CS-Note* 4

Bill J Ellis

November 15, 2001

A Guided Tour of the SPADE Proof Checker

1 Introduction

The Spark Approach is a method for producing software with a high level of integrity. This is achieved by subjecting the produced code to both static flow analysis and rigours formal proof. The formal proof of program code is reduced to a problem of proving a number of mathematical theorems, called verification conditions (VCs). These VCs are typically tackled interactively inside the SPADE Proof Checker.

It is our intention to improve the power of SPADE by adding support for new techniques in automatic theorem proving. Such techniques have already been implemented inside a system called CLAM. Thus the idea is to combine SPADE and CLAM to produce a sound starting platform, upon which further theorem proving techniques may be developed.

The integration of these systems will require a sound and flexible architecture. Thus this note has been produced for SPADE and a corresponding note for CLAM (CSnote 5) to explore possible architectures and the problems that might occur.

2 The External Perspective

The external perspective considers SPADE from the users viewpoint. Though the user should have an appreciation of the techniques being performed inside SPADE, the detailed method of their actual implementation is not required.

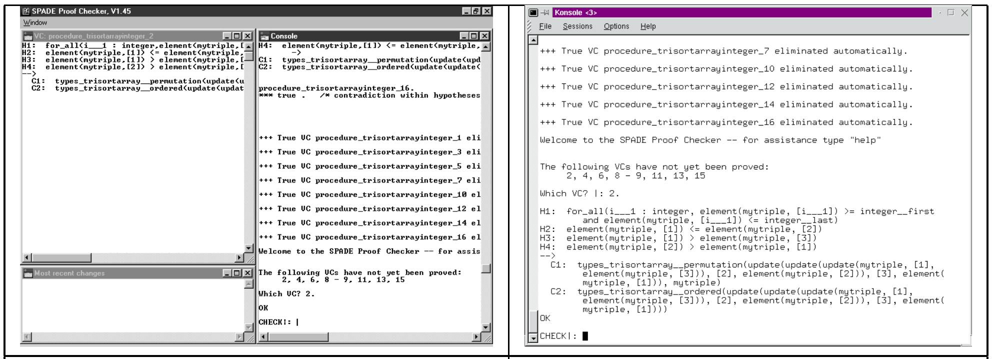
2.1 Overview

2.1.1 Interactive Theorem Prover

SPADE is classified as an interactive theorem prover. This means that the emphasis is on the user to provide guidance in the search for proofs. Although powerful theorem proving commands may still be made available, the user is always relied upon to make any key search decisions.

2.1.2 Interface

SPADE is available for two different operating systems, Windows NT and Solaris. In both cases the sole interface is a command prompt. The Windows NT version provides separate windows to display the status of the current proof. While this is a helpful feature, it does not begin to hide the command prompt design of SPADE. A screen shot of these two versions of SPADE is shown in Figure 1.



Windows NT	Solaris

Figure 1: Comparing Windows NT and Solaris interfaces

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the CLAM-SPARK project (EPSRC GR/R24081).

2.2 Black-box View

Figure 2 shows the external components that interact with SPADE. These are indicated as actors (stick-men) playing a role described in text underneath. The notes beside actors indicate the type of files used in the interaction. This also outlines the key functions of SPADE. These are described as text in use cases (ellipses).

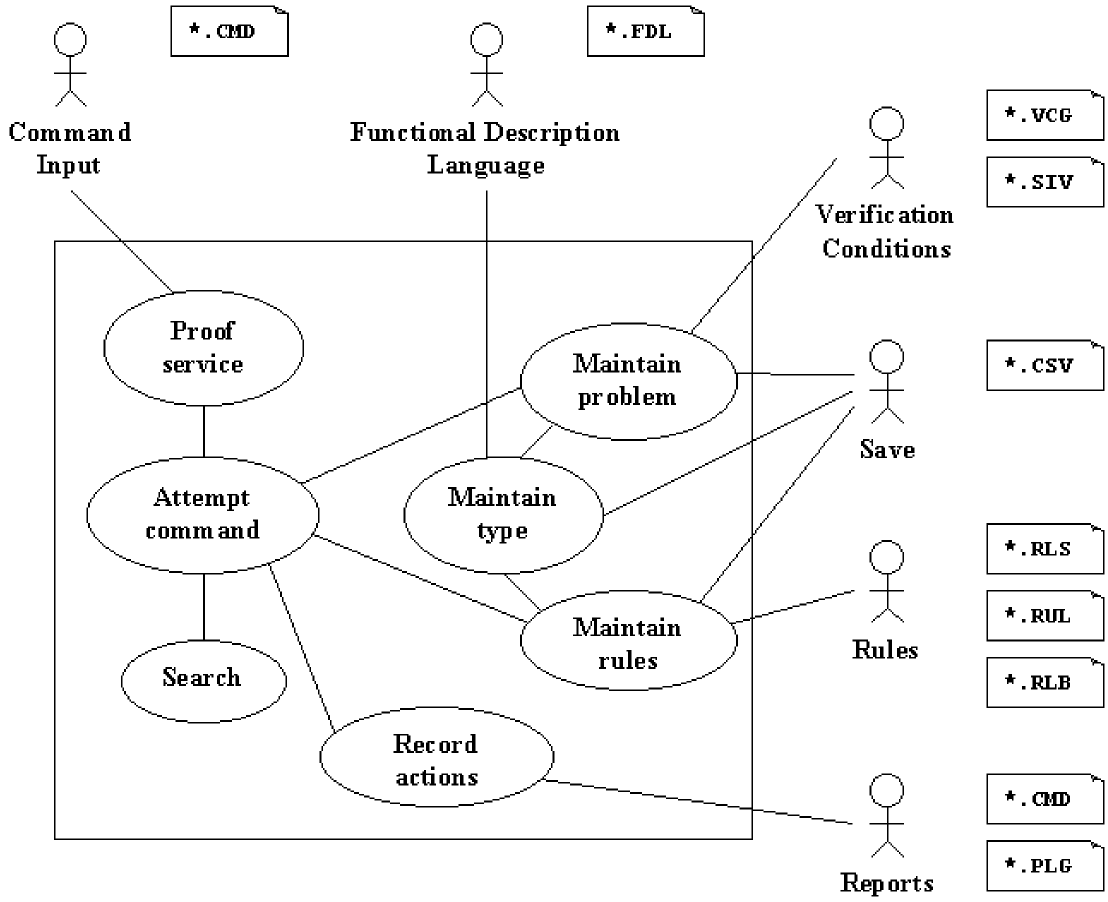


Figure 2: Black box view of SPADE

2.2.1 External components

These are the external users or systems that interact with SPADE.

- **Command input -** The system is controlled via command input. Typically the input will be from a human user, interacting with the system to guide proof attempts. However, the user may source a command log to repeat a previously issued command sequence.
  - ***.CMD -** A log of commands recorded during a previous session with SPADE. The astute user may source these logs to repeat a common sequence of commands in a proof.
- **Functional Description Language -** Every VC and rule is described syntactically using a number of identifiers. The specific details of these identifiers are described using Functional Description Language (FDL). For example, a VC may refer to a proof function. The semantic details of this proof function will be provided through corresponding FDL.
  - ***.FDL -** A number of lines of FDL, assigning semantics to the identifiers used in the syntactic VCs and rules.
- **Verification Conditions -** The VCs to be solved are generated automatically by the Spark EXAMINER from analysing the program unit. Often a number of the generated VCs will be quite trivial, and can be

automatically proved by the Spark SIMPLIFIER. Thus only the remaining VCs need be supplied to SPADE for proving.

  - ***.VCG -** The VCs generated for the program unit by the EXAMINER. Note that a corresponding *.FDL file will also be generated to elaborate on the details of the VCs.
  - ***.SIV -** The VCs generated by the EXAMINER, after being passed through the SIMPLIFIER to solve any trivial VCs.

- **Save -** At any point during a proof attempt the user can save the current state of the system. This might be used to quickly return to a previous state after exploring an unsuccessful proof path. However, it exists primarily as a courtesy to the users, allowing them to save their proof and switch off the computer.

  - ***.CSV -** A previously saved proof attempt.

- **Rules -** To prove VCs it is often necessary to manipulate the conjecture using a number of mathematical rules. A selection of commonly used rules are built-in to SPADE as default. Where these rules are not sufficient the user may create and load their own rule files.

  - ***.RLS -** Any rules that are automatically generated by a Spark program, including the built-in rules provided with SPADE. Rules are sometimes automatically generated by the the EXAMINER when producing the VCs. These generated rules capture any additional knowledge known about the VCs that do not form part of the VC itself. For example, rules may be generated to provide legal assignments for constant variables used in the VCs.
  - ***.RUL -** Rules that have been created by the user to assist in a proof.
  - ***.RLB -** These are pre-compiled rule libraries. They are created using a special program that comes with SPADE called BUILDLIB. By using compiled rules the time consuming parsing and type checks applied for normal rule files can be avoided. Further, this can be used to load a comprehensive collection of definitions, where only a small subset of these are directly referred to in the program unit.

- **Reports -** During a proof session the users actions are recorded. These can then be summarised in a report using the Proof ObliGation Summariser tool (POGS).

  - ***.CMD -** A record of every piece of text typed by the user. Note that even mis-typed commands are recorded.

  - ***.PLG -** A proof log recording every command that advances a proof, showing the changes the command made.

2.2.2 Functionality

This is a brief note on the key functions of the SPADE system.

- **Proof service -** Where supporting a user, the quality of a system is largely influenced by the interaction service it provides. Thus an important aspect of this system is accommodating the user, as well as performing the theorem proving.

- **Maintain problem -** The problem being tackled is a number of VCs that together when proved will show the correctness of a program unit.

- **Maintain rules -** The library of rules is a vital component in allowing many VCs to be solved. A collection of built-in rules is loaded as default, with additional rules being loaded by the user if required.

- **Maintain type -** The problem and rules are bound together by a common type mechanism.

- **Attempt command -** The user commands are attempted, affecting the system state as requested.

- **Search -** The commands issued by the user are at a high level. They do not describe in intricate detail the techniques being applied. This is often compounded by the user providing partial commands. For example, selecting an entire family of rules to try, where only one of the rules in this family has the effect desired. Thus in many cases an amount of searching needs to be conducted to perform the user command.

- **Record actions -** Record the users actions to provide a log sufficiently descriptive to repeat the proof attempt.

3 The Internal Perspective

The internal perspective considers SPADE from the programmers viewpoint. It is concerned with how the system is actually implemented rather than its practical use.

3.1 White-box View

Figure 3 shows the internal components of SPADE, their dependencies and general architecture. Each box describes a discrete object of the system, with the file names relating to this object shown inside brackets [1]. In many cases objects implement the commands of the system. Where this is the case the object is named after the system command and shown in inverse video. Note that the database is not considered in any detail here. A detailed discussion of internal representation is key to this note, and thus given a full discussion in section 6 on page 10.

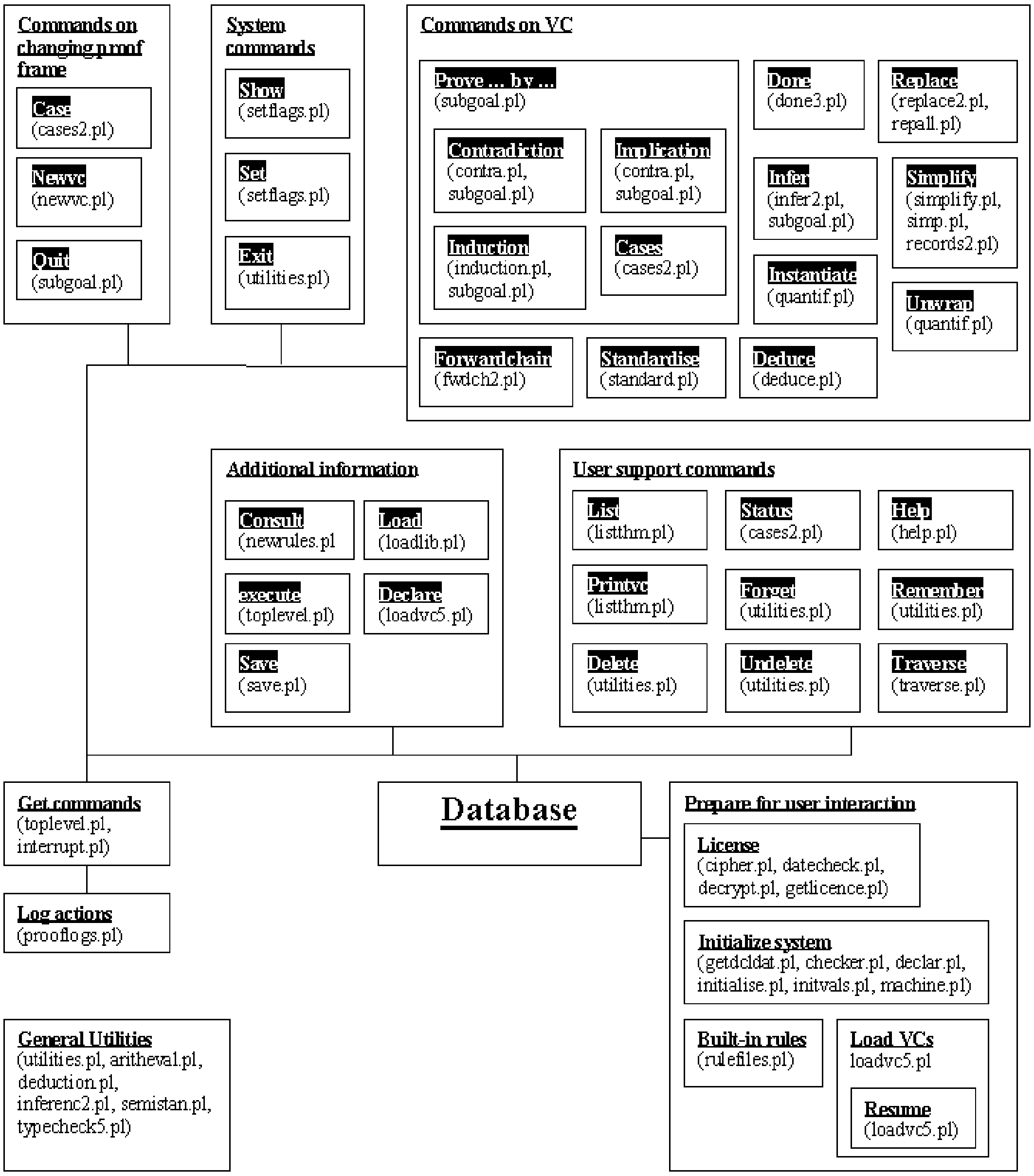


Figure 3: White box view of SPADE

[1] SPADE is not written in an object oriented language. Thus the objects here are artificially created by considering collections of related functionality.

3.1.1 Observations

- **Implemented in Prolog -** SPADE is implemented in Prolog, in particular it is currently executed on Poplog-Prolog. The command prompt used to interact with the system is in fact the Prolog interpreter.
- **The Database -** A key feature of Prolog is the ability to `assert` and `retract` predicates. This allows information to be stored that might otherwise be lost during the unwinding of unified variables in backtracking. This mechanism can be effectively used to store information required during the execution of a program. It is therefore not surprising that SPADE relies solely on this for storing all of its data [2].
- **Commands and Code -** There is a strong relationship between the commands that modify VCs and the files that implement them. Where one of these commands has many different options these options are again separated into different files. Although this relationship is not as strong for other kinds of commands, this practise of mapping code directly to the commands of the system is well observed.
- **Utilitys -** There is a number of general purpose predicates used in implementing many of the commands. Although these are often difficult to identify, they can usually be traced to a central location of similar general purpose predicates.

4 Proof Construction

Once the VCs have been loaded into SPADE, along with any number of user rule files, proofs are constructed. This process is performed with a high degree of interaction with the user. It is this behaviour that earns SPADE the classification of an interactive theorem prover.

The inclusion of a human user brings a distinct advantage to the theorem proving process. Aspects of human behaviour, such as intuition or gut feeling, are invaluable in searching for a proof, yet are very difficult to quantify. However, this is not to suggest that humans can effortlessly find the proof. There is still a need to explore possible paths and make new choices based on unfavourable results.

Supporting this very human approach to problem solving is a key influence on the way proofs are constructed in SPADE. Recognising this helps justify the SPADE approach to proof construction.

4.1 Searching for Proof In SPADE

Figure 4 illustrates how proofs are constructed in SPADE. The left hand side indicates the effects of traversing between theorem proving depths while the right hand side indicates the theorem proving state at each depth.

4.1.1 Depth in SPADE

An important aspect of constructing proofs in SPADE is that of proof depth. Going to a lower depth suggests breaking down a big problem into a smaller more manageable problem. Rising to a higher depth is possible once all of the smaller problems at a given depth have been successfully completed.

At the very top there is a depth above SPADE, where the program units exist, *Outside*. Using the EXAMINER the correctness of these program units can be described as VCs.

Loading SPADE with a number of VCs will enter at its highest level, *Top*. Proving every VC at *Top* will prove the program unit correct. At this point SPADE can exit, having successfully completed its task.

At *Top* no theorem proving is possible. The user can only select which VC to tackle. Once chosen, this VC is brought down to the highest theorem proving depth, *Proving: 0*. If proved this VC will be removed from *Top*.

At every theorem proving depth *Proving: N* “newgoaling” may be used to start a new problem at a lower depth. Note that the term “newgoaling” is used as opposed to “subgoaling”, as proving the new goal need not necessarily prove the original goal [3]. The benefit of newgoaling is to prove a new conjecture at *Proving: N+1* that when added to the current conjecture at *Proving: N* makes it easier to prove.

4.1.2 One Way Proof With Save At Depth Boundaries

At the first theorem proving depth, *Proving: 0*, there will always be a single conjecture. This is an exact copy of one of the VCs brought down from *Top*. At any other theorem proving depth, *Proving: N*, there may be any number of conjectures.

[2] The only exception is using environment variables to store paths to the built-in rules and help files.

[3] The process of newgoaling can be used to introduce new lemmas. This is a helpful facility prompted by the existence of the cut rule: $\frac{\Gamma\alpha\vdash\beta \quad \Gamma\vdash\alpha}{\Gamma\vdash\beta}$. Unusually, the same mechanism is also used to perform true subgoaling (Where the new “lemmas” are really modifications of the original conjecture).

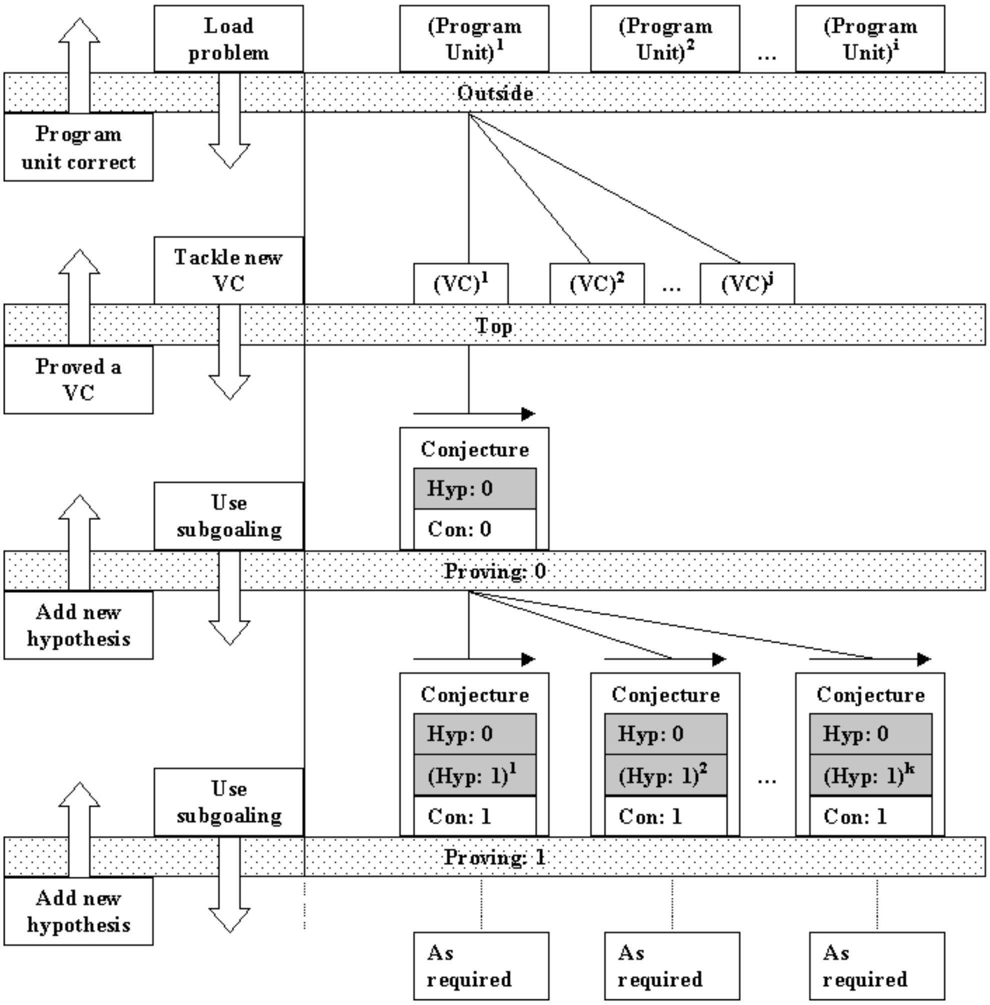


Figure 4: Proof construction in SPADE

As a golden rule there is only ever one conjecture being tackled at any given time. Thus the user tackles each VC at *Top* one at a time, and each conjecture at *Proving: N* one at a time [4].

Further, at each theorem proving depth *Proving: N*, all of a users theorem proving efforts are in one direction. After performing some theorem proving technique that modifies the current state there is no way to undo back to the previous state. This is noted on the diagram as one way arrows above each conjecture.

However, by newgoaling at *Proving: N*, the user is taken to a lower depth *Proving: N+1*. Similarly by choosing which VC to tackle at *Top*, the user is taken to a lower depth *Proving: 0*. This can be thought of as crossing a depth boundary.

At the point of crossing a depth boundary, the state of the conjecture being tackled just before entering the new depth is maintained and the state of all new conjectures just after entering the new depth is maintained. Thus it is always possible to start again on a VC by reloading from *Top*. Or start again on a newgoal conjecture created at *Proving: N+1* or to abandon the newgoal attempt and return to the conjecture at *Proving: N* just before newgoaling.

4.1.3 Tackling a VC

The VC is described as a single conjecture, containing a number of hypothesis and conclusions. Constructing a proof will formally show that the conclusions can be derived from the hypothesis.

The user issues commands to the system to gradually modify the hypothesis and conclusions in a search for proof. At any point in tackling a conjecture the user can newgoal to a lower depth. This allows the user to try and prove auxiliary conjectures that may help in proving the current conjecture. Depending on the method used to newgoal there may be a number of conjectures to prove at the new depth. Each new conjecture will share the hypothesis of the conjecture newgoaled from, possibly with some new hypothesis and with a different conclusion.

If the newgoal attempt is successful the conclusion of the newgoal is added to the conjecture newgoaled from as a hypothesis. If the newgoal has been carefully selected this new hypothesis should help in proving this conjecture. If the user decides to abandon a newgoal attempt they can return to the state saved at the depth boundary.

4.1.4 Proof Logs

As a user interacts with SPADE every piece of text entered is recorded in a proof log. This quite unsophisticated logging mechanism can be used to achieve some useful functionality.

Where a user explores a wrong path for a conjecture they may want to undo only the last few steps, instead of restarting from the last depth boundary. This can be achieved by restarting the proof of the conjecture from the very start, removing the last few lines of the command log, and then sourcing this command log. All of the users actions on this conjecture up to the appropriate point will be repeated. Thus when finished sourcing the command log the user will have managed to not do, rather than undo, the few steps.

It may also be the case that a user has spotted a common method used to prove a number of similar VCs. They can conduct this automatically by selecting each conjecture and sourcing the relevant part of the proof log. Note that this may require some editing, as all hypothesis and conclusions will be referred to by number in the proof log.

[4] Interestingly, it need not be case that each program unit at *Outside* is tackled one at a time. Two versions of SPADE can run at the same time.

5 Library Mechanism

The library mechanism describes both the different collections of data required by a system and how these are accessed. In SPADE there are two different librarys. The VCs, to describe the problem, and the rules, to help solve the problem.

However, neither of these two libraries can be considered in isolation. For example, a proof function refereed to in a VC may be assigned definitions as part of the rules. In practise such crossovers are essential, as only the rules that describe entitys in the VCs will have potential to assist in a proof.

Associated with each VC or rule there is a deeper meaning than the syntactic representation. For example, a proof function must record the types of its arguments. This leads to a problem of whether the VCs, rules or both should have copies of these meanings. This problem is solved through the Function Description Language.

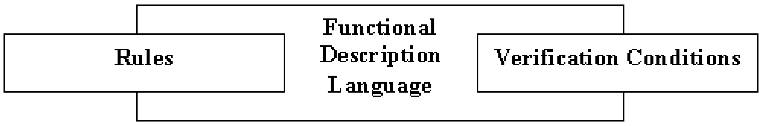


Figure 5: Functional Description Language adding common semantics to rules and Verification Conditions

The VCs and rules themselves are described syntactically. Where the VCs and rules are describing the same entity it will have the same name. The meaning behind these names are described in FDL. This is illustrated in Figure 5.

5.1 Functional Description Language

FDL is a surprisingly powerful descriptive language, bearing a close similarity to Pascal. It is used to assign meanings to the syntactic VCs and rules used inside SPADE. It is worth noting that FDL is also used to describe proof annotations in Spark programs.

As FDL is used to describe the meanings of both the VCs and rules it might be loaded as a wholly separate component. However, in practise the FDL files loaded into SPADE are either tied to the VCs or rules.

- **FDL tied to automatically generated VCs -** Where the Examiner automatically generates VCs it also generates corresponding FDL.
- **FDL tied to rule libraries -** Where the user creates their own rule libraries they must also provide corresponding FDL.

A full discussion of FDL is outside the scope of this note. It can be thought of as a long list of declarations as might be seen in any other program, but without any code making use of these declarations. It provides a means to describe in detail each word used in the VCs or rules. A short segment of FDL is shown in Figure 6.

```
title function runtimefunction;

  function round__(real) : integer;
  type types_runtime__vector = array [integer] of integer;
  const types_runtime__top : integer = pending;
  const types_runtime__bottom : integer = pending;
  const integer__last : integer = pending;
  const integer__first : integer = pending;
  var myarray : types_runtime__vector;
  var myindex : integer;

end;
```

Figure 6: Some FDL

5.2 Verification Conditions

A VCG file contains a syntactic description of the hypothesis and conclusions of every VC. A single VC as part of a VCG file is shown in Figure 7.

```
For path(s) from start to finish:

function_inc_1.
H1:    true .
H2:    value >= integer__first .
H3:    value <= integer__last .
        ->
C1:    value + 1 = value + 1 .
```

Figure 7: A Verification Condition

5.3 Rules

A rule file contains a syntactic description of a number of rules. To aid in quickly searching through these rules a limited amount to type information is also provided as part of the rule file.

Each rule is declared as part of a rule family. This allows easy access to groups of rules that manipulate similar formula. Each rule family must declare the input type of every function it uses. This is a general type used to help search through the rules, rather than the proper semantic description of FDL. The general types are specified thus:

Type specifier	**Types allowed**
i	Integer.
ir	Integer or Real.
ire	Integer, Real or Enumeration.
e	Enumeration.
any	Any type.

The rules themselves are presented using one of a number of possible forms:

X may_be_replaced_by Y.
X may_be_replaced_by Y if [CONDITIONS].
X & Y are_interchangeable.
X & Y are_interchangeable if [CONDITIONS].
X may_be_deduced.
X may_be_deduced if [CONDITIONS].
X may_be_deduced_from Y.
X may_be_deduced_from Y if [CONDITIONS].

Note the similarity between (...`may_be_replaced_by`...) and (...`may_be_deduced`...). The two different forms are used to declare rules available for rewriting and for inferring respectively. This separation offers greater control in maintaining the rules used for rewriting. For example, where a rule can not be stored as a rewrite rule, as it has unfavourable search properties, it can still be held in the system, only as a deduced rule.

An extract of a rule file is shown in Figure 8. Note how the first section of this extract introduces the rule family and general types while the second section declares the actual rules.

```
rule_family arith:
    X * Y   requires [ X:ir, Y:ir ] &
    X + Y   requires [ X:ir, Y:ir ] &
    X - Y   requires [ X:ir, Y:ir ] &
    X div Y requires [ X:i,  Y:i  ] &
    X / Y   requires [ X:ir, Y:ir ].

arith(1):    X*1            may_be_replaced_by X.
arith(2):    1*X            may_be_replaced_by X.
arith(3):    X+0            may_be_replaced_by X.
arith(4):    0+X            may_be_replaced_by X.
arith(5):    X - 0          may_be_replaced_by X.
arith(6):    X*0            may_be_replaced_by 0.
arith(7):    0*X            may_be_replaced_by 0.
arith(8):    X div 1        may_be_replaced_by X.
arith(9):    (X*N) div N    may_be_replaced_by X        if [ N<>0 ].
arith(10):   (N*X) div N    may_be_replaced_by X        if [ N<>0 ].
arith(11):    X/1           may_be_replaced_by X.
arith(12):    (X/Y)*Y       may_be_replaced_by X        if [ Y<>0 ].
arith(13):    Y*(X/Y)       may_be_replaced_by X        if [ Y<>0 ].
arith(14):    (X*Y)/Y       may_be_replaced_by X        if [ Y<>0 ].
arith(15):    (Y*X)/Y       may_be_replaced_by X        if [ Y<>0 ].
```

Figure 8: Declare a rule family

6 Data Representation

The database is at the very centre of the the SPADE system. Recognising how it operates is key to understanding the whole system. Figure 9 shows a high level view of the data structures held in SPADE. A description of the predicates used to actually hold these structures is provided in the Appendix.

The working area is at the centre of the database. This is where the actual theorem proving takes place. As mentioned above, proofs are constructed one at a time in one direction, with states being saved at proof boundaries. By only tackling one proof at a time there need only be one working area. As proofs are in one direction this working area need only store the current state of the hypothesis and conclusions.

The working area can be filled by tackling a new VC. When subgoaling the working area can be replaced with another subgoal at the current depth or by abandoning the subgoal attempt and recovering the state just prior to subgoaling.

The global items associated with the working area store additional information about the current conjecture. For example, which hypothesis, if any, the user has selected to hide when displaying the current conjecture. These are cleared and edited accordingly to ensure they match the current conjecture. They are not saved as part of the working area.

The rules available to the system are recorded as a number of rule stubs. These stubs can be quickly searched to find potential matching rules before consulting the corresponding rule files. Note that the built-in rules are essentially considered in the same manner as user loaded rules. However, the built-in rule stubs are assigned different names internally and are stored statically, thus do not technically exist in the database.

The semantics of the VCs and rules are held in a common area. This literally assigns a semantic definition to every syntactic word used in describing the VCs are rules.

Along with all the essential theorem proving information, SPADE also holds various bookkeeping information in the database. This records the files being worked on, the current system settings, etc.

7 Integrating SPADE With Other Systems

A number of initial insights to the problems of integrating SPADE and CLAM are made. The new system shall be referred to here as NUSPADE [5]. This refers to the complete system, including any parts of SPADE or CLAM that

[5]This either stands for Nearly Ultimate Spade or New Spade, or perhaps both. The name signifies that the new system is an extension to SPADE, rather than a whole new program that happens to interact with SPADE.

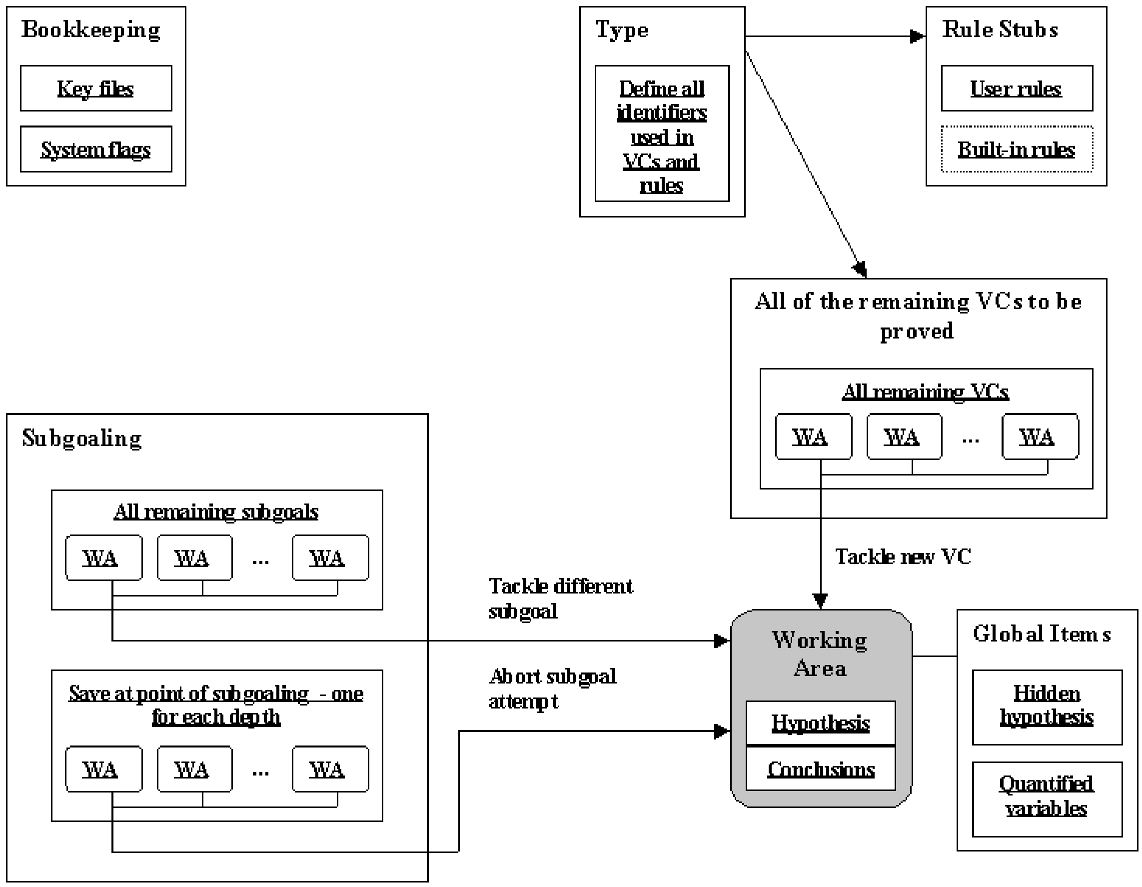


Figure 9: Data representation inside SPADE

may or may not be used.

7.1 The Nature of SPADE

Prolog is certainly an advantageous language to build a theorem prover. Resolution proof, search mechanism, strong structure inspection predicates, readily accessible database, are all useful features that would require quite substantial effort to implement in an imperative language such as C++.

However, Prolog is not as accommodating for system building. Features such as data/functional abstraction are not implemented in a formal manner that matches the header files of C, let alone the classes of C++. Further SPADE has been in development for quite some time, with code dating back as far as 1984. The current state of the art for theorem proving is continually evolving, making it difficult to pursue a coherent architecture over such a long period of time.

Fortunately, despite these potential software engineering problems, SPADE has managed to stay true to its architecture. There is a clear separation between the different commands of the system and the files they are implemented in.

Thus it can be observed that SPADE is a large system, best modified by respecting its original architecture. It is foreseen that it might be helpful to interact with SPADE at a low level. This should be done with great caution, and preferably without affecting the system state.

7.2 Accessing Data

The VCs tackled inside SPADE must first be loaded from the appropriate files. Further, it is likely that additional rules will also be loaded for a given proof session.

In NUSPADE this information could be extracted from the files in the same manner used by SPADE. However, this is not a trivial operation. The parsing of these files and their correct interpretation would be require a substantial section of code.

Thus if NUSPADE is to load problems in exactly the same manner as SPADE it may be sensible to copy the appropriate routines. In the short term this solution may be practical. However in the long term the SPADE versions of these routines may change, requiring copying these changes to NUSPADE.

There may be an alternative solution if SPADE, CLAM, and NUSPADE all share the same Prolog environment. In this case NUSPADE could directly access the appropriate part of the global database. Of course, changes to SPADE may still impact on the database, requiring other amendments to NUSPADE.

7.3 Dual Operation

There is some enthusiasm in making NUSPADE appear as a super-set of SPADE. This way users of SPADE may be more keen to adopt NUSPADE, as they can revert to SPADE whenever they like. The precise details of this dual behaviour is likely to be a significant design decision.

It is expected that NUSPADE will operate using more information than is held in SPADE. However, it is not clear what this means at the point of invoking an NUSPADE command. Will the current state in SPADE be the starting point from which information will grow in NUSPADE or will NUSPADE expect to start with information greater than that commonly held in SPADE.

If NUSPADE expects more information than SPADE has to offer then it may not be possible to revert to true SPADE. It is likely that some NUSPADE code will remain to secretly inspect the users actions and make NUSPADE specific entries in the database.

7.4 Interact With SPADE

The general idea of NUSPADE is to find a proof fragment that will help in advancing the current problem in SPADE. Once discovered this proof fragment may be executed inside SPADE at the push of a button.

Theorem proving in SPADE is geared up for a human user. For example, in a helpful manner SPADE asks for conformation on many actions. Thus the proof fragment devised by NUSPADE will need to cater for these unusual, but entirely predictable, interaction commands.

More significantly, NUSPADE will need to be aware of the names of rules in SPADE and the numbers of the relevant hypothesis and conclusions. In general, NUSPADE will either need to be able to convert from its theorem proving language to something recognisable by SPADE or work in the same language as SPADE.

7.5 NUSPADE Rules

At any point in using SPADE there will be a number of rules made available. Together these make up everything SPADE can use in theorem proving. If NUSPADE finds a proof using rules not know to SPADE then a corresponding proof in SPADE does not exist. The missing rules must first be loaded into SPADE.

There is thus an observation that any rule used in NUSPADE must be able to exist inside SPADE. In this context it may be sensible to adopt SPADE rule conventions inside NUSPADE.

8 Conclusions

SPADE is neither a hugely complex program nor a particularly straight forward one. The complexity of SPADE is inherent from its subject matter, while its clarity is gained by design. The subject matter makes it difficult to significantly reduce the complexity of SPADE. However, it would be quite possible through good intentions to push SPADE beyond its original design and begin sacrificing its clarity.

In our case the development of SPADE itself is handled elsewhere. Our emphasis is on developing a new system that uses SPADE as a complete unit. However, it can still be argued that interaction with SPADE should be reduced to the minimum necessary.

The purpose of this note was to help in designing an architecture for combining SPADE and CLAM. To achieve this it was felt necessary to consider the detailed operation of both of these systems. However, in view of the above argument for limited interaction with SPADE, it is perhaps likely that the majority of this information will be used indirectly rather than as a key route to building the new system.

Some initial insights are supplied concerning the building of the new system. However, many of these are really questions concerning the behaviour of CLAM and the intended behaviour of the new system. Thus the key to progressing this is to consider CLAM and to better appreciate, even informally specify, the operation of the new system.

A Appendix

A.1 The Working Area

The hypothesis and conclusions of the current conjecture in the working area are recorded:

conc(NUMBER, FORMULA)	
This predicate stores all of the conclusion formulas for the VC currently being tackled. Each conclusion is assigned a number, starting at 1.	
NUMBER	The number of this conclusion within this VC.
FORMULA	The formula of this conclusion.
Examples	
`conc(1, false)`	
`conc(3, >=(0, integer_first))`	

hyp(NUMBER, FORMULA)	
This predicate stores all of the hypothesis formulas for the VC currently being tackled. Each hypothesis is assigned a number, starting at 1.	
NUMBER	The number of this hypothesis within this VC.
FORMULA	The formula of this hypothesis.
Examples	
`hyp(1, true)`	
`hyp(11,>=(element(myarray, [myindex]), integer_first)`	

For both cosmetic and practical reasons the user can control the visibility of hypothesis in the working area:

forgotten(HYP_NUMBER)	
As the number of hypothesis grows the user may make the current goal easier to read by hiding any hypothesis. These hypothesis will only be hidden from display, and can still be used in the proof attempt. This is implemented by the printing routine skipping over any hypothesis that have been forgotten.	
HYP_NUMBER	The number of a hypothesis to hide.
Examples	
`forgotten(3)`	

deleted(HYP_NUMBER)	
As the number of hypothesis grows the user may make the current goal smaller by deleting hypothesis. Similar to forget, deleted hypothesis will not be show on the display. However, unlike forget, deleted hypothesis can not be used in the proof attempt. This is implemented by replacing the deleted hypothesis with the trivial hypothesis true. This is an essential trick to ensure that the numbering of hypothesis from 1 to the max hypothesis has no gaps. The deleted hypothesis are not printed by the system.	
HYP_NUMBER	The number of a hypothesis that has been deleted.
Examples	
`deleted(2)`	

deleted_hyp(HYP_NUMBER,FORMULA)	
Where a hypothesis is deleted it can no longer be used in the proof attempt. It is therefore helpful to be able to recover previously deleted hypothesis. This is achieved by storing the number of each deleted hypothesis and its formula. As the deleted hypothesis has technically only been replaced by the dummy hypothesis true, the hypothesis can be recovered by copying it back into this space, and removing the corresponding deleted predicate.	
HYP_NUMBER	The number of a hypothesis that has been deleted.
FORMULA	The formula of the deleted hypothesis.
Examples	
`deleted_hyp(2, >=(myindex, integer__first))`	

Where unwrapping quantified expressions new variables are introduced. These variables have no FDL declaration, and so are not part of the FDL type information.

qvar(QVAR)	
Both universally quantified expressions and existenetually quantified expressions can be unwrapped. This removes the quantifier and deals with the quantified variable. For existenetually quantified expressions the variable must grounded to a constant, for universally quantified expressions the variable is left uninstantiated. The user can instantiate these variables to whatever they require. These uninstantiated variables are recorded, and used to determine what the user can instantiate and restrict the unwrapping of quantified expressions that contain uninstantiated quantified variables.	
QVAR	An uninstantiated quantified variable.
Examples	
`qvar(int_I___1_1)`	

current_root(ROOT, CURRENT_NUMBER)	
This predicate ensures that new quantified variables do not clash with any existing quantified variables. It prescribes an increasing number to use in the new name each time a quantified variable with a common root component is unwrapped.	
ROOT	Root identifier for a quantified variable. This is held as a list of numbers that correspond to characters.
CURRENT_NUMBER	The last number used to create a new variable when unwrapping a quantified expression with this root identifier.
Examples	
`current_root([105, 110, 116, 95, 111], 1)` [6]	

A.2 Subgoaling

A single VC can be tackled as a number of smaller problems by subgoaling. Each of these subgoals and their role as part of the bigger problem must be recorded:

case_pointer(CASE_POINTER)	
Records the depth that the user is currently working at. Due to the way proofs are constructed in SPADE this must also be deepest depth.	
CASE_POINTER	A number corresponding to the depth the user is working at.
Examples	
`case_pointer(0)`	

[6] [105, 110, 116, 95, 111] is the string "int_o".

on_case(CASE_POINTER, CASE_NUMBER, TEMPORARY_HYPOTHESIS_NUMBER)	
A proof by cases can create a number of new subgoals to prove. Each of these new subgoals will have new and potentially different hypothesis. Like the VCs of the program, only one of these subgoals may be tackled at a time. Further, it is quite possible to attempt a proof of a case subgoal by again using a proof by cases. Thus at each depth where a proof by cases is being conducted the case being tackled must be recorded. This is stored by holding the depth that these cases occur, the number of the case being tackled, and the number of the hypothesis that has been inserted to accommodate this case.	
CASE_POINTER	A number corresponding to the depth that these cases occur.
CASE_NUMBER	The number of the case being worked on at this depth.
TEMPORARY_HYPOTHESIS_NUMBER	The number of the hypothesis inserted to accommodate this case.
Examples	
`on_case(2, 1, 7)`	

proved_for_case(CASE_POINTER, CASE_NUMBER)	
At each depth where there is a proof by cases it may be the case that a number of cases have already been proved. This is recorded by storing the location of each case as they are proved.	
CASE_POINTER	A number corresponding to the depth where a case has been proved.
CASE_NUMBER	The number of the case at this depth that has been proved.
Examples	
`proved_for_case(1, 1)`	

saved_vc(CASE_POINTER, FACT)	
At the point of subgoaling the goal currently being tackled by the user is saved. This saved goal will be returned to, intact if the subgoal proof is abandoned, or with new hypothesis inserted if the subgoal proof is successful.	
CASE_POINTER	A number corresponding to the depth where the VC was saved.
FACT	A fact for this VC, as hypothesis or conclusion.
Examples	
`saved_vc(1, hyp(2, >=(myindex, integer__first)))`	
`saved_vc(1, conc(1, >=(0, integer__first)))`	

subgoal_formula(CASE_DEPTH, FORMULA, CONC_NUMBER, METHOD)	
The user can issue the command status to display their current position in a proof attempt. This predicate gives easy access to the subgoal information required by status.	
CASE_DEPTH	Depth of a subgoal proof attempt.
FORMULA	The formula of the conclusion being tackled in the subgoal.
CONC_NUMBER	The number of this conclusion in the goal being subgoaled.
METHOD	A string describing the method used to subgoal. • **CONTRADICTION -** Subgoal is a proof by contradiction. • **CASES -** Subgoal is a proof by cases. • **IMPLICATION -** Subgoal is a proof by implication. • **INDUCTION -** Subgoal is a proof by induction. • **QUANTIFICATION -** Subgoal is a proof by quantification.
Examples	
`subgoal_formula(1, >=(0, integer__first), 1, CASES)`	
`subgoal_formula(1, types_trisortarray__ordered(mytriple), 2, CONTRADICTION)`	

A.3 All the verification conditions

Storing the names and details of every VC that still needs to be proved to show the program unit correct:

is_vc(VC_NAME)	
The name of every VC that has yet to be proved is held here.	
VC_NAME	Name for VC, based on the Ada name for the program unit and the number of each VC.
Examples	
`is_vc(procedure_trisortarrayinteger_2)`	

vc(VC_NAME, HYP_OR_CONC)	
For every named VC remaining to be proved this holds the actual details of the VC.	
VC_NAME	Name for VC, based on the Ada name for the program unit and the number of each VC.
HYP_OR_CONC	A part of this VC, as hypothesis or conclusion.
Examples	
`vc(function_runtimefunction_1, hyp(2, >=(myindex, integer__first)))`	
`vc(function_runtimefunction_2, conc(2, <=(0, integer__last)))`	

vcs_to_prove(VCLIST)	
This records the numbers of every VC that still needs to be proved. These are held as a list of lists. Each embedded list contains an unbroken sequence of VC numbers, with both the numbers in the lists and the lists of sequences being ordered.	
VCLIST	A list of lists holding the VCs that still need to be proved.
Examples	
`vcs_to_prove([[2], [4], [6], [8, 9], [11, 12, 13, 14, 15]])`	

A.4 Rule Stubs

The rules of the system are held as a collection of stubs. When matched the appropriate rule file is consulted to find the rule details:

user_rulefile(FILE, RULE) (Use 1)	
This records for each user loaded family of rules the file where these rules can be found.	
FILE	File from which the family of rules where loaded from.
RULE	The rule family name with a free variable, ranging over all of the rules of this family.
Examples	
`user_rulefile(trisortarray.rul, perm(A1))`	

user_rulefile(EXPR, FILE) (Use 2)	
There may be a number of rules in a family of rules. These rules may be used to advance a proof attempt. To determine which rules are applicable at any given point, without scanning through all the rule files, the minimum structure matching information is held. The minimum structure matching information is the most general expression of the matching side of each rewrite rule.	
EXPR	The most general expression, specified as a function name with each argument as an uninstantiated variable. Note that zero arity functions are possible, for example in rewriting a constant identifier to its value.
FILE	File that this rule this rule was loaded from.
Examples	
`user_rulefile(permutation(A1, A2), trisortarray.rul)`	
`user_rulefile(types_runtime__top, runtimefunction.rls)`	

user_classification(EXPR, FNAME, RNAME, CONSTRAINTS)	
The detail of each rule used by the system is always loaded on demand. The system only stores a small amount of information sufficient to identify when a rule might be applicable, and where it can be found. While **user_rulefile(···) (Use 2)** records the most general expression this predicate also records a general type. Interestingly, SPADE only seems to use the type information after having consulted the rule files. Doing this the other way round is expected to be more efficient, as rule files need not be consulted where the rule structure matches but type does not.	
EXPR	The most general expression, specified as a function name with each argument as an uninstantiated variable. Note that zero arity functions are possible, for example in rewriting a constant identifier to its value.
FNAME	File that this rule this rule was loaded from.
RNAME	The rule family name with a free variable, ranging over all of the rules of this family.
CONSTRAINTS	The general type constraints for each variable in the expression. This is held as a list of tuples pairing the variable with the type specifier.
Examples	
`user_classification(>=(A1, A2), runtimefunctionn.rls, runtimefunct_rules(A3), [:(A1, ire), :(A2, ire)])`	
`user_classification(permutation(A1, A2), trisortarray.rul, perm(A3), [:(A1, any), :(A2, any)])`	

ruleused([NAME,RULE])	
If in conducting a proof the user makes use of a supplied rule then it is recored using this predicate. This is important in recording which rules have been used in a proof, making it possible to detect and amend proof attempts that made use of a rule that has subsequently been found to be wrong. Note that every rule used is recorded - even if the rule was used on a subgoal that is subsequently abandoned.	
NAME	File where this rule was found.
RULE	The name of the rule used, as family name with ground number.
Examples	
`ruleused([trisortarray.rul, ord(1)])`	
`ruleused([$SPADE_CHECKER/SPECIAL.RUL, eq(1)])`	

banned_rule(FILENAME, RULENAME)	
As rules are loaded into SPADE there is the possibility that a rule is rejected. There are a number of reasons that a rule may be rejected, and these reasons are affected by state. For example if a new rule has a name clash with an existing rule and it is not the same rule then both rules are subsequently rejected.	
FILENAME	File where this rule was found.
RULENAME	The name of the rule used, as family name with ground number.
Examples	
`banned_rule(runtimefunction.rls, runtimefunct_rules(1))`	

A.5 Functional Description Language

The FDL that binds together the VCs and rules is stored in a large number of predicates:

enumeration(TYPE, ENUMERATION)	
Records the enumerations for each enumeration type. The attributes of the enumeration are also held in **vc_const(···)**.	
TYPE	Name for the type.
ENUMERATION	Ordered list of legal values for this type.
Examples	
`enumeration(day, [mon, tue, wed, thu, fri, sat, sun])`	

used_ident(IDENTIFIER, CLASS)	
Records every identifier currently loaded. Associated with each identifier is a note of the identifiers class.	
IDENTIFIER	Name for the identifier.
CLASS	The class of the identifier. Available classes are: • **libident -** Identifiers introduced as part of a loaded library. • **var_const -** A constant value. Each of these will have a corresponding entry as **var_const(···)**. • **type -** A type. Each of these will have a corresponding entry as **type(···)**. • **function -** A function. Each of these will have a corresponding entry as **function(···)**. • **record_function -** A record_function. Each of these will have a corresponding entry as **record_function(···)**.
Examples	
`used_ident(day, type)`	
`used_ident(mon, var_const)`	
`used_ident(types_trisortarray_permutation, function)`	

var_const(CONST, CORE_TYPE, SPECIFIER)	
Records the details of every constant and variable.	
CONST	Name for the variable or constant.
CORE_TYPE	The type of the variable or constant.
SPECIFIER	The kind of identifier. Available classes are: • **c -** An identifier declared as being constant. • **v -** An identifier declared as being variable. • **p -** A variable introduced as a result of unwrapping quantified expressions. • **tv -** The type of variables in quantified expressions during interactive traversal. (This is a special case. The predicate is used as a storage location for traversing, and cleared when traversing is completed.)
Examples	
`var_const(mon, day, c)`	
`var_const(mytriple, types_trisortarray_triple, v)`	

type(TYPE_IDENTIFIER, TYPE_STRUCTURE)	
Records the details of every type.	
TYPE_IDENTIFIER	Name for the type.
TYPE_STRUCTURE	A structure describing the type of this identifier. Possible forms are: • **abstract -** Awaiting a type declaration. Most often seen where constants are used in FDL and assigned the value pending. • **enumerated -** An enumerated type. • **array([index_type], element_type) -** An array, indexed with values of type index_type and containing elements of type element_type. • **record([[componemt_name,componemt_type], ...]) -** A record, denoted by a list of two element lists. Each two element list describes a component of the record with its name as componemt_name and its type as componemt_type. There will be a two element lists for every component in the record. • **sequence(sequence_type) -** Denotes a sequence of type sequence_type. • **set(sequence_type) -** Denotes a set of type set_type.
Examples	
`type(char, abstract)`	
`type(day, enumerated)`	
`type(types_runtime_vector, array([integer], integer))`	
`type(string,record([[length, integer], [contents, contents_array]]))`	
`type(enumseq, sequence(dayenum))`	
`type(intset, set(integer))`	

type_alias(TYPE, RANGE_TYPE)	
Records the details of type aliases. These are used to represent the base type of subtypes. Note that any range information present in the subtype is discarded. Thus the subtype becomes only an alias for its base type. Where the subtype information is required for proving purposes it must be explicitly added.	
TYPE	Name for the subtype.
RANGE_TYPE	The name of the base type for this subtype.
Examples	
`type_alias(indexrange, integer)`	

function(FUNCTION, ARG_TYPES, RESULT_TYPE)	
Records the type details of every function. Calling details are kept in **function_template(···)**.	
FUNCTION	Name of the function.
ARG_TYPES	An ordered list of the type names for each of the functions arguments.
RESULT_TYPE	The return type of the function.
Examples	
`function(round__, [real], integer)`	
`function('types_trisortarray__permutation, [types_trisortarray__triple, types_trisortarray__triple], boolean)`	

function_template(FUNCTION_CALL, VAR_LIST, FUNCTION))	
Records how to call the function. Type details of the function are kept in **function(···)**.	
FUNCTION_CALL	How to call the function, specified as the function name with each argument as an uninstantiated variable.
VAR_LIST	Lists the variables used in describing the arguments to the function call.
FUNCTION	Name of the function.
Examples	
`function_template(round_(A1), [A1], round__)`	

record_function(NUMBER, FUNCTION, MODE, FIELD, ARGUMENTS, TYPE)	
Records the automatically generated functions that allow access and updating of record types. It is correct to think of these as a special form of the array operations **update(···)** and **element(···)**.	
NUMBER	The number of a component that this record function makes use of. This starts at 1 for the first component of each record.
FUNCTION	How to call the function, specified as the function name with each argument as an uninstantiated variable.
MODE	The type of function described in this predicate. Possible forms are: • **update** - The function name will take the form upf_COMPONENT_NAME(RECORD, NEW_VALUE_FOR_COMPONENT). This will change the value of COMPONENT_NAME in record RECORD to NEW_VALUE_FOR_COMPONENT and return the new record. • **access** - The function name will take the form fld_COMPONENT_NAME(RECORD). This will return the value of COMPONENT_NAME in record RECORD.
FIELD	The name of the record component.
ARGUMENTS	A list of the uninstantiated variables used to describe how to call the function.
TYPE	The name of the record structure.
Examples	
`record_function(3, upf_day(A1, A2), update, day, [A1, A2], daterecord)`	
`record_function(3, fld_day(A1), access, day, [A1], daterecord)`	
`record_function(1, upf_time(A1, A2), update, time, [A1, A2], daterecord)`	

A.6 Bookkeeping

There are a number of predicates used to record system settings and other non theorem proving information:

csvfile_name(CSVFILE)	
The file used to save the system state is fixed here. It is set to the location that the checker is executed.	
CSVFILE	Full path and file name of the fixed save file (*.CSV).
Examples	
`csvfile_name(runtimefunction.csv)`	

fdl_file_title(TITLE)	
Stores the titles of loaded FDL files. The location of the FDL files is held in **fdlfile_name(···)**.	
TITLE	The title specified inside an FDL file (*.FDL).
Examples	
`fdl_file_title(function_runtimefunction)`	

fdlfile_name(FDLNAME)	
Stores the location of loaded FDL files. The title of the FDL file is held in **fdl_file_title(···)**.	
FDLNAME	Full path and file name of the loaded FDL file (*.FDL).
Examples	
`fdlfile_name(runtimefunction.fdl)`	

library_already_loaded(FULL_FILE_NAME)	
Holds the name of each loaded library file.	
FULL_FILE_NAME	Full path and file name of the loaded library file (*.RLB).
Examples	
`library_already_loaded(z8002logic2.rlb)`	

logfile_name(LOGFILE)	
Holds the name of the proof log file.	
FULL_FILE_NAME	Full path and file name of the proof log file (*.PLG).
Examples	
`logfile_name(runtimefunction.plg)`	

command_log_filename(CMDLOG)	
Holds the name of the command log file.	
CMDLOG	Full path and file name of the command log file (*.CMD).
Examples	
`command_log_filename(runtimefunction.cmd)`	

vcgfile_name(VCGFILE)	
Holds the name of the file containing the VCs being tackled. The VCs may be provided straight from the Examiner or after they have been put through the Simplifier.	
VCGFILE	Full path and file name of the VC file (*.VCG or *.SIV).
Examples	
`vcgfile_name(runtimefunction.vcg)`	