in the Semantic Web, where reasoning is frequently distributed and knowledge is drawn *ad hoc* from a variety of sources. Consequently we have opted for a lightweight provenance ontology rather than replicating the full complexity of, say, PROV. However, as Companions' provenance needs grow, we will expand the ontology as needed, drawing on existing work for inspiration. In particular, PROV's tripartite distinction between entities, agents, and activities might be useful for our needs.

8. Conclusion

One of the most important aspects of a cognitive architecture is how it deals with knowledge. In this paper, we have outlined the Companion knowledge stack, including the ways Companions represent, access, and manage their knowledge. The CycL language, microtheories, and NextKB give Companions a solid foundation of expressive representations, Plan B provides efficient unification-based retrieval, and recent developments such as the provenance cache and Archivist have expanded Companions' ability to manage learned knowledge. All of this supports the goal of knowledge autonomy, and we are currently exploring what further capabilities are needed to let a Companion learn in more ways on its own.

While we have grounded our discussion in the Companion cognitive architecture, many of the same challenges will apply to any cognitive architecture that moves past hand-crafted knowledge and into more autonomous learning. There are many sources of knowledge out there, and cognitive architectures are uniquely positioned to not only learn from them, but to do so in a principled way.

Our experiences developing the Companion knowledge stack suggest a few lessons that other cognitive architecture developers may find useful.

First, simple design decisions can have a large impact on where future complexity lies. For example, the decision to use plaintext KRF files and version control software to store knowledge has shaped how the knowledge stack has evolved. Version control simplifies the responsibilities of the Archivist greatly, shifting the design discussion to where and when knowledge should be stored, rather than how. At the same time, the challenges of working with independent KRF files required the creation of the provenance cache to resolve. Such tradeoffs are difficult to anticipate, but they are worth considering in light of a cognitive architecture's long-term goals.

Second, use existing software whenever appropriate. This suggestion may seem obvious, but it is worth repeating, especially when dealing with bespoke software like cognitive architectures. Not every problem is worth solving on one's own, and off-the-shelf tools can fill important gaps. PlanB would not work nearly as well without AllegroCache or a similar high-performance database under the hood, while Subversion has proven its worth as both a development tool and a knowledge repository. External dependencies come with a cost, but they are well worth considering.

Finally, our experience suggests that it is useful to "close the loop" whenever possible. Tasks that cause friction for the developers are likely the result of a missing abstraction or capability. It is often worth the time to fix these sources of friction when possible. This not only helps developers move on to more worthwhile problems, but it can open up new avenues for autonomy, as tasks that were once the sole responsibility of a person can be delegated to the architecture itself.

CRediT authorship contribution statement

Constantine Nakos: Software, Writing – original draft, Conceptualization, Writing – review & editing, Methodology. **Kenneth D. Forbus:** Writing – review & editing, Project administration, Conceptualization, Software, Funding acquisition, Supervision, Methodology.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Constantine Nakos reports financial support was provided by Office of Naval Research. Kenneth D. Forbus reports financial support was provided by Air Force Office of Scientific Research. Kenneth D. Forbus reports financial support was provided by Office of Naval Research. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank Abhishek Sharma for his helpful comments about Cyc, Madeline Usher for the design and implementation of the Archivist, Samuel Hill for the conversations that led to the creation of the provenance cache, and the many QRG members who have contributed to NextKB, PlanB, and the entire Companion knowledge stack over the years. The authors would also like to thank the reviewers for their insightful comments. This research was sponsored by the US Office of Naval Research under grant #N00014-20-1-2447 and by the US Air Force Office of Scientific Research under award number FA95550-20-1-0091.

Data availability

No data was used for the research described in the article.

References

Anderson, J. R., & Gluck, K. A. (2013). What role do cognitive architectures play in intelligent tutoring systems?. In *Cognition and Instruction* (pp. 227–261). Psychology Press.
Baker, C. F., Fillmore, C. J., & Lowe, J. B. (1998). The berkeley framenet project. COLING 1998 Volume 1: The 17th International Conference on Computational Linguistics.
Blass, J. (2023). *Extracting and applying legal rules from precedent cases. Doctoral dissertation, Department of Computer Science*. Evanston, IL: Northwestern University.
Chen, K. (2023). *Visual understanding using analogical learning over qualitative representations*. Northwestern University, Evanston, IL: Department of Computer Science. Doctoral dissertation.
Crouse, M. (2021). *Question-answering with structural analogy. Doctoral dissertation, Department of Computer Science*. Evanston, IL: Northwestern University.
Dehghani, M. (2009). *A cognitive model of recognition-based moral decision making. Doctoral dissertation, Department of Electrical Engineering and Computer Science*. Evanston, IL: Northwestern University.
Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational*

Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers) (pp. 4171-4186). Minneapolis, MN: Association for Computational Linguistics.

Etzioni, O., & Weld, D. (1994). A softbot-based interface to the internet. *Communications of the ACM, 37*(7), 72–76. https://doi.org/10.1145/176789.176797

Forbus, K. (2016). *Software social organisms: Implications for measuring, 37* pp. 85–90). AI progress. AI Magazine.

Forbus, K.D., Hinrichs, T.R., de Kleer, J., Usher, J.M. (2010). FIRE: Infrastructure for experience-based systems with common sense. In: *AAAI Fall Symposium: Commonsense Knowledge*.

Forbus, K. D., & Hinrichs, T. (2017). Analogy and qualitative representations in the Companion cognitive architecture. *AI Magazine, 38*, 34–42.

Forbus, K., & Kuehne, S. (2007). Episodic memory: A final frontier (Abbreviated version). In: *Proceedings of AI for Interactive Digital Entertainment (AIIDE07)*. Palo Alto, CA.

Forbus, K., Gentner, D., & Law, K. (1995). MAC/FAC: A model of similarity-based retrieval. *Cognitive Science, 19*, 141–205.

Forbus, K., Usher, J., Lovett, A., Lockwood, K., & Wetzel, K. (2011). CogSketch: Sketch understanding for cognitive science research and for education. *Topics in Cognitive Science, 3*, 648–666.

Forbus, K. D., Ferguson, R. W., Lovett, A., & Gentner, D. (2016). Extending SME to handle large-scale cognitive modeling. *Cognitive Science, 41*, 1152–1201.

Guha, R. V. (1992). *Contexts: A formalization and some applications.. Doctoral dissertation, Department of Computer Science*. Stanford, CA: Stanford University.

Hancock, W., & Forbus, K. (2021). Qualitative spatiotemporal representations of episodic memory for strategic reasoning. In: *Proceedings of the 34th International Workshop on Qualitative Reasoning*. Montreal, Canada.

Hancock, W., Forbus, K., & Hinrichs, T. (2020). Towards qualitative spatiotemporal representations for episodic memory. In: *Proceedings of the 33rd International Workshop on Qualitative Reasoning*. Santiago de Compostela, Spain.

Herschel, M., Diestelkämper, R., & Ben Lahmar, H. (2017). A survey on provenance: What for? What form? What from? *The VLDB Journal, 26*, 881–906.

Hinrichs, T., & Forbus, K. (2019). Experimentation in a model-based game. In: *Proceedings of the Seventh Annual Conference on Advances in Cognitive Systems*. Cambridge, MA.

Hinrichs, T., & Forbus, K. (2011). Transfer learning through analogy in games. *AI Magazine, 32*(1), 72–83.

Hinrichs, T., & Forbus, K. (2014). X goes first: Teaching simple games through multimodal interaction. *Advances in Cognitive Systems, 3*, 31–46.

Kandaswamy, S., & Forbus, K. (2012). Modeling learning of relational abstractions via structural alignment. In: Proceedings of the 34th Annual Conference of the Cognitive Science Society (CogSci). Sapporo, Japan.

Kirk, J. R., & Laird, J. E. (2014). Interactive task learning for simple games. *Advances in Cognitive Systems, 3*(13–30), 5.

Laird, J. E. (2012). *The Soar cognitive architecture*. Cambridge, MA: MIT Press.

Laird, J. E., Lebiere, C., & Rosenbloom, P. S. (2017). A standard model of the mind: Toward a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. *AI Magazine, 38*(4), 13–26.

Langley, P., Laird, J. E., & Rogers, S. (2009). Cognitive architectures: Research issues and challenges. *Cognitive Systems Research, 10*(2), 141–160.

Lenat, D. B. (1995). CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM, 38*, 33–38.

Lenat, D. (1998). *The dimensions of context-space (Technical report)*. Austin, TX: Cycorp Inc.

Lenat, D. B., & Guha, R. V. (1991). The evolution of CycL, the Cyc representation language. *ACM SIGART Bulletin, 2*(3), 84–87.

McGuinness, D. L., Ding, L., Da Silva, P. P., & Chang, C. (2007). PML 2: A modular explanation interlingua. *ExaCt, 2007*, 49–55.

McShane, M., & Nirenburg, S. (2021). *Linguistics for the Age of AI*. MIT Press, 10.7551/mitpress/13618.001.0001.

Moreau, L., Groth, P., Cheney, J., Lebo, T., & Miles, S. (2015). The rationale of PROV. *Journal of Web Semantics, 35*, 235–257.

Nakos, C., Kuthalam, M., & Forbus, K. (2022). A framework for interactive natural language debugging. In: *Proceedings of the Tenth Annual Conference on Advances in Cognitive Systems*. Arlington, VA.

Olson, T., Salas-Damian, R., & Forbus, K. (2024). A defeasible deontic calculus for resolving norm conflicts. In: *Proceedings of the Eleventh Annual Conference on Advances in Cognitive Systems*. Palermo, Italy.

Olson, T., & Forbus, K. (2021). Learning norms via natural language teachings. In: *Proceedings of the Ninth Annual Conference on Advances in Cognitive Systems*.

Olson, T., & Forbus, K. (2023). Mitigating adversarial norm training with moral axioms. In: *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence*. Washington, DC.

Olson, J. R., & Rueter, H. H. (1987). Extracting expertise from experts: Methods for knowledge acquisition. *Expert Systems, 4*(3), 152–168.

Pérez, B., Rubio, J., & Sáenz-Adán, C. (2018). A systematic review of provenance systems. *Knowledge and Information Systems, 57*, 495–543.

Pinheiro da Silva, P., McGuinness, D. L., & McCool, R. (2003). Knowledge provenance infrastructure. *IEEE Data Engineering Bulletin, 26*, 26–32.

Ribeiro, D. (2023). *Reasoning and structured explanations in natural language via analogical and neural learning. Doctoral dissertation, Department of Computer Science*. Evanston, IL: Northwestern University.

Ribeiro, D. & Forbus, K. (2021). Combining analogy with language models for knowledge extraction. In: *Proceedings of the Third Conference on Automatic Knowledge Base Construction*. Virtual.

Shafer, G. (1976). *A mathematical theory of evidence*. Princeton, NJ: Princeton University Press.

Siegel, N., Goolsbey, K., Kahlert, R., & Matthews, G. (2004). *The Cyc system: Notes on architecture (Technical report)*. Austin, TX: Cycorp Inc.

Sikos, L. F., & Philp, D. (2020). Provenance-aware knowledge representation: A survey of data models and contextualized knowledge graphs. *Data Science and Engineering, 5*, 293–316.

Tomai, E., & Forbus, K. (2009). EA NLU: Practical language understanding for cognitive modeling. In: *Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference*. Sanibel Island, FL.

Wilson, J. R., Chen, K., Crouse, M., Nakos, C., Ribeiro, D., Rabkina, I., & Forbus, K. D. (2019). In *Analogical question answering in a multimodal information kiosk*. Cambridge, MA.