# Efficient Forward Chaining for Declarative Rules in a Multi-Agent Modelling Language

**Steve Wallis and Scott Moss**
**s.wallis@mmu.ac.uk, s.moss@mmu.ac.uk**
**Centre for Policy Modelling, Manchester Metropolitan University, UK**
**CPM Report: 004, 21 October 1994 (revised 16 February 1995)**

## Abstract

Most rule-based systems are imperative, or, if they are mainly declarative, also provide some imperative features. It is argued that a strictly declarative formalism with a sound logical basis is preferable.

This paper addresses the task of implementing forward chaining efficiently for declarative rulebases. Deductions that can be invalidated by new information are particularly difficult to deal with consistently and efficiently.

The paper shows that efficient forward chaining is possible, by utilising the implicit structure of the rulebases to optimise the most common circumstances. A language, called SDML, which utilises such declarative rulebases to simulate multi-agent models, is described.

# 1 Introduction

This paper starts with some motivations for using strictly declarative rules. Section 3 then describes how knowledge is represented declaratively in SDML[1], using rulebases and databases, and how these structures are used to construct models.

Section 4 describes how SDML fires rules using forward and backward chaining. Rulebases are typically regarded as unstructured collections of rules, but they contain implicit structure, as defined by the dependencies between rules. In addition to helping users construct rulebases and detect errors, they are extremely useful for efficient forward chaining.

Section 5 describes how SDML can make provisional deductions that may be invalidated by new information. This is commonly referred to as *non-monotonic reasoning* [9]. If a deduction is made assuming that a predicate is not true, and this predicate is later deduced to be true, then the deduction must be retracted. It would be very restrictive if the deduction could only have been made if the predicate had already been proven to be false.

Section 6 provides some measurements of the speed of SDML, and compares it with an imperative rule-based system. Section 7 compares the features of SDML with other systems. Finally, Section 8 presents some conclusions.

# 2 Motivations for strictly declarative rules

The "procedural-declarative controversy" has often been discussed in Artificial Intelligence (in [18] for example). Imperative (or procedural) structures specify how to do something, whereas declarative knowledge specifies what is true. Imperative techniques are usually justified on the grounds of efficiency or because some kinds of knowledge are hard to represent

1. SDML stands for "Strictly Declarative Modelling Language".

declaratively. Declarative knowledge tends to be more modular (because imperative structures combine knowledge and control information) and economic (because the same knowledge can be used in many different ways). More importantly, they have better semantics being more closely related to logic, leading to the following advantages for declarative rules in general (Section 8 examines how these motivations apply to SDML):

- All models which purport to represent some aspect of the world make simplifications. Knowledge which is assumed to be irrelevant, because it is too detailed or seemingly unrelated, is left out. Other knowledge is left out because it is difficult to represent or utilise, or because the modeller is not aware of it. Assumptions can also be made about what knowledge to deduce when there are multiple valid possibilities. Such assumptions are often hidden within imperative rules, but may have to be stated more explicitly if rules are declarative.
- Contradictory and incorrect knowledge can be easily deduced by imperative rules, because “facts” can be retracted without following through the consequences. Such errors tend to occur less frequently with declarative rules, they can often be detected automatically, and they are easier for modellers to correct.
- In some declarative systems, rules continue firing until there are no more deductions that can be made. Modellers can therefore be more confident that something is not true if it has not been deduced.
- The order in which rules fire may be affected by the order they appear in a rulebase or by some external algorithm. Ordering often affects the results of firing imperative rules. If firing is completed for declarative rules, then the results are usually the same no matter what order they were fired in.

All these motivations entail that modellers who use declarative rules can have more confidence that their models are correct and that they yield correct results.

# 3 Representation of knowledge in SDML

This section firstly describes how units of information are represented in SDML as *clauses*, and then how clauses are used to construct databases and rulebases. Finally, this section describes how models are composed of multiple databases and rulebases.

## 3.1 Clauses

There are three basic types in SDML, using which all knowledge is represented:

- **clause**. A clause consists of a *keyword* plus a number of *arguments*.
- **number** (integers or floating point).
- **variable**, used to represent indefinite information. Variables can be bound to specific clauses or numbers, or to other variables.

SDML provides *type hierarchies*, which allow other types to be defined in terms of these basic types. For example, **symbol** is defined as a subtype of **clause**; a symbol is regarded as a clause with no arguments. Real world objects are typically represented in SDML as instances of subtypes of **symbol**.

An argument of a clause may be a symbol, number, variable or subclause. Clauses may be nested to an arbitrary depth, enabling a large amount of knowledge to be represented by a single clause. Information is often represented using type **list**, defined as a subtype of **clause**. Matrices are regarded as lists of lists of numbers.

Although symbols, lists and matrices are special kinds of clauses from the modeller’s point of view, more efficient representations are used in the implementation of SDML. Thus, clauses are simple and uniform, yet powerful and efficient, structures for the representation of knowledge.

## 3.2 Databases

In SDML, a database is a collection of clauses. Each clause on a database represents something that is known to be true.[2] Clauses can be placed in databases explicitly by the modeller, or they can be deduced in the course of a simulation (by rules firing as described in Section 4).

Thus, clauses can be used to represent logical predicates, and each clause on a database represents a predicate that is definitely true. For convenience, let us say that such clauses *are* true. The absence of a clause from a database does not necessarily mean that it is false, but merely that it has not been deduced to be true. It should be noted that some clauses, such as those representing lists, do not correspond to logical predicates and should only be placed on databases as arguments of other clauses.

The keywords of clauses that can be used in a database are defined, along with the legal types of the arguments, in a *clause definition group*. Modellers use clause definitions to specify how specific kinds of knowledge are to be represented, as clauses on databases. Checks are performed automatically to ensure that clauses adhere to the syntax specified in their definitions.

As mentioned earlier, clauses can contain subclauses, and a large amount of information can be stored in a single clause on a database. In practice, most clause definitions restrict their arguments to numbers or symbols (or specific subtypes), and such cases are optimised for more efficient access.

There are three main operations that can be used to access databases:

- *Retrieve*. This operation finds all the clauses in a database that match a particular clause, using unification. If there are any variables in either clause, then they are bound to corresponding terms in the other clause. A mapping from variables to the terms they are bound to is known as a *bindings dictionary.* Retrieval may yield zero, one or many solutions, with a different bindings dictionary for each solution. Retrieving a clause determines whether it is known to be true, and what values the variables must have for the clause to be true if it contains any.
- *Assert*. This operation adds a clause to a database, unless it is already present. Asserting a clause notes that it is true.
- *Retract*. This operation removes all clauses which unify with a particular clause from a database. Retracting a clause notes that it may not be true.

All three of the above operations can be performed by a modeller, using an appropriate tool, to check or modify knowledge on a database. However, clauses cannot be retracted during a simulation.[3] Every clause on a database represents the knowledge that something is true; if the simulation later deduces that it is false then the clause should not have been asserted in the

2. This may seem overly restrictive. However, many different kinds of knowledge can be represented in this way — for example, it is true that A believes something; it is true that B would be true if C were true; it is true that D is definitely false. In the current release of SDML, there is no standard notation for any of these kinds of knowledge, and the modeller must create appropriate clause definitions. In a later release, it is planned to provide additional databases for things that are known to be false.
3. Except by the assumption manager, which can retract clauses which depend on assumptions that turn out to be false, as described in Section 5.

first place. Logical inconsistencies would arise if asserting a clause caused further deductions to be made and the clause was later retracted.

In the real world, things that are true at one time (or date) can become false later. Instead of allowing such knowledge to be retracted, the period of time during which it is known to be true is noted. This is done using *subdatabases*, each of which contains clauses that are true at a particular time. Knowledge that is always true, or will remain true after a particular time, is represented on *permanent databases*. Every permanent database has a subdatabase corresponding to each time period of the simulation.[4] Clauses in a permanent database are inherited by its subdatabases, so that retrieving from a subdatabase determines whether a predicate is true at the appropriate time.

Retaining all the subdatabases generated during a simulation could waste a lot of memory, so a facility is provided to automatically dispose of old subdatabases that are no longer required. In order to ensure that this does not result in incorrect results, an error is reported if it is attempted to retrieve from a database that has been disposed of.

In common with the rest of SDML, databases were designed to provide modellers with a large amount of flexibility and a sound logical basis, without much loss of efficiency. Databases are represented in such a way that common operations, such as retrieving clauses with numbers and symbols as arguments, can be dealt with particularly efficiently. Internal representations can easily be changed, for example to interface with a relational database.

## 3.3 Rulebases

A rulebase in SDML is a collection of rules, each of which has a name, a number of *antecedents* and a number of *consequents*. The antecedents of a rule are represented by a single clause, with subclauses if there is more than one antecedent. Another clause represents the consequents of the rule. Every rule states that if its antecedents are true for any bindings of the variables used in them, then its consequents are also true for the same bindings of variables. Rules are strictly declarative; they declare relationships between clauses representing logical predicates about the world.

Section 4 describes how rules are fired, in order to deduce further items of knowledge from existing knowledge represented on databases. Firing rules cannot retract any knowledge that has already been deduced.

Clauses in rules often correspond to clauses on databases, but some *primitive* keywords have also been provided. Definitions of primitives are contained in a clause definition group, which is also used to specify the syntax of clauses that can be placed on databases. For example, the primitive **sum** can be used for addition; a clause with this keyword is true if the first argument (a number or matrix) is the sum of the remaining arguments. This primitive behaves as if there is an infinite database containing clauses corresponding to all correct additions, such as **sum 4 2 2**. Similar clauses, containing variables, can be used as antecedents of rules to represent mathematical relationships. Such clauses are not permitted as consequents; strange behaviour could result if it was possible to deduce that **sum 5 2 2** is true!

Other primitives are provided to enable relationships to be specified between knowledge on different databases. For example, the primitive **last** refers to the subdatabase corresponding to the previous time period. A clause with keyword **last** is true if its subclause was true just

4. At present, only evenly spaced discrete time periods can be represented, and it is efficient to maintain separate subdatabases for each time period. More complex representations of time will be supported in future releases, which may be reflected by different internal representations of databases.

before the current time. If such clauses were permitted in the consequents of rules, firing them could cause antecedents of rules at previous times could become true, and firing these rules could cause rules at even earlier times to be fired. Since SDML is used for simulation, and past events should not be affected by things happening in the future, primitives such as **last** are not permitted in consequents. Once a time period has been simulated, rules at that period cannot be fired again. It should be noted that this restriction does not prevent deductions from being made about previous times, but only affects where such knowledge is represented. This restriction also benefits efficiency, since it is only necessary to check whether rules can fire at the current time.

Logical primitives can be used to construct more complex antecedents and consequents from simpler ones. These primitives are described below:

- **and**. A clause with this keyword and a number of subclauses as arguments is true if all subclauses are true.
- **or**. A clause with this keyword and a number of subclauses as arguments is true if any subclause is true.[5]
- **not**. A clause with this keyword and a single subclause as its argument is true if the subclause is false.[6]
- **notKnown**. A clause with this keyword and a single subclause as its argument is true if the subclause is not known to be true. This may be because the subclause is false, or because there are no rules to deduce that it is true.[7]

Two primitives are provided for negation to avoid making the closed world assumption [13] that anything that cannot be proven is false. The **not** primitive can be used for knowledge that is deduced to be false, whereas **notKnown** can be used to explicitly state any assumptions that have to be made.

Ordinary clauses corresponding to logical predicates are combined using such logical primitives ensuring a sound logical basis is provided for SDML. It is, of course, important to ensure that all primitives are logically consistent. Many primitives, like **sum**, behave as if they retrieve from a database. Some of them, particularly those which manipulate lists, were originally written as rules but later reimplemented as primitives for improved efficiency.

## 3.4 Models

In SDML, models comprise a number of *agents*, which represent distinct entities (such as people or organisations) that are capable of making decisions and interacting with each other and the environment. Their behaviour is specified using rules and they communicate by placing information on databases. Agents have their own private rulebases and databases, and also have access to a public database contained in the model. The model also has its own rulebases, which can be used to represent the environment.

This modular approach of SDML offers a number of advantages over a single rulebase and database:

---

5. The primitive **or** could be used to conclude that at least one consequent is true, for example that the weather is hot or cold. The same knowledge can be expressed using the two rules in figure 7. Currently, this primitive cannot be used in consequents. If permitted, one rule's consequent could affect consequents as well as antecedents of other rules.
6. The primitive **not** will be implemented in a later release of SDML, providing access to databases containing knowledge about things known to be false (see footnote 2 on page 3).
7. The primitive **notKnown** cannot currently be used in consequents. If permitted, it could be used to check that something has not been deduced to be true, yielding an error if it has.

- In the real world, entities often have private information that they do not wish to reveal to everybody else. Private databases are ideal for such information.
- A large modelling task can often be simplified by dividing it into more manageable chunks. A single rulebase can be harder to find errors in and modify.
- Modularity can also aid efficiency. Data can be located more quickly and fewer rules need to be considered for firing at once.

Additionally, this approach enables rules to write other rules, without resorting to imperative techniques. In SDML, agents have *meta-level* rulebases and *object-level* rulebases. Typically, meta-level rules are used for learning and generating strategies, whereas object-level rules utilise such strategies to make day-to-day decisions. For example, a meta-level rule may state that a particular strategy is used (represented by a consequent stating that a particular rule is in the object-level rulebase) in a particular situation (if certain antecedents are true). Conversely, meta-level rules can make decisions based on current or previous strategies. Since antecedents and consequents are represented as clauses, rules on rulebases can be manipulated as easily as clauses on databases.

It should be noted that all permanent databases in a model, whether public or private, have subdatabases corresponding to particular time periods. Similarly, permanent object-level rulebases have *subrulebases*, enabling meta-level rules to generate different object-level rules at different times.

The modularity of SDML is further enhanced by its object-oriented facilities, detailed discussion of which are beyond the scope of this paper. General rules can be specified in rulebases associated with agent types. Rules defined in general types are inherited by specific types and individual agents.

# 4 Firing rules

Rulebases were described in Section 3.3. This section describes the techniques used in SDML to fire rules, in order to make deductions. It firstly describes how declarative rules can be fired using forward chaining, and then shows how this basic mechanism is speeded up. Finally, this section describes how backward chaining is performed and how it is integrated with forward chaining.

## 4.1 Forward chaining

Forward chaining rules are fired by finding the bindings of variables such that the antecedents are true, and deducing that the consequents are true when the bindings are applied. The same rule may fire many times with different bindings; a rule together with a bindings dictionary is known as an *instance* of the rule. Firing one instance may cause the antecedents of another instance to become true, and forward chaining continues until there are no more deductions to be made, i.e. until every instance with true antecedents also has true consequents.

The bindings such that the antecedents are true are found by *evaluating* the antecedents clause. Clauses with primitive keywords are evaluated by executing code in the implementation language of SDML (currently Smalltalk); other clauses are evaluated by retrieving them from databases. How to evaluate a clause (whether to use a primitive or retrieve from a private or public database) is determined by a clause definition group. Evaluating any clause determines the conditions under which it is true, i.e. the bindings for variables such that the clause is true. The clause may succeed many times, with different combinations of bindings, until all solutions are found.

Rules often have several antecedents which must all be true, with consistent bindings for their variables, for the rule to fire. They are represented as subclauses of a clause with primitive keyword **and**. If the first subclause succeeds, the second subclause is evaluated with those bindings applied, the resulting bindings are applied to the third subclause, and so on. When the entire antecedents clause succeeds, the rule is fired. After a rule fires or a subclause fails, SDML backtracks to search for other bindings for clauses that have already succeeded.

Whenever the antecedents succeed, the bindings are applied to the consequents, which are then *performed* in order to fire this instance of the rule. Performing a clause notes that it is true, which usually entails asserting it to a database. If there are several consequents, they are also represented as subclauses of a clause with keyword **and**; this clause is performed by performing its subclauses. The keyword **and** is one of a small number of primitives that are permitted in consequents; others are used to assert clauses to specific databases (or rules to rulebases). For example, a clause with keyword **permanent** specifies that its subclause is permanently true from the current time to the end of the simulation, so this clause is performed by performing the subclause on a permanent database.

It would be possible to move on to a different rule as soon as a rule has fired for the first time. However, it is usually faster to backtrack to find further instances than to search again from scratch. It should be noted that this approach could not so suitable for imperative rules because firing one instance could cause the antecedents of another instance of the same rule to become false. Further checks would be required, and the overall results of the rulebase could be affected.

Imperative rule-based systems often allow the user to specify a *conflict resolution strategy*. This strategy is used when there is a choice of rules for which the antecedents are true, in order to decide which one to fire. Such a strategy can introduce large overheads. Firstly, many instances of rules may be found with true antecedents, but then discarded if they are not chosen. Secondly, comparing the instances in order to decide which to fire may take a considerable amount of time. To avoid these overheads, some imperative systems use the order in which rules appear in a rulebase and the order in which clauses are asserted to a database to resolve such conflicts. Needless to say, the latter approach is very *ad hoc*.

Such a trade-off between flexibility and speed is not an issue when rules are strictly declarative. The order in which instances fire can be solely determined by efficiency considerations. This gives SDML a lot of scope for optimisation, as explained in the next section.

There is, however, one aspect in which SDML modellers do have control over ordering when firing rules. Antecedents are evaluated in the order in which they appear in a rule, since the most efficient ordering would be hard to determine automatically (although some work has been done in this area, see [11]). Due to the declarative semantics of SDML, changing the order of antecedents cannot yield different results. Some primitives cannot be evaluated when certain arguments are uninstantiated because they have an infinite number of solutions (e.g. **sum ?a ?b 4**). Such antecedents have to be delayed until enough variables have been instantiated for them to be evaluated.[8]

8. An automatic reordering facility is planned for a later release. At present, such primitives instead generate errors. Therefore, the order of antecedents does not matter as long as they can be evaluated without error.

## 4.2 Compilation

In SDML, rules can either be interpreted or compiled. Compiled rules usually run much faster than interpreted rules, but interpretation is useful for debugging. Of course, the time spent during compilation may sometimes outweigh the speed-up, particularly for rules that are only fired once (such as those generated by meta-level rules). This problem is reduced, however, by caching the code generated by compilation; the same code can be used for the same rule in a different rulebase.

The compiler utilises type information supplied by clause definitions in order to optimise common situations. For example, database retrieval can often be compiled into simple operations to fetch specified clauses or to check whether particular clauses exist, avoiding the relatively time consuming unification process. Some primitives, such as **sum**, can be compiled into particularly efficient code.

Checks must be made to ensure that a clause adheres to the syntax specified in the clause definition group before asserting it to a database. Such checks can usually be fully performed at compile time, but code is generated to perform such checks during rule firing when necessary.

## 4.3 Dependencies and partitioning

Rulebases can be analysed in order to find relationships between rules, which can then be used to increase the speed of firing them. Consider two forward chaining rules: $R_1$ and $R_2$. If firing $R_1$ may assert a clause to a database that can be retrieved by $R_2$, then $R_2$ is a *dependent* of $R_1$ and conversely $R_1$ is a *determiner* of $R_2$. This relationship can be discovered by searching for a clause in the consequents of $R_1$ which unifies with a clause in the antecedents of $R_2$. Only clauses representing predicates about the current time are considered by this search; knowledge in antecedents about previous time periods is ignored because it cannot be affected by any rules at the current time. If there are any matching clauses, then firing $R_1$ may cause $R_2$ to fire; otherwise $R_1$ cannot directly affect $R_2$.

A *dependency graph* can be constructed by finding all the dependencies in a rulebase. This graph indicates which rules may directly or indirectly affect each other. Rules are arranged in *partitions*, so that all rules in the same partition are directly or indirectly dependent upon each other (unless the partition contains a single rule, which need not be a dependent of itself). The partitions are partially ordered so that no rule in an earlier partition is dependent upon a rule in a later partition. An example dependency graph is illustrated in figure 1, with arrows from determiners to dependents and each partition shaded.

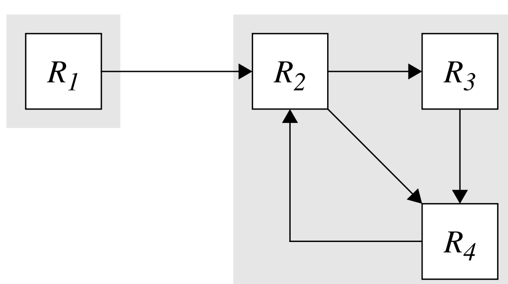


**Figure 1: An example dependency graph.**

Partitioning can be utilised to improve the efficiency of forward chaining. The rules are fired in each partition, continuing until no more instances of rules in the partition can be fired (until every instance with true antecedents also has true consequents), before moving on to the next partition. In the rulebase of figure 1, rule $R_1$ is fired before the rules in the other partition. Since no rule in a later partition can affect the antecedents of a rule in an earlier partition, there cannot be any further instances to fire after all the partitions have been dealt with. Partitioning increases speed by reducing the number of rules to be considered for firing at one time, reducing the likelihood that rules will be considered too early (before the antecedents have been deduced to be true), and therefore reducing the number of attempts to fire the same rule. It is obviously most effective in rulebases with many small partitions.

Dependencies can also be used to optimise the firing of rules within partitions, to try to minimise the number of attempts to fire each rule. If the rules within a partition are ordered, then the system can try each rule in that ordering in turn and repeat until no more rules can fire. An algorithm is therefore required to order the rules efficiently, to try to minimise the number of iterations. If a rule occurs earlier in the sequence than its dependents, then its consequents can be used by the dependents in the same iteration; otherwise they cannot be used until the next iteration. Therefore, a good heuristic is to put dependents later in the sequence than the rules they depend on. If the dependency graph is displayed graphically, with rules shown in order from left to right, then the aim is to minimise the number of dependency links from right to left. Indeed, the same rule ordering algorithm is used by SDML to display dependency graphs, as shown in figure 2. The chosen ordering is more efficient than ($R_2$, $R_4$, $R_3$), which would take two iterations before the results of deductions made in $R_2$ can affect $R_3$, its results can affect $R_4$, and its results can affect $R_2$ again. This is all done in one iteration with the ordering shown in figure 2.

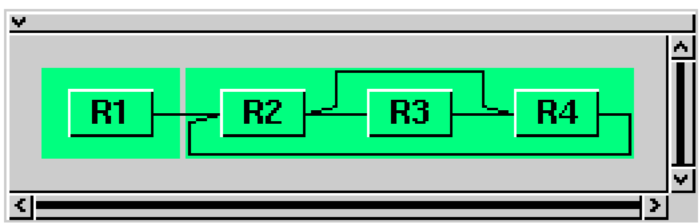


**Figure 2: The dependency graph in figure 1, as displayed by SDML.**

Dependencies can also be used to avoid trying to fire rules when the antecedents cannot have changed since the last time firing was attempted. For example, in the rulebase with the dependency graph shown in figure 3, if rule $R_3$ has not fired since it was last attempted to fire rule $R_4$, it is not necessary to check whether $R_4$ can fire again. Similarly, all the instances of rule $R_1$ can be fired in a single iteration using backtracking, whereas it is necessary to iterate if rule $R_8$ succeeds.

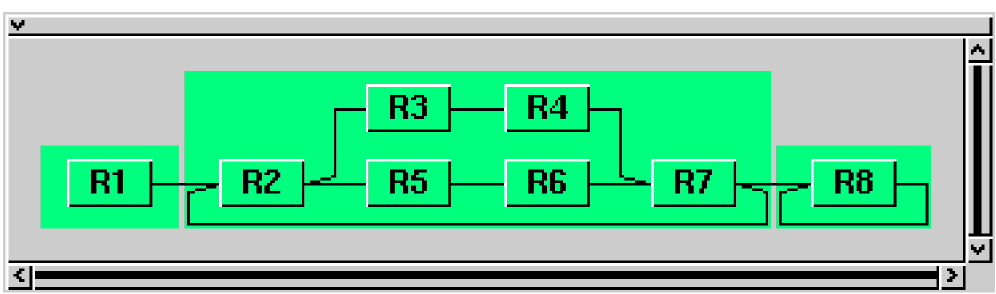


**Figure 3: The dependency graph of another rulebase, displayed by SDML.**

SDML's facility to display dependency graphs is useful for the construction and debugging of rulebases. Dependency graphs are easier to navigate than unstructured lists of rules, and they are useful for understanding the deductions that are made when rules are fired. Errors can be spotted when two rules are unexpectedly shown to be dependent, or when the modeller thinks rules should affect each other but no dependency is shown. In our experience, well structured rulebases, with relatively few dependencies and small partitions, are more likely to terminate when rules are fired and are more robust to the addition of new rules, as well as being generally more efficient. Therefore, it is worthwhile for modellers to invest some time and effort in ensuring that their rulebases are well structured.

Dependencies and partitions are not original to SDML. Similar techniques were devised by Simon [15] to order variables when solving systems of simultaneous equations. Mobal [10] can similarly generate "predicate topologies" (ignoring arguments of predicates) which are used to restrict hypotheses when learning. Dependencies are used in [2] to reduce inter-process communication by parallel logic programs and in [4] to facilitate rules asserting and retracting other rules in the same rulebase. However, to the authors' knowledge, SDML is the first rule-based system to use dependencies to determine rule-firing order.

## 4.4 Optimising cycles in rulebases

Dependencies and partitioning can be very effective at reducing the number of times rules need to be considered for firing, and thereby enhancing efficiency. Nevertheless, sometimes many iterations are necessary in order to fire all instances of rules in a partition. Partitions containing cycles can become very inefficient unless further optimisation techniques are used.

Consider the rule in figure 4, called **"Number generator"**, which states that if **?value** is a low natural number less than a particular limit and **?newValue** is **?value** plus one, then **?newValue** is also a low natural number. Assume that the clauses **lowNaturalNumber 1**and **limit 20** have been asserted to a database. Consider firing this rule, in a partition on its own, using forward chaining. On the first iteration, the antecedents are evaluated twice yielding the bindings **{?value = 1, ?n = 20, ?newValue = 1}**, and the clause **lowNaturalNumber 2** is asserted to the database. On the second iteration, evaluating the antecedents again would succeed twice, yielding the same bindings in addition to **{?value = 2, ?n = 20, ?newValue = 2}**. On the third iteration, the antecedents would succeed three times, and so on. The rule can only fire once on each iteration (and no times at all on the final iteration), since no clause can be asserted to the same database more than once, but the antecedents would succeed 209 times before the partition is completed!

| |
|---|
| Antecedents:<br>**and**<br>   **lowNaturalNumber ?value**<br>   **limit ?n**<br>   **less ?value ?n**<br>   **sum ?newValue ?value 1** |
| Consequents:<br>**lowNaturalNumber ?newValue** |

**Figure 4: A rule which triggers itself: "Number generator".**

After the first iteration, a rule instance can only fire if it depends on some knowledge that has been deduced since the rule was encountered last time. This fact can be used to avoid retrieving clauses which cannot have any effect. In rule **“Number generator”**, **lowNaturalNumber ?value** is the only clause in the antecedents that can be affected by any rule in the partition. Therefore, when this clause is evaluated, clauses asserted to the database before the rule was encountered last time (*previous* clauses) are ignored; only *recent* clauses can be retrieved. Thus, the rest of the antecedents are only evaluated once on each iteration, and they only succeed 19 times. Of course, databases must be represented in such a way as to facilitate the fast retrieval of recent clauses. In SDML, databases contain lists of clauses (with other information attached) ordered according to when they were asserted. Thus, recent clauses can be found first and it is not necessary to backtrack once a previous clause has been found. Thus, the time taken to fire this rule by SDML is proportional to the limit (plus a small constant); i.e. these techniques reduce the time taken to complete the partition from $O(n^2)$ to $O(n)$.

Sometimes, more than one clause in the antecedents of a rule may be affected by other rules in the same partition. The optimal way to fire the rule at a particular iteration depends on which rules have been fired since it was last encountered. SDML generates structures called *recency arrays*, in which each *recency* corresponds to a clause in the antecedents. Each recency is either **recent**, **previous**, **all** or **none**, and specifies whether recent and/or previous clauses should be retrieved. Different recency arrays can be used with the same compiled rule, depending on the combination of rules that have recently fired. Recency arrays are generated automatically when required, and are cached for speed.

For example, there may be another rule called **“Determine limit”** in the same partition as **“Number generator”** which asserts a clause with keyword **limit** to the database. Consider attempting to fire **“Number generator”** when both rules have fired since it was last encountered. Recency arrays for this rule have two elements corresponding to the first two clauses of the rule. When the antecedents are evaluated, either or both of the clauses retrieved should be recent. This can be ensured by evaluating the antecedents twice with the recency arrays **(recent, all)** and **(previous, recent)**. If only one of the rules has fired recently, then the antecedents are evaluated once, using the recency array **(recent, all)** or **(all, recent)** as appropriate.[9]

Information about antecedents that can retrieve clauses from databases, and the order in which they are evaluated, is gathered when rules are being compiled. This information, together with information gathered when dependencies are computed, is used to generate recency arrays for particular combinations of rules which have fired recently. This procedure is quite complicated, particularly when backward chaining rules and assumptions are taken into account, and is beyond the scope of this paper. Since compilation information is required by this mechanism, recency arrays cannot be used when rules are being interpreted.

Thus, forward chaining can be performed efficiently for rulebases containing cycles, without resorting to imperative techniques. If clauses could be retracted, then the problems addressed by recencies need not arise; a clause which triggers a rule could be retracted to prevent the rule from firing again. Recencies cannot make every declarative rulebase efficient, but this technique can be very successful in well-behaved partitions without many cycles.

## 4.5 Backward chaining

Rules can be fired using backward chaining in order to determine if something (a goal) is true. If the goal contains any variables, backward chaining finds the values of those variables

9. If there cannot be any recent clauses then obviously **previous** and **all** are equivalent, but it is slightly more efficient to use the recency **all**, and avoid checking.

such that it is true. In SDML, backward chaining can be used to evaluate a clause to determine the bindings such that it is true. This entails finding all rules with consequents which unify with this clause and evaluating the antecedents of these rules using the resulting bindings. Backward chaining is done the same way in SDML as in Prolog.

Backward chaining is sometimes much faster than forward chaining, particularly for rules which can deduce a lot of irrelevant knowledge. For some rules, backward chaining must be used because there are an infinite number of valid instances. For example, the clause **appended ?list1 ?list2 ?list3** specifies that **?list1** is the result of appending the other two lists. Forward chaining cannot be used to append all lists of arbitrary lengths. Backward chaining rules can be fired recursively to determine **?list1** if **?list2** and **?list3** are instantiated to lists, or all valid combinations of **?list2** and **?list3** if **?list1** is instantiated.

In other circumstances, forward chaining is much faster than backward chaining, particularly for rules which deduce knowledge that will be used many times and rules with multiple consequents. In SDML, it is the modeller's responsibility to decide where to use forward and where to use backward chaining (in clause definition groups). This facilitates further optimisation of dependency graphs than would be possible if SDML could decide which to use in the middle of a rulebase. The order in which forward chaining rules are fired is governed by dependencies, whereas backward chaining rules are fired when required from within other rules. It is therefore useful to construct a *forward dependency graph* by discovering the dependencies between forward chaining rules. When searching for the determiners of a particular rule, backward chaining rules which may be fired by it are also considered, taking into account any bindings of variables which can be ascertained statically. In general, this procedure yields a simpler and hence more efficient forward dependency graph than the one that would be generated by eliminating backward chaining rules from the graph of dependencies between all rules.

In some systems such as Mobal [10], deductions made using backward chaining are stored on databases, to avoid having to make the same deductions again. However, storing all such deductions could use up a lot of memory and it would be slower for deductions that are only made once. Furthermore, checking the database may only find some bindings of the variables such that a goal is true, and backward chaining would still be required to find out if there are any more. Therefore, the results of firing backward chaining rules are not stored by SDML, and modellers must use forward chaining rules when they want the results of deductions to be stored on a database.

# 5 Assumption handling

The primitive **notKnown** was introduced in Section 3.3. A clause with this keyword is true if its subclause is not known to be true. Deductions made on the basis that this clause is true may become invalid after further rules have fired. SDML uses *assumptions* to implement this primitive.

Another primitive that is implemented using assumptions is **total**, which finds the total value of a variable for all bindings such that its subclause is known to be true. If further clauses are asserted to a database, the subclause may succeed with more bindings yielding a different total. Assumptions are also used to implement primitives which find the maximum, minimum or average value of a variable, or generate a sorted list of all bindings of a variable.

The assumptions mechanism is designed to ensure that all valid deductions are made and any invalid deductions that have been made are retracted. The efficiency of the basic mechanism is improved by reducing the likelihood that invalid deductions are made in the first place.

## 5.1 Creating assumptions

An *assumable clause* is a clause that may succeed assuming that something is true. An assumable clause together with the bindings of variables in the clause before it is evaluated is known as an *assumption context*. If the evaluation succeeds, an assumption is generated corresponding to the assumption context and the result of the evaluation. After more rules have been fired, an assumable clause may no longer succeed or may yield a different result if it is evaluated again. Every clauses which is asserted to a database due to the assumable clause succeeding is tagged with an assumption, so that the clause can be retracted from the database if the assumption turns out to be false.

The rule in figure 5 contains an assumable clause with keyword **notKnown**. When this rule is fired, the assumable clause may be evaluated several times with different bindings for the variable **?value**. If the subclause succeeds when the variable is bound to particular values (because an appropriate **lowNaturalNumber** clause is found on the database) then the assumable clause fails. If the subclause fails, the assumable clause succeeds, an assumption is generated and a clause with keyword **highNaturalNumber** is asserted to the database tagged with this assumption. An assumable clause with keyword **notKnown** can only succeed or fail; no variables can be bound so at most one assumption can be generated for each assumption context.

| Antecedents:<br>**and**<br>    **interestingNaturalNumber ?value**<br>    **notKnown**<br>        **lowNaturalNumber ?value** |
|---|
| Consequents:<br>**highNaturalNumber ?value** |

**Figure 5: A rule demonstrating assumptions.**

When the assumable clause with keyword **total** in figure 6 is evaluated, the subclause is evaluated and the resulting bindings of **?value** are added up. The result is unified with **?totalValue**, and the assumable clause succeeds on the assumption that this is the correct total. Further clauses may be asserted with keyword **lowNaturalNumber**, causing a different total and corresponding assumption to be generated when the rule is fired again. Thus there may be multiple assumptions corresponding to the same assumption context, but only one of them can ultimately be true.

| Antecedents:<br>**total ?totalValue ?value**<br>    **lowNaturalNumber ?value** |
|---|
| Consequents:<br>**lowNumberTotal ?newValue** |

**Figure 6: Another rule demonstrating assumptions.**

## 5.2 Assumption tags

Sometimes a single item of knowledge depends on more than one assumption, so it must be possible to represent combinations of assumptions. They are represented using *assumption tags*. The tag $a_1 \wedge a_2$ represents the conjunction of assumptions $a_1$ and $a_2$, and is true if both assumptions are true. The tag $a_1 \vee a_2$ represents the disjunction of assumptions $a_1$ and $a_2$, and is true if either assumption is true. A single assumption can also be represented using an assumption tag, as can the booleans *true* and *false*.[10]

Evaluating any clause yields an assumption tag corresponding to each bindings dictionary, which together encapsulate the conditions under which the clause is definitely true. In practice, most clauses yield the tag representing *true*. Tags are combined when complex clauses are evaluated. For example, if evaluating clause **c**$_1$ yields tag $t_1$ and evaluating **c**$_2$ yields $t_2$, then evaluating **and c**$_1$ **c**$_2$ yields the tag $t_1 \wedge t_2$ (reduced to a canonical form).

When a clause is asserted to a database as a consequent of a rule, the assumption tag $t_1$ determined by evaluating the antecedents is stored with the clause on the database. If the same clause has already been asserted to the database with tag $t_2$, the combined tag $t_1 \vee t_2$ replaces $t_2$ on the database (unless the combined tag is the same as $t_2$, in which case the assertion has no effect and dependents of the rule are not triggered). When a clause is retrieved from a database by an antecedent of a rule, the assumption tag is also retrieved and used to construct the assumption tag asserted with the rule's consequents. Thus, further deductions that can be made on the basis of assumptions are also made, but are tagged in case the assumptions turn out to be false.

Sometimes whether the subclause of an assumable clause is true may also be dependent on assumptions; i.e. evaluating the subclause may succeed with an assumption tag other than *true*. When evaluating an assumable clause with keyword **notKnown**, the disjunction of the assumption tags found by evaluating the subclause represents the conditions under which the assumable clause is false; this disjunction is known as a *conflict tag*. If the conflict tag is not *true*, then the assumable clause may be true so an assumption is generated and the conflict tag is remembered for later use.

Evaluating the subclause of an assumable clause with keyword **total** may yield many different values and corresponding assumption tags. The correct total will depend upon which assumption tags turn out to be true, so it is necessary to generate assumptions corresponding to every combination of assumption tags which may turn out to be true. If there are many different combinations then the mechanism is obviously inefficient, but this rarely occurs in practice.

The space overheads incurred by tagging every clause on a database are fairly small due to the use of a compact representation, and since tags are often *true*. Every database retrieval or assertion entails an "$\wedge$" or "$\vee$" operation, but these operations are carried out relatively quickly.[11] The overheads of assumptions are far more significant when many deductions are made on the basis of false assumptions (see Section 5.4).

---

10. The tag representing *false* is equivalent to evaluation failing, and clauses are not asserted with this tag. However, this tag is used for other purposes, such as representing no conflicting assumptions.
11. The conjunction of one or more assumptions is represented as an integer, with one bit representing each assumption. A disjunction is represented as an array of integers. This representation facilitates the use of bit manipulation operations such as "AND" and "OR", which are particularly fast when there are a fairly small number of assumptions (but Smalltalk can represent integers of an arbitrary size and manipulate them reasonably quickly).

## 5.3 Resolving assumptions

Eventually it is necessary to *resolve* the assumptions, in order to determine which of them are true and therefore which clauses need to be retracted. This can be done when all the rules in a rulebase have finished firing, since the semantics of SDML do not allow antecedents to be affected by rules firing later on (in a different rulebase at the same time period or any rulebase at a later period). When rule firing has finished, the databases contain clauses representing every deduction that could have been made according to the rulebase. Resolving assumptions consists of three steps:

1. Finding conflicts. For every assumption, a conflict tag representing all the conditions under which that assumption is false must be determined. All the conflicting assumptions can be found by evaluating the assumptions' subclauses (and they can therefore be accumulated during the process of firing rules).[12]

| | |
|---|---|
| Antecedents:<br>**notKnown**<br>**weather hot** | Antecedents:<br>**notKnown**<br>**weather cold** |
| Consequents:<br>**weather cold** | Consequents:<br>**weather hot** |

**Figure 7: Two rules with conflicting assumptions.**

2. Finding a resolution, in which each assumption is true or false. For any valid resolution, every assumption is true if and only if its conflict tag is false. A resolution can often be found very quickly, by deducing that assumptions which have the conflict tag *true* are false and vice versa, substituting in other conflict tags, and repeating. However, finding a resolution sometimes involves a search process[13], and there may be more than one valid resolution. For example, the assumptions in the rules of figure 7 contradict each other, and if no other rules affect whether the weather is hot or cold then either assumption could be correct. Since SDML is used for simulation, it is most useful to choose one resolution, but all valid resolutions could be found if required. On the other hand, sometimes there is no valid resolution, often due to self-conflicting rules such as the one shown in figure 8. If there is no other rule to deduce that the weather is hot, this conflict cannot be resolved and SDML reports an error.
3. Retracting clauses relying on false assumptions. This entails searching databases for clauses with tags other than *true*[14], and retracting them or replacing the tags by *true* depending on the resolution of the assumptions.

After these three steps have taken place, SDML discards the assumptions and moves on

12. In the current implementation, all the results and tags of evaluating subclauses are discovered during rule firing. However, for assumptions with keywords such as **total**, combining this information to yield conflict tags is relatively time consuming, and therefore this process is delayed until after the rules have fired so that it is only done once.
13. Searching is potentially slow, so other techniques are used in the resolution algorithm. If an assumption is contradicted by itself, then it must also be contradicted by another assumption if a resolution exists. Thus, inferences can be made about other assumptions used in its conflict tag, inferences can be made about these assumptions' conflicts, and so on. The resolution algorithm also discovers completely independent sets of assumptions, and searches each set separately. Due to such techniques, assumption resolution has been relatively fast so far.
14. The search is limited to databases that have had clauses asserted with such tags, for efficiency.

Antecedents:
**notKnown**
**weather hot**

Consequents:
**weather hot**

**Figure 8: A rule with a self-conflicting assumption.**

to the next rulebase.

## 5.4 Optimisations

The basic assumptions mechanism described above suffers from some efficiency problems. In particular, a lot of time and memory can be spent making deductions on the basis of false assumptions. Techniques to improve the efficiency of assumptions are described in this section.

### 5.4.1 Partitioning

Section 4.3 described how dependencies between rules can be used to partition a rulebase, so that rules in later partitions cannot affect rules in earlier ones. Partitioning yields two advantages with respect to assumptions:

- Assumptions can be resolved at the end of each partition, rather than at the end of the entire rulebase. Thus, any clauses that were asserted on the basis of assumptions which have been resolved to be false are retracted before moving on to the next partition. Therefore, the effect of making a wrong assumption is reduced.
- If an assumable clause cannot be affected by any rule in the same partition (because its subclause is not dependent on any rule in the partition) then it is not necessary to generate an assumption. If the clause is true when it is evaluated the first time, it will remain true. In well structured rulebases, assumptions are only required for a minority of assumable clauses. Indeed, many partitions do not require assumptions to be used at all.

It is possible for one partition to have several possible resolutions, and for some of these resolutions to result in unresolvable conflicts in later partitions. When such a contradiction is discovered, SDML backtracks to a previous partition that may have caused the contradiction and tries to resolve that partition's assumptions differently. This could undermine the advantages of partitioning in certain rulebases, but has not proved a problem in practice.

### 5.4.2 Extending dependency graphs

It is useful to include assumable clauses as well as rules in dependency graphs. An assumable clause is a dependent of a rule if the rule may assert a clause that can be retrieved by evaluating the assumable clause. An assumable clause's dependents are the rules or other assumable clauses enclosing it.

When an assumable clause is encountered within a rule in a later partition, it is not necessary to create any assumptions (or assumption contexts) for the reasons described in the previous section. They are only required in partitions containing cycles. An assumable clause may have various assumption contexts, corresponding to different bindings of variables used within it. Each assumption context caches the results of evaluating the subclause, when it is encountered for the first time within an enclosing rule or assumable clause. If the same context is encountered again, the cached results can be used in order to avoid evaluating the subclause again.

If one of the determiners of an assumable clause fires, the results cached in that clause's contexts are quickly updated using recencies as described in Section 4.4. This may cause the overall results of some assumption contexts to change, creating new assumptions, and affecting the dependents of the assumable clause.[15] When these dependents attempt to fire, overall changes to these assumption contexts can also be propagated using recencies.

Including assumable clauses in dependency graphs reduces the number of times that an assumable clause or enclosing rule attempts to fire, and using recencies increases the speed of evaluation when it is required.

### 5.4.3 Ordering within partitions

The order in which rules attempt to fire can affect the number of assumptions that are created. Whereas it sometimes takes several cycles around rules in a partition before an assumption is contradicted, assumptions are often contradicted the first time a rule fires. Reordering the rules in the partition may prevent the assumption from being made in the first place. A good heuristic is to order rules and assumable clauses within a partition so that rules which contain assumable clauses are placed after rules which do not.

This heuristic can be combined with the heuristic described in Section 4.3 that dependents should be placed after rules they depend on, to try to find an optimal ordering.[16] It may be possible to achieve further efficiency gains by detecting time being wasted and reordering dynamically.

### 5.4.4 Invalid combinations of assumptions

Some combinations of assumptions cannot be true. In particular, no assumption context can have more than one true assumption, and an assumption cannot be true if its conflict tag is. Assumption tags containing such invalid combinations of assumptions can be simplified or possibilities filtered out altogether. If such filtering is carried out too often, more time can be wasted than saved, but it is worthwhile before performing the consequents of every rule. This can prevent many false deductions from being made.

### 5.4.5 Validities

In well-structured rulebases, most assumptions that are made are true. It is therefore worthwhile to assume that they are true until there is evidence to the contrary. This is implemented by giving every assumption a *validity*, which is one of the following: "assumed true", "assumed false", "known true", "known false" or "unknown". Assumptions are initially assumed true if they have no conflicting assumptions. If one assumption is assumed true, and another assumption relies on it being false, then the latter assumption is assumed false (if it has already been created; otherwise it is not created in the first place).[17] If an assumption's conflicting assumptions are known to be true, then the assumption is known to be false. Assump-

---

15. A further optimisation is possible by only causing the enclosers (rules and/or assumable clauses) in which the contexts were encountered to attempt to fire. If there are any other dependents, they cannot be affected.
16. The current implementation does this by ordering to minimise backward dependencies, then adjusting the start of the sequence so that most dependents of assumable clauses are at the end. The partition takes at most one extra iteration to complete. This algorithm seems to work well.
17. Infinite cycles must be avoided in which assumptions repeatedly change validity from true to false and back again. To break such cycles, if an assumption's conflicting assumptions change validity from true to false, the assumption's validity becomes unknown.

tions that are assumed or known to be false can be filtered out. This is done whenever clauses are retrieved from databases so that assumptions which are no longer assumed true are not propagated further.

The rules in a partition cease firing when all the consequences of assumptions with true or unknown validities have been deduced. Resolution is only used to determine which of the assumptions with unknown validities are true or false. If there is no valid resolution, then it may be because wrong validities have been assumed. The assumed validities are changed to unknown and rules in the partition are fired again, enabling new deductions to be made which allow the assumptions to be resolved.

When validities are used, the consequences of some assumptions are not deduced. If an assumption that was assumed false gets a true or unknown validity again, then the implementation must ensure that those further deductions are made by refiring rules. Fortunately, this is rarely necessary in well-structured rulebases.

## 5.5 Random and arbitrary decisions

It is clearly useful for rules to be able to specify that a random choice is to be made. In SDML, such facilities must be declarative and logically consistent. A primitive **randomNumber** is provided, which behaves as if it retrieves from an infinite database associated with each agent and time period of the simulation, containing all terms (except those containing variables) and corresponding random numbers in the range zero to one. It is implemented by generating a random number the first time a particular term is specified, and returning the same number if the same term is used again. Thus, evaluating the same clause from within the same rulebase at the same time period yields the same bindings, and the facility works even in a partition containing cycles. Every time a different random number is required, a different term, specifying what the number corresponds to, must be specified.

Sometimes, it is required to choose one of several possibilities, any of which could be true. This can be done by generating a sorted list of the possibilities (using an assumable clause), and indexing this list by a random number generated using the **randomNumber** primitive and converted to an integer in the appropriate range. However, some choices may cause conflicts between assumptions for which there is no valid resolution, and this method does not allow other choices to be tried.[18]

An alternative approach is to use backward chaining rules to specify the possible choices and a primitive (called **randomChoice**) to randomly select one of the choices. The primitive can generate an assumption corresponding to each possibility, and randomly resolve the assumptions so that only one of them is true. Validities can be used to optimise this primitive by randomly choosing one to assume true and backtracking if this leads to contradictions. The primitive must be defined in such a way that if the same assumption context is encountered again, the same choice is made.

The first rule in figure 9 specifies that all low natural numbers are possible choices, and the second rule specifies that one of the possible choices, randomly selected, is the natural choice. An additional term (in this case the symbol **natural**) is used as an argument of the **possibleChoice** and **randomChoice** clauses to identify the choice, so that different choices can be specified in the same rulebase (as with **randomNumber**).

When a fair random choice is required, it is necessary to fire all applicable backward

18. It would not be sensible to allow the **randomNumber** primitive to backtrack to try different random numbers, because there are (theoretically) infinity numbers between zero and one.

| Antecedents: **lowNaturalNumber ?value** |
|---|
| Consequents: **possibleChoice ?value natural** |

| Antecedents: **randomChoice ?value natural** |
|---|
| Consequents: **naturalChoice ?value** |

**Figure 9: Rules to define possible choices and make a random choice.**

chaining rules completely to determine all possible choices, and there must be a chance for the random choice to be changed if new possibilities are discovered after the assumption context was first encountered. Sometimes an arbitrary choice, which need not be selected randomly, is required. If the first possibility discovered is adequate, then a lot of time can be avoided investigating other possibilities. Unlike with the Prolog "cut", however, backtracking can be used to make other choices if required. In SDML, arbitrary choices can be made by simply using the **arbitraryChoice** primitive instead of **randomChoice**.

It may be noted that arbitrary choices can be made in SDML, without using **arbitraryChoice**, by attempting to fire the rules in figure 7 for example. The rule in figure 10 specifies that a low natural number is the natural choice unless there is a different natural choice. If validities are used, an assumption corresponding to the first natural number retrieved is assumed true, which prevents other assumptions from being made. The **arbitraryChoice** primitive is merely a more efficient mechanism to make such decisions.

```
Antecedents:
and
   lowNaturalNumber ?choice\
   notKnown
      and
         naturalChoice ?value|
         notKnown
            equal ?value ?choice
---
Consequents:
naturalChoice ?choice
```

**Figure 10: Another way of making an arbitrary choice.**

Arbitrary choices can be affected by criteria such as rule ordering and the order in which data is retrieved from databases. It could be argued that such facilities should not be provided in a strictly declarative language. However, arbitrary decisions only affect the order in which resolutions are found. If backtracking is used, then the same final states of the databases will be obtained irrespective of ordering (if rule firing terminates).[19] For some problems, arbitrary choices enable one solution to be found much faster, and one solution is often sufficient.

19. This property obviously does not apply when the **randomNumber** primitive is used. See a forthcoming paper for the property's proof.

# 6 Performance

This section describes some benchmarks which can be used to compare the efficiency of different rule-based systems. Measurements of speed and memory usage, for SDML and an imperative system, are shown. The imperative system that SDML is compared with is an implementation of OPS5 which uses the standard RETE [6] match algorithm to decide which rules can fire. These measurements show that declarative rules are viable for efficient forward chaining.

## 6.1 Benchmarks

Four benchmarks are presented here. The first two are designed to measure the raw speed of firing rules using forward chaining.

- *cycle1000*. Firing 1000 instances of the same rule in a cycle. This is done by firing the rule in figure 11 after the clause **forwardKeyword 0** has been asserted to the database.

| Antecedents: |
|---|
| **and**<br>**forwardKeyword ?a**<br>**less ?a 1000**<br>**sum ?b ?a 1** |
| Consequents:<br>**forwardKeyword ?b** |

**Figure 11: Rule for benchmark *cycle1000* (similar to “Number generator” in figure 4).**

- *nocycle1000*. Firing 1000 instances of a rule, without a cycle. This is done by firing the rule in figure 12 after 1000 clauses with the keyword **forwardKeyword** have been asserted.

| Antecedents: |
|---|
| **forwardKeyword ?a** |
| Consequents:<br>**otherForwardKeyword ?a** |

**Figure 12: Rule for benchmark no*cycle1000*.**

However, the rules used in the above benchmarks are artificial and much simpler than those used to solve real problems. Therefore, two further benchmarks which solve specific problems are also used.

- *tourney*. Scheduling a bridge tournament of 16 players, so that every player partners every other player once, and plays against every other player twice.
- *rubik*. Scrambles a rubik’s cube and solves it, by rotating faces until every block on the same face is the same colour.[20]

The latter benchmarks were previously implemented in OPS5[21], and have been used to

20. An average is taken over 10 different sequences of scrambling operations. These sequences are generated randomly, and the same sequences used for each implementation of the benchmark. This is done to avoid bias, which may arise due to some positions being solved in fewer moves.

compare the efficiency of different OPS5 implementations (in [1] for example). OPS5 is an imperative rule-based language, and most of the rules used in the OPS5 benchmarks retract or modify information already on the database, often immediately making the antecedents false. Nevertheless, the SDML implementations use the same algorithms and there is a one-to-one correspondence between most of the rules in both languages.

The SDML versions of the benchmarks use some facilities that are not present in OPS5, such as **arbitraryChoice** (to arbitrarily choose who plays together or which block to move into position next) and lists (particularly useful for the efficient representation and manipulation of rubik's cubes). Subdatabases can be used for different stages in the problems' solutions, since backtracking is not used (in either OPS5 or SDML) after scheduling any bridge hands or rotating the cube. Backward chaining, which is not available in OPS5, is used for some of the SDML rules.[22]

## 6.2 Measurements

The benchmarks described above have been performed on a Sun SPARCstation 5, using both SDML and a version of OPS5 called CParaOPS5[23]. The SDML rules were compiled into Objectworks\Smalltalk®, whereas the OPS5 rules were compiled into C. Measurements for each of the benchmarks are shown in table 2. The timings exclude compilation time and the memory usages exclude any memory occupied by the compiled code. A rulebase in the SDML implementation of *rubik* performs a considerable amount of initialization that can be reused if multiple cubes are solved[24], so measurements excluding this rulebase are also shown. SDML's facility to dispose of old databases was not used in these benchmarks.

| | SDML | | | CParaOPS5 | | |
|---|---|---|---|---|---|---|
| | Memory (K) | Time (s) | Rate (rules / s) | Memory (K) | Time (s) | Rate (rules / s) |
| *cycle1000* | 119 | 1.78 | 560 | 3144 | 0.92 | 1090 |
| *nocycle1000* | 95 | 0.67 | 1500 | 2547 | 3.43 | 290 |
| *tourney* | 28 | 1.28 | | 3780 | 3.79 | |
| *rubik*<br>(without set-up) | 202<br>153 | 7.20<br>5.57 | | 1245 | 8.95 | |

**Table 1: Measurements for firing compiled rules**

SDML performs many optimisations at compile time, including computing dependen-

21. *tourney* was implemented in OPS5 by Bill Barabash, and *rubik* by James Allen. The OPS5 implementation of *tourney* is inefficient because it generates a lot of possibilities that are immediately thrown away; this inefficiency is easily overcome in SDML using **arbitraryChoice**. There are no such problems with *rubik*, so the efficiency comparisons are fairer.
22. None of the backward chaining rules are recursive, so the benchmarks could have been implemented without using backward chaining simply by incorporating their antecedents in the enclosing rules. Therefore, the benchmarks are useful for evaluating the efficiency of forward chaining.
23. CParaOPS5 can run in parallel, but is run in single processor mode for comparison with SDML.
24. A backward chaining rule to rotate a cube face is fired for a general cube (containing uninstantiated variables). This generates clauses corresponding to each possible rotation which are stored on a database; unification can then be used to rotate specific cubes.

cies. Incremental compilation, and caching of compiled rules, are used to limit the amount of recompilation that has to be done. Nevertheless, it is important that compilation can also be carried out reasonably efficiently, particularly when meta-level rules are used to generate new object-level rules. For each benchmark, the time taken to compile all the rulebases and the size of the compiled code are shown in table 2. It should be noted that the memory measurements for SDML exclude space occupied by the primitives and other code necessary to fire the rules. In the current release, the entire SDML language and environment (including most of Smalltalk) is required to fire rules, adding about 2.5MB to the memory usage. However, the measurements show that the additional space occupied by the compiled rules is reasonably low.

| | SDML | | CParaOPS5 | |
|---|---|---|---|---|
| | Memory (K) | Time (s) | Memory (K) | Time (s) |
| *cycle1000* | 1.8 | 0.42 | 128 | 2.5 |
| *nocycle1000* | 4.2 | 0.54 | 128 | 2.6 |
| *tourney* | 35 | 7.4 | 176 | 7.4 |
| *rubik* | 162 | 40.6 | 512 | 46.0 |

**Table 2: Measurements for compiling rules**

The measurements show that SDML is reasonably efficient for certain kinds of problems, and therefore that forward chaining is viable for declarative rules. They are particularly good considering that SDML is written in and compiles into Smalltalk, whereas OPS5 uses C.[25] However, we do not claim that declarative rules are more efficient in general. It would obviously require more benchmarks and more comparisons with other systems to justify such an assertion.

# 7 Comparisons with other approaches

## 7.1 Logic

Declarative rules are closely related to logic. Indeed, simple rules in SDML are equivalent to logical implications. However, not all deductions that can be logically deduced can be deduced by firing rules. For example, if the consequents of a rule are false, then the antecedents could also be deduced to be false, but this deduction cannot be made by firing the rule.[26] Also some situations that lead to contradictions in SDML may be solved using logic. For example, the rule in figure 8 logically entails that the weather is hot, but this cannot be deduced by firing this rule except by assuming that the weather is not hot! In a logical sense, any set of deductions that is not contradicted by rules and base facts is a possible solution, but SDML only finds solutions that can be derived from the base facts via a sequence of valid rules. Thus, rules are one way of constraining the search space.

25. Some estimates of the speed difference between implementations of C and Smalltalk are as high as a factor of 10, but a factor of 2–5 is probably more accurate for modern implementations. It is planned to use C for at least some critical sections of code in future releases of SDML.
26. Of course, rules could also be fired in this manner, but this is not done in SDML for efficiency reasons.

An important property of SDML is that any solution that is found is logically consistent, and could have been deduced using logic. It is also important that all solutions for which there is a valid sequence of valid rules can be deduced using SDML. However, formal proofs of these properties are the subject of ongoing research.

SDML has similarities to different kinds of logic. Intuitionistic logic [16], in which predicates are true if they have a constructive proof, is similar because a sequence of rules can be considered a proof. Since SDML has facilities to deal with time, it is similar to temporal logic [12]. SDML's agents can represent and manipulate beliefs about the world and other agents, resembling belief logic [7, 8]. SDML is also similar to default logic [14], because predicates can be assumed until there is evidence that they are not true. The exact logical status of SDML is also part of work in progress.

## 7.2 Truth maintenance systems

Assumption resolution is a form of truth maintenance. A traditional truth maintenance system (TMS) [5] represents every fact or deduction as a node in a network. At any point in time, each node is assumed to be in (true) or out (false). Each node has a justification representing nodes that must be in and nodes that must be out for that node to be in. Making a new deduction can cause the status of a node to change from in to out, changing the status of other nodes, and so on. However, propagating such changes throughout a network, and refiring rules to take them into account, can be very inefficient.

An assumption-based truth maintenance system (ATMS) [3] is designed to overcome some of these inefficiencies. An ATMS follows through the consequences of every assumption being true, and is therefore similar to the basic assumptions mechanism in SDML. Such systems are particularly efficient when many solutions need to be found or when it is difficult to find a solution due to multiple conflicting assumptions. However, when using SDML for simulation tasks, one solution is normally sufficient and complex conflicting relationships between assumptions rarely arise.

SDML uses a different approach to overcome the inefficiency problems of truth maintenance. Assumptions are only generated where necessary, corresponding to fewer nodes in a TMS network. Partitioning is used to divide the truth maintenance task into subtasks. The observation that most assumptions end up being true is used to limit the search space. These optimisations, described in Section 5.4, usually entail that assumption resolution is trivial.

## 7.3 Other declarative rule-based systems

Most mainly declarative rule-based systems also provide some imperative facilities. In Prolog, rules are fired in the order they are stored, and the "cut" facility can be used to prevent later rules or instances of the same rule from being fired. LDL [11] overcomes these imperative features of Prolog by automatically ordering rules, and using a declarative form of cut called "choice" which is similar to SDML's **arbitraryChoice** primitive. However, even LDL resorts to imperative techniques for asserting information to and retracting it from databases. The semantics of the Prolog "not" are often criticised; clauses with this keyword are interpreted as true if their subclauses fail to find a solution according to the current database state. Both Prolog and LDL use backward chaining. Forward chaining can be performed in these languages, but only by using imperative database operations.

Mobal [10] is a declarative rule-based system, designed for knowledge acquisition and machine learning, which provides both forward and backward chaining. Mobal has similar facilities to SDML's assumable clauses, known as "autoepistemic operators". For example, SDML's **notKnown** is equivalent to the disjunction of “unknown” and “not” in Mobal. Unfortunately, Mobal is inefficient on large databases; the benchmarks *cycle1000* and *nocycle1000* both take over 2 minutes. Consequently, the user can limit the amount of forward and backward chaining that is carried out, affecting the results of autoepistemic operators. These operators themselves are less efficient than in SDML; Mobal sometimes unnecessarily propagates the results of these operators, only to retract them again when their results change due to another fact being deduced.

# 8 Conclusions

Section 2 listed some motivations for using declarative rules. Here we consider whether SDML fulfils these expectations.

SDML forces all the criteria on which deductions are to made to be explicitly stated in the rules. Any deductions that rely on the absence of knowledge have to be expressed using assumable clauses. Assumptions that are made when rules are fired can be examined by the modeller to determine if they are reasonable.

Conflicts between rules are detected automatically by SDML, and a contradictory rulebase for which there is no logically consistent solution always yields an error. Of course, if there are errors in the rules, wrong deductions can be made. However, tagging each clause with the rule that asserted it (and the time period) eases the task of locating the source of the error. A debugger, with single stepping facilities, can be used to refire the rule, and the rule must yield the same erroneous clause even if further rules have fired in the meantime.

In SDML, rules continue firing until there are no more deductions that can be made. If a clause is not present on the database when firing finishes, then either it is false, a rule that would deduce it is missing, or there is an error in a rule.

Rule ordering is less critical on the results of firing rules in SDML than in imperative rule-based systems. If there is a solution, one will always be found. Ordering only makes any difference when there are multiple solutions, and this does not occur often in practice. SDML can search for all solutions if required, and these solutions will be independent of rule ordering.

Therefore, using declarative rules enables modellers to have more confidence that their models are correct and that they yield correct results. Therefore, strictly declarative rules are advantageous as long as they are easy to use and implemented efficiently.

Williams and Bainbridge [17] listed five supposed disadvantages of rule-based systems: (1) control knowledge is often not clear; (2) lack of structure makes management of large knowledge bases difficult; (3) not all human problem solving methods are easily represented; (4) matching (finding instances of rules with true antecedents) is inherently inefficient; (5) it is all but impossible to determine properties of behaviour except by testing, making them unsuitable for safety-critical applications. This paper shows that using declarative rules and multi-agent models alleviates problems (1), (2) and (5) — and can even yield improved efficiency.

This paper describes several techniques to facilitate efficient forward chaining of declarative rules. The benchmarks show that reasonable efficiency is obtainable. However, rulebases have to be well-structured to take advantage of these techniques. Such rulebases are more easily created for multi-agent models than for models with only one rulebase. It is not always easy to construct rules declaratively, and further work is planned to make SDML easier to use.

# Acknowledgements

Thanks to Bruce Edmonds at the Centre for Policy Modelling for his comments, and to everybody else who has used SDML. Thanks to Anurag Acharya for supplying the OPS5 benchmarks and information about them.

# References

[1] A. Acharya and M. Tambe. Production systems on message passing computers: Simulation results and analysis. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 246–256, August 1989.

[2] T. Castagnetti and P. Ciancarini. Static analysis of a parallel logic language based on the blackboard model. *Journal of Parallel and Distributed Computing*, 13(4):412–423, Digital Equipment Corporation 1991.

[3] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[4] S. K. Debray. Flow analysis of dynamic logic programs. *The Journal of Logic Programming*, 7(2):149–176, Sept. 1989.

[5] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[6] C. L. Forgy. RETE: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[7] J. Hintikka. *Knowledge and Belie*f. Cornell University Press, Ithaca, New York, USA, 1962.

[8] K. Konolige. *A Deduction Model of Belief*. Pitman Publishers/Morgan Kaufman Publishers, San Mateo, California, USA, 1986.

[9] W. Lukaszewicz. *Non-Monotonic Reasoning: Formalization of Commonsense Reasoning*. Ellis Horwood series in artificial intelligence. Ellis Horwood Limited, 1990.

[10] K. Morik, S. Wrobel, J.-U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning*. Academic Press, 1993.

[11] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.

[12] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, Providence, USA, November 1977.

[13] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.

[14] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.

[15] H. A. Simon. Causal ordering and identifiability. In Hood and Koopmans, editors, *Studies in Econometric Method*. Cowles Foundation monograph No. 14, 1953.

[16] D. van Dalen. Intuitionistic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic, Vol. III*, chapter 4. D. Reidel, Dordrecht, Netherlands, 1983.

[17] T. Williams and B. Bainbridge. Rule based systems. In G. A. Ringland and D. A. Duce, editors, *Approaches to Knowledge Representation: An Introduction*, chapter 5, pages 101–115. Research Studies Press Ltd., Letchworth, Herts., UK, 1988.

[18] T. Winograd. Frame representations and the declarative/procedural controversy. In E. G. Bobrow and A. M. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*, pages 185–210. Academic Press, New York, USA, 1975.