Research report 94

THE BB-SR SYSTEM

Iain D Craig

(RR94)

Abstract

This paper briefly describes the BB-SR system. BB-SR integrates the blackboard model of problem solving with a powerful object-oriented knowledge representation system. The combined blackboard and knowledge base system provides a sophisticated and powerful environment within which to examine self-reflection in problem solving and within which to develop systems to solve complex problems.

Department of Computer Science
University of Warwick
Coventry
CV4 7AL, UK

Feb 1987

THE BB-SR SYSTEM

Iain D. Craig

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL, UK

ABSTRACT

This paper briefly describes the BB-SR system. BB-SR integrates the blackboard model of problem solving with a powerful object-oriented knowledge represenation system. The combined blackboard and knowledge base system provides a sophisticated and powerful environment within which to examine self-reflection in problem solving and within which to develop systems to solve complex problems.

February 16, 1987

THE BB-SR SYSTEM

Iain D. Craig

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL, UK

1. INTRODUCTION

This paper describes the BB-SR system, a powerful blackboard system building environment currently under development. BB-SR integrates the blackboard model of problem solving with a knowledge base. The knowledge base is expressed in a flexible and extensible knowledge representation language called SRL.

The paper is organised as follows. In the next section, the two system components, the NBB blackboard interpreter and the SRL representation language are descibed. In section three, the BB-SR systems is described. BB-SR combines its components, but requires extra structure in each to operate successfully.

2. SYSTEM COMPONENTS

This section is concerned with a description of the two primary components of BB-SR: the NBB blackboard interpreter and the SRL representation language.

2.1. NBB

2.1.1. The NBB Interpreter

NBB is a blackboard system interpreter, similar to BB-1 (REFS). The interpreter is designed as a performance program: it interprets blackboard system structures (KSs, KSARs, etc.) in an efficient fashion. The interpreter is designed to operate as a stand-alone problem solving system or as an integral component of BB-SR. The conversion from BB-SR to stand-alone mode is achieved via the setting of a few interface switches.

Blackboard structures are represented in NBB as data structures. Each data structure has associated with it a set of interpretation functions. For example, the KS structure has functions which perform triggering, precondition matching and action executiuon. The interpretation functions are *not* held in the data structures but form part of the NBB interpreter. Interpreter functions are not called in data-directed fashion: instead, they are assembled into a more conventional interpreter framework.

The structure of the NBB interpreter does not mean, however, that it is inflexible. As will be described below, it is desgined so that it can be reconfigured: this requires that it be constructed in a highly modular fashion and that the individual functions within each module be available as part of the module interface. The relationship between functions and modules within the interpreter is directly analogous to that between methods and classes in an object-oriented programming language. That this is no coincidence will become clear in section three.

NBB uses the blackboard control model described in [Hayes-Roth, 1985].

2.1.2. Knowledge Sources

As in all blackboard systems, reasoning in NBB is performed by Knowledge Sources (KSs). Knowledge Sources represent large chunks of problem solving knowledge and have internal structure. NBB retains the conventional tri-partite stucture of KSs: that is, they have a trigger, a precondition and an action.

Triggers in NBB respond to blackboard events. At present, blackboard events can be either the addition of a completely new entry to the blackboard (a NEW event), the addition of an attribute-value pair to an existing blackboard entry (an ADD event), or the modification of a value stored in an attribute of an entry (a MODIFY event). Events occur on only one level of the blackboard at any time (this is because NBB is a serial implementation of the architecture) and events may only be about one entry -- either an existing entry or the entry created by a NEW event.

When an event occurs, the blackboard level on which the entry resides, the type of event (NEW, ADD or MODIFY) and the name of the entry involved in the event are recorded by the interpreter for use by triggers. Each KS trigger pattern may contain predicates to test the event type and the level on which the event occurred. Trigger patterns may contain other predicates, for example, predicates which test the name of the entry causing the current event. Trigger patterns are, though, intended to be quick checks of the event which has just occurred. If a KS does not care about the attributes of the current event, it may have a trigger consisting of the constant *t* which always evaluates to *true:* the trigger, in this case, is always satisfied.

When the trigger pattern of a KS has been matched, the **Context-Vars** (context variables) of the KS are evaluated. Context variables appear either singly or in sets. A separate KSAR (Knowledge Source Activation Record) is created for each context variable or set of context variables been bound. Once the context variables have been bound and a KSAR created for each variable or set of variables, the KSARs are added to the *triggered* list in the NBB agenda. Context variables can be initialised to values produced by arbitrary LISP expressions.

Preconditions are evaluated for all those KSARs which have been created by the triggering mechanism. When a KS's precondition has evaluated to true, the KSAR created by triggering the KS is moved into the *eligible* list in the NBB agenda. All KSARs in the *eligible* list are eligible for execution. Only KSARs whose preconditions have evaluated to true appear in the eligible list. When the precondition has been evaluated to true, two additional things happen before the KSAR is actually made eligible.

First, the **obviation** conditions are checked. Each KS may optionally specify a set of obviation conditions. Obviation conditions are predicates which determine whether or not the KSAR containing the instance of a triggered KS should be considered for execution. If one or more obviation conditions evaluate to true, the KSAR is put back into the triggered list: this prevents it from being considered for execution until both its precondition and obviation conditions evaluate to true.

The second thing which happens is that the list of **KS variables** is evaluated. The KS variables slot of each KS contains a list of local variable names and initialisation expressions. The list may be **NIL,** in which case, no initialisation is performed. At KS variable initialisation time, the expressions are evaluated and bound to the local variables. Variables may be introduced into a KS by mentioning them in triggers and preconditions (this is usually done by performing a local variable assignment using the **$VSET** assignment primitive) or by giving them initialisation expressions in the KS variables slot. KS variables initialised in the variables slot are available only to the KS's action. The initialisation mechanism allows local variables to be initialised to the results of arbitrary computations.

Local variables in KSs are lexically scoped. That is, the maximum scope which they may have is the entire KS. Local variables introduced elsewhere have a scope which extends from the point at which they are first defined or used as far as the end of the KS.

Functions are provided by NBB for assigning and de-referencing local and context variables. The interpreter also provides predicates to test whether or not a given variable is bound in the current KSAR.

The preconditions, obviation conditions and local variables are evaluated on every interpreter cycle. This repeated evaluation ensures that the agenda is consistent at all times.

NBB **actions** are represented as sequences of production rules. Each production rule has a name unique within the KS, a set of conditions and a set of actions. Rule conditions are interpreted as conjunctions of clauses. Actions are interpreted as if they were LISP PROGNs. If a rule is unconditionally to be executed, its condition is specified as *t:* the same convention applies to triggers and preconditions.

Actions have the primary responsibility for changing the blackboard state. NBB provides one function, **PROPOSE** to alter the blackboard state. PROPOSE is the only function in NBB which automatically causes KS triggering. PROPOSE can take on different forms according to the type of event to be caused

(or: according to the type of blackboard alteration required).

In the simplest case, PROPOSE posts a new blackboard entry (a NEW event). PROPOSE must, here, specify the abstraction level on which the new entry is to reside and a list of attribute-value pairs which are to comprise the new entry.

In the other cases (ADD and MODIFY), PROPOSE must supply the name of the entry to be updated, the name of the attribute to be added or updated and the value to be supplied to the attribute. When the entry to be modified is the same as the one which caused the KS to trigger, the entry can be directly accessed via the variable **$TRIGGER.ENTRY** which always points to the entry which caused the KS to trigger.

Rules in KS actions always execute in sequence. When a rule condition evaluates to false, the rule is skipped and the next evaluated. It is possible to have arbitrary pieces of LISP code as rule actions. Rule actions can, for example, update external queues and displays.

2.1.3. Events

KS actions are executed using the bindings for local and context variables found in the KSAR representing the particular KS instance. When a KS action executes, it causes blackboard events. As soon as an event occurs, it causes triggering of further KSs. The events caused by any particular KS instantiation are, however, recorded for future use. An **event list** is maintained by the KS action execution module for the recording of events. Events recorded in the event list contain information about the time at which the event occurred, the level on which it occurred, the identifier of the causing KS and so on. This information is put into the causing KSAR when the actions of its KS have terminated. Event lists are intended used to generate explanations and justifications as well as to assist learning mechanisms: they also serve as a useful debugging aid during system development.

2.1.4. Entries

Entries represent solution components on the blackboard. They contain information derived from executing KS actions and may be linked to form solution islands. Entries are also the data items which cause KSs to trigger. Entries contain information which represents a partial state of the solution.

The primary methods for accessing entries are the triggering mechanism and the PROPOSE sub-actions (adding attribute-value pairs and modifying values in attributes). It is also necessary to provide facilities so that preconditions and rule conditions can access entry data. The **$GET** function is provided for this purpose. ($GET e1 Size) returns the value of the Size attribute of the entry referred to by e1: the first argument to $GET must always *evaluate* to a valid entry identifier. In addition to $GET, there are other entry-manipulating functions: their use is dangerous and users are discouraged from applying them in the system documentation*.

2.2. SRL

SRL is an object-oriented knowledge representation. Knowledge expressed ion SRL is structured as hierarchies of **units.** Units communicate with each other via a message-passing protocol. Associated with each unit is a **meta-unit.** Meta-units contain information describing units at the object-level. Each meta-unit has a back pointer to the object-level unit it describes.

Message-passing in SRL works by fetching the method associated with the message selector from the meta-unit of the recipient unit. If there is such a method, it is applied to the message arguments. If, on the other hand, there is no such method, an attempt is made to inherit it from the chain of meta-units which comprise the **type** chain of the original meta-unit. If there is no method in the type chain, an error is raised.

Each unit (either at object- or at meta-level) has a **type** slot. The type slot refers to the class to which the unit belongs: the class is, of course, represented as a unit. Type pointers in SRL correspond to **IsA, SuperClass** or **AKO** slots in other unit- or frame-based representations.

---

* They are present in the system interface to ease the writing of external modules -- user-interfaces, for example.

The power of SRL is derived from the fact that meta-units define the behaviours of units. A meta-unit may, itself, have a meta-meta-unit: the meta-meta-unit describes the behaviour of the meta-unit. It is possible to extend the basic representation by the definition of new units with new meta-classes. For example, by the definition of a new meta-unit, it is possible to convert the basic SRL property list representation into a representation containing facets as well as slots. It is, furthermore, possible to define unit systems in which each slot is associated with a defining unit: this permits semantic checking of the form described in [Greiner, 1980] to be performed.

In SRL, each unit represents a concept. Each concept has an extension and an intension: both intension and extension are maintained by SRL. They are useful, for example, in learning and in enforcing semantics.

SRL is designed to be an open and extensible system. The basic SRL interpreter is itself represented in the SRL system: the interpreter provides the default behaviours for all units. The SRL representation of the interpreter is called the **Default-Meta-Unit.** All meta-units should have this unit as the end of their type chain.

If a unit has no associated meta-unit, Default-Meta-Unit is inferred as its meta-unit by the system.

3. BB-SR

BB-SR is the name given to the composite system based on the NBB interpreter and the SRL representation system. Apart from increasing the power of the blackboard component, the composite system is intended for research into so-called *self-reflective* systems: that is, systems which reason about themselves. In a later paper, BB-SR's capacity for self-reflection will be discussed.

The integration of blackboard interpreter and knowledge base is achieved by providing additional facilities in the NBB interpreter for handling SRL objects and for sending messages automatically to the SRL knowledge base when the need arises; there are also architectural extensions to the control blackboard employed by NBB which support graceful integration with the knowledge base.

In BB-SR, domain and control knowledge is represented in declarative form in the knowledge base. This knowledge is in the form of facts, concepts and relations. The BB-SR knowledge base may also contain inference rules which operate independently of the tasks performed in the performance component of the system (on the blackboard, that is).

In addition to domain and control knowledge, the BB-SR knowledge base contains structures representing blackboard objects*. Specifically, it contains unit classes for KSs, KSARs, events and entries. It also contains descriptions of the abstraction levels required by the problem and by the control blackboard. The unit classes are instantiated whenever an object of the appropriate class is created in the NBB interpreter. Thus, whenever a KSAR is created as the result of KS triggering, it is converted into an equivalent declarative form and added to the knowledge base.

The presence of blackboard objects in the knowledge base permits BB-SR to reason about its control and problem solving behaviour in more abstract and comprehensive terms than the blackboard architecture alone permits. It also allows explanation and learning processes and KSs to operate on a richer representation of problem solving decisions than is possible in the stand-alone blackboard. Because blackboard objects are present in the knowledge base and because they are linked into a conceptual network, it is possible to reason about these structures in a deeper fashion than is possible within the purely syntactic environment provided by the NBB interpreter.

Blackboard objects are represented within the context of domain-dependent and independent knowledge. Within this context, there is a vocabulary with which to describe actions, states and events. This vocabulary can be exploited by explanation and learning processes. The presence of this vocabulary assists the blackboard control component by imposing extra, semantically-derived criteria for performing control decisions.

---

* The BB* system [Hayes-Roth, 1986] also contains a similar representation but does not go as far in its self-description as BB-SR.

The most surprising feature of BB-SR is that it contains a model of itself. The model of the declarative component is derived from the use of meta-units in SRL. The NBB interpreter is also represented as a set of units in the knowledge base: this is why the interpreter was constructed in a modular fashion. It is theoretically possible for BB-SR to reason about its own structure and state and to treat itself as a theory within which to solve problems.

4. SUMMARY

This paper has briefly described the BB-SR system, a blackboard problem solving system with an integrated knowledge base. The system contains models of itself and contains knowledge about its own structures. The system is currently under development, with the SRL and NBB sub-systems now in completed state. In the immediate future, a user-environment will be constructed. The environment will contain a filer and a set of editors. The environment will eventually contain checking facilities and facilities for maintaining applications: these facilities will be written in SRL.

REFERENCES

[Hayes-Roth, 1985] B. Hayes-Roth. A Blackboard Architecture for Control. AI Journal, Vol. 26, pp. 251-321, 1985

[Hayes-Roth, 1986] B. Hayes-Roth, A. Garvey, M.V. Johnson Jr., and M. Hewett. A Layered Environment for Reasoning about Action. Report No. KSL 86-38, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.

[Greiner, 1980] R. Greiner and D.B. Lenat. A Representation Language Language. Proc. First National Conference on Artificial Intelligence, pp. 165-169, 1980.