Cogn Tech Work (2003) 5: 140–151
DOI 10.1007/s10111-003-0120-4

ORIGINAL ARTICLE

H. Putzer · R. Onken

COSA – A generic cognitive system architecture based on a cognitive model of human behavior

Received: 1 July 2002 / Accepted: 25 January 2003 / Published online: 11 April 2003

Abstract This article presents an overview of COSA, a cognitive system architecture, which is a generic framework proposing a unified architecture for cognitive systems. Conventional automation and similar systems lack the ability of cooperation and cognition, leading to serious deficiencies when acting in complex environments, especially in the context of human-computer interaction. Cognitive systems based on cognitive automation can overcome these deficiencies. Designing such artificial cognitive systems can be considered a very complex software development process. Although a number of developments of artificial cognitive systems have already demonstrated great functional potentials in field tests, the engineering approach of this kind of software is still a candidate for further improvement. Therefore, wide-spread application of cognitive systems has not been achieved yet. This article presents a new engineering approach for cognitive systems, implemented by the COSA framework, which may be a crucial step forward to achieve a wide-spread application of cognitive systems. The approach is based on a new concept of generating cognitive behaviour, the cognitive process (CP). The CP can be regarded as a model of the human information processing loop whose behaviour is solely driven by "a-priori knowledge". The main features of COSA are the implementation of the CP as its kernel and the separation of architecture from application leading to reduced development time and increased knowledge reuse. Additionally, separating the knowledge modelling process from behaviour generation enables the knowledge designer to use the knowledge representation that is best suited to his modelling problem. A first application based on COSA implements an autonomous unmanned air vehicle accomplishing a military reconnaissance mission. Some of the application experiences with the new approach are presented.

Keywords Cognitive engineering · Architecture · Knowledge modelling · Autonomous unmanned air vehicle · SOAR · CORBA · UML

H. Putzer (✉) · R. Onken (✉)
Institut für Systemdynamik und Flugmechanik,
Universität der Bundeswehr München,
85577, Neubiberg, Germany
E-mails: henrik.putzer@unibw-muenchen.de,
reiner.onken@unibw-muenchen.de

1 Introduction

1.1 Motivation

The continuously increasing complexity of environments and the extension of the range of operation put great demands on the cockpit crew and aircraft performance. As a consequence, more and more automated functions were introduced in aircraft cockpits in the past and this tendency is still ongoing. The result so far is that productivity (effectiveness and safety) could be increased. Investigations on modern aircraft cockpits show, however, that a further increase in the use of automation will not necessarily result in increased productivity because automation itself has become a highly complex element within the already complex environment of the cockpit. Meanwhile, in some cases automation has in fact become the key factor for decreased safety (e.g. "mode confusion") (Fig. 1).

Extensive research has been done to analyse this situation and it was found that highly complex *conventional automation* may act in unpredictable ways, e.g. not consistent with the pilot's mental models, and may provide the flight crew members with too little feedback (Wiener 1993; Sarter and Woods 1995; Billings 1991). In most cases, conventional automation is used in more or less stand-alone systems. There is no integrated approach. The coupling of elements of automated functions and the complexity of automation as such increase the load on the human operator. A new integrated approach based on *cognitive automation* overcomes these deficiencies

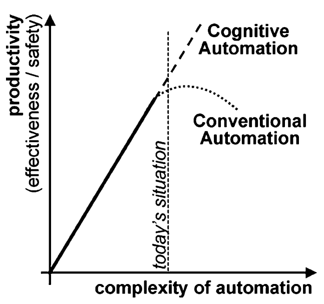


Fig. 1 Productivity gain through different kinds of automation

by making use of cognition, working in cooperation with the human operator and showing goal-consistent and transparent behaviour.

1.2 Some comments on cognitive automation

Cognitive automation is an extension of conventional automation (as reported in Walsdorf et al. 1999; Frey et al. 2001) and follows the human knowledge-processing scheme as discussed, for example, by Rasmussen (1983). As shown in Fig. 2, cognitive automation implements the whole process of building an internal comprehensive representation of the relevant parts of its external world (left column in Fig. 2). This can be considered the principal basis upon which all considerations and actions of the system, especially for the crucial part of goal-driven knowledge-based behaviour (top level row in Fig. 2), are created.

Another important feature of systems based on cognitive automation is the ability to cooperate with the environment, especially with the human operator. Synergic benefit is gained by supporting the strengths of both the human operator (instinct, abstraction, creativity, etc.) and the automatic system (objectivity, stress resistance, parallel processing, complex calculations, etc.). Cooperation based on cognitive automation means implicitly and explicitly communicating with the crew in order to continuously match the technical system's knowledge about the situation against the flight mission goals and the intents and actions of the crew. If necessary or requested by the crew, actions are taken which comply with the crew's goals. Further details about the concept of cognitive automation as well as the application and evaluation of this concept can be found in the literature (Walsdorf et al. 1999; Putzer and Onken 2001; Frey et al. 2001).

Although the application of cognitive automation in this article is aircraft guidance, the approach of cognitive automation is equally appropriate in other domains with technical processes controlled by human operators such as road vehicle guidance, power plant control, or even business management.

1.3 Scope of this article

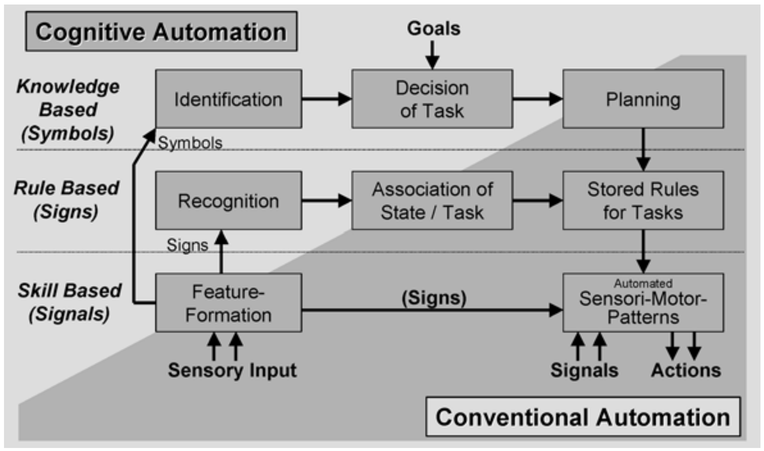
The theoretical foundation of cognitive systems and their application is a key aspect in the area of human-machine interaction leading to assistant systems, tutoring systems, or intelligent, cooperative agents. It is still a great challenge, however, to design and implement these systems. Usually, the system design leads to implementations with no or little reuse, for the sake of cost reduction, when the next generation of the same kind of system is designed.

This article understands the design of cognitive systems as a software engineering process that is founded on the concept of cognitive processing. The following approach consists of two main steps.

1. As a conceptual basis a generic structure of the cognitive process is generated. This is covered in Sect. 2, The cognitive process.
2. A framework is implemented using the concept of the cognitive process as its kernel. This is presented in Sects. 3 and 4, Design principles and Architecture.

The subsequent Sect. 5, Implementation, discusses the implementation of the framework along with a first application, which is an autonomous, unmanned air vehicle (UAV).

Fig. 2 Cognitive automation



2 The cognitive process

2.1 Introduction

The cognitive process (CP) is a technical process which mimics human information processing. This concept is based on knowledge about human behaviour and known design philosophies for cognitive systems. It is not designed to verify or to comply in detail with the theories about physiological processes and features of the human brain. It is, rather, motivated by the need to model behaviour that is similar to the main stream of behavioural characteristics of humans. In essence, this should lead to a behaviour of the artificial process that results from a profound understanding of the human operator's intentions and needs, and, in turn, the system's behaviour should be well understood by a human operator.

The central component of the CP is the "body", the oval part in Fig. 3, which hosts a large amount of data. These data represent the knowledge of the system that is specific to the CP-subprocesses and to the application. The inner oval (slightly darker) contains the *a-priori knowledge* that is fed into the CP before any processing starts. This a-priori knowledge is the source of the behaviour of the application. The outer oval (light grey) contains the situational knowledge which is created during runtime. This kind of knowledge is also called the *cognitive yield* because it results from the operation of the CP subprocesses.

Processing of the knowledge is done by the so-called *transformators*, which are represented by dashed and solid lined arrows around the body in Fig. 3. These transformators can read from the whole body to retrieve their necessary input but usually get their most important input from the dashed area within the body, the cognitive yield. Transformators write their results into designated areas of the body at the arrowhead. The functions of all transformators are designed according to the recognition-act cycle (in Fig. 3, clockwise around the body): *interpretation*, *goal determination*, *planning*, and *plan realization* (Walsdorf et al. 1999; Jennings and Wooldridge 1998). Additionally, two specialized transformators establish the interaction with the outside world: via the *input interface* the data get into the body and via the *output interface* external actuators and other output devices can be controlled.

The execution of all transformators is assumed to be in parallel and event-driven. As soon as a transformator detects a significant change at its primary input this change is evaluated using the available knowledge and the output is adjusted. A direct consequence is that there will be no unnecessary recalculation of the plan, if changes in the situation stay within predicted bounds.

2.2 Tranformators

For a better understanding of the functions and interactions of all transformators within the cognitive process a closer look at each of them is given.

Interpretation This transformator generates the complete picture of the situation. As a simple example, one can imagine taking sensory data from a radar device and creating a structure within the area of "beliefs" that reflects the knowledge (e.g. position, heading and speed, behaviour) about other aircraft. Besides natural objects, this transformator may also establish structures reflecting relations. For example, a structure describing the binary relation of distances between the own vehicle and other objects could be established.

Goal determination Primarily using the beliefs as input, this transformator determines which relevant goals should actually be pursued. This is done by evaluating the desires from the a-priori knowledge. For example, in a military scenario, if there is an incoming missile, the desire of surviving (e.g. by evading) is activated. Activated desires are called goals.

Planning Using the goals as triggering elements and the a-priori knowledge about strategy models, this transformator generates a plan of how to achieve these

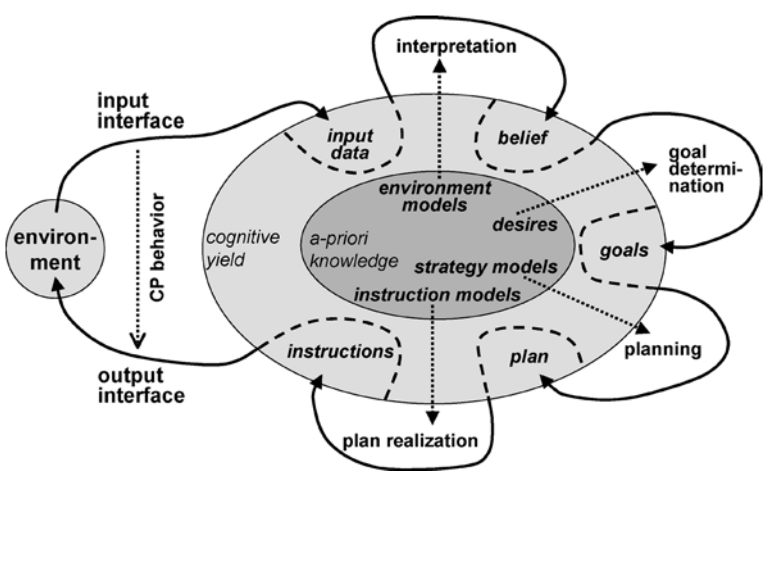


Fig. 3 The cognitive process

goals. This plan may contain parallel tasks as well as sequential ones. It may also contain alternative operations. However, the plan is hierarchically structured and has a time span up to the end of the current mission.

Plan realization The primary input to this transformator is the plan. The tasks to be processed at each point in time are selected from the hierarchical structure of the plan, associated with the appropriate instruction model, and converted into simple instructions that constitute the output of the CP.

2.3 The body

If necessary and if modelled by the a-priori knowledge, the behaviour of these transformators covers all three levels of (human) behaviour according to Fig. 2. It is solely based on a-priori knowledge specific to each transformator. Without this knowledge the transformators will do nothing. The basic idea about the purpose of the body is that simply by adding knowledge all transformators know how to process this knowledge in combination with newly added knowledge with the previous knowledge.

The organization of the knowledge within the body follows the *object-oriented* paradigm: it has a uniform structure in terms of models (= objects). Each model can be instantiated. Instances have data members describing their state. The model also comes with template functions that describe the behaviour of all instances. This concerns the whole life cycle of instances including creation, behaviour during life time, and removal. Tying up all *micro-behaviours* of all model instances forms the *macro-behaviour* of the CP.

2.4 Application of the CP

It can be stated that the concept of the CP will be adequate for a wide range of cognitive system applications. It is independent of domains and characteristics of the application. Thus, the CP serves as an unchanged core element in arbitrary applications such as tutorial systems and assistant systems or even autonomous systems within any domain such as aeronautical applications, driving, and managing power plants. The domain and the kind of application shows up solely in the a-priory knowledge that is put into the CP. Before doing so, the CP concept has to be transformed into a design, architecture, and implementation of an operational framework, which we call COSA. These steps are described in the following sections.

3 Design principals

Now that there is a common concept for cognitive systems available in the form of the cognitive process, this concept has to be implemented into an operational framework. This framework has to provide:

- A new software engineering concept (based on the cognitive model represented by the CP)
- An architecture separating cognitive functions from the application on the basis of implementing the system's behavioural traits exclusively from a-priori knowledge
- Separating knowledge modelling from behaviour generation by introducing the concept of the front-end for knowledge modelling

Besides presenting the concept of the CP, the framework design and implementation is the main topic of this article. This kind of unified framework is most suitable to make cognitive system design common property.

3.1 User requirements

In most cases of current developments special designs are used to accomplish certain isolated functions of cognitive system applications. With COSA we try to design a system that uses results from several sources and combines them to get an improved, holistic, and unified architecture for cognitive systems in general. This section will give an overview on the design and the design goals from a higher-level perspective leaving detailed aspects to be explained in later sections.

The first step towards this framework was the analysis of prior designs of cognitive systems like CAMA (Walsdorf and Onken 1998; Schulte and Stütz 1998) and other state-of-the-art systems. Further user design requirements were derived from three different groups of users: system developers, knowledge engineers, and system end-users.

- **System developers** need flexibility to add functionality or external components like displays, database interfaces, or image analysis core functions. For this task they need good documentation, simple interfaces for internal and external subsystems, and good modularity and support to easily extend and maintain the system. These design goals influenced the architecture and implementation as described in Sect. 4.
- **Knowledge engineers** design the a-priori knowledge to implement the application system behaviour. They need an interface to any potential modelling language to choose the one that is best suited for their particular application domain. Further on, they need a design methodology to model the knowledge and a structured concept to partition the knowledge. This is supported by the *front-end*, which is described in Sect. 4.4.
- **System end-users** of the resulting cognitive application put some more general requirements on the system. These are described in the following paragraphs along

with other features derived from our own experience with artificial cognition.

3.2 Features of the framework

The COSA framework is designed to ensure a unified architecture of cognitive systems. Thus, it can serve as a basis for a wide range of applications in cognitive engineering. In its actual implementation it is based on the CP, which is described in Sect. 2. The CP as the core element of the system makes all its properties available for the resulting framework.

The CP approach follows the human-centred design by making the behaviour of the resulting system similar to human information processing. This ensures system behaviour that is best comprehensible to human operators.

The core element of COSA, which is the CP, is designed to carry out high-level knowledge processing. It is not designed to implement number-crunching algorithms or high frequency control loops. As it will turn out, this is not a restriction for an application because there are flexible interfaces to external modules (COSA components) that can cope with such functions.

Certification is not yet addressed in the first implementation of COSA. It can be stated, though, that COSA supports approaches to strengthen system integrity.

4 Architecture

4.1 Overview

4.1.1 Subsystems

The COSA framework is a component architecture. It is built on a distribution layer of CORBA, which is an industry standard for distributed systems (Puder and Römer 1999). CORBA ensures clear external and internal interfaces and decoupling of functional components.

Components are grouped into subsystems of the architecture. These subsystems are marked at the left side in Fig. 4 and are described as follows.

- The *kernel* encapsulates the knowledge processing engine by which the CP is implemented. This group does not contain any knowledge specific for any application domain. This part of the architecture only ‘‘knows’’ how to convert knowledge into behaviour. It is the organizational element in terms of the controller that integrates all distributed components of the framework to form one integrated system. At runtime the kernel manages all registered COSA components and compilers via special interfaces (see arrows starting at the ‘‘manager’’ of the ‘‘controller’’ component in Fig. 4).
- The *application* consists of components of a special kind, the so-called ‘‘COSA components’’. These are the only components that contain domain-specific knowledge. Under control of the kernel during start-up this knowledge is translated by compilers into a-priori knowledge which complies with a format that can be processed by the CP. In addition to knowledge, COSA components may contain servers for calculations (external to the CP) or interfaces to other systems. These could be data base interfaces, HMI-interfaces, image processing algorithms, or closed-loop feedback control.
- The *front-end* covers all tools for knowledge modelling and debugging. Thus, it serves as the front-end for the knowledge engineer for application development and debugging. Furthermore the front-end provides a compiler to convert the designed knowledge to run on top of the cognitive process.

With these subsystems an important feature of the COSA framework is described: the separation of appli-

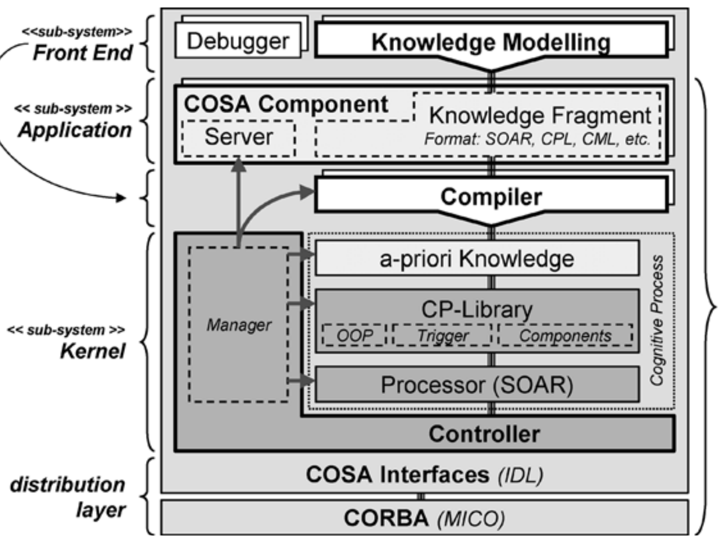


Fig. 4 Block diagram of COSA’s component architecture

cation and architecture. The reusable architecture saves time during the development of cognitive systems, because basic architectural problems are already solved. Further on, it provides behaviour generation on the basis of explicit knowledge (environment models, desires, etc.) leading to great flexibility for the application development and the reusability of components.

4.1.2 Abstraction levels of knowledge

Another view of the block diagram of COSA in Fig. 4 (vertical triple line) identifies hierarchical levels of knowledge abstractions.

- The top level and most abstract layer is given by the knowledge designed by the knowledge engineer in a high-level language by means of the front-end. This knowledge is incorporated into COSA components with no change. The set of knowledge fragments from all COSA components of an application is exactly the domain specific knowledge of that application.
- The next level is the knowledge translation by compilers. The result of the compilation is a-priori knowledge in a format that can be processed within the CP. This translation is "behaviour preserving", i.e. the behaviour that evolves from original and translated knowledge has to be the same.
- By providing the interfaces of the CP as described in Sect. 1.3, the CP library is the link to the next level of abstraction. Here, the a-priori knowledge on the basis of the CP library is converted into behaviour by the processor. The level of abstraction and the processing speed is determined by the choice of the processor. For the current implementation, SOAR, a production system (Sect. 4.2.2), is chosen. For other requirements with less abstraction but more complex CP library implementations and possibly more speed, other processors like the JAVA virtual machines can be used.

This overview gives important aspects of the architecture on an abstract level. More details on the architecture and functionality of all components within their groups are given in the next sections.

4.2 Kernel

This subsystem has two major responsibilities.

1. Manage all other components of the framework – this is done by the *controller* with the help of its managers and adapters.
2. Implement the CP – this is done by the *SOAR processor* and the *CP library*.

In the following section a detailed look is taken at these components of the "kernel" subsystem: controller, processor, and CP library.

4.2.1 Controller

The controller can be regarded as the central manager in the COSA architecture. At start-up all components like compilers provided by front-ends and COSA components register with the controller. After the start-up process the controller takes control over all components, searches for the knowledge of COSA components, translates it with the appropriate compiler, and loads the result into the processor. During this starting phase (*offline phase*) dependencies of the components are checked and resolved.

After the starting phase the control is transferred to the processor which processes all loaded knowledge to produce behaviour (*runtime* or *online phase*).

4.2.2 Processor

As the central element of the "kernel" a component is needed that can process knowledge to yield behaviour. SOAR (Newell 1990; Laird et al. 1987), which is developed and maintained by the University of Michigan, is a good candidate because it meets all requirements. The main reasons for selecting SOAR as the processor are the following: it is easy to learn, the developing community is very active and reacts quickly to queries, and, last but not least, SOAR comes with portable source code and can be easily integrated into C/ C++ environments. Additionally, SOAR provides interfaces to integrate basic features that are not supported in its pure implementation.

SOAR stores its "knowledge" about the situation in its working memory, which has a uniform and symbolic structure similar to conceptual graphs. The "behaviour" is uniformly stored in rules; thus, SOAR is a kind of production system but with a very special multi-stage processing loop so that even advanced features like learning are supported.

Like other production systems, SOAR offers a very fine-grained interface: the rules. With this feature, extending existing knowledge models is as easy as writing new rules and loading them into SOAR (even at runtime). In SOAR, all loaded productions can fire in parallel so this supports the idea of cognitive automation: apply *all* knowledge at *any* situation.

It turns out that maintenance can be done as easy as extensions: the debug code is a set of rules that can be loaded into SOAR at any time. This code can trace values, set breakpoints, or just print out portions of the working memory. This is supported by a symbolic representation of the working memory that is understandable by human beings.

Like many other architectures that build on production systems, SOAR has deficiencies in implementing number crunching algorithms (e.g. frequency analysis, closed-loop feedback control) or image processing. But this is not the job of the core processor within COSA. Instead, the processor is used to implement higher decision levels while the above-mentioned functionality is implemented in extensions that are external to the CP residing in COSA components.

Further on, some key features on the knowledge level are added to SOAR by loading a set of basic productions called the CP library, thereby giving it, for example, the means to organize knowledge in components and to implement the CP, which should be the kernel of COSA.

4.2.3 CP library

The higher the abstraction of programming languages, the more functionality can be covered with a single expression. Thus, we do not use C + + but rather the more abstract SOAR language to simplify the implementation of the CP as it is described in Sect. 1.3.

This step of building the abstract interface level of the CP is the function of the CP library. This library is combined with the a-priori knowledge and both are executed by the SOAR processor. Three main features of the CP library lead to the required functionality.

1. **Timers/triggers** handle simultaneous events and synchronization throughout the whole system. They are needed internally to the CP library and are not used in the actual application.
2. The **OO approach** introduces object-oriented design philosophies to SOAR. With this feature, SOAR supports "models" (= classes) as they are described in Sect. 1.3. Models can be instantiated, which is a similar process to instantiating a class in C + +. Instances contain data describing their individual state. Models also contain the behaviour for all instances throughout the whole lifecycle: creation, behaviour during life time, and removal.
3. **Component management** enables the kernel to keep track of activated components and determine dependencies and priorities. This is done in cooperation with the controller component of the "kernel".

As the CP library is implemented as a set of SOAR productions in separate files, these can be loaded into any running SOAR processor to yield a CP scheme. Again, it is emphasized that the CP as a basic element for cognitive applications does not include any domain-specific knowledge.

4.3 Distribution layer

COSA's distribution layer is based on CORBA (Puder and Römer 1999), an industry standard for distributed systems that connects distributed software objects. It serves as middleware to connect the controller component to other components containing knowledge, I/O interfaces, or providing compilers. These components can be distributed over a network and may differ in programming language, operating system, and computer platform.

To use the SOAR processor within the CORBA environment, the interfaces of the processor have to be mapped to the middleware layer. The main feature of COSA's middleware layer are the following.

– The **client-server-structure** puts the controller into a central position. Components of the application register with the controller on start-up so that they can be used to retrieve a-priori knowledge or to access other interface functions.
– The **knowledge mapping** is the link between the different knowledge representations in SOAR and in the transport layer CORBA. It defines the representation CORBA uses to transport any piece of knowledge via the network from the processor to other objects and vice versa. The mapping is done by implementing a graph structure for CORBA.
– The **encapsulation of callbacks** connects I/O functions and internal events of the knowledge processor to published member functions of distributed objects. This is done by dispatching calls to registered member functions of objects.

With these features COSA can dispatch tasks to processing units external to the kernel, which are servers as a part of COSA components (Fig. 4). COSA components and with them all servers can be distributed objects connected via a computer network. This enables the flexibility to integrate servers, black boxes, and external systems to do number crunching, implement high frequency control loops, interface data base systems, or connect any other subsystem to the application.

4.4 Front-end

The main purpose of the front-end is to decouple different representations of the knowledge needed for knowledge modelling and knowledge processing. This allows the knowledge engineer to use the modelling tool or environment best suited to his work, without the necessity to know or learn the format in which the processor is programmed. This way object-oriented or procedural approaches for any standard of knowledge modelling can be supported or even mixed to yield a joint behaviour.

4.4.1 Theory behind front-ends

The idea of the concept of a front-end is based on a direct consequence of behaviourism: it does not matter which syntax or mechanism is used to describe the knowledge as long as the evolving behaviour is the same.

Thus the description of an intended behaviour by means of a specific knowledge representation or format should be separated from the process of generating the behaviour from the modelled knowledge. This is exactly the purpose of the front-end. The behaviour generation is delegated to the COSA kernel while the various front-ends provide the modelling facilities.

This separation is used to reach a high abstraction for the knowledge modelling process. Modelling takes place in terms of "mental concepts" such as beliefs or goals as they are defined by the CP. This can be seen as a further step in the evolution of software engineering methods (Fig. 5; compare Balzert 2000). This increase of abstraction enables human beings to understand und cover an increasing complexity of applications as is necessary for cognitive systems.

4.4.2 *How to define a front-end*

A front-end as it is invented for the COSA framework is defined by the following five aspects: concept, language, compiler, method, and tools.

– The **concept** represents the basic idea of how a cognitive system should produce its behaviour. For example, the CP can be used as the concept.
– A **language** is needed to express the knowledge that is to be modelled in a formal syntax. Text-based representations as well as graphical notations are possible. Examples will be given in Sect. 4.4.3.
– The **compiler** translates the language with its underlying concept into the knowledge format of the processor such that it can be executed by the kernel. As mentioned before, the compiler is one of the most crucial elements of a front-end: the compiler not only has to convert knowledge formats but also underlying concepts. A fragment of knowledge and its compiled version both have to represent the same knowledge in terms of producing the same behaviour.
– For knowledge modelling a **method** is needed. This method defines a guideline for the knowledge engineer on how to create an application using the given concept and language of the front-end.
– **Tools** are optional elements of the front-end. They support the knowledge modelling process. The minimal tool is a text editor to produce a textual representation.

In more sophisticated systems it might be advantageous to provide several front-ends at the same time so the knowledge engineer can use the front-end and with it the design method which is best suited to his design problem. At runtime, the controller in COSA's "kernel" searches all registered front-ends for the appropriate compiler to translate the knowledge. All compilers produce code fragments that are integrated in the processor to yield the system's behaviour.

At the current state of the project, besides using native SOAR representations, there are two other front-ends. One is called the cognitive process language (CPL), which is based on SOAR and extends SOAR with elements of the object-oriented paradigm. An example discussing and using the CPL front-end is presented in the next section (Sect. 4.4.3). The other front-end is based on the standard of the CommonKADS Markup Language (CML; Schreiber et al. 1999). It is still at an experimental stage.

4.4.3 *Example using CPL*

To be more concrete on the knowledge modelling aspect, a short example is presented using the language, method, and tools of the CPL front-end. The example will be a rough sketch of the behaviour of an autonomous unmanned air vehicle (UAV) that decides to evade an aircraft classified to be dangerously close. Most of the example can be found in the screenshot of the graphical tool "visualCPL" in Fig. 6, which was used to design the example. The tool "visualCPL" is part of the CPL front-end.

The CPL method is based on the ideas of "agent-oriented software engineering" (AOSE; Wooldridge 1997) and adjusted to the concepts of the CP. The goal of the method is the design of the a-priori knowledge from which the behaviour of the resulting system is generated by the kernel. The method starts by defining the *static model* within four steps.

1. The main scope is to model goal-oriented behaviour. Thus, the first step defines the goals that the system tries to achieve. For the given example the goal is to evade from a dangerously close aircraft. This leads to the description of the "evade-goal".
2. Considering the next transformator "planning" (Fig. 1), the ability to achieve the "evade-goal" has to be implemented as a parameterized procedure composed of elementary actions. This results in the a-priori knowledge (strategy models) modelling how the UAV safely evades the aircraft. This knowledge is represented by the "class evade-plan" in Fig. 6.

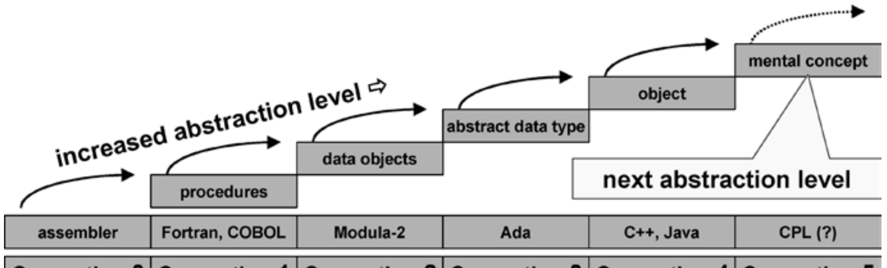
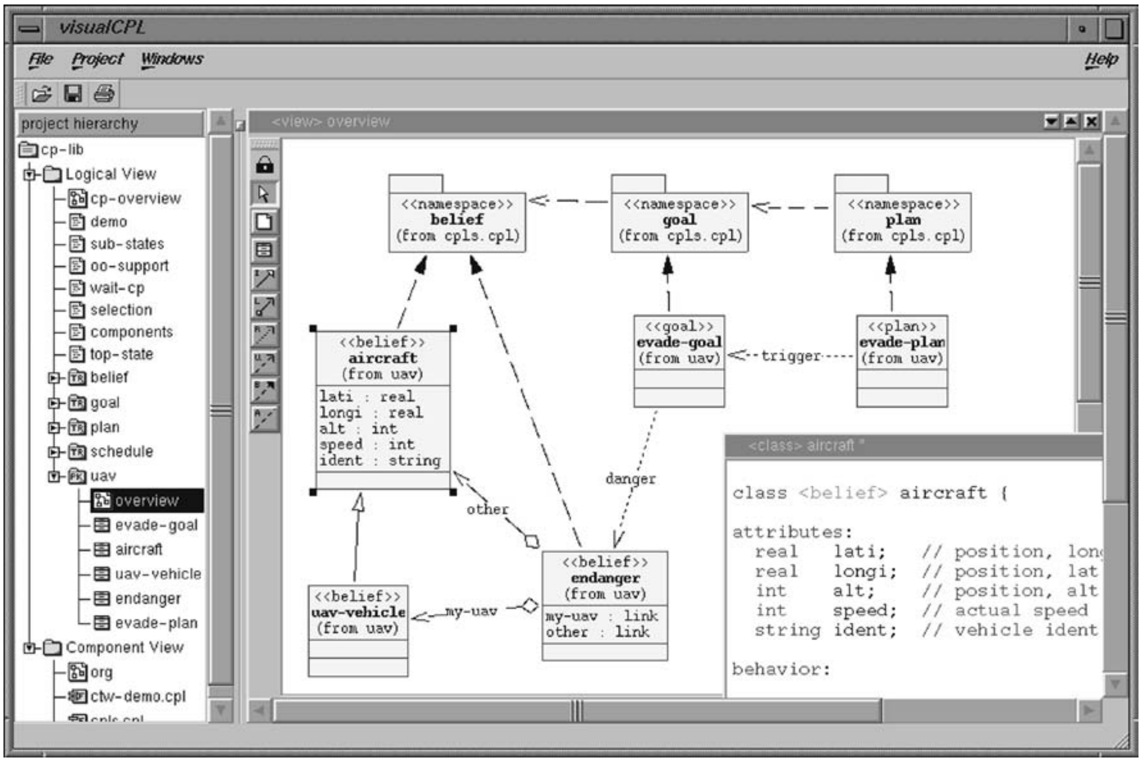


Fig. 5 Evolution of software design methods over 6 generations of paradigms



```
class <belief> aircraft {

attributes:
  real    lati;    // position, lon
  real    longi;   // position, lat
  int     alt;     // position, alt
  int     speed;   // actual speed
  string  ident;   // vehicle ident

behavior:
```

Fig. 6 visualCPL – a graphical tool of the CPL front-end

3. The transformator "plan-realization" gets its a-priori knowledge (instruction models) by analysing the strategy models and implementing how all elementary actions are to be executed. This knowledge is implemented in the package of UAV control. To keep the example simple this is not displayed in Fig. 6.
4. As the last step towards the static model the environment models that form the belief of the system are built. Environment models build the UAV's information about the environment needed to support (1) the decision on goals modelled in step one, (2) the planning modelled in step two, or (3) the execution of instructions modelled in step three.

Within our example there is a class representing any (not further classified) aircraft (class "aircraft") and a derived class to represent the own vehicle, a UAV (class "uav-vehicle"). Furthermore there is an abstract class representing a danger (class "endanger"), which arises by a separation violation of the own vehicle and any other vehicle. In this simple example "separation violation" is the definition for "dangerous". Technically, the class "endanger" does not model a physical element of the environment but represents a binary relation.

These four steps result in the *static model* consisting of classes as described along with their specific attributes. A final step in the CPL method completes the model by analysing the behaviour of all classes resulting in the *dynamic model*: For each class the behaviour and its "ongoing interactions" with other classes are modelled.

Each class is to be considered in detail and the behaviour (triggered by a specific situation) of creation and removal is to be defined. For example, the "uav-vehicle" is created as soon as the interface to the hardware of the UAV is attached to the kernel. Another example is the class "aircraft" which creates an instance within the beliefs as soon as radar sensors detect the aircraft. This instance is removed upon disappearance of the radar blip.

The "ongoing interaction" for the class "endanger" includes distance measurements between the instance of "uav-vehicle" and any other instance of "aircraft". In the simplest case, an "evade-goal" is instantiated if the distance falls below a certain threshold, e.g. 15 nautical miles. A more sophisticated behaviour could take the speeds and directions of the vehicles into account as well as their military classification of friend or foe. This can simply be implemented by changing the constructors of the class "endanger".

Within visualCPL this step of building the dynamical model is supported by syntax-highlighting editors taking the text representation of CPL code (lower right in Fig. 6). Graphical representations may follow in the future. All graphical representations used in visualCPL rely on the standard of the unified modelling language (UML; D'Souza and Wills 1998).

5 Implementation

5.1 The COSA framework

The operating system used for our implementation is IRIX, a UNIX version from SGI. As the programming environment the IRIX native tools are used along with the latest versions of the packages listed below. These are distributed with source code and are portable to Windows and to many UNIX derivatives.

– **SOAR** as the implementation of the unified theory of cognition (Newell 1990)
– **MICO** which is a free and very good implementation of the CORBA standard (Puder and Römer 1999).
– **QT** library is used for the (graphical) user interface (Dalheimer 2002)
– **STL,** the Standard Template Library, provides basic types and containers. It is part of the IRIX native C+ + environment but is also distributed freely for other platforms (Musser et al. 1996).

The documentation for most parts is done with DOXYGEN (Heesch 2001), which generates the documentation from the C and C+ + structures by using code comments of a special format. Design tools are also used for some aspects of the implementation, such as Rational Rose (Quantrani 1999) and the already mentioned CPL tool "visualCPL". Both are based on the unified modelling language UML (D'Souza and Wills 1998).

5.2 Actual application

A first and relatively simple application is implemented on the basis of COSA. It is called COSY$^{\text{flight}}$, an autonomous "cognitive system for the flight-domain". It models an autonomous UAV during a military reconnaissance mission. The test bed used is a flight simulator with a simulated dynamic environment.

The focus of this application is to verify core features of COSA's architecture to some extent. In the context of this article, emphasizing the realization of the CP, the design excludes the human-UAV interaction and pertinent human factor issues. Rather, the focus lies with the autonomous cognitive behaviour of the UAV itself that evolves from the knowledge processed by the CP to yield goal-based behaviour complying with the actual situation.

5.2.1 The simulated environment

To better understand the environment of COSY$^{\text{flight}}$ along with the interfaces used, Fig. 7 shows all communication paths internal and external to the UAV.

– An external data link for messages to and from the human operator to receive a mission order or updates. The operator communicates with the UAV in the same way as she would communicate with a human pilot sitting in an aircraft's cockpit (except for a formal communication protocol to ease the UAV implementation).
– Internal connections to sensors (including TCAS sensors) to get information about the situation and, in turn, to build an internal representation.
– Internal connections to the radio unit for communication with the operator or other entities in the environment.
– Internal connection to the flight control system to control the UAV.
– Internal connection to data bases.

5.2.2 The scenario

The behaviour of the UAV is solely based on a-priori knowledge as it is proposed by the concepts of the CP. This knowledge enables the UAV to execute procedural tasks such as the execution of checklists. Further on, the UAV is able to act according to its goals and to cope with unexpected situations.

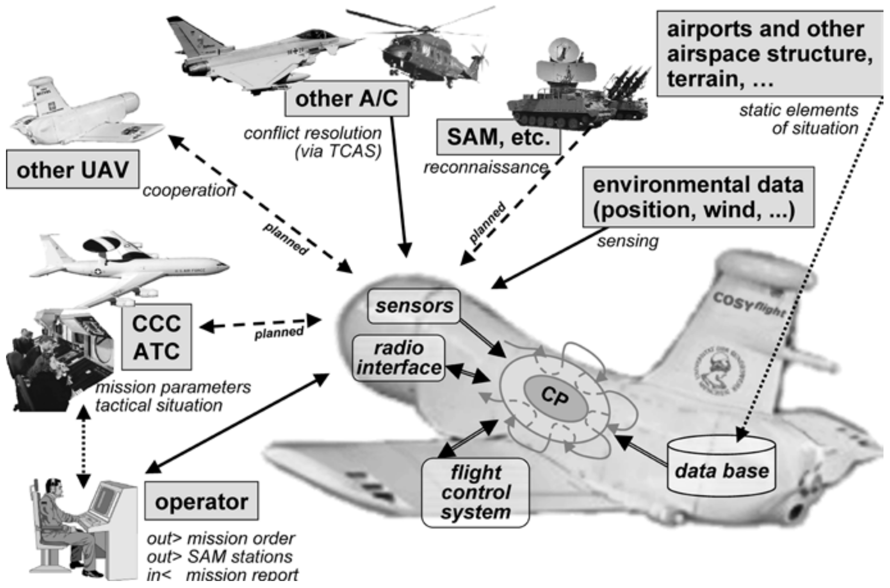


Fig. 7 Environment and communication of COSY$^{\text{flight}}$

The actual scenario as shown in Fig. 8 contains the following elements.

- The mission begins at a military *airfield* where the UAV communicates with the operator to receive the mission order and to report its state.
- After takeoff the *UAV* follows the flight path which is generated autonomously. The flight plan includes all mission waypoints (if possible) and other waypoints to control the flight path. It also includes flight segments to avoid *SAM-1*, a coverage zone of a surface-to-air missile station known at takeoff time.
- Suddenly, *unexpected traffic* is reported by the simulated on board traffic collision avoidance system (TCAS). The UAV autonomously classifies the other aircraft, calculates an evasion route and follows this new route.
- Passing the waypoint *overfly1* the UAV achieves the first mission goal.
- With *SAM-2* an unexpected new SAM station is reported by the operator. Depending on its mission goals the UAV autonomously reacts on this event by planning the new route *route B* or by continuing on *route A*. Except for the communication at the airfield, the notification of SAM-2 is the only interaction between the UAV and its operator. For the future it is planned to receive the new SAM station via a communication link to the command and control centre (CCC in Fig. 7) or by using on-board sensors and reconnaissance capabilities.
- At *reconnaissance2* and in case of route A, likewise on *reconnaissance1*, a picture is taken by the on-board camera.
- Via the flight path, the UAV returns to its base airfield, lands, and transmits its mission report to the operator.

For this rather simple, but illustrative, scenario the event of the appearing SAM-2 is one of the situations where the UAV demonstrates its goal-based behaviour: route A (dotted lines in Fig. 8) is taken if it is more important to accomplish the predominant mission goal of getting the picture than to warrant safety. In the other case "route B" (dashed line in Fig. 8) is taken if safety is the more important goal. These goals, including context-dependent weights, are explicitly modelled as part of the a-priori knowledge, based on the COSA front-end. They exclusively drive the COSY$^{\text{flight}}$ behaviour. This is hardly found in current applications.

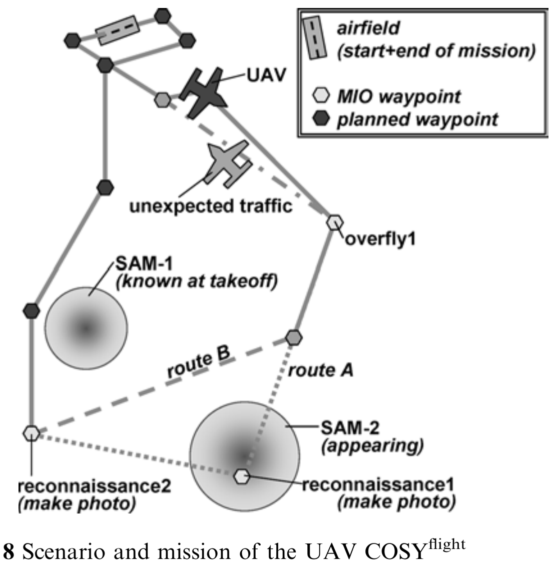


Fig. 8 Scenario and mission of the UAV COSY$^{\text{flight}}$

6 Results and prospects

It has been demonstrated in the literature that conventional automation and similar systems in complex environments reveal significant deficiencies. To overcome these deficiencies the concept of the cognitive process (CP) was developed. This concept takes into account the research results gained by projects on cognitive systems.

The CP is used as the basis for the design of the COSA framework, which is a uniform architecture for cognitive systems. COSA follows certain paradigms:

- A new software engineering concept based on the CP
- Knowledge reuse based on object orientation by encapsulation of knowledge in higher abstraction entities (objects)
- Context-dependent behaviour generated solely on the basis of knowledge
- Separation between architecture for cognitive mechanisms and application (similar to human cognition)
- Separation of knowledge modelling and behaviour generation by a front-end

Meeting the requirements of the three user groups system developers, knowledge engineers, and system end-users (see Sect. 3), COSA became a highly flexible and usable framework with an implementation based on free libraries. The implementation of the CP as the kernel for the uniform architecture for cognitive systems such as a SOAR library fits especially well.

The evaluation of COSA includes an application that models an autonomous unmanned air vehicle (UAV) called COSY$^{\text{flight}}$. The UAV communicates and interacts with its simulated environment to achieve its goals. Besides safety goals, one of its other goals is to fulfil a given military reconnaissance mission.

Results from the research up to now can be summarized as follows.

- The COSA framework eases the creation of cognitive systems, since basic architectural problems are resolved by it.
- The CP as the core concept to model and produce behaviour so far has proven to be a suitable approach.
- The CPL front-end on the basis of the COSA framework can reduce the time for development of

cognitive systems. This is accomplished by its high abstraction for knowledge modelling introducing "mental concepts" as building blocks. This approach eases knowledge reuse and makes it easier to understand complex knowledge models.

- With the application COSY^{flight} the usability of the COSA framework and the CPL front-end was shown.
- In terms of cognitive system modelling COSY^{flight} is based on a-priori knowledge. This and nothing else determines the system's behaviour as a reaction on the situational context.
- As it is proposed by the CPL method, the behaviour shown by COSY^{flight} is driven by explicit goals. This is rarely found in current applications.

For the near future, improvements of COSA are planned in many details of implementation, especially concerning the CP library. Although the computing performance of the system was not a design goal of high priority, the system at its actual implementation is capable of running in simulated real-time environments. There is still potential for some improvements, however, especially within the interface between MICO and SOAR. A research-related goal is the improvement of the language front-end and the modelling paradigm of mental concepts. Concerning the CPL tools the extension of the graphical design tool visualCPL and the introduction of symbolic debugging facilities are planned. Future work will also concentrate on the completion of COSY^{flight} and applications of other domains.

References

Balzert H (2000) Lehrbuch der Software-Technik 1, 2nd edn. Spektrum, Heidelberg

Billings CE (1991) Human centered automation: a concept and guidelines. NASA Technical Memorandum 103885, Moffett Field, CA, August 1991

D'Souza DF, Wills AC (1998) Objects, components and frameworks with UML. Addison-Wesley, MA

Dalheimer K (2002) Programming with QT. Write portable GUI applications on UNIX and WIN32. O'Reilly

Frey A, Lenz A, Putzer H, Walsdorf A, Onken R (2001) In flight evaluation of CAMA – the crew assistant military aircraft. In: Proceedings of Deutscher Luft- und Raumfahrtkongress, 17–20 Sept. 2001, Hamburg

van Heesch D (2001) DOXYGEN (homepage). Internet Resource Locator, http://www.stack.nl/~dimitri/doxygen

Jennings NR, Wooldridge MJ (1998) Agent technology. Foundations, applications and markets. Springer, Berlin Heidelberg New York

Laird JE, Newell A, Rosembloom PS (1987) Soar: an architecture for general intelligence. Artif Intell 33(1):1–64

Musser DR, Derge GJ, Saini A (1996) The STL tutorial and reference guide. Addison-Wesley, MA

Newell A (1990) Unified theories of cognition. Harvard University Press, Cambridge, MA

Puder A, Römer K (1999) MiCO – Mico is Corba. Academic Press / Morgan Kaufmann; www.mico.org

Putzer H, Onken R (2001) COSA – a generic approach towards a cognitive system architecture. In: Proceedings of 8th European conference on cognitive science approaches to process control CSAPC '01, 24–26 Sept. 2001, Universität der Bundeswehr, Munich

Quantrani T (1999) Visual modeling with Rational Rose 2000 and UML. Addison-Wesley, MA

Rasmussen J (1983) Skills, rules and knowledge; signals, signs, ad symbols, and other distinctions in human performance models. IEEE Trans Syst Man Cybern 13:257–266

Sarter NB, Woods DD (1995) Strong, silent, and "out-of-the-loop" properties of advanced (cockpit) automation and their impact on human-automation interaction. Technical Report 95-TR-01, Ohio State University Cognitive Systems Engineering Laboratory, February 1995

Schreiber G, Akkermans H, Anjewierden A, de Hoog R, Shadbolt N, Van de Velde W, Wielinga B (1999) Knowledge engineering and management. MIT Press, Cambridge, MA

Schulte A, Stütz P (1998) Evaluation of the cockpit assistant military aircraft CAMA. In: NATO system concepts and integration panel symposium. Sensor data fusion and integration of the human element, 14–17 Sept. 1998, Ottawa, Canada

Walsdorf A, Onken R (1998) Intelligent crew assistant for military transport aircraft. In: NATO RTO system concepts and integration panel symposium. Sensor data fusion and integration of the human element, 13–17 Sept. 1998, Ottawa, Canada

Walsdorf A, Putzer H, Onken R (1999) The cognitive process and its application within cockpit assistant systems. In: IEEE/IEEJ/JSAI international conference on intelligent transportation systems, 5–8 Oct. 1999, Tokyo, Japan

Wiener EL (1993) Human factors in aviation. Academic Press, San Diego, CA

Wooldridge M (1997) Agent-based software engineering. IEE Trans Softw Eng 144(1):26–37