# Rationalising Over-Determined Intelligence

Richard Harrison, Colin O'Halloran, Alistair McEwan, Jim Woodcock

QinetiQ/FST/CR041616/1.0
**26 March 2004**

# Administration page

| **Customer Information** | |
|---|---|
| Customer reference number | C/EGC/N02501 |
| Project title | Flight Clearance of Autonomous UAVs |
| Customer Organisation | MOD |
| Customer contact | Dr C Leach, RT(RAO RD WPE and OP1) |
| Contract number | FST/EGC/077 |
| Milestone number | EGC02/26/002/06 |
| Date due | 30 April 2004 |

| **Principal author** | |
|---|---|
| R D Harrison | +44 (0) 1684 897254 |
| Systems Assurance Group, Woodward Building, QinetiQ Malvern | rdharrison2@QinetiQ.com |
| Prof C Ohalloran | +44 (0)  1684 894320 |
| Systems Assurance Group, Woodward Building, QinetiQ Malvern | cmohalloran@QinetiQ.com |
| Prof J Woodcock | 44 (0) 1227 82 4197 |
| Computing Laboratory, University of Kent, Canterbury | J.C.P.Woodcock@kent.ac.uk |
| A McEwan | 44 (0) 1227 82 7234 |
| Computing Laboratory, University of Kent, Canterbury | A.A.McEwan@kent.ac.uk |

| **Release Authority** | |
|---|---|
| Name | M Downes |
| Post | Business Group Manager, Platform Systems |
| Date of issue | 26 March 2004 |

**Record of changes**

| Issue | Date | Detail of Changes |
|---|---|---|
| 1.0 | 26 March 2004 | Initial release to customer |

# Abstract

The objective of the present phase of the CRP project “Flight clearance of autonomous UAVs” is to certify candidate Machine Intelligence algorithms using formal mathematical assessment techniques. The meaning of formal in this context is that it reduces the certification problem to small verifiable steps that can be carried out by a machine. The certification of such Machine Intelligence algorithms falls into two parts: the formal mathematical validation of the safety of the Machine Intelligence algorithm; and the formal mathematical verification of the implementation of the algorithm. This report addresses the objective of this phase of the project by describing a subset of the Soar language that is essentially certifiable and by providing a formal semantics for programs written in this subset that can be verified for healthiness properties, such as deadlock or livelock. In particular the concept of over-determined machine intelligence is taken to be over specialisation leading to rule redundancy, which this report discusses and shows can be automatically detected as a healthiness property. The formal semantics for the subset of the Soar language are provided by a prototype translator from Soar into an non-monotonic inference engine in the formal language of Communicating Sequential Processes, CSP. Further such Soar programs can be verified against critical properties identified by a system safety case for an autonomous UAV. Finally the formal representation of Soar programs written in the subset can be verifiably implemented on an FPGA via its semantic representation in CSP.

# Executive Summary

The objective of the present phase of the project is to certify candidate Machine Intelligence algorithms using formal mathematical assessment techniques. The meaning of formal in this context is that it reduces the certification problem to small verifiable steps that can be carried out by a machine.

The certification of such Machine Intelligence algorithms falls into two parts: the formal mathematical validation of the safety of the Machine Intelligence algorithm; and the formal mathematical verification of the implementation of the algorithm.

The objective of the work that this report discusses is to provide the validation of a Machine Intelligence algorithm for the purpose of certifying it as safe. A candidate Soar program is developed, as currently, then the Soar program is translated into the CSP framework that has been developed. The translator will check that the Soar program satisfies certain syntactic and semantic constraints that allow it to be analysed, else it is rejected with relevant error messages. If rejected the Soar program cannot be certified and therefore needs to be modified to satisfy the constraints imposed by the translator.

If the translator accepts the Soar program it will produce a representation of the Soar program within a formal CSP[1] model. The model with the representation of the Soar program can then be subjected to a pre-defined set of automated checks that determine healthiness and safety. If the checks are all successful then the CSP representation of the Soar program can be transformed into a form suitable for direct compilation into a Field Programmable Gate Array, FPGA. An FPGA has low power requirements (consequently requiring less cooling) and has massive parallelism (which means that the CSP representation of the Soar program will be very efficient).

In a previous report [1] from this project, a generic CSP model that accepts a set of rules representing a Soar program was produced. The model at that time was far from complete and translation of the Soar program was done manually. This report is a snapshot of the design of a prototype translator from the Soar language to an enhanced inference model in CSP. The translator implicitly defines a subset of Soar that is analysable and the subsequent restrictions on a Soar programmer are discussed. The report also discusses what it means for a Machine Intelligence to be over-determined and how it can be mechanically detected.

The report addresses the objective of this phase of the project by describing the subset of Soar that is essentially certifiable and by providing a formal semantics for programs written in this subset that can be verified for healthiness properties. Further, such Soar programs can be verified against critical properties identified by a system safety case for an autonomous UAV. This is a powerful and novel verification technique for Machine Intelligence.

Based on the results reported in this report it is recommended that:

- the formal model of Soar in CSP is extended;
- the prototype translator is extended;

---

[1] CSP, Communicating Sequential Processes is a mathematical theory and language that describes patterns of communication, or interaction.

- the analysis capabilities are validated against the Soar RoadSearch algorithm for autonomous UAVs.

# List of contents

# 1 Introduction

## 1.1 Contractual information

This report constitutes milestone EGC02/26/002/06 for the Weapons, Platforms and Effectors Corporate Research Programme “Flight clearance of autonomous UAVs”.

## 1.2 Objectives

The objective of the present phase of the project is to certify candidate Machine Intelligence algorithms using formal mathematical assessment techniques. The meaning of formal in this context is that it reduces the certification problem to small verifiable steps that can be carried out by a machine.

The certification of such Machine Intelligence algorithms falls into two parts: the formal mathematical validation of the safety of the Machine Intelligence algorithm; and the formal mathematical verification of the implementation of the algorithm.

The objective of the work that this report discusses is to provide the validation of a Machine Intelligence algorithm for the purpose of certifying it as safe. Figure 1 describes the relevant process for certifying Soar programs, first a candidate Soar program is developed, as currently, then the Soar program is translated into the CSP framework that has been developed. The translator will check that the Soar program satisfies certain syntactic and semantic constraints that allow it to be analysed, else it is rejected with relevant error messages. If rejected the Soar program cannot be certified and therefore needs to be modified to satisfy the constraints imposed by the translator.

If the translator accepts the Soar program it will produce a representation of the Soar program within a formal CSP[2] model. The model with the representation of the Soar program can then be subjected to a pre-defined set of automated checks that determine healthiness and safety. An example of a healthiness condition is that the Soar program will not reach a point where it cannot make any more progress, i.e. it is in a deadlock with its environment. Clearly in some circumstances this will be a safety issue, but not necessarily in all circumstances. Another important healthiness condition, unrelated to safety, is whether the program is more constrained than it needs to be. If the program is over constrained then it will not be able to respond as flexibly as it should, largely defeating the point of using Machine Intelligence. Specific safety properties will be to demonstrate that a Soar program can never perform certain dangerous actions. What actions are dangerous depends upon the specific safety analysis that must be conducted on the whole system with respect to a set of scenarios.

If one of the checks performed on the CSP model fails then the analysis tool, called FDR[3], reports a counterexample, i.e. under what circumstances the Soar program will violate the property being checked. The counterexample can be used to correct the Soar program and then translate it back into CSP for re-checking. If the checks are all successful then the CSP representation of the Soar program can be

---

[2] CSP, Communicating Sequential Processes is a mathematical theory and language that describes patterns of communication, or interaction.

[3] FDR stand for Failures Divergence Refinement, it performs an exhaustive state space exploration with respect to the Failures Divergence semantic model of CSP.

transformed into a form suitable for direct compilation into a Field Programmable Gate Array, FPGA. An FPGA has low power requirements (consequently requiring less cooling) and has massive parallelism (which means that the CSP representation of the Soar program will be very efficient).

In a previous report [1] from this project, a generic CSP model that accepts a set of rules representing a Soar program was produced. The model at that time was far from complete and translation of the Soar program was done manually. This report is a snapshot of the design of a prototype translator from the Soar language to an enhanced inference model in CSP. The translator implicitly defines a subset of Soar that is analysable and the subsequent restrictions on a Soar programmer are discussed. The report also discusses what it means for a Machine Intelligence to be over-determined and how it can be mechanically detected.

The report addresses the objective of this phase of the project by describing the subset of Soar that is essentially certifiable and by providing a formal semantics for programs written in this subset that can be verified for healthiness properties. Further, such Soar programs can be verified against critical properties identified by a system safety case for an autonomous UAV. This is a powerful and novel verification technique for Machine Intelligence.

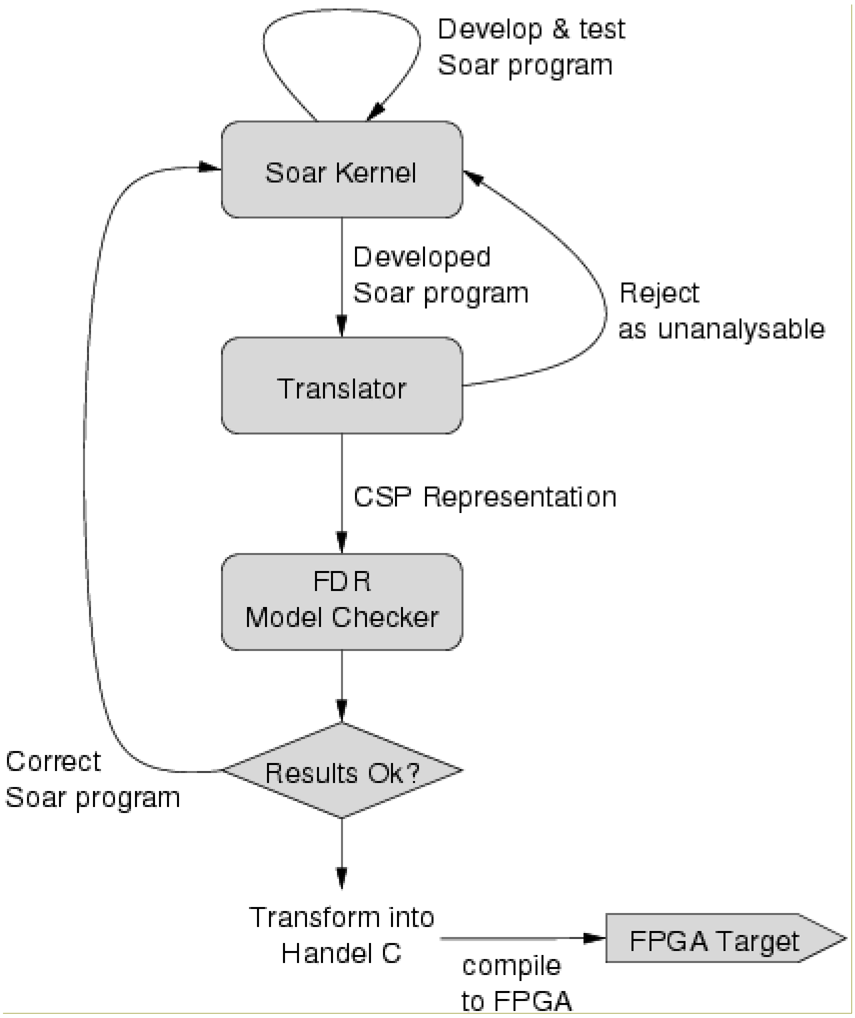

*Figure 1 the development and analysis process.*

## 1.3 Report outline

The report starts with an overview of the Soar language, the task of sorting blocks by a Soar agent is used to explain the Soar language. This is followed by section 3 which presents an enhanced CSP model that accepts a set of rules that represent a Soar program. In particular the different healthiness conditions are discussed including the issue of over-determined intelligence, (which itself is discussed in more depth in section 6). The enhancements to the model have increased the subset of the Soar language that can be accepted and analysed for safety and healthiness properties. However not all Soar programs can be analysed within the CSP model, hence section 4 explains the current limitations in terms of the Soar constructs that are disallowed for analysis purposes.

Section 5 presents the design of a prototype translator from Soar into the CSP model. This is followed, in Section 6, by an in depth discussion about what does over-determined intelligence mean in the context of a Soar program. To illustrate the issues and the approach taken a simple system is presented of the safety critical “tea maid”. In section 7 an initial evaluation of the Blocks World Soar agent described in section 2.2 is performed to illustrate the types of healthiness conditions and how a safety property can be checked. Finally, the application of the translator to an algorithm that defines flight paths for UAVs searching for moving vehicles (the RoadSearch Soar algorithm) is then discussed. The report finishes with conclusions and recommendations.

# 2 Soar

This section gives an overview of the Soar language and architecture, followed by a description of our current CSP model of Soar and the constraints we have applied to the Soar language. The section then ends with a return to the “Blocks-World” example, discussed in a previous report [1] and a simple Soar agent created to illustrate later examples

## 2.1 Soar Language Overview

Soar (State-Operator-And-Result) is a cognitive architecture that provides the foundations for building systems that exhibit general intelligent behaviour. At the first approximation, Soar is a rule-based system with (long-term) knowledge stored as "if-then" *production rules*. However, Soar also provides a flexible automatic *sub-goaling mechanism* as well as a general *symbolic learning mechanism* (known as 'chunking'). Finally, Soar provides a *belief maintenance mechanism* to automatically update beliefs when their basis no longer holds. For more information, see [3] and [4].

### 2.1.1 Soar Production Rules and Working Memory

A Soar system is entirely specified by a set of production rules encoding domain knowledge. The rules are similar to "if-then" statements, the "if" part specifying a set of conditions that must be met by the current situation and the "then" part specifying a set of actions to perform once the conditions are met.

In Soar the current situation is represented by a working memory organised as objects. Objects are described using attribute-value pairs, where objects may appear as values, and attributes may have multiple (but distinct) values. Attributes are named using strings such as `size` while values may be integers, floats, strings or identifiers (denoting objects). As such the working memory may be viewed as a directed, fully connected graph of objects rooted at a top 'state' node as in Figure 2.

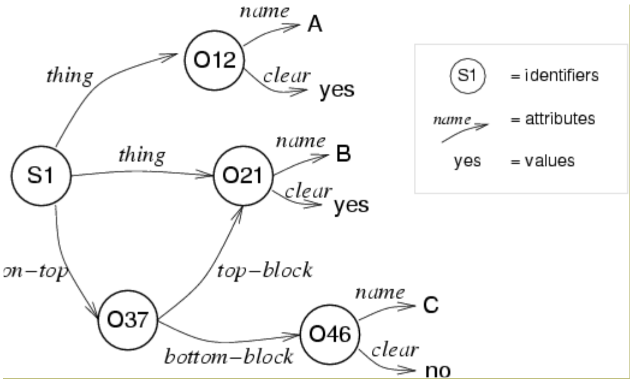

*Figure 2:*

*An example working memory composed of four objects including the top-state ‘S1’*

The language of Soar production rules is very rich, with support for partial descriptions of objects, unification of variables, predicates, negation and destructive actions (removal of working memory elements - WMEs). Although the syntax allows for generalised constructs such as attribute paths, disjunctions and conjunctions, these may all be expanded out into multiple conditions, actions or rules so that all conditions and actions are essentially 3-tuples of (identifier,attribute,value).

Variables may appear in the place of identifiers, attributes or values. As identifiers are created automatically by Soar, the identifier will always be a variable. Within a rule, variables unify so that the conditions and actions may describe the structure of working memory. Variables provide a mechanism to generalise rules and pass identifiers or constants matched in the conditions to the actions. Any free variables appearing as attributes or values in the actions are taken to be identifiers and are created automatically by Soar.

Conditions may be negated, specifying that the condition does not match working memory. Similarly, actions may specify the removal of working memory elements (WMEs). Removal of WMEs is recursive, removing all elements no longer linked to a top 'state' node (a kind of garbage collection). Lastly, conditions may be conjoined allowing for subtle "negative" conjunctions of conditions as in "not(A and B)".

### 2.1.2 The Soar Synchrony Model and Belief Maintenance

The firing of all production rules is synchronised in Soar. Rules are matched in parallel against the working memory, fired in parallel and their actions executed before the next round of rules may match and fire. This is known in Soar as an *elaboration cycle*. As attributes may have multiple values most conflicts are avoided; remove-add conflicts are simply resolved with removal overriding addition.

An important aspect of Soar is belief maintenance. This is an automatic mechanism that retracts non-deliberative beliefs (removes/adds WMEs) when their basis no longer holds. Non-deliberative beliefs are taken to be the actions of production rules that do not depend directly upon a decision. Such actions lend "support" to WMEs so long as their conditions still hold. When a WME has no support then belief maintenance will automatically remove the element.

Belief maintenance in Soar often results in unexpected behaviour. For instance, a WME may have negative support resulting from non-deliberative remove actions. When this support is removed the WME may magically reappear!

### 2.1.3 The Soar Decision Cycle

Central to Soar is the Problem Space Hypothesis that claims that all symbolic goal-orientated behaviour can be cast as a search in a problem space. Here, a problem is taken to consist of a set of states and a set of operators to move amongst these states. States correspond to the details of the current situation (internal and external) pertinent to the current goal while operators correspond to deliberate acts of cognition.

In this context, Soar introduces a *decision cycle* as an attempt to achieve rational behaviour[4] in terms of operator selection given a current state and a body of long-term knowledge. The cycle is composed of a *propose* phase in which operators are proposed, followed by a *fixed decision* and finally an *apply* phase in which the chosen operator is applied. Within the *propose* and *apply* phases, production rules

---

[4] Specifically, the principle of Rationality is defined as: "If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action."

fire synchronously as above until no more rules are eligible to fire. A fixed decision function chooses between proposed operators, using operator preferences created during the propose phase to guide the selection. When an unambiguous selection is not possible, Soar considers this to be an "impasse" signalling the need for a sub-goal that may or may not result in progress.

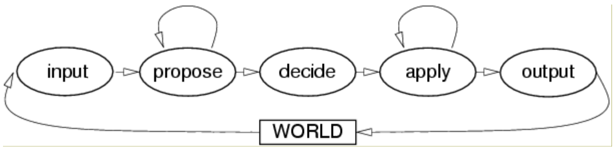

*Figure 3 the Soar decision cycle.*

The decision cycle is an acknowledged cognitive “bottleneck”, as it forces a choice to be made between operators that may actually proceed in parallel. However, the cycle is a safe, general approach compared to the opposite extreme in which pure parallelism depends upon specific executive strategies to achieve serial behaviour. An interesting discussion on cognitive bottlenecks applicable to Soar may be found in [5], providing a functional analysis and possible improvements to the general architecture.

### 2.1.4 Soar Sub-Goals and Impasses

As described above, sub-goals in Soar arise quite naturally from an inability to select unambiguously from the set of proposed operators. Whenever Soar is unable to make such a decision, the system halts and a new working memory state is created. This new state is actually a sub-state of the current state and will hold the work of the sub-goal until its completion. The initial state of the sub-goal contains a complete description of the immediate cause of the impasse, such as operators that could not be decided among. Importantly, sub-states always contain a link back to the super-state using the attribute `superstate`.

The sub-state behaves just as a normal state, proposing and applying operators. In solving the sub-goal additional impasses may be encountered, each leading to a new sub-goal and sub-state. Thus, it is possible for Soar to have a *stack of sub-goals* (and sub-states). Each sub-state has a single super-state and each state may have at most one sub-state.

To resolve an impasse, the sub-goal must generate *results* that allow problem solving at higher levels to proceed. In essence, the sub-goal must modify its super-state leading to the selection of a new operator. When this happens, Soar automatically removes the sub-state (and all its sub-structure) before applying the new operator and continuing as before. Note that any results (WMEs) linked to the superstate will persist after the sub-state has been removed. For example, the results of the sub-goal might be a plan for achieving the original goal, which another sub-goal may then need to access in order to apply it and so on.

### 2.1.5 Soar Input and Output

Interaction with the external environment is achieved in Soar through predefined, dedicated structures in the working memory. All inputs to a Soar system are affected *before* the propose phase and all outputs are affected *after* the apply phase. It is then up to the system designer to define the interface to Soar in terms of expected input and output structures and their interpretation. For example, it is common to "run a Soar system till output" so that multiple internal decisions may occur up to some bound before an output is required.

In the context of multi-agent systems the model of action is "action as command", in which a Soar agent produces actuator commands in response to sensor perceptions.

## 2.2 Return to Blocks-World

For illustration purposes we have resurrected a simple example of a planning problem space known as Blocks World (previously examined in [1]). In Blocks World the objective is to build a tower from a set of blocks given the usual laws of physics. In our version, three blocks named A, B and C initially rest on a table and the goal is to build a tower with A on top, B in the middle, and C on the bottom. The Soar agent must then come up with a sequence of legal block movements that achieve the task. For illustration, the effective state-space as defined by block movements is given by Figure 4.

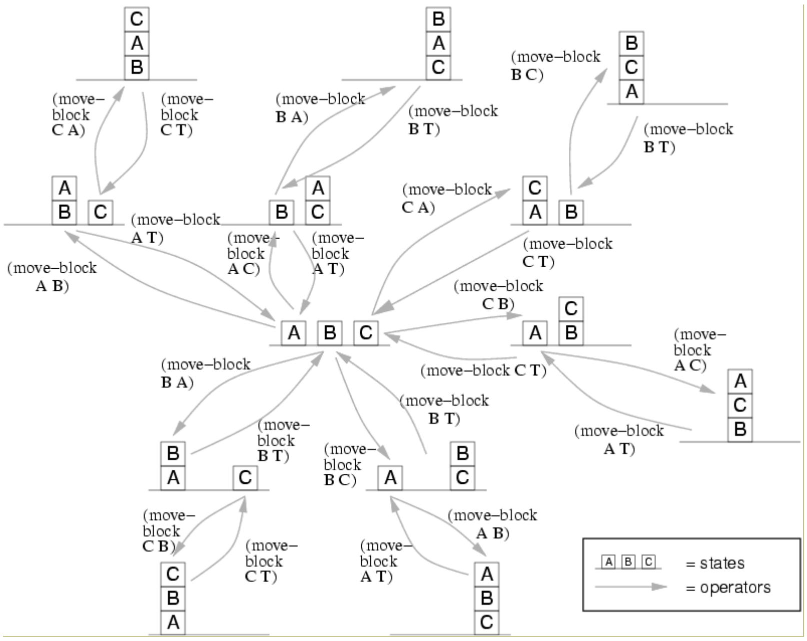

*Figure 4 The Blocks World state space of block movements*

We have written a very simple Soar agent that performs this task. To keep things simple the agent is internalised and there are no inputs or outputs to handle. In this sense, the agent represents a *planner who manipulates an internal representation of the blocks to formulate a plan of block movements*. Currently, the agent performs little more than a random walk through the state space and only contains 16 Soar productions. However, it does maintain a representation of the problem using Soar productions that may be injected with faults. In addition, we have added two belief-maintained rules that tell the agent when a block is "in place" (and should not be moved if possible).

We have also performed a manual translation of this Soar agent to our CSP model's target environment. This translation (a CSP script) serves both as a "proof-of-concept" and as an oracle against which we may validate our prototype translator output. This CSP script may be found in Appendix B.

# 3 A Refined CSP Model of Soar

We have chosen to model Soar as a generic *inference engine*, adding specific features of Soar to our model as deemed appropriate. The model of a Soar agent is defined indirectly in CSP as data to a generic CSP model. As in Soar, the data takes the form of simple firing rules produced by an automatic translation from the original Soar production rules. These firing rules are essentially the same as the original production rules although not as expressive on an individual basis.

In modelling Soar, we have chosen a largely static approach and as a consequence most dynamic behaviour must be evaluated statically or at least limited. In particular, the CSP firing rules must be grounded (all variables expanded out to constants) and we assume learning has been turned off. To allow for such a static evaluation and to control the remaining dynamic behaviour we have chosen to place constraints on the Soar language. These constraints are the subject of section 4 and effectively define a subset of the Soar language that is currently analysable.

## 3.1 Healthiness Properties

As an initial starting point to our analysis of Soar agents, we have decided to look at healthiness properties of a Soar agent: behaviour that would be considered "bad" in any context. This is akin to an exception analysis of a conventional program. This analysis is at a low level and we have had to be careful not to include behaviours that are due just to the current implementation of Soar. At the system level, we are more interested in behaviour such as unintended impasses, which might correspond to *holes* in the agent's problem space - a definite problem.

The following is a list of properties we currently check for in our analysis, and motivates the design of our CSP model of Soar. These properties are initial observations and could be refined or extended as needed. In describing the properties below we make use of the terms “I-support” and “O-support” with reference to Soar production rules and WMEs. These are roughly defined as follows:

- A Soar production is an “O-support” rule if and only if it tests for an operator (i.e. a decision) in its conditions, else it is an “I-support” rule.
- A Soar WME has “O-support” if it was created/removed by an “O-support” rule, and “I-support” if it was created/removed by an “I-support” rule.

While the first definition can be implemented by a clear-cut syntactic check, the second definition is more problematic as a given WME may be created/removed by both “I-support” and “O-support” rules. “O-support” is associated with deliberative actions and persistence, while “I-support” is associated with non-deliberative actions and the belief maintenance mechanism.

### 3.1.1 Livelock (with respect to decisions and outputs)

This is when a Soar system performs an infinite sequence of elaborations during the *propose* or *apply* phases of the decision cycle. Here, production rules fire continuously and the system never reaches acquiescence (from which a decision is made). This may be due to:

1. An enabling-disabling loop of production rules.

2. An infinite elaboration of the working memory state.

Enabling-disabling loops are commonly associated with belief maintained “I-support” rules but may equally occur with just deliberative "O-support" rules. We may outlaw much of this looping behaviour by putting constraints on the Soar language. In particular, we may make the belief maintained "I-support" rules more or less monotonic, i.e. the rules only ever *add* knowledge and do not cause other belief maintained rules to retract. When looping behaviour does occur we may detect it as *system livelock*, in which the system is capable of performing an infinite sequence of rule firings without making a decision.

Infinite elaboration of the state is commonly associated with run-away computations such as counting and implies infinite state - clearly a problem for our static approach. In tackling this problem, we may try to place constraints on the Soar language. When constraints on the language are not feasible we may take a different approach and place bounds on the dynamic behaviour within the model (and signal an error when the bound is exceeded).

### 3.1.2 Unresolvable Impasses

When a Soar agent is unable to make an unambiguous decision over which operator to select next, Soar considers this to be an *impasse*. An impasse indicates a lack of *current* knowledge that the Soar system attempts to resolve by automatically formulating a sub-goal with the express purpose of gaining the required knowledge to proceed, e.g. by enabling new production rules from its *long-term* knowledge.

In most cases the impasse is intended and results in a sub-goal that makes local progress, eventually enabling global progress. Sometimes the impasse simply represents *waiting* for the situation to change; presumably affected by another agent. We therefore have to evaluate all such impasses in a broader context.

When Soar is unable to resolve an impasse, this is usually apparent by an unending spawning of repeated sub-goals, normally *state no change* impasses in which a new operator is not selected. It is hypothesised here, that all unresolvable impasses eventually culminate in repeated *state no change* impasses. For example, an *operator no change* impasse in which an operator cannot be *applied* will invariably result in an *state no change* impasse next time round the decision cycle.

The current CSP model of Soar does not address the issue of sub-goals and we reserve this for future work. Most likely, sub-goals and their associated problem spaces will provide an intuition for decomposition of our analysis. Currently, all impasses - unresolvable or otherwise - are detected and raised as failures.

### 3.1.3 Cognitive dissonance (or illegal actions)

What happens when an agent issues an actuator command that cannot be actuated? In Soar, this question is left up to the environment to answer. For example in the TankSoar [6] game the environment raises a warning and ignores the command. In a real system, an illegal action may result in more than a friendly warning. It is therefore important in our analysis not to ignore illegal actions. In Soar, an illegal action may be due to a simple bug in the agent or due to a disparity in the cognitive agent’s representation of the real world, i.e. dissonance.

As an aside, the flight control technologies being developed elsewhere in this programme, are being designed to provide agents with information on the UAV’s current capability. This may be either via warnings, raised in response to illegal actions, or via a continuously updated parameter set which informs the agent of the vehicles current limitations.

### 3.1.4 Redundant Productions.

The purpose of expressing Soar's rules as rules within a CSP inference model is to essentially perform a reachability analysis to determine whether "bad" states can be reached through executing Soar rules. A "bad" state would typically be an unsafe situation. Although the reachability analysis offers assurances that the rule set is sufficient to achieve safety, it offers no guarantee that each of the rules are necessary. If a rule within the CSP model is unnecessary then it could be removed without affecting critical functional properties. A complication arises because a single Soar rule can give rise to multiple rules in the CSP model. However checking each of these in turn provides evidence of whether the Soar rule is redundant and hence whether the machine intelligence embodied in the Soar rules is over-determined. These issues are discussed in more detail in section 6.

## 3.2 The modelling approach

Our CSP model of Soar starts with the concept of a "datamap", a static vision of all possible WMEs that may exist for a given problem space. The datamap is a directed graph with WMEs as nodes and matching Soar identifiers defining the links between nodes. There is always one root node, the top 'state' node, and the graph must be fully connected. Usually the graph is simply a tree, but it is not uncommon to find many-to-one links or even cycles. In the past, Soar datamaps have been used successfully for documentation and even validation (see [7],[8]). Importantly, there are tools available to generate such datamaps semi-automatically. Figure 5 shows a simplified datamap of the Blocks World Soar agent with WMEs annotated with their attribute and possible values.

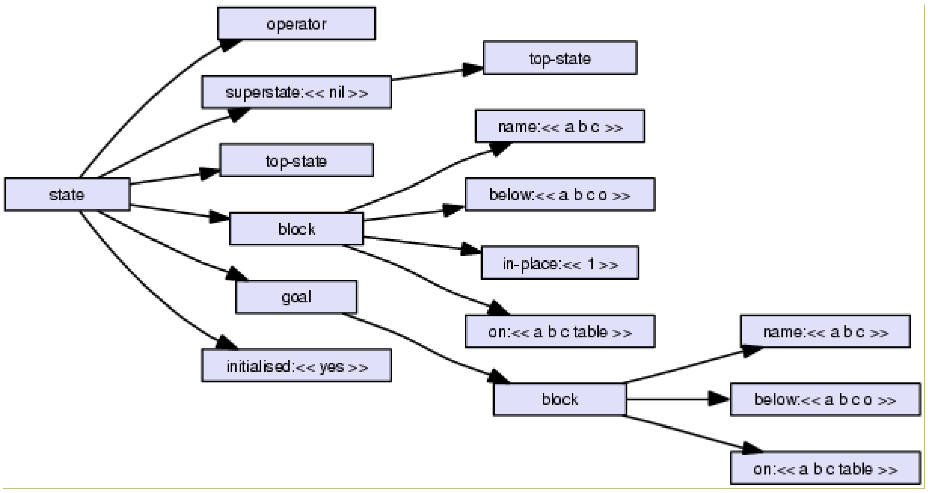

*Figure 5 A simplified datamap of the Blocks World Soar agent*

In this context, we may construct a model of Soar at the level of *atomic facts* corresponding to WMEs and the actual *instances* of production rules that operate over these facts. We may choose to incorporate parts of the Soar architecture into the model or simply encode them as extra rules. Following a translation to this low-level framework, we may then use a model checker to exhaustively search the space of possible rule firings for some of the properties described above. Any failures found should then always be interpreted within the original Soar agent as the model may introduce artificial failures through abstraction, i.e. *false negatives*.

The remainder of this section is divided into an overview of our approach and a report on the current status of the model. Issues pertaining to the automatic translation of Soar productions to this framework we reserve for section 5.

### 3.2.1 Our Approach

The primary focus of our CSP model of Soar is the atomic facts or rather WMEs that define the state space of a Soar agent. We will assume for the time being that a given state is defined by the truth status of each of the datamap WME nodes. In particular, there is no way to break links between nodes, as Soar requires working memory (as defined in our datamap by active/true nodes) to be fully connected at all times.

Even for the Blocks-World example, the apparent state space is huge. However, the actual state space is almost always much smaller - the graph must always be fully connected and leaf nodes usually only have one or two possible values at a time.

If the state space of a Soar agent is defined by the truth status of each datamap WME, then it is the Soar production rule instances that allow the agent to transition from one state to another. Here, an "instance" is a grounding of variables to constants. For example, in the Blocks World Soar agent each production rule proposing to move a block corresponds to about six possible instances – representing potential, perhaps impossible scenarios.

While variables denoting Soar constants may be grounded using the datamap, variables that denote Soar identifiers cannot, as Soar identifiers are only generated at run-time. Instead, we associate a Soar identifier with a single WME node in the datamap (that in some sense *declares* the identifier variable). In most cases the WME node is uniquely determined by the context. In other cases the variable may correspond to more than one WME node. For example, in Blocks World the top 'state' attribute `block` may refer to up to three blocks (not shown in the simplified datamap of Figure 5). This is common in Soar – such attributes are known as *multi-attributes* when the same attribute name may reference multiple values.

The first enhancement we make to our generic inference engine is to add the ability to "forget" previously learnt facts (allow it to be non-monotonic). This is an essential component of the model, and enables us to cope with destructive actions within production rules, i.e. removal of WMEs. Without the ability to forget it would be very hard to represent change within a Soar agent and harder still to translate Soar productions to our CSP model. The mechanism we use to model destructive actions is to introduce special rules to represent the action of forgetting facts.

To model belief maintenance within Soar it is important to understand which production rules it applies to and how the mechanism works. First of all, to repeat part of section 2.1, belief maintenance applies precisely to productions that do not depend directly upon a decision (a syntactic check). Such rules are known as operator application or "O-support" rules and are understood to have persistent actions. All other rules have belief maintenance and only "support" their actions as long as their conditions still match. For example, when a block that is "in place" is moved, the belief loses support and Soar automatically removes the WME representing the belief.

As a mechanism, belief maintenance is usually described in stative terms of "support" given by production rules for WMEs. However, it may also be described as a set of implicit rules that accompany the normal production rules. In simple terms, whenever we forget a fact that a belief maintained rule depends on we also have the opportunity to forget its conclusions. Of course, a given fact may be supported by more than one rule and we have to face the issue of "collective" support.

In our modelling, “support” is taken to be the ability to *re-learn* a fact. If we assume such implicit belief maintenance rules fire only once, then all WMEs that have collective support will eventually be *re-learnt*. Of course, this assumes the system is always capable of reaching a stable state. When it is not we may detect this using our model checker - and hopefully attribute it to a failure of one of the healthiness properties of section 3.1 above.

For illustration, the following is an example belief maintenance rule taken from our translated Blocks World Soar agent (see Appendix B). The rule implements the forgetting of the “in place” property of a single block and describes its potential effects – in this case the forgetting of the “in place” properties of other blocks. This rule is *special* and has the effect of forgetting its antecedents, hence the rule will only fire once.

```
({Disable.wme.st.block_3.inplace.yes,
  wme.st.block_3.inplace.yes},
 {Disable.wme.st.block_2.inplace.yes,
  Disable.wme.st.block_1.inplace.yes})
```

The next enhancement to our model is to give operators first class status by modelling them as defined events as in `operator.Moveblock.a.table`. This enables us to talk about operator proposals and decisions as actual events and eases the modelling. We may then model Soar's fixed decision function as the vital link between operator proposal and *decide* events. The operators and their parameters (name, subject etc) may be extracted semi-automatically from the datamap. In a similar manner we may also give the standard memory-mapped input and output structures special status as defined events.

There is a high degree of synchronisation in the Soar implementation. While this is an essential feature for the Soar programmer interested in achieving rational behaviour, it does impose a cognitive bottleneck and in fact complicates our modelling and analysis of Soar. In our model of Soar, we have chosen to abstract away from most of this synchronisation and are able to explore all possible execution sequences. This is an important step and may facilitate a massively parallel implementation of a Soar agent on a device such as Field Programmable Gate Array.

### 3.2.2 Current Status of the Model

The current CSP model of Soar is capable of detecting most of the healthiness properties described above. However, the model is still immature and does not support certain key features of Soar including sub-goaling, negated conditions, operator preferences and many-to-one links (in which a WME may have more than one parent in the datamap). In addition, the current treatment of belief maintenance introduces an explosion in state space. The remainder of this subsection discusses these limitations.

The sub-goaling feature as described in section 2.1.4 is very complicated and we do not foresee its inclusion in the CSP model in the near future. However, subgoals will surely be invaluable in the longer term for decomposition of our analysis. For the time being, the analysis of a Soar agent containing sub-goals must reduce to an analysis of its sub-goals (or rather problem spaces) and an informal argument made to the safety of their combination. For example, we may at least check an agent always enters and exits a sub-goal on the right conditions.

Negated conditions require a slight but significant enhancement to the CSP model which we have withheld from the current model in order to consider its implications. Negated conditions are an essential feature of nearly all Soar systems and we intend to introduce them in a limited fashion (see section 4.2). For example, it is hard to initialise a Soar system through operator applications without a check to tell that the system isn't already initialised.

As well as proposing operators for selection, Soar productions may also specify *preferences* between the proposed operators, which are then used by the fixed decision function. The addition of such operator preferences to the CSP model would probably only involve a slight enhancement to the CSP decide function. Alternatively, we may ignore operator preferences and consider all proposed operators as having indifferent preference. However, this prevents us from detecting certain types of impasses that are a result of *incomplete* operator preferences.

Recall that in section 2.1 we described the removal of WMEs as recursive (or garbage collection) in that the removal of a parent WME may result in the removal of all its children, grandchildren and so on if they are no longer linked to the top 'state' node. We have not addressed this feature in general yet – instead we currently put constraints on the Soar language to enable a temporary solution. If we were to allow many-to-one links we would certainly need to address the feature properly (as such links interact with the recursive removal). An approach to handling such links would probably call for special rules just as for belief maintenance.

Finally, our treatment of belief maintenance introduces an explosion in the state space of the model even for the Blocks World Soar agent. This seems to be due to competition between the belief maintenance rules and normal inferences and is symptomatic of a larger issue – the importance of synchronisation and control within Soar and artificial intelligence in general. In particular, we have not modelled elaboration cycles in Soar that separate the firing of normal rules from belief maintenance. While we may reduce the importance of control in Soar through constraints such as those in the following section, we may suffer from issues such as “stale” information without an alternative model of parallel control.

# 4 Soar Language Constraints for analysis

This section describes the constraints that we have imposed on the Soar language for our modelling and analysis. Some constraints are considered as *fundamental* and non-negotiable, while others reflect *current limitations* of the CSP model or prototype translator. We have therefore divided up this section accordingly. For each constraint, we give a synopsis, a description and our rationale.

## 4.1 Fundamental Constraints

1. We do not allow WME identifiers to be used as attributes.

This feature does not seem to be used in practice; its meaning is unclear and would complicate our CSP model unnecessarily. The Soar User Manual gives an example of its use to provide meta-information about an attribute in the form of an object.

2. We do not allow negative actions in belief-maintained "I-support" rules, i.e. reject '-' preferences.

As explained in section 2.1.2, negative (or destructive) actions and belief maintenance is unintuitive and may result in strange, unexpected behaviour. Often, such actions may just as well be performed using (deliberative) "O-support" rules. We have therefore decided to outlaw negative actions within "I-support" rules. Incidentally, this also has the effect of simplifying the model as belief maintenance becomes more predictable.

3. A given WME may only be created/removed by an "I-support" or "O-support" rule but not both.

When a given WME may receive both "I-support" and "O-support" we often get unintuitive behaviour when such support is removed. In particular, later positive "I-support" may override earlier negative "O-support".

We have decided to outlaw such behaviours with the aim of removing potential "bad" behaviour as in section 3.1. In our limited experience such clashes of support appear to be rare and unnecessary. The constraint also greatly simplifies our modelling as it allows us to partition WMEs by their behaviour. There may well be exceptions to the rule, for instance where we want to "firm up" non-deliberative beliefs to deliberative beliefs.

4. An "O-support" WME may not be the child of (augment) a "I-support" WME.

Following on from the previous constraint, recall that the removal of WMEs involves a recursive removal of all (unlinked) children. This implies potential conflicts between "I-support" and "O-support" – the removal of an "I-support" WME may in turn cause the removal of an "O-support" WME and visa versa. In the later case, the "I-support" WME that augments some "O-support" WME must actually depend on it and will be removed by the action of belief maintenance anyway, and hence is not a "conflict".

We have decided to outlaw the first case in which an "O-support" WME augments an "I-support" WME and may get removed simply through belief maintenance. While this may not directly contribute to "bad" behaviour, it again complicates our modelling and seems unnecessary.

5. We do not allow unbounded variables to be used in attribute tests, for instance in "generic" Soar productions.

We have decided to outlaw, or at least curtail, the use of unbounded variables in attribute tests. Here, an "unbounded" variable whose possible values may not be determined within the scope of the production rule. Although quite natural, unbounded variables seem disingenuous with "safety-critical" systems and would undermine our static approach to modelling Soar.

Given this constraint, an attribute must test for a constant, a disjunction of constants, or test for a variable bounded elsewhere within the production.

## 4.2 Current Limitations

1. The "datamap" (or production rules) may not contain any cyclic links.

To keep things simple we do not cater for cycles within the datamap or production rules. This does not appear to be a significant restriction for production rules. Indeed, in the RoadSearch [2] case study (discussed later in 7.2) we only found 2 productions containing cycles – one of which was a simple self-loop placed on the standard attribute `topstate`, used to point to the top-level state. We have not assessed the impact on the RoadSearch datamaps as we do not yet have *complete* datamaps for the RoadSearch case study. However, it is likely there will be an impact as the datamap must contain all *possible* links between WMEs.

2. We do not allow negated conditions in production rules, i.e. `-^on table`.

Without negated conditions, our modelling becomes much easier. However, this is a major limitation as negated conditions are very common and useful. Further, it is hard to see how they could be eliminated from a Soar system without a great deal of effort.

We therefore intend to weaken the constraint above to allow negated conditions to be used where they do not substantially complicate our modelling of Soar. We will most probably only allow "O-support" (deliberative) conditions to be negated. In particular, we wish to prevent negated conditions from causing problems for belief maintenance. From inspecting the RoadSearch case study this seems a good compromise.

For the time being, all negated conditions must be converted or eliminated. In the Blocks World Soar agent we managed to reduce the few negated conditions to negated tests as in "`^on <> table`". This matches any WME with attribute `on` and any value but `table`.

3. We do not allow many-to-one links between WMEs, i.e. a given WME must have no more than one parent.

As described in section 3.2, the current CSP model does not address recursive removal of WMEs in general. This has prevented us from handling many-to-one links within the datamap (or production rules for that matter) as they interact with the recursive removal procedure as described in section 2.1.1.

# 5 Translator Design

## 5.1 Overview

This section describes the design of a prototype translator from Soar to our CSP model of Soar. It is the translator that is responsible for the static expansion of Soar production rules to CSP firing rules as well as the definition of certain datatypes and channels. Together, the products of the translator are then used as data to our generic CSP model of Soar to form a complete CSP script representing the Soar program. Of course, if the Soar agent interacts with an external environment the analyst must also write a CSP process to model the behaviour of the environment. This is usually a simple matter of mapping Soar output events to appropriate input events.

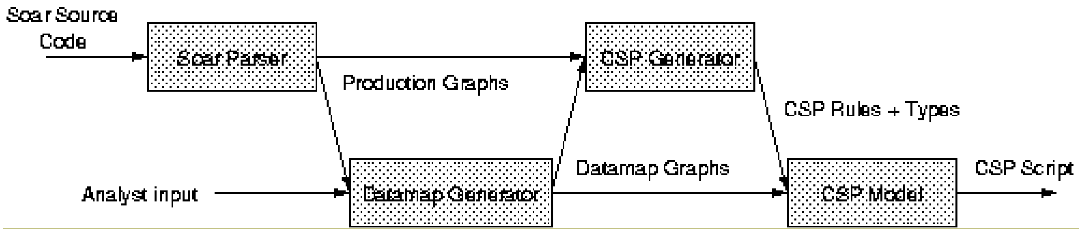

*Figure 6 The architecture of our prototype translator.*

Although referred to as a notional "translator", our prototype is actually a collection of tools as shown in Figure 6 above. The Soar Parser is a general purpose parser/simplifier we have written and covers the entire language of Soar productions (see Appendix A for the Soar grammar). The Datamap Generator is a third-party tool (VisualSoar [8]) used to automate the generation of Soar datamaps needed for the static expansion of Soar productions. The CSP Generator is the heart of the translator, enforcing our Soar language constraints but carries out only part of the translation – the actual work of expanding Soar productions is performed within CSP using set comprehension.

The following sections describe the internal representation used to represent Soar productions and datamaps, and then the actual process of generation.

## 5.2 The Internal Representation

The representation we use for Soar productions is essentially the simple "internal form" as used by the Soar kernel. In this form a Soar production consists of a set of conditions and actions, each a basic 4-tuple of (preference,identifier,attribute,value) created by expanding attribute paths, synthesising implicit variables and so on. In fact, the "internal form" of any production may be viewed in an interactive Soar session by using the `print` command.

We make heavy use of the concept of a Soar "test". This is the basic datatype used by the Soar language for the specification of identifiers, attributes and values. The syntax for Soar tests lacks punctuation and is hard to read. However, we do have an authoritative LALR(1)[5] grammar for the Soar language, part of the on-line documentation of the Soar kernel [9] and listed in Appendix A. Inspection of the Soar kernel source code confirms that its does in fact adhere to the grammar and so we have used it as the basis for our prototype Soar parser/simplifier.

For reference, a Soar "test" may be one of:

- A Soar constant (integer, float, symbolic), e.g. `23`, `1.134` or `house`
- A Soar variable, e.g. `<block>`, `<s>`
- A relational predicate, e.g. `<> table`, `> 0`, `< <var>`
- A disjunction of constants, e.g. `<< a b c >>`, `<< 1 2 3 >>`
- A conjunction of the above, e.g. `{ <name> << a b c >> }`

To illustrate the use of Soar tests and the expansion of productions to the internal representation we will use a production from our Soar Blocks World agent:

```
sp {apply*move-block*to-table
   (state <s> ^operator <op>
              ^block <moving> {<> <moving> <origin>})
   (<op> ^name move-block
        ^block <m>
        ^destination table)
   (<moving> ^name <m> ^on <o> ^below o)
   (<origin> ^name <o>         ^below <m>)
-->
   (<moving> ^on table ^on <o> -)
   (<origin> ^below o ^below <m> - )}
```

This is an operator application ("O-support") production rule used to apply the `move-block` operator in the case when moving blocks to the table. The production name `apply*move-block*to-table` is followed by a set of conditions, an arrow, and then a set of actions. Conditions and actions are delimited by parentheses and contain an identifier variable followed by a list of attribute-value pairs. Attributes are always indicated using a hat symbol `^` as they may take multiple values. Finally, the top 'state' node, here `<s>`, must always be identified in the conditions by the marker "`state`".

In the production above the conditions test for an operator `<op>` that moves some block named `<m>` to the table. They also test for two distinct blocks `<moving>` and `<origin>` (note the conjunctive test for `<origin>`), and a situation in which `<moving>` has the same name as the moving block, is clear (denoted by "`o`") and is sitting on the block `<origin>`. Given these firing conditions, the production effects the block movement in the actions by updating the state of each block.

Once parsed and simplified, the production produces a set of 4-tuple conditions and actions that may then be wired together to form a *production graph*. This is the actual internal representation we use in our CSP generator and allows us to make additional semantic checks and determine violations of our Soar language constraints. The production graph is wired up in a parenting process as follows.

---

[5] LALR(1) is a type of grammar that may be parsed Left to Right only using one Look Ahead token.

First, the parents of a node are all those nodes that contain the node's identifier as a value test; the only constraint being that action nodes may not parent condition nodes. We may then link all parents to their children to form the complete graph as in Figure 7. Each node is labelled with its attribute and value tests separated by a colon ':', conditions are given a light shade and actions a dark shade (preferences are also indicated using colour).

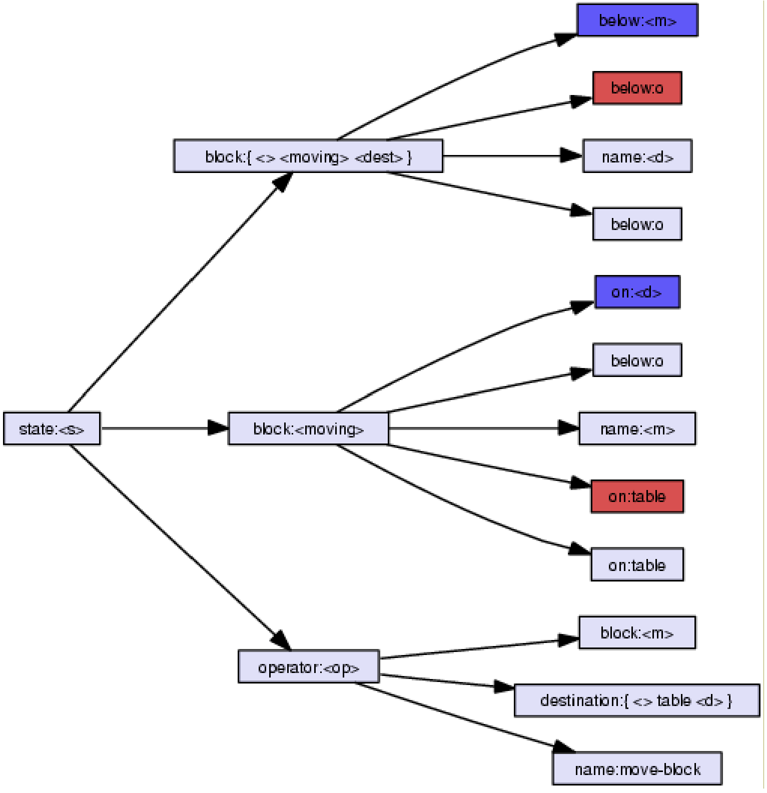

*Figure 7 The production graph for* `apply*move-block*from-table*to-block.`

Our prototype Soar Parser is written in Perl using a parser compiler known as yapp[6]. This allows us to separate out the grammar from the business of constructing conditions and actions. The semantic actions that construct these simplified conditions and actions are based on the Soar kernel parser source code but are much simplified for our purposes.

While constructing and debugging the Soar Parser we made extensive use of a graph visualisation tool known as Graphviz [11]. We intend to include this tool as part of the prototype translator, and use it to output production graphs such as Figure 7 that violate the Soar language constraints of section 4 (e.g. cycles, many-to-one links) so that the analyst may quickly determine the cause.

## 5.3 Generation

The process of generation is analogous to the work of a compiler; there is a target environment – our CSP model of Soar, to which we must map our source language of Soar productions. The CSP generator also takes on the activities of a compiler.

---

[6] Yapp – Yet Another Perl Parser compiler, a yacc-like clone for perl [10].

Beyond merely parsing the source language, we will perform a semantic analysis, code generation and perhaps optimisation. Semantic analysis will be used to check certain assumptions of the modelling, for instance containment of productions within the datamap. Code generation will produce the actual CSP, mapping Soar symbols to legal CSP symbols, constructing CSP datatypes/channels and producing the CSP firing rules. Again, the CSP Generator is written in Perl.

### 5.3.1 Target Environment

The target environment, our CSP model of Soar, has to some extent been described by section 3. However, in generating the data for our generic CSP model we may also make extensive use of the CSP functional language CSP$_{m}$. This enables us to make a trade-off between processing within our translator and processing within the CSP, i.e. the model checker FDR. Currently, though, besides the static expansion of productions through set comprehensions, we make little use of CSP in our generation. The major features of our current target environment are as follows:

- WMEs are *named* by an attribute path + value.
- All operators must be pre-declared by "name" and parameters, e.g. Moveblock.{a,b,c}.{a,b,c,table}.
- Datamap values are made accessible through the function "vals".
- Datamap multi-attribute values are made accessible through the function "multi".
- CSP firing rules are specified using a 3-tuple of (name,{conditions},{actions}).
- WME conditions and actions are specified using the channel "wme".
- Negative actions are distinguished using the channel "Disable".
- Operator proposals and applications are specified using the channels "propose" and "operator" respectively.

To illustrate the above, the following is an example of a set comprehension from our "proof-of-concept" CSP translation of the Blocks World Soar agent (see Appendix B).

```
    propose_moveblock_2 = {
        (propose_moveblock_totable,
            {
                wme.B1.name.A1,
                wme.B1.on.O1,
                wme.B1.below.o
            },
            {
                propose.Moveblock.A1.table
            }
        ) | B1 <- multi2(st.block),
            A1 <- vals(B1.name), O1 <- vals(B1.on),
            O1 != table
    }
```

Here, the expression expands to a set of CSP firing rules that together represent the Soar production `propose*move-block*to-table`. A vertical bar '|' in the set comprehension separates the template CSP firing rule from a list of variable bindings used in the expansion. A CSP firing rule is then produced for all possible variable bindings.

Expansion is over the possible values for a multi-attribute “block” and the possible values for leaf nodes in the production graph, i.e. “name” and “on”. Note that the relational test “`<> table`“ contained in the production graph is passed into the CSP set comprehension as the constraint “`O1 != table`”, and that the operator proposal actions are reduced to a single CSP “propose” event.

### 5.3.2 Semantic Analysis

The semantic analysis phase of our prototype translator has not been designed yet but is expected to involve the analysis of production graphs against the datamap graphs supplied by the analyst. In particular, we intend to enforce some of the fundamental constraints concerning potential conflicts over “I-support” and “O-support” as well as the issue of “unbound” Soar variables, something that may be hard and undesirable to do within the CSP model of Soar (we want to protect the analyst).

Given that the static expansion is performed within the CSP as above, an obvious check to perform will be to check for containment of production graphs within the datamaps. This may be a simple matter of checking that each production graph matches a datamap (which may already be done using the VisualSoar tool [8]), or may even include reasoning over production graphs to check for undeclared “multi-attributes”. Related to the latter check is the question of the “completeness” of the datamaps, i.e. whether our static expansion of Soar productions covers every possible production “instance”. This is an open question – for the time being we make the assumption that our datamaps are complete after analyst input.

Finally, we are likely to want to perform some kind of data refinement. For instance, we cannot currently handle inequalities and a semantic analysis could be used to determine range abstractions such as “`x_lt_9`”, perhaps with analyst assistance.

### 5.3.3 Code Generation

Code generation within the prototype translator is currently rather bespoke and not quite complete. However, the translator is capable of producing the set comprehensions for CSP firing rules as in section 5.3.1 above, and of mapping Soar’s symbols to legal CSP symbols. Still to complete are the construction of the CSP datatypes/channels from an analysis of the datamaps and the actual realisation of a complete CSP script.

The generation of the CSP set comprehensions is currently performed in a single pass through the production graph as follows. The algorithm will be subject to change once the current limitations of section 4.2 are resolved and is provided for illustration only. It assumes all nodes in the production graph have been previously decorated with semantic information such as “multi-attribute” status.

The algorithm consists of a depth-first traversal of the production graph, updating a symbol table and set of outputs (conditions, actions, proposals, applications, constraints). As such it is quite complicated and should really be separated into multiple passes. The key behaviour is as follows:

- Soar variables are *declared* whenever they appear as equality tests.

- Each Soar variable declaration binds a CSP variable using the ‘vals’ function, repeated declarations necessitate an implicit equality constraint.
- For Soar variables denoting identifiers, the ‘vals’ function uses the special value `na`.
- Soar tests are converted to CSP constraints over the Soar variable declared in the test.

Once a traversal of the production graph is complete we realise the collected outputs as a CSP set comprehension. In particular, the CSP constraints must be reordered to meet “declaration before use” dependencies using a topological sort. Any operator proposals or operator applications must be output as the special `propose` and `operator` events. We use the special value `na` to specify any fields that were not defined in the production.

# 6 Over-determined Intelligence

## 6.1 What is over-determined intelligence?

In this report we are taking over-determined intelligence to mean that the collection of Soar rules have given rise to a system that is more specialised than it needs to be. For example in the "blocks world" case study, if the blocks were different colours then we might have a Soar program that could only arrange white blocks but was unable to deal with red blocks. Clearly the objective would be to have a Soar program that could arrange blocks irrespective of their colour and only being able to arrange white blocks is a characteristic of an "intelligence" that is over-determined.

The problem is that we can recognise the above example of over determination, but other examples might be more complex or subtle. Indeed some examples of over determination might be subjective depending upon the context in which the machine intelligence operated. Instead an objective characterisation of over-determined intelligence is required that can be determined mechanistically. The characterisation that we choose is of rule redundancy. This will not capture all types of over determination, but it will catch over specialisation due to too many rules being added. This characterisation of over determination is similar to the concept of completeness in Inductive Logic Programming when a set of rules should cover all positive examples from a learning set.

A working definition of a redundant rule is:

**Definition 1** *A rule is said to be redundant if it cannot, does not, or need not fire, and the observable behaviour of the program is the same if the rule is removed.*

In this section, we explore this definition, identifying several types of redundancy, and showing how their presence can be detected.

## 6.2 Classes of redundant rules

Before discussing redundant rules further a few definitions are needed.

**Definition 2** *Conjunctive normal form for rule sets*

*A rule set is in conjunctive normal form if it is a conjunction (sequence of 'and's) consisting of one or more rules, each of which is a disjunction ('or's) of one or more hypotheses leading to the same conclusion.*

We will also adopt the syntax for a rule in definition 3. This definition expresses a rule as an implication: if the hypothesis h is true, then the conclusion c can be inferred. In this definition, h may be a conjunction of several facts, but c is a single fact. This syntax shows that rules, and sets of rules, can be described in terms of logical propositions.

**Definition 3** *Syntax of a rule: h=>c*

### 6.2.1 Non-firing rules

The first major class of redundancy is rules that do not fire. A rule may not fire for one of several reasons:

- the hypothesis of the rule is encapsulated by another rule;

- the hypothesis is trivially false;
- the system cannot evolve into a state in which the hypothesis can be asserted.

If ‘A’ is a hypothesis for the rule “A => C” and ‘B’ is the hypothesis for the rule “B => C”, such that ‘A’ is a stronger hypothesis than ‘B’ (i.e. “A and B” is logically equivalent to ‘A’), then the rule “B => C” is absorbed by the rule “A=> C”.

The second case of a non-firing rule is one where the hypothesis of the rule is trivially false. Such a situation can arise, for instance, when the hypothesis of a rule is two conflicting facts such as “block A is on top of block B” and “block B is on top of block A”.

An important result with regards to these two areas of redundancy is that they can be removed automatically using a simple decision procedure.

The third possibility for a non-firing rule is that the rule is unreachable. This situation can arise if, for instance, the system never evolves into a state where the hypothesis is known to be true, even though the hypothesis itself is not trivially false. An unreachable hypothesis is a consequence of the behaviour of the system, rather than a property of the rule itself, a property that is not readily detectable using conjunctive normal form. Instead, FDR can be used to assert that a state exists in the system where the rule can fire.

### 6.2.2 Unnecessary rules

The second major class of rules that can be shown to be redundant are *unnecessary rules*. A rule is unnecessary if it establishes an irrelevant fact, one that neither appears in the hypothesis of another rule, nor is observable in the external world.

**Example** *Two unnecessry rules*

```
r5 =  h => c
r6 =  c => h
```

In the example above we have a pair of potentially unnecessary rules. The first rule, concludes ***c*** when ***h*** is true; while the second concludes ***h*** when ***c*** is true. If no other rule in the set requires ***c*** in a hypothesis, and ***c*** is not externally visible, then these two rules do not conclude any useful information about the state of the system, and can be removed.

### 6.2.3 Abstract modes

The final area of redundancy discussed in this paper concerns abstract modes within a set of rules. Unlike the other areas of redundancy discussed in this paper, this is not readily detectable using tools, but relies on well chosen modelling decisions. However, when abstract modes are built into the rule set, the correctness of the abstractions employed can be verified using FDR. By identifying suitable abstract modes, it is often possible to introduce several new rules to replace a large set of existing rules.

***Definition 4*** *Abstract modes*

*An abstract mode exists in a rule set when one conclusion can be inferred from a set of disjoint hypotheses that are continuous over a range of senses, and no other rules depend on those hypotheses.*

The following example contains a simple definition of a fuel tank in an aircraft. The fuel tank emits sensor measurements indicating its fuel level, upon which two control systems rely. One is a flight control system, relying on detailed fuel information; the other is a fuel control system, concerned with making sure that fuel can only be drawn from, or put in, to the tank when appropriate. These relationships can be expressed in CSP as:

```
channel level : 0..20

Fuel_Control   [|   {|level|}   |]   Fuel_Tank   [|   {|level|}   |]
Flight_Control
```

Senses emitted by the tank indicate the level in the tank, an integer between 0 and 20 (where 0 corresponds to an empty tank and 20 a full tank). It is the responsibility of the fuel control system to ensure that fuel cannot be drawn from an empty tank, and cannot be filled into a full tank. This requirement is described by the rules below. The full enumeration of this rule set amounts to 40 rules.

```
level.0 => fill
level.1 => fill
level.1 => draw
...
level.19 => fill
level.19 => draw

level.20 => draw
```

The structure of these rules can be abstracted: the conclusion is the same across the range 1-19-which can be described by the abstract mode, *normal*. A new rule set is given below.

Sensor information is used to establish the mode of operation, and actuations are defined in these terms. When a sensor signal is received indicating the system is no longer in a given mode, the mode is forgotten. The abstract rule set consists of 12 rules.

```
level.0 => not normal
level.0 and not normal and level.1 => not level.1
level.1 and not normal => normal
level.1 and normal and level.0 => not level.0
level.19 and not normal => normal
level.19 and normal and level.20 => not level.19
level.20 => not normal
level.20 and not normal and level.19 => not level.19
normal and not level.0 and not level.20 => fill
normal and not level.0 and not level.20 => draw
level.0 and not normal => fill
level.20 and not normal => draw
```

## 6.3 A simple example

In this sub-section, we present a simple example of a control system, and an associated rule set. We then apply the techniques discussed in section 6.1 to expose, and remove, areas of redundancy. The example is a simple kettle which can be used to boil water, to pour water, or, occasionally needs refilled. Unconstrained use of the kettle leads to a hazardous situation. To solve this problem, a control system, in the disguise of a maid, is deployed.

The water level in the kettle can range between the values 0 and 4, where 0 is the minimum and 4 is the maximum; the minimum safe level is 1, and maximum safe level is 3 before burnout or spillage become a threat. The kettle indicates that it is ready for operation, and confirms an initial (safe) water level of 2.

### 6.3.1 The kettle

```
max = 4
min = 0

safe_min = 1
safe_max = 3

start_level = 2

inc(x)=   if(x==max) then x else x+1
dec(x)=   if(x==min) then x else x-1

Kettle =
kettle_ready -> indicator.start_level -> Kettle'(start_level)
```

The CSP above defines a kettle in terms of the behaviour relevant to some observer. It starts with some event that indicates that the kettle is ready, perhaps some initialisation event. In this simple example the kettle magically always starts with water at a certain level, indicated by the constant start_level which, in this case, is defined to be 2. The kettle then enters into normal operating mode that is defined by the process Kettle' defined below.

In normal operating mode a user of the kettle may choose to pour water from the kettle. After doing so, the kettle emits a sensor signal to confirm the water level has reduced, and then recurses. Alternatively, a user may choose to boil water. This act results in some steam being produced-which also causes a drop in the water level. The third option is to fill the kettle with some more water, resulting in the water level increasing. The remaining options describe the two hazards: should the water rise above the safe level, then the kettle will overflow and water will spill out. Alternatively, should it drop below the safe level, the kettle will burnout. After each hazard, the kettle deadlocks.

```
Kettle'(level)=
    pour -> indicator.dec(level) -> Kettle'(dec(level))
    []
    boil_water -> indicator.dec(level) -> Kettle'(dec(level))
    []
    fill -> indicator.inc(level) -> Kettle'(inc(level))
    []
    level > safe_max & spill -> STOP
    []
    level < safe_min & burnout -> STOP
```

The safety criterion is to ensure that the kettle never enters a hazardous state, i.e. will never overflow or burnout.

```
assert STOP [T= Kettle \ (All_Events \ Disaster_Events)
```

Using FDR, an analysis provides a counter example, a trace that leads to the kettle burning out. Unconstrained use of the kettle is therefore not safe. The solution is a control system designed to allow a user to pour the kettle when it is in a safe state, and fill and boil the kettle when it is appropriate to do so, such as below[7].

```
  Rule1 = indicator.2 => pour
  Rule2 = indicator.2 => boil_water
  Rule3 = indicator.2 => fill
  Rule4 = indicator.3 => pour
  Rule5 = indicator.3 => boil_water
  Rule6 = indicator.1 => fill
  Rule7 = indicator.3 and boil_water => pour
  Rule8 = indicator.2 and indicator.3 => pour
  Rule9 = indicator.2 and not indicator.2 => fill
```

The implementation of the control system is the parallel composition of processes firing individual rules. The composition of the control system and the kettle is given below. As the control system is responsible for filling and boiling the kettle, these events are hidden from the external view of the system. The only external observation is the act of pouring the kettle: the functionality required by a user of the kettle.

```
Controlled_Kettle =
(Maid[|{|fill,boil,pour,indicator|}|]Kettle)\{|fill,boil,indicator|}
```

The process Controlled_Kettle is the parallel composition of the processes Maid and Kettle such that they only progress individually if they are both ready to perform the events “fill”, “boil”, “pour” and “indicator”. Any other event, such as Kettle_ready, can occur independently. The events “fill”, “boil”, “pour” and “indicator” are hidden (using the hide operator ‘\’) to prevent outside interference with the system definition.

A safety analysis on the new system reveals that now the kettle is safe: the control system enforces the condition that filling and boiling are done when appropriate. Furthermore, a second refinement check evidences that a user can pour the kettle when desired, and it does not contravene the safety condition in doing so.

---

[7] Of course the above rules are contrived to illustrate the issues of redundancy and would in practice never occur for such a small set of rules. For a distributed control system with hundreds, or even thousands of rules this could occur. However it is likely that in practice redundancy will arise from subtlties of rule expression for a particular problem. How these might manifest themselves is beyond the scope of this report.

Although safe, the implementation of the control system should also be optimal: it should not contain redundant rules. Using a decision procedure, rule9 is shown to have a false hypothesis and can be removed. Furthermore, FDR confirms rule8 has an unreachable hypothesis and can also be safely removed. In fact FDR would also show that rule9 would be unreachable because of the inconsistent hypothesis.

Rule 4 presents an interesting quandary. Reducing to the normal form suggests this is absorbed by rule 7, and should be removed. However, while doing so still observes the safety criteria, it contravenes the standard property of liveness. Liveness is the property that a system does not idle forever performing no useful function. It should be concluded, therefore, that this rule can be removed if only the safety properties are of interest, but not if the liveness property must also be observed. This example shows that when a rule is removed through conjunctive normal form and absorption, checks should be made to ensure that any liveness properties of interest are still preserved. This suggests that using the reachability analysis provided by FDR is necessary and sufficient to guarantee safety and liveness in a rule based system.

# 7 Analysis & Examples

This section covers an initial evaluation of our CSP model of Soar using the Blocks World Soar agent described in section 2.2. We also report on the progress towards the analysis of a real Soar agent, the RoadSearch case study.

## 7.1 Blocks World

Before writing the prototype translator described in section 5 we wrote a manual “proof-of-concept” translation of our Blocks-World agent to our CSP model target environment. This enabled us to develop the CSP model, design the target environment as well as the translation process. As our prototype translator is not capable of producing complete CSP scripts, we have used the manual translation in the following evaluation. Those familiar with CSP may consult Appendix B for the source before continuing.

The manual translation contains nine out of the sixteen Soar productions defining our Blocks World agent. There are three operator proposal rules and four operator application rules, and only two belief maintenance rules. Out of the seven rules excluded from the translation, one is an operator selection rule and four are standard state elaboration rules that are not used by our simple agent. The remaining two rules are monitor rules whose actions are just function calls (and would be ignored by our translator anyway).

In our translation, each Soar production expands to about six CSP firing rules on average with forty-seven rules in total. There are twenty-two belief maintenance rules, generated automatically by the target environment using a set comprehension. There are 89 facts in total, composed of 47 WMEs, 10 operator applications, 10 operator proposals, and 22 disable triggers (one for each WME that may be forgotten). Note that there are over twice the number of WMEs to disable triggers. This is because our Soar agent contains an explicit representation of the Blocks World *goal* that may be initialised (using “O-support”) to a large number of configurations but never forgotten.

For our evaluation we initially tried to model check the CSP script for the properties of section 3.1. This uncovered several issues with the CSP model including the need to prevent CSP firing rules from firing unless they are making “progress”, i.e. creating or removing WMEs. For the rules that can fire without making progress we now add an “oracle” process that prevents the rule from firing unless there is (or *has been*) progress to be made.

We found that our low-level treatment of belief maintenance introduced an explosion in the state space of the model. As belief maintenance is performed on the level of WMEs our model introduces a huge number of intermediate states that could not exist within a Soar system. In fact, the situation is so bad that our model checker FDR reaches 37 million states without finishing.

We have remedied this situation by giving belief maintenance precedence over normal rule firings and hiding the intermediate states. However, this is only a temporary solution to allow us to evaluate other parts of the CSP model. We need to pull the low-level model of belief maintenance up to at least the level of atomic rule firings. To do this we will make good use of the fundamental constraints of section 4.1 concerning “I-support” and “O-support”. The state-space of the Blocks

World CSP model with this temporary solution has between 4,000 and 20,000 states and is easily model checked.

Our initial model checking for the healthiness properties of section 3.1 has been successful. We are currently able to check for all enabling-disabling loops, but only certain types of impasse (our *decide module* is not complete yet). To check for these properties we must always perform two separate checks, first one for the enabling-disabling loops and then one for impasses. In the first check we hide all rule-firing events and check for livelock. In the second check we simply check for deadlock.

Without any faults injected, the model checker did not detect any bad behaviour as expected from running Soar. We then proceeded to inject faults into the Soar productions, comparing the results of the model checker to that of running Soar. We found it hard to inject enabling-disabling loops but were able to detect the behaviour by adding conflicting rules. We confirmed the inadequacy of our decide module, but did find an interesting impasse as follows.

When we injected a fault into the `apply*move-block*to-table`, the model checker revealed a sequence of decisions (block movements) that led to an impasse. For the fault, we removed an action asserting the fact that when you move a block to the table the block below it becomes clear. As a block must be clear to be moved it is simple to see that the Soar agent could get into a state in which it thinks no blocks are clear, hence an impasse. The model checker found the following (simplified) sequence:

```
infer.(propose_initialise,{…},{…})
decide.Initialise
infer.(apply_initialise,{…},{…})
tock
infer.(propose_moveblock_toblock,{…},{…})
decide.Moveblock.b.a
infer.(apply_moveblock_fromtable_toblock,{…},{…})
tock
infer.(propose_moveblock_toblock,{…},{…})
decide.Moveblock.c.b
infer.(apply_moveblock_fromtable_toblock,{…},{…})
tock
infer.(propose_moveblock_totable,{…},{…})
decide.Moveblock.c.table
infer.(apply_moveblock_totable,{…},{…})
tock
```

Running Soar with the faulty production rule revealed the same impasse, but we had to run the Soar agent for longer. Although this is a trivial example, it does illustrate the usefulness of our analysis.

## 7.2 RoadSearch Case Study

In a first effort to analyse an software agent operating within a UAV context, we have revisited the implementation of a Soar agent that provides a plan for searching a network of roads for a moving road vehicle using multiple UAVs [2]. This agent was developed for use in piloted simulation trials examining the management of multiple UAVs from a single airborne operator.

The RoadSearch problem is complex and open-ended problem, and the Soar agent solution adopted a relatively simple approach, appropriate to the reqiuirements of piloted simulation. In spite of this, its properties are representative of those anticipated in a fully-developed search and planning agent.

The algorithm uses map data, which describes sections of road as a series of latitude and longitude coordinates. From this, Soar rules establish which road sections are connected to which. Given the position that the moving target was last sighted and the direction in which it was headed the Soar planning rules perform the following actions:

- Establish the road section that the vehicle was on at the time it was sighted, and its direction of travel, given the initial position and heading.
- Establish the network of road sections connected to the initial position in the direction of travel, and in the opposite direction.
- Issue a request for each available UAV to search a specified road section
- Issue the sequence of latitude/longitude coordinates to be used by each UAV route following algorithm
- Replan when the initial position, direction or search distance data changes.

Underlying the algorithm are the following assumptions:

- The vehicle is located on a road section at all times
- The vehicle is in the part of the road network specified by the original direction of travel, i.e. it has not doubled-back
- There is no implicit completion criterion, the UAVs must be switched to another task once the vehicle is located
- There are sufficient UAVs to cover the number of branches in the network

Unfortunately, the current limitations of our CSP model and prototype translator have prevented us from performing any detailed analysis of the RoadSearch Soar agent. However, we can report some statistics on the impact of our Soar language constraints (including current limitations) on the agent. These are as follows.

Out of 262 Soar productions, our Soar general-purpose parser managed to correctly parse and simplify all but one of the productions. The one production that did not parse contained an operator preference `@` (reconsider) that is deprecated in the Soar language and we do not support. The remaining 261 productions failed on a number of constraints that we currently check. This is unsurprising since the algorithm was developed without analysis or certification in mind. However it is believed that the functionality of the algorithm can be expressed within the subset of Soar defined by the translator. The RoadSearch algorithm is being restructured by its developers, for separate reasons, and they are currently taking our constraints into account. The new Soar program should be able to perform more efficiently and also be analysable.

We also managed to auto-generate some documentation for the RoadSearch case study using the tool SoarDoc [7]. This enabled us to understand the structure of the Soar agent, namely its problem spaces and operators. We have started to produce datamaps for the various problem spaces but these are still incomplete and require analyst input to complete them. Figure 8 shows an initial auto-generated datamap for the deployment problem space. The datamap shows structure that could potentially be exploited to formulate smaller checks that could then be composed into a check for the overall Soar program.

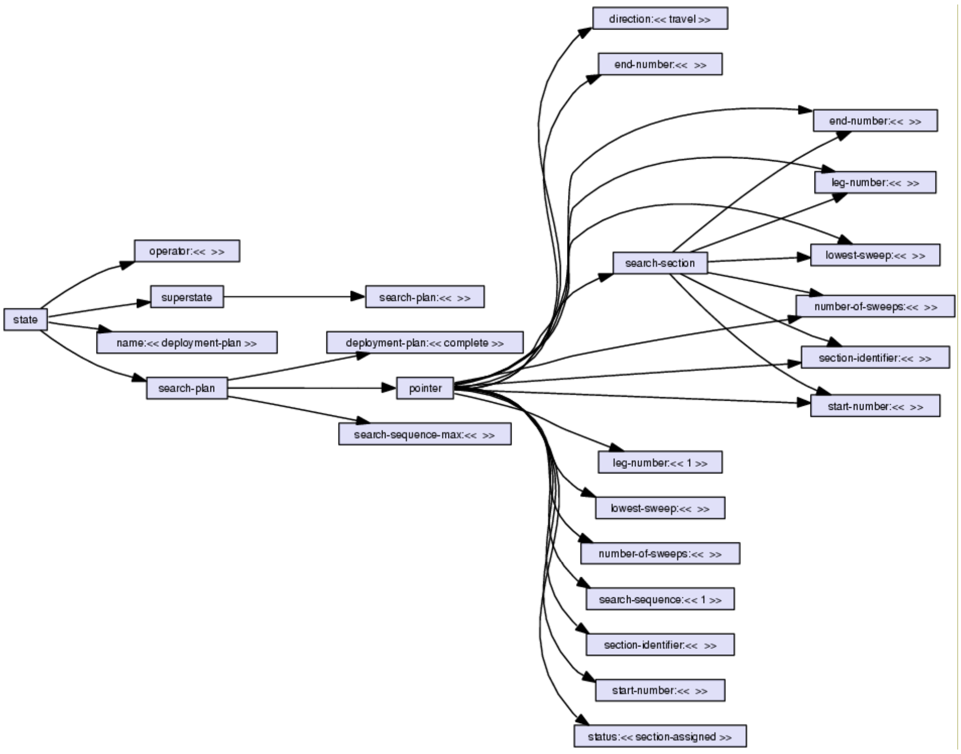

*Figure 8 An auto-generated datamap of the deployment problem space.*

# 8 Conclusions

## 8.1 Summary

In this report enhancements to the CSP inference model have been made along with a set of healthiness properties for Soar programs that can be mechanically checked. However the ability to analyse Soar causes certain restrictions to apply to the type of Soar program that can be written. This report lists the restrictions along with the rationale for the restrictions.

The design of a prototype translator is also presented that essentially provides the semantics for the subset of the Soar language identified previously. The healthiness property of non-redundancy of rules is adopted as an objective characteristic of over-determined intelligence. The reason for this is that redundant rules indicate that the Soar program has become over specialised. A precise definition of rule redundancy has been given followed by the implications for detection and elimination.

The case study of the “Blocks World” has been translated into the present CSP inference model and analysed for the healthiness properties previously discussed. The RoadSearch algorithm for autonomous vehicles was also assessed, unfortunately it failed to meet the constraints necessary for analysis. Fortunately the algorithm is to be changed, for separate reasons, from the present monolithic program into many smaller interacting algorithms that should satisfy the analysis constraints.

## 8.2 Conclusions

The provision of a translator and the formulation of healthiness checks that can be automatically performed provide a novel and powerful analysis capability for Soar programs that express Machine Intelligence. The mapping embodied by the translator enables assurance arguments to be made about the translator and the semantics of the Soar subset to be validated. The healthiness checks that can be carried out mechanically will enable the analysis of Soar programs for typical problems that frequently affect their operation. Furthermore, the Soar program can also be assessed against critical properties that have been identified as part of a system safety case. Finally the formal representation of Soar programs written in the subset can be verifiably implemented on an FPGA via its semantic representation in CSP; the potential route for achieving this is discussed in [1].

Based on this work the demonstrations of autonomy with pre-learned intelligence can, in principle, be assessed. The most significant problem is the size of the state space that potentially must be explored. However the use of multiple agents should significantly mitigate this problem as well as speed up the Soar application, such as the road searching Soar application developed by Blue Bear Ltd. The translator also attempts to mitigate the size of the state space by identifying structure within the problem space that the Soar program operates. It is conjectured that the assessment of such a Soar program can be similarly structured into smaller checks that can be composed together to give an equivalent check of the whole system.

# 9 Recommendations

Based on the results reported in this report it is recommended that:

- the formal model of Soar in CSP is extended to:
    - extend it to enable full analysis of “Blocks World”;
    - extend it to fix the current limitations of section 3.2.2 and 4.2;
    - consider issues such as the completeness of datamaps;
- extend the prototype translator to:
    - complete the semantic analysis to check constraints and modelling assumptions;
    - complete the code generation phase;
- perform an analysis of part of RoadSearch to:
    - determine whether we can construct a complete datamap;
    - investigate the input/output interface;
    - determine the ease of eliminating false-negatives;
    - validate the Soar language constraints as practical;
- investigate checking for over-determined intelligence by:
    - validating on a simple agent for the  “Blocks World” problem;
    - validating against the “RoadSearch” problem.

# 10 References

[1] C M O'Halloran. *Feasible methods to quantify autonomy outcomes*, QinetiQ/FST/CR034761.

[2] P R Smith *Marvin – Smart Algorithms For Combat UAVs* DERA/SA/EA1668/01-02/1.0, January 2002

[3] Jill Fain Lehman, John Laird, Paul Rosenbloom. *A Gentle Introduction to Soar: an Architecture for Human Cognition*. In D. Scarborough & S. Sternberg (Eds.), An invitation to Cognitive Science, Volume 4 Methods, models, and conceptual issues. New York: MIT Press, 1998

[4] John Laird, Clare Bates Congdon and Karen J. Coulter. *The Soar User's Manual* (version 8.2), University of Michigan, 1999.

[5] Richard L. Lewis. *Does the Mind Need a Bottleneck? Toward a functional analysis of bottlenecks, executive processes, and control structures*. NASA Ames Research Center, Cognitive Group, November 2001.

[6] TankSoar, http://www.eecs.umich.edu/~soar/sitemaker/projects/soardoc/samples/TankSoar/

[7] SoarDoc, http://www.eecs.umich.edu/~soar/sitemaker/projects/soardoc/soardoc.html.

[8] VisualSoar, http://www.eecs.umich.edu/~soar/sitemaker/projects/visualsoar/.

[9] Soar Kernel Documentation http://www.eecs.umich.edu/~soar/sitemaker/docs/doxygen/kernel/.

[10] Francois Desarmenien. Yapp: Yet Another Perl Parser compiler, http://search.cpan.org/~fdesar/Parse-Yapp-1.05/.

[11] AT&T Research Labs. Graphviz - Open source graph drawing software. http://www.research.att.com/sw/tools/graphviz/.

# A. Appendix A

```
# The Soar Production Grammar
#
# The main lexical tokens are as follows:
#
#   VARIABLE        = <[A-Za-z][A-Za-z0-9$%&*+-/:<=>?_]*>
#   SYM_CONSTANT    = [A-Za-z][A-Za-z0-9$%&*+-/:<=>?_]*
#   INT_CONSTANT    = [+-]?[0-9]+
#   FLOAT_CONSTANT  = [+-]?[0-9]+\.[0-9]*{[eE][+-]?[0-9]*}?
#
# Compound literals include:
#
#   '-->', '<<', '>>', '<>', '<=', '>=', '<=>'
#
# Whitespace is [ \t\n\r\f]
#
# Comments begin with '#' and extend to end of line
#
# SYM_CONSTANTs may also appear between vertical bars '|'
# as in |hello world|.

%token      VARIABLE
%token      SYM_CONSTANT
%token      INT_CONSTANT
%token      FLOAT_CONSTANT

%%

prods:      prods production
    |       production
;

# 'sp' '{' production_name lhs '-->' rhs '}'
production:  SYM_CONSTANT '{' SYM_CONSTANT lhs '-->' rhs '}'
;

# LEFT HAND SIDE (LHS)

lhs:    conds
;

# <cond>+
conds:  conds cond
    |   cond
;

cond:   minus positive_cond
;

positive_cond:  conds_for_one_id
    |           '{' conds '}'
;

conds_for_one_id:   '(' id_test attr_value_tests ')'
;

# [state|impasse] [<test>]
id_test:    #empty
    |       test
    |       test test
;

test:   conjunctive_test | simple_test
;

conjunctive_test:   '{' simple_tests '}'
;

# <simple_test>+
simple_tests:   simple_tests simple_test
```

```
    |           simple_test
;

simple_test:    disjunctive_test | relational_test
;


disjunctive_test:   '<<' constants '>>'
;

# <constant>+
constants:  constants constant
    |       constant
;

constant:   SYM_CONSTANT | INT_CONSTANT | FLOAT_CONSTANT
;

relational_test:    relation single_test
;

# [<relation>]
relation:   #empty
    |       '<>' | '<' | '>' | '<=' | '>=' | '=' | '<=>'
;

single_test:    VARIABLE | constant
;

# <attr_value_test>*
attr_value_tests:   #empty
    |               attr_value_tests attr_value_test
;

attr_value_test:    minus '^' attr_path  value_tests
;

# <attr_test> [. <attr_test>]*
attr_path:      attr_path '.' test
    |           test
;

# <value_test>*
value_tests:    #empty
    |           value_tests value_test
;

value_test:     test plus
    |           conds_for_one_id plus
;

# RIGHT HAND SIDE (RHS)

rhs:    rhs_actions
;

# <rhs_action>*
rhs_actions:    #empty
    |           rhs_actions rhs_action
;

rhs_action:     '(' VARIABLE attr_value_makes ')'
    |           function_call
;

function_call:  '(' function_name rhs_values ')'
;

# <rhs_value>*
rhs_values:     #empty
    |           rhs_values rhs_value
;

rhs_value:      constant
```

```
    |           function_call
    |           VARIABLE
;

# <attr_value_make>+
attr_value_makes:   attr_value_makes attr_value_make
    |               attr_value_make
;

attr_value_make:    '^' rhs_attr_path value_makes
;

# <rhs_value> [. <rhs_value>]*
rhs_attr_path:  rhs_attr_path '.' rhs_value
    |           rhs_value
;

# <value_make>+
value_makes:    value_makes value_make
    |           value_make
;

value_make:     rhs_value preferences
;

# "comma" signals a default preference of '+'
preferences:    comma
    |           preference_specifiers
;

comma:  #empty
    |   ','
;

# <preference_specifier>+
preference_specifiers:  preference_specifiers preference_specifier
    |                   preference_specifier
;

# NOTE:
# There is a reduce/reduce conflict here between "unary" and
# "binary" preference. The Soar User Manual specifies that "binary"
# preference should take precedence!

preference_specifier:   unary_preference comma
    |                   binary_preference rhs_value
;

# not supporting '@' (reconsider) preference.
unary_preference: '+' | '-' | '!' | '~' | '>' | '=' | '<'
;

# not supporting '&' (??) preference.
binary_preference:  '>' | '=' | '<'
;

# [-]
minus:  #empty
    |   '-'
;

# [+]
plus:  #empty
    |   '+'
;

%%
```

# B. Appendix B

```
-- CSP script produced using Soar2Csp version 0.0!

-- < header info such list of all Soar production rules, perhaps some
--   documentation from soardoc (if available) >


-------------------------------------------------------------------------------
-- TYPES and CONSTANTS
-------------------------------------------------------------------------------

-- The following attributes and values are reserved:
-- My CSP:
--  infer wme propose tock deductions ddeductions
--  bool fired i ....
-- CSPm:
--  true false not and or
--  if then else let within channel datatype nametype include assert print
--  length null head tail concat elem
--  union inter diff member card set
--  transparent chase normal extensions productions
--  ?? attribute embed module subtype ??


-- All user and reserved attributes (lowercase)
-- Notes:
--  "st" (state) is the 'null' attribute.
--  The use of '_' is limited to resolving multi-attributes.

datatype Attr =
    -- user attributes
    st | below | block | block_1 | block_2 | block_3 |
    goal | initialised | inplace | name | on |
    -- reserved attributes
    superstate | topstate


-- All user values (*distinct* from attributes!)
-- Note: "na" (not applicable) is the 'null' value.

datatype Val =
    na | a | b | c | table | o | yes | no

-- All operators - declared as bespoke imperatives with parameters
-- Notes:
--  We know exactly what proposals will be made and what operators
--  are expected from the Soar production rules & datamap!
--  For rules that simply "piggy-back" off an operator (e.g. by name),
--  we may use the 'null' value "na" instead of blindly expanding rules.

datatype Operator = Moveblock.{a,b,c}.{a,b,c,table} | Initialise

-- propose, decide and signal an Operator as above.

channel propose, decide, operator : Operator


-- a working memory element (WME) (attribute path + value)
-- Notes:
--  to keep type rectangular we represent shorter paths using 'null' attribute
--  "st":
--      <s> ^block.name a    = wme.st.block.name.a
--      <s> ^superstate nil = wme.st.st.superstate.nil

channel wme : Attr.Attr.Attr.Val

-- special "disable" action representing the forgetting a WME

channel Disable : {|wme|}
```

```
-- Some standard functions

pick({x}) = x




--------------------------------------------------------------------------------
-- THE SOAR DATAMAP
--------------------------------------------------------------------------------

-- The multi-attributes

-- whenever we know an attribute may be a multi-valued attribute
-- we use the 'multi' function to return the (suffix'd) attributes.

-- multi1 : Attr        -> Attr
-- multi2 : Attr.Attr   -> Attr.Attr

multi2(st.block)    = {st.attr_ | attr_ <- {block_1,block_2,block_3}}
multi2(goal.block)  = {goal.attr_ | attr_ <- {block_1,block_2,block_3}}

-- multi(st.st.mattr_) = {st.st.attr_  | attr_ <- multi1(mattr_)}
-- multi(st.mattr_)    = {st.attr_     | attr_ <- multi2(mattr_)}



-- define attribute values
-- vals1 : Attr             -> Set(Val)
-- vals : Attr.Attr        -> Set(Val)
-- vals : Attr.Attr.Attr   -> Set(Val)

-- some value types
OnVals = {a,b,c,table}
BelowVals = {a,b,c,o}

vals1(_)               = {}

vals2(block_1.name)     = {a}
vals2(block_1.on)       = OnVals
vals2(block_1.below)    = BelowVals
vals2(block_1.inplace)  = {yes}
vals2(block_2.name)     = {b}
vals2(block_2.on)       = OnVals
vals2(block_2.below)    = BelowVals
vals2(block_2.inplace)  = {yes}
vals2(block_3.name)     = {c}
vals2(block_3.on)       = OnVals
vals2(block_3.below)    = BelowVals
vals2(block_3.inplace)  = {yes}

vals3(goal.block_1.name)     = {a}
vals3(goal.block_1.on)       = OnVals
vals3(goal.block_1.below)    = BelowVals
vals3(goal.block_2.name)     = {b}
vals3(goal.block_2.on)       = OnVals
vals3(goal.block_2.below)    = BelowVals
vals3(goal.block_3.name)     = {c}
vals3(goal.block_3.on)       = OnVals
vals3(goal.block_3.below)    = BelowVals

vals(st.st.attr_)   = vals1(attr_)
vals(st.attr_)      = vals2(attr_)
vals(attr_)         = vals3(attr_)



--------------------------------------------------------------------------------
-- SOAR PRODUCTION RULES
--------------------------------------------------------------------------------

-- The names of all Soar production rules
-- Note: We use '_' for '*' and filter out non alphanumerics such as '-'.

datatype RuleName =
    observe_block_inplace_1 | observe_block_inplace_2 |
    propose_moveblock_toblock | propose_moveblock_totable |
        selection_dontmove_inplace |
    apply_moveblock_totable | apply_moveblock_fromtable_toblock |
        apply_moveblock_fromblock_toblock |
    propose_initialise | apply_initialise |
    test_rule |
    belief_maintenance

-- nametype Rule = (RuleName,{|wme,disable,operator|},{|wme,disable,propose|})


-- The actual rules
-- Note: We use the first two parts of the production name for the name of the CSPm
-- set, with an incremental suffix for duplicates, e.g. "observe_block_1".

-----------------------------------
-- sp {observe*block*in-place*1
--     (state <s> ^goal.block <gb>)
--     (<gb> ^name <n> ^on table)
--     (<s> ^block <b>)
--     (<b> ^name <n> ^on table)
-- -->
--     (<b> ^in-place 1)}
-----------------------------------

-- expands to three rules.

observe_block_1 = {
    (observe_block_inplace_1,
        {
            wme.GB.name.N1,
            wme.GB.on.table,
            wme.B.name.N2,
            wme.B.on.table
        },
        {
            wme.B.inplace.yes
        }
    ) | GB <- multi2(goal.block), B <- multi2(st.block),
        N1 <- vals(GB.name), N2 <- vals(B.name),
        N1 == N2
}


-----------------------------------
-- sp {observe*block*in-place*2
--     (state <s> ^goal.block <gb>)
--     (<gb> ^name <n> ^on <on>)
--     (<s> ^block <b1> <b2>)
--     (<b1> ^name <n> ^on <on>)
--     (<b2> ^name <on> ^in-place)
-- -->
--     (<b1> ^in-place 1)}
-----------------------------------

-- expands to nine rules, three of which represent (presumably!) impossible
-- goals - "a on a", "b on b", "c on c"
-- we could refine the datamap by removing these possibilities!

observe_block_2 = {
    (observe_block_inplace_2,
        {
            wme.GB.name.N1,
            wme.GB.on.O1,
            wme.B1.name.N2,
            wme.B1.on.O2,
            wme.B2.name.O3,
            wme.B2.inplace.I1
        },
        {
            wme.B1.inplace.yes
        }
    ) | GB <- multi2(goal.block), B1 <- multi2(st.block), B2 <- multi2(st.block),
        N1 <- vals(GB.name), O1 <- vals(GB.on),
        N2 <- vals(B1.name), O2 <- vals(B1.on),
        O3 <- vals(B2.name), I1 <- vals(B2.inplace),
        N1 == N2, O1 == O2, O1 == O3
}


---------------------------------------------------
-- sp {elaborate*state*name
--     (state <s> ^superstate.operator.name <name>)
-- -->
--     (<s> ^name <name>)
-- }
---------------------------------------------------

-----------------------------
-- elaborate_1 = {
--     (elaborate_state_name,
--         {
--             superstate_???
--         },
--         {
--             ????
--
--         }
--     ) | ????
-- }
-----------------------------



-------------------------------------------------------------------------------
-- MOVE-BLOCK.SOAR
-------------------------------------------------------------------------------

-----------------------------------------------------
-- sp {propose*move-block*to-block
--     (state <s> ^block <block> {<> <block> <dest>})
--     (<block> ^name <a> ^below o)
--     (<dest>  ^name <b> ^below o)
-- -->
--     (<s> ^operator <o> + =)
--     (<o> ^name move-block
--          ^block <a>
--          ^destination <b>)}
-----------------------------------------------------

-- expands to six rules
-- N.B. we translate the operator preference to our bespoke form
-- N.B. preferences beyond '=' (indifferent) are ignored as yet!

propose_moveblock_1 = {
    (propose_moveblock_toblock,
        {
            wme.B1.name.A1,
            wme.B1.below.o,
            wme.D1.name.B2,
            wme.D1.below.o
        },
        {
            propose.Moveblock.A1.B2
        }
    ) | B1 <- multi2(st.block), D1 <- multi2(st.block), B1 != D1,
        A1 <- vals(B1.name), B2 <- vals(D1.name)
}


------------------------------------------------
-- sp {propose*move-block*to-table
--     (state <s> ^block <block>)
--     (<block> ^name <a> ^on <> table ^below o)
-- -->
--     (<s> ^operator <o> + =)
--     (<o> ^name move-block
--          ^block <a>
--          ^destination table)}
------------------------------------------------

-- expands to nine rules, three of which represent (presumably!) impossible
-- situations ("a on a", "b on b", "c on c")
-- N.B. we translate the operator preference to our bespoke form
-- N.B. preferences beyond '=' (indifferent) are ignored as yet!

propose_moveblock_2 = {
    (propose_moveblock_totable,
        {
            wme.B1.name.A1,
            wme.B1.on.O1,
            wme.B1.below.o
        },
        {
            propose.Moveblock.A1.table
        }
    ) | B1 <- multi2(st.block),
        A1 <- vals(B1.name), O1 <- vals(B1.on),
        O1 != table
}


----------------------------------------
-- sp {selection*dont-move*in-place
--     (state <s> ^operator <o> +)
--     (<o> ^name move-block ^block <n>)
--     (<s> ^block <b>)
--     (<b> ^name <n> ^in-place)
-- -->
--     (<s> ^operator <o> <)}
----------------------------------------

-- do we bother modelling operator preferences??
-- if we do then can we use belief maintenance as for normal I-support rules??
-- i.e. using "Disable.propose.??"

-------------------------------------------------------
-- selection_1 = {
--     (selection_dontmove_inplace,
--         {
--             propose.Moveblock.N1.D1
--             B1.name.N2,
--             B1.inplace.I1
--         },
--         {
--             ?? operator pref ??
--         }
--     ) | B1 <- multi(block.na.na),
--         N1 <- {a,b,c}, D1 <- {a,b,c,table},
--         N2 <- vals(B1.name), I1 <- vals(B1.inplace),
--         N1 == N2
-- }
-------------------------------------------------------


--------------------------------------------------------------------------------
-- APPLY.SOAR
--------------------------------------------------------------------------------

---------------------------------------------
-- sp {apply*move-block*to-table
--     (state <s> ^operator <op>
--                ^block <moving> {<> <moving> <origin>})
--     (<op> ^name move-block
--           ^block <m>
--           ^destination table)
--     (<moving> ^name <m> ^on <o> ^below o)
--     (<origin> ^name <o>         ^below <m>)
-- -->
--     (<moving> ^on table ^on <o> -)
--     (<origin> ^below o  ^below <m> -)}
----------------------------------------------

-- expands to six rules, all of which *must* represent the possible scenarios
-- N.B. these rules should be expanded again if we require only *one* consequent!

-- note that we put the disables right in the "o-support" inference!
--  - we had better know about it if there is a clash between positive and
--      negative support for a WME!
--  - these are the "trigger" disables and are *not* forgotten by destructive
--      rules (but may be reset by positive "o-support"!)
--  - wmes that appear within the RHS of "o-support" rules are considered
--      to be "o-support" wmes
--  - "o-support" wmes (& Disable.wme) behave differently to normal wmes:
--      - "o-support" Disable.wme's are *not* forgotten by destructive
--          rules or "I-support" normal rules,
--          but must be reset by positive ("O-support") rules!
--  - hence, conflicts between "I-support" & (negative) "o-support" could come out as
--      livelock?? i.e. the "I-support" inference pulls up the wme, then the
--      Disable.wme pulls it down, then the "I-support" inference pulls it up
--      again...
--      - what about conflicts between "I-support" and positive "O-support"??
--      - need to add some more Soar semantics!!



apply_moveblock_1 = {
    --------------------------------------
    -- (test_rule,
    --     {
    --         operator.Moveblock.a.table
    --     },
    --     {
    --         wme.st.block_1.on.c
    --     }
    -- ),
    -- (test_rule,
    --     {
    --         operator.Moveblock.a.table
    --     },
    --     {
    --         Disable.wme.st.block_1.on.c
    --     }
    -- ),
    --------------------------------------
    -------------------------------------------
    -- (test_rule,
    --     {
    --         operator.Moveblock.a.table
    --     },
    --     {
    --         Disable.wme.st.st.initialised.no
    --     }
    -- ),
    -------------------------------------------
    (apply_moveblock_totable,
        {
            operator.Moveblock.M1.table,
            wme.B1.name.M2,
            wme.B1.below.o,
            wme.B1.on.O1,
            wme.B2.name.O2,
            wme.B2.below.M3
        },
        {
            --  removal  not  caught,  in  soar  results  in  only  offering  last  operator
again!
            --  i.e. operator-no-change impasse
            wme.B1.on.table,
            -- removal caught, results in deadlock & no proposals!
            wme.B2.below.o,
            -- removal not caught, in soar as above
            Disable.wme.B1.on.O1,
            -- removal not caught, in soar no issue either!
            Disable.wme.B2.below.M1
        }
    ) | B1 <- multi2(st.block), B2 <- multi2(st.block), B1 != B2,
        M1 <- {a,b,c},
        M2 <- vals(B1.name), O1 <- vals(B1.on),
        O2 <- vals(B2.name), M3 <- vals(B2.below),
        M1 == M2, M2 == M3, O1 == O2
}


----------------------------------------------
-- sp {apply*move-block*from-table*to-block
--     (state <s> ^operator <op>
--                ^block <moving> {<> <moving> <dest>})
--     (<op> ^name move-block
--           ^block <m>
--           ^destination { <> table <d> })
--     (<moving> ^name <m> ^on table ^below o)
--     (<dest>   ^name <d>           ^below o)
-- -->
--     (<moving> ^on <d>    ^on table -)
--     (<dest>   ^below <m> ^below o -)}
----------------------------------------------

-- again, expands to six rules, all of which *must* represent the possible scenarios
-- N.B. these rules should be expanded (*2) again if we require only *one*
consequent!

apply_moveblock_2 = {
    (apply_moveblock_fromtable_toblock,
        {
            operator.Moveblock.M1.D1,
            wme.B1.name.M2,
            wme.B1.on.table,
            wme.B1.below.o,
            wme.B2.name.D2,
            wme.B2.below.o
        },
        {
            wme.B1.on.D1,
            wme.B2.below.M1,
            Disable.wme.B1.on.table,
            Disable.wme.B2.below.o
        }
    ) | B1 <- multi2(st.block), B2 <- multi2(st.block), B1 != B2,
        M1 <- {a,b,c}, D1 <- {a,b,c,table}, D1 != table,
        M2 <- vals(B1.name), D2 <- vals(B2.name),
        M1 == M2, D1 == D2
}


-----------------------------------------------------------
-- sp {apply*move-block*from-block*to-block
--     (state <s> ^operator <op>
--                ^block <moving> {<> <moving> <origin>}
--                       {<> <moving> <> <origin> <dest>})
--     (<op> ^name move-block
--           ^block <m>
--           ^destination { <> table <d> })
--     (<moving> ^name <m> ^on { <> table <o> } ^below o)
--     (<dest>   ^name <d>                      ^below o)
--     (<origin> ^name <o>                      ^below <m>)
-- -->
--     (<moving> ^on <d>    ^on <o> -)
--     (<dest>   ^below <m> ^below o -)
--     (<origin> ^below o   ^below <m> -)}
-----------------------------------------------------------

-- again, expands to six rules, all of which *must* represent the possible scenarios
-- N.B. these rules should be expanded (*3) again if we require only *one*
consequent!
```

```
apply_moveblock_3 = {
    (apply_moveblock_fromblock_toblock,
        {
            operator.Moveblock.M1.D1,
            wme.B1.name.M2,
            wme.B1.on.O1,
            wme.B1.below.o,
            wme.B2.name.D2,
            wme.B2.below.o,
            wme.B3.name.O2,
            wme.B3.below.M3
        },
        {
            wme.B1.on.D1,
            wme.B2.below.M1,
            wme.B3.below.o,
            Disable.wme.B1.on.O1,
            Disable.wme.B2.below.o,
            Disable.wme.B3.below.M1
        }
    ) | B1 <- multi2(st.block), B2 <- multi2(st.block), B3 <- multi2(st.block),
        B1 != B2, B3 != B1, B3 != B2,
        M1 <- {a,b,c}, D1 <- {a,b,c,table}, D1 != table,
        M2 <- vals(B1.name), O1 <- vals(B1.on),
        D2 <- vals(B2.name),
        O2 <- vals(B3.name), M3 <- vals(B3.below),
        D1 != table, O1 != table, M1 == M2, M2 == M3, D1 == D2, O1 == O2
}



--------------------------------------------------------------------------------
-- MONITOR.SOAR
-- all have no real RHS action!
--------------------------------------------------------------------------------

----------------------------------------------------
-- sp {monitor*goal*achieved
--     (state <s> ^goal <g>)
--     (<g> ^block <ga> <gb> <gc>)
--     (<ga> ^name a ^on <a1> ^below <a2>)
--     (<gb> ^name b ^on <b1> ^below <b2>)
--     (<gc> ^name c ^on <c1> ^below <c2>)
--     (<s> ^block <a> <b> <c>)
--     (<a> ^name a ^on <a1> ^below <a2>)
--     (<b> ^name b ^on <b1> ^below <b2>)
--     (<c> ^name c ^on <c1> ^below <c2>)
-- -->
--     (write (crlf) |The problem has been solved.|)
--     (halt)}
----------------------------------------------------

--------------------------------------------------------------------------------
-- INITIALISE.SOAR
--------------------------------------------------------------------------------


----------------------------------------------------------------------
-- sp {propose*initialise
--     (state <s> -^initialised)
-- -->
--     (<s> ^operator <o> + =)
--     (<o> ^name initialise)}
----------------------------------------------------------------------

-- as we don't handle negated attribute conditions yet,
-- we convert (by hand!) "-^initialised" to "^initialised no" and make sure
-- wme.st.st.initialised.no is initially true!!

propose_initialise_1 = {
    (propose_initialise,
        {
            wme.st.st.initialised.no
        },
        {
            propose.Initialise
        }
    )
}



---------------------------------------------------------------------
-- sp {apply*initialise
--     (state <s> ^operator.name initialise)
-- -->
--     (<s> ^initialised yes)
--     # Initial State
--     (<s> ^block <a> <b> <c>)
--     (<a> ^name a ^on table ^below o)
--     (<b> ^name b ^on table ^below o)
--     (<c> ^name c ^on table ^below o)
--     (write (crlf) |Initial state has A, B & C on the table.|)
--     # Goal State
--     (<s> ^goal <g>)
--     (<g> ^block <ga> <gb> <gc>)
--     (<ga> ^name a ^on b     ^below o)
--     (<gb> ^name b ^on c     ^below a)
--     (<gc> ^name c ^on table ^below b)
--     (write (crlf) |The goal is to get A on B on C on the table.|)}
---------------------------------------------------------------------

-- as we don't handle negated attribute conditions yet (see propose*initialise)
-- we add (by hand!) "^initialised no -" to the RHS actions.

-- note the implicit "<a> != <b> != <c>" when creating multi-attribute WMEs!
-- we need to resolve multi-attributes to unique solutions or we get non-det
-- rules!!


apply_initialise_1 = {
    (apply_initialise,
        {
            operator.Initialise
        },
        {
            wme.st.st.initialised.yes,
            Disable.wme.st.st.initialised.no,
            wme.B1.name.a,
            wme.B1.on.c,
            wme.B1.below.o,
            wme.B2.name.b,
            wme.B2.on.table,
            wme.B2.below.o,
            wme.B3.name.c,
            wme.B3.on.table,
            wme.B3.below.a,
            wme.GA.name.a,
            wme.GA.on.b,
            wme.GA.below.o,
            wme.GB.name.b,
            wme.GB.on.c,
            wme.GB.below.a,
            wme.GC.name.c,
            wme.GC.on.table,
            wme.GC.below.b
        }
    ) | B1 <- multi2(st.block), B2 <- multi2(st.block), B3 <- multi2(st.block),
        B1 != B2, B3 != B1, B3 != B2,
        member(a,vals(B1.name)), member(table,vals(B1.on)), member(o,vals(B1.below)),
        member(b,vals(B2.name)), member(table,vals(B2.on)), member(o,vals(B2.below)),
        member(c,vals(B3.name)), member(table,vals(B3.on)), member(o,vals(B3.below)),
        GA <- multi2(goal.block), GB <- multi2(goal.block), GC <- multi2(goal.block),
        GA != GB, GC != GB, GC != GA,
        member(a,vals(GA.name)), member(b,vals(GA.on)), member(o,vals(GA.below)),
        member(b,vals(GB.name)), member(c,vals(GB.on)), member(a,vals(GB.below)),
        member(c,vals(GC.name)), member(table,vals(GC.on)), member(b,vals(GC.below))
```

```
}



--------------------------------------------------------------------------------
-- BELIEF MAINTENANCE RULES
--------------------------------------------------------------------------------

-- Calculate "destructive" belief maintenance rules from (I-support) rule dependencies.
--
-- Each rule implements the forgetting ("disabling") of an WME and its (potential)
--  immediate effects on dependent "I-support" rules (i.e. new "disable" actions).
-- The rules have semantics: (<dis>, <wme> --> <cons>)
--             <dis>, <wme> |- not* <dis>, not <wme>, <cons>
--      where <dis> is the "disable" action, <wme> is the WME, and <cons> is a
--      set of dependent "disable" actions (possibly empty).
-- * When <dis> is an "O-support" WME, the consequent is actually "<dis>" as we
--  expect forgetting of an "O-support" WME to be persistent (until an "O-support"
--  rule explicitly cancels the "disable").
--
-- Conflicts:
--
-- Although we currently assume a given WME will either have "O-support" (it is
--  added/removed by an "O-support" rule) or "I-support" (it is added by an
--  "I-support" rule), we may currently detect conflicts between "O-support" rules.
-- Here, two "O-support" rules compete to forget and recall an "O-support" WME
--  and we may end up with non-determinism (one rule disables the other) or
--  livelock as each undoes the actions of the other.
--
-- If an "I-support" rule gives (positive) support for an "O-support" WME then we
-- may detect similar conflicts as above (likely livelock), or worse, the ensuing
-- "disable" action may forget previous (positive) "O-support" - masking possible
-- behaviour! For this reason, we assume a given WME will either have "O-support"
-- *or* "I-support". This may be confirmed with a simple syntactic check!
--
-- If we want to model this issues of support, we must answer some questions:
--  1. What happens when the positive "I-support" is lost?
--      - the "disable" action for an "O-support" WME is currently permanent
--  2. What happens when there was previous positive "O-support"? how can we know?
--      do we care?
--
-- Algorithm:
--
--  Assuming any (I-support) rule may be retracted in its life-time, we construct
--  the set of all WMEs that may be forgotten from the consequents of these rules
--  and any explicit "disable" actions in a set of (O-support) trigger rules.
--
--  Each "starter" WME then contributes a (belief maintenance) rule as above with
--  all its (potentially) dependent (I-support) rule's consequents as "disable"
--  actions (excluding itself).
--
--  This is a pessimistic, simple model that could be refined and developed.

dependencies(triggers_,rules_) =
    let
        starters_ = Union({  { t_ | (_,_,C_) <- triggers_, Disable.t_ <- C_ },
                            { wme.i | (_,_,C_) <- rules_, wme.i <- C_ } })
    within
    {(belief_maintenance,
        {
            Disable.F_,
            F_
        },
        {
            Disable.wme.c_ | (_,A_,CC_) <- rules_, member(F_,A_), wme.c_ <- CC_,
                (wme.c_) != F_
        }
    ) | F_ <- starters_
}
```

```
-- example: if one block is no longer in place then any block sitting on it
--          may no longer be in place! (see observe*block*in-place*2)
---------------------------------------
-- (Dis.???,
--     {Disable.wme.st.block_3.inplace.yes,
--      wme.st.block_3.inplace.yes},
--     {Disable.wme.st.block_2.inplace.yes,
--      Disable.wme.st.block_1.inplace.yes}
-- )
---------------------------------------





-------------------------------------------------------------------------------
-- SOAR MODEL
-------------------------------------------------------------------------------

-- A Soar production rule is an "O-support" rule iff it tests for an operator in
--  the LHS, else it is an "I-support" rule.
-- A Soar working memory element (WME) is an "O-support" WME iff it appears in the
--  RHS of an "O-support" rule, else it is an "I-support" WME.
--
-- O-SUPPORT Rules:
--
-- "O-support" rules have semantics: (<operator>, <wmes> --> <cons>)
--          <operator>, <wmes>, not <cons> |- <cons>
--      where <wmes> is a set of any WMEs (possibly empty) and <cons> is a set of
--      WMEs and "disable" actions.
--
-- Note that we require 'not <cons>', i.e. one or more of the consequents are
--  false. This prevents the rule from continually firing unless progress is being
--  made, hence 'livelock' will not occur unless caused by rule interactions.
--
-- When the rule has only one consequent (positive or negative), then it is a
--  simple matter to prevent the rule from firing unnecessarily.
-- When the rule has multiple consequents, then we require coordination to get
--  the required behaviour. Dividing up the rule into multiple rules, each with one
--  consequent proved problematic as "O-support" rules have negative actions.
-- To get this behaviour we use instead an 'oracle' that insists that one of the
--  consequents of such a rule must be 'false' for the rule to re-fire. Note that
--  this only guarantees that the consequents *were* previously false.
-- For "blocks world" we only have to use oracles for "apply_2" rules as the other
--  "O-support" rules are incapable of firing while making no progress, under the
--  assumption that negative actions are given priority over positive actions. Here,
--  all the rules contain a negative action undermining a condition.
--
-- I-SUPPORT Rules:
--
-- "I-support" rules have semantics: (<wmes> --> <cons>)
--          <wmes>, not <cons> |- <cons>
--      where <wmes> and <cons> are sets of WMEs, where <cons> only contains
--      "I-support" WMEs.
--
-- Again, we require 'not <cons>', i.e. one or more of the consequents are
--  false, to prevent the rule from continually firing unless progress is
--  being made.
--
-- Again, the issue of 'not <cons>' and multiple consequents has to be resolved.
-- However, as "I-support" rules contain no negative actions we have decided to
--  split up "I-support" rules with multiple consequents by default. We may then
--  manually apply 'oracles' to such rules that cause (hopefully rare)
--  "false-negatives". Also, "I-support" rules are maintained by implicit
--  belief maintenace, the monitoring of the consequents will be harder.
-- For instance, one consequent could enable an "O-support" rule disabling the
--  rules before another consequent could enable an operator proposal, hence a
--  "false" impasse.


-- All the Soar production rules as above.
the_rules = Union({
    observe_block_1,observe_block_2,
    propose_moveblock_1,propose_moveblock_2,
    apply_moveblock_1,apply_moveblock_2,apply_moveblock_3,
    propose_initialise_1,apply_initialise_1
})

-- "O-support" rules (see above for definition)
o_support = { (R_,A_,C_) | (R_,A_,C_) <- the_rules, inter(A_,{|operator|}) != {}}

-- "I-support" rules
i_support = diff(the_rules,o_support)

-- "O-support" WMEs (see above for definition)
o_support_wme = Union({ C_ | (_,_,C_) <- o_support})
o_supported(i) = member(wme.i,o_support_wme) or member(Disable.wme.i,o_support_wme)

deductions = the_rules
ddeductions = dependencies(o_support,i_support)

-- inferences

channel infer : union(deductions,ddeductions)


channel tock


-- count things...

-- Rules (69): 19 o_support, 28 i_support, 22 belief maintenance

-- 89!! (
all_facts = Union({ A_,C_ | (_,A_,C_) <- union(deductions,ddeductions)})

-- 22
all_disable = inter(all_facts,{|Disable|})

-- 10
all_proposals = inter(all_facts,{|propose|})

-- 10
all_operators = inter(all_facts,{|operator|})

-- 47!!
all_wmes = inter(all_facts,{|wme|})




-------------------------------------------------------------------------------
-- WORKING MEMORY ELEMENT (WME)
-------------------------------------------------------------------------------

alpha_WME(i) = {
    infer.(R_,A_,C_) | (R_,A_,C_) <- union(deductions,ddeductions),
                       member(wme.i,C_) or member(wme.i,A_)
}

-- blocks all normal inferences with 'i' in antecedents
WME(i,false) =
    let
        Infer_from      = {(R_,A_,C_) | (R_,A_,C_) <- deductions, member(wme.i,C_)}
        Infer_from_d    = {(R_,A_,C_) | (R_,A_,C_) <- ddeductions, member(wme.i,C_)}
    within
        -- learn 'i'
        ( [] r_ <- Infer_from @ infer.r_ -> WME(i,true) )
    []
        -- learn 'i' - is this ever needed?
        ( [] r_ <- Infer_from_d @ infer.r_ -> WME(i,true) )

-- blocks all normal inferences with *only* 'i' in consequents
WME(i,true) =
    let
        Applicable      = {(R_,A_,C_) | (R_,A_,C_) <- deductions, member(wme.i,A_)
                                              or (card(C_) > 1 and member(wme.i,C_))}
        Forget          = {(R_,A_,C_) | (R_,A_,C_) <- ddeductions, member(wme.i,A_)}
    within
        -- allow learn 'c of C'
        ( [] r_ <- Applicable @ infer.r_ -> WME(i,true) )
    []
        -- forget 'i'
        ( [] r_ <- Forget @ infer.r_ -> WME(i,false) )

-------------------------------------------------------------------------------
-- DISABLE ACTION
-------------------------------------------------------------------------------

-- A "disable" action is a similar to a WME, but differs as follows:
--  - a "disable" action is cancelled implicitly by a normal inference
--      of the right type, i.e. "O-support" for "O-support" WMEs
--  - an "O-support" (persistent) "disable" action is not forgotten by a
--      "destructive" inference when one of the antecedents

alpha_DISABLE(i) =
    let
        -- the rules that may request or cancel a "disable" action.
        Rules = if o_supported(i) then o_support else i_support
    within
        Union({
    { infer.(R_,A_,C_) | (R_,A_,C_) <- union(deductions,ddeductions),
                        member(Disable.wme.i,C_) or member(Disable.wme.i,A_) },
    { infer.(R_,A_,C_) | (R_,A_,C_) <- Rules, member(wme.i,C_)}
})


-- blocks all normal inferences (when true) with *only* 'disable.i' in consequents
DISABLE(i,bool) =
    let
        -- the rules that may request or cancel a "disable" action.
        Rules   = if o_supported(i) then o_support else i_support
        Request = {(R_,A_,C_) | (R_,A_,C_) <- Rules, (not bool or card(C_) > 1),
                                                    member(Disable.wme.i,C_)}
        Cancel  = {(R_,A_,C_) | (R_,A_,C_) <- Rules, member(wme.i,C_)}
        -- the rules that trigger and perform the "disable" action.
        Trigger = {(R_,A_,C_) | (R_,A_,C_) <- ddeductions, member(Disable.wme.i,C_)}
        Perform = {(R_,A_,C_) | (R_,A_,C_) <- ddeductions, member(Disable.wme.i,A_)}
    within
        -- request or trigger a "disable" action.
        ( [] r_ <- union(Request,Trigger) @ infer.r_ -> DISABLE(i,true) )
    []
        -- cancel a "disable" action.
        ( [] r_ <- Cancel @ infer.r_ -> DISABLE(i,false) )
    []
        -- perform a "disable" action.
        bool & ( [] r_ <- Perform @ infer.r_ -> DISABLE(i,o_supported(i)) )


-------------------------------------------------------------------------------
-- ORACLE
-------------------------------------------------------------------------------

-- When an "O-support" rule fires, prevent it from continually firing unless
-- progress is being made, hence 'livelock' should not occur unless caused by
-- rule interactions.
--
-- The 'oracle' only allows a rule to re-fire when one of its consequents has
-- previously been 'false'. As "O-support" WMEs are explicitly added and removed,
-- we may simply monitor the firing of other "O-support" rules that add or remove
-- the consequents (and *not* the belief maintenance rules!)

alpha_ORACLE(rule_@@(_,_,RHS_)) =
    let
        Pos = { infer.rule_, infer.(R_,A_,C_) | (R_,A_,C_) <- o_support, wme.F_ <-
RHS_,
                                                    member(Disable.wme.F_,C_) }
        Neg = { infer.(R_,A_,C_) | (R_,A_,C_) <- o_support, Disable.wme.F_ <- RHS_,
                                        member(wme.F_,C_) }
    within
        union(Pos,Neg)
```

```
ORACLE(rule_@@(_,_,RHS_),fired_) =
    let
        Pos = { (R_,A_,C_) | wme.F_ <- RHS_, (R_,A_,C_) <- o_support,
                                  member(Disable.wme.F_,C_) }
        Neg = { (R_,A_,C_) | Disable.wme.F_ <- RHS_, (R_,A_,C_) <- o_support,
                                  member(wme.F_,C_) }
    within
        -- a rule may fire once
        not(fired_) & infer.rule_ -> ORACLE(rule_,true)
    []
        -- the rule may fire again once one of its consequents goes 'false'
        ( [] r_ <- union(Pos,Neg) @ infer.r_ -> ORACLE(rule_,false) )


------------------------------------------------------------------------------
-- INFERENCES
------------------------------------------------------------------------------


INFERENCES(initials_) =
    let
        facts_      = Union({ A_,C_ | (_,A_,C_) <- union(deductions,ddeductions)})
        wmes_       = { i | wme.i <- facts_ }
        disables_   = { i | Disable.wme.i <- facts_ }
        -- we add oracles to "O-support" rules that have multiple consequents
        -- and do not disable themselves
        oracles_    = { (R_,A_,C_) | (R_,A_,C_) <- o_support, card(C_) > 1,
                                      Disable.wme.F_ <- C_, not member(wme.F_,A_) }
        alpha_WMES  = Union({alpha_WME(i) | i <- wmes_})
        WMES        = || i : wmes_ @ [alpha_WME(i)] WME(i,member(i,initials_))
        alpha_DISABLES  = Union({alpha_DISABLE(i) | i <- disables_})
        DISABLES    = || i : disables_ @ [alpha_DISABLE(i)] DISABLE(i,false)
        alpha_ORACLES   = Union({alpha_ORACLE(r_) | r_ <- oracles_})
        ORACLES     = || r_ : oracles_ @ [alpha_ORACLE(r_)] ORACLE(r_,false)
    within
        (WMES [alpha_WMES || alpha_DISABLES] DISABLES)
            [union(alpha_WMES,alpha_DISABLES) || alpha_ORACLES] ORACLES



------------------------------------------------------------------------------
-- DECIDE MODULE
------------------------------------------------------------------------------

-- Model the decision cycle which maps operator proposals to operator decisions,
-- conveys external input/output to and from the system and ??
--
-- The decision cycle consists of separate "phases" of production rule firing,
-- with one phases separated from the next by system aquiescence (no more rules
-- are eligible to fire).
--
-- Although we could perhaps


alpha_DECIDE = { infer.(R_,A_,C_) | (R_,A_,C_) <- deductions,
                    inter({|propose,operator|},union(A_,C_)) != {} }


-- maps proposals to applicable operator inferences before tock'ing.

DECIDE =
    let
        proposals(op_)    =    {    (R_,A_,C_)    |    (R_,A_,C_)    <-    deductions,
member(propose.op_,C_) }
    within
        (  []  op_  :  Operator,  r_  :  proposals(op_)  @  infer.r_  ->  decide.op_  ->
DECIDED(op_) )

DECIDED(op) =
    let
        operators = { (R_,A_,C_) | (R_,A_,C_) <- deductions, member(operator.op,A_) }
    within
```

```
            ( [] r_ <- operators @ infer.r_ -> DECIDED(op) )
        []
            tock -> DECIDE


normal_goal = {
    st.st.initialised.yes,
    goal.block_1.name.a,
    goal.block_1.on.b,
    goal.block_1.below.o,
    goal.block_2.name.b,
    goal.block_2.on.c,
    goal.block_2.below.a,
    goal.block_3.name.c,
    goal.block_3.on.table,
    goal.block_3.below.b,
    st.block_1.name.a,
    st.block_1.on.table,
    st.block_1.below.o,
    st.block_2.name.b,
    st.block_2.on.table,
    st.block_2.below.o,
    st.block_3.name.c,
    st.block_3.on.table,
    st.block_3.below.o
}

no_goal = {
    st.st.initialised.no
}

initially_true = no_goal


transparent chase

belief_rules = { infer.r_ | r_ <- ddeductions }

CHASED = chase(INFERENCES(initially_true) \ belief_rules)

-- this process is deterministic!
TEST = CHASED [| alpha_DECIDE |] DECIDE

LIVELOCK_TEST = TEST \ { infer.r_ | r_ <- Union({deductions}) }


SYSTEM = INFERENCES(initially_true) [| alpha_DECIDE |] DECIDE
```

# Initial distribution list

| **External** | |
|---|---|
| Dr C Leach, RT(RAO RD WPE and OP1) | MOD |
| R Piller, RPC(WPE) | Dstl |
| Dstl Knowledge Services | |
| **QinetiQ** | |
| Information Resources | |
| Dr R Trumper, Channel Manager | Portsdown West |
| Prof. C O'Halloran | Malvern |
| R Harrison | Malvern |
| Prof. J Woodcock | Kent University |
| A McEwan | Kent University |
| A T McCallum | Bedford |
| Dr Y Patel | Bedford |
| Dr J Platts | Bedford |
| M Downes, BGM Platform Systems | Bedford |
| R Reading | Bedford |
| Dr S W Willcox | Blue Bear Systems Research, Ltd |

# Report documentation page

| | | | |
|---|---|---|---|
| Originator's Report Number | QinetiQ/FST/CR041616/1.0 | | |
| Originator's Name and Location | A T McCallum, Room 47, Building 109, The Enclave, Thurleigh, BEDFORD, MK44 2FQ | | |
| Customer Contract Number and Period Covered | FST/EGC/077 | | |
| Customer Sponsor's Post/Name and Location | Dr C Leach, DG(R&T)-(RD EGC) | | |
| Report Protective Marking and any other markings<br>QinetiQ Proprietary | Date of issue<br>26 March 2004 | Pagination<br>Cover + 62 | No. of references<br>[11] |
| Report Title<br>Rationalising Over-Determined Intelligence | | | |
| Translation / Conference details (if translation give foreign title / if part of conference then give conference particulars)<br>None | | | |
| Title Protective Marking | QinetiQ Proprietary | | |
| Authors | R D Harrison, C M O'Halloran, J Woodcock and A McEwan | | |
| Downgrading Statement | None | | |
| Secondary Release Limitations | DEFCON 705 (Edn 11/02) | | |
| Announcement Limitations | None | | |
| Keywords / Descriptors | UAV, Agents, Autonomy, Clearance, Formal Methods | | |

Abstract

The certification of Machine Intelligence algorithms falls into two parts: the formal mathematical validation of the safety of the Machine Intelligence algorithm; and the formal mathematical verification of the implementation of the algorithm. This report describes a subset of the Soar language that is essentially certifiable and, by providing a formal semantics for programs written in this subset, that can be verified for healthiness properties, such as deadlock or livelock. In particular, the concept of over-determined machine intelligence is taken to be over specialisation leading to rule redundancy, which this report discusses and shows can be automatically detected as a healthiness property. The formal semantics for the subset of the Soar language are provided by a prototype translator from Soar into an non-monotonic inference engine in the formal language of Communicating Sequential Processes, CSP. Further such Soar programs can be verified against critical properties identified by a system safety case for an autonomous UAV.

| | |
|---|---|
| Abstract Protective Marking: | UNCLASSIFIED QinetiQ Proprietary |

This form meets DRIC-SPEC 1000 issue 7

**Blank page**