Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 217 (2008) 221–234
www.elsevier.com/locate/entcs

The HiVe Writer

Tony Cant[1,2], Ben Long, Jim McCarthy, Brendan Mahony and Kylie Williams

Command, Control, Communications and Intelligence Division
Defence Science and Technology Organisation
PO Box 1500, Edinburgh, South Australia 5111

Abstract

Critical systems require assurance that key security, safety or mission requirements are met. Tools are necessary to provide this assurance. The HiVe Writer supports model-based documentation for complex critical systems. The HiVe Writer forms the functional foundation for the ambitious HiVe (Hierarchical Verification Environment) project which aims to provide a unified framework in which entire design projects can be described with the highest level of assurance. The primary innovation in the HiVe Writer is a centrally-managed design model: any design, explanatory and technical documents created within the tool are constrained to be consistent with this design model and therefore with each other. This paper gives a detailed description of the HiVe Writer, showing how it supports model-based editing of structured technical documents and, in particular, requirements formulation.

Keywords: Critical Systems, Formal Methods, Design Verification, Requirements Formulation.

1 Introduction

The Australian Government Department of Defence procures a number of computer-based systems that are *critical* in the sense that they have requirements whose violation could have grave consequences.

- *security-critical systems* need to satisfy a *security policy* [4] enforcing a number of security requirements, such as the prevention of unauthorised access to confidential data or the maintenance of operational integrity.
- *safety-critical systems* need to satisfy a *safety case* [7] ensuring that system design eliminates or reduces the chance that hazardous system states (that could lead to injury or death) are reached.

[1] The authors wish to thank the Defence Materiel Organisation for sponsorship and funding of The HiVe.
[2] Email: Tony.Cant@dsto.defence.gov.au

doi:10.1016/j.entcs.2008.06.051

- *mission-critical systems* must ensure that certain goals, mission or performance requirements (critical to success of the mission) are met.

The central challenge for critical systems is to be able both to achieve assurance and to transfer this assurance to other parties. Assurance is achieved by carrying out a range of specification and verification activities in the early stages of development that show how critical system requirements are met. The transfer of assurance involves an appropriate mix of informal, semiformal and formal arguments that will convince a third party, such as an evaluator or certifier.

An example of a tool supporting informal arguments is Telelogic's DOORS [2]. DOORS provides a framework in which users can generate structured documentation consisting of a series of numbered informal requirements.

Semi-formal arguments are supported by tools such as Telelogic Rhapsody, which combines requirements management and design and simulation of UML [10] schematics within the one tool. The MathWorks Simulink [8] is another example of a semi-formal development tool and supports the design and simulation of functional block diagrams.

Formal methods provide well-defined languages such as Z [6] and CSP [5] that allow precise specifications to be written and subsequent rigorous verification procedures — typically *model checking* [3] or *theorem proving* [11,9] — to be performed.

Regardless of whether informal, semiformal or formal arguments are used, or whether a single formalism or multiple formalisms are used, the need for assurance requires documentation that is well structured and that provides consistency between all arguments. The authors' contention, based on extensive research and application, is that the following are the key desiderata for a design tool to support the achievement and transfer of assurance for critical systems:

Powerful Modelling: The tool should support the design of hierarchical, concurrent and real-time systems.

Trustworthy Modelling: The tool should support reasoning and proof as well as simulation.

High-Assurance: The tool should support the specification and verification of critical requirements, as well as the ability to flow requirements to system components.

Communication: The tool should be document-driven, supporting different views for different audiences.

Synergy with other tools: The tool should allow the interaction with other tools, such as theorem provers, model checkers and simulation tools.

The *Hierarchical Verification Environment* (HiVe) project aims to realise these goals by providing a unified framework in which entire design projects can be captured. It will encourage designers to write design, explanatory and technical documents in parallel, thereby helping the user to produce output that will convince others of the correctness of the design. All documents will be created (and remain) consistent with a centrally-managed document-driven design model, which will al-

low consistent presentation of informal, semi-formal, and formal arguments, and from which various tools involving simulation, model checking and theorem proving can be invoked. The HiVe *Writer*, discussed in this paper, is the component specifically concerned with document generation for achieving this.

In Section 2 we introduce the fundamental principles on which the Writer is based, and give an intuitive description of the mechanisms through which it realises the goals outlined above. Section 3 sketches the functional decomposition of the Writer's implementation. In Section 4, we consider the use of the Writer, from a user interface viewpoint, in a case study based on the development of the Def (Aust) 5679 Safety Standard [1]. Section 5 presents some conclusions.

2 The Writer: a mechanistic underview

The fundamental principles underlying the HiVe Writer are embodied in the dual notions of *model-based editing* and *literate modelling*:

(i) The Writer builds a *model* as an approach to complex documentation tasks. This helps the user to maintain consistency between documents since they all refer to the same underlying model. For the same reason it helps the user to oversee and implement change management. Finally, the model itself provides assurance of consistency in the design process.

(ii) The Writer builds *literate documentation* as an approach to complex modelling tasks. This helps to promote the user's comprehension of the design and implementation. Equally importantly, it helps to communicate the requirements, the design, and the results of assurance activities to other stakeholders in the process.

The HiVe Writer can thus be regarded as an application of *lightweight formal methods*: the tool enforces consistency as the model-based documentation is built, but the user is not required to be an expert in the application of formal methods.

Our approach to implementing a tool that upholds the above principles utilises highly structured documents that are correlated through a central fact repository. In this section we briefly elaborate on how this works.

2.1 Structure algebraically

The document structure is induced from the use of what we call *structured text*. At the simplest level, structured text is the language generated by a repository of *syntax constructors*. The constructors are *sorted*: i.e., each constructor argument only accepts terms from a particular subset of structured text (labelled by a so-called *sort*), and the constructor returns a term of definite sort. Documents created in the HiVe Writer are written in structured text, thereby acquiring a hierarchical structure from the sort-directed nesting of constructors.

Figure 1 illustrates the hierarchy exhibited by the mathematical expression

$$x + y = 3 \wedge 2x - y = 3\,.$$

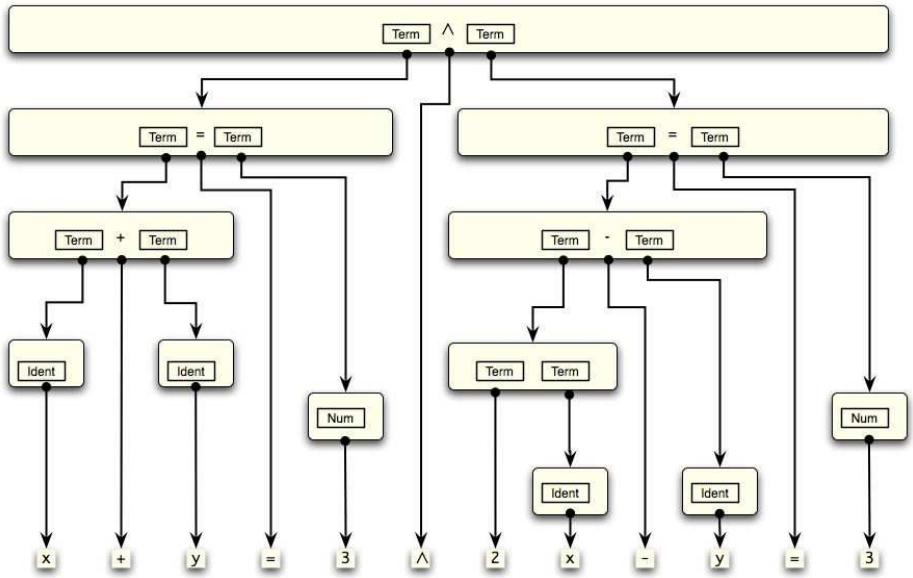

Fig. 1. Structured text.

The structure, in this simple example of a term algebra, branches according to the algebraic nature of the operators: the top level is a conjunction of two subexpressions; each subexpression is an equality; and so on. For example, the top-level conjunction is achieved through the constructor $\wedge$, for which a production is indicated via a *dummy box* inscribed with the required sort.

The syntax constructors need not just build mathematical terms, but could act on formatted text or whatever is desired. One example, featuring in Section 4, is a simple requirements language. It is used to enforce a strict documentation standard on requirements gathering which prevents the user from writing requirements on development artifacts that cannot be held to account.

To remove any prejudice of usage, we introduce the term *element* to denote the fundamental data underlying a "syntax constructor": i.e., a *name*, a *result* sort and the list of argument sorts known as its *arity*. For example, the top-level element in Figure 1 could be denoted

$$\texttt{conjunction} :: \text{Term} \leftarrow [\text{Term},\text{Term}]$$

In the HiVe Writer, elements are used at all levels in document production. Technically, they provide the constructors of a *many-sorted algebra*. Elements must be defined for *all* language elements required to build documents; in particular, those required to declare other elements. So, a sort is an element, as are all the constructors required to build the sort list for the element arity.

2.2 Present for readability

Truly literate documentation must be attractive for readability. Structured text would not be suitable for producing literate documents if it did not allow the syntactic freedom to display documents in their most natural and informative style as

appropriate to a given audience. Indeed, in Figure 1 the raw element data does not appear – rather, a corresponding presentation using appropriate mathematical symbols is given. For example, the element `conjunction` is presented by $\Box \wedge \Box$.

The HiVe Writer requires the user to define a *production* corresponding to each element, and it is provided at the point of declaration of the given element. The production has the form of a sequence of the element's arguments sandwiched by delimiters. The delimiters of the production can be arbitrary Unicode modified by the standard text attributes (bold, italic, color, etc). This provides sufficient freedom for most application domains, and is particularly straightforward for rendering to the screen. Thus, the document as presented on the screen is highly readable, and can be faithful to the printed form.

In fact, the user can define more than one production for a given element if desired. The different productions are distinguished by assignment to separate *presentation styles*. The mandatory production required at element declaration is assigned to the default style. An ordering on the different styles can be introduced, in which the default style provides a top. If a given element does not have a production in a given style, the style ordering is followed until a production is found. The ability to change presentation at a global level can be very useful: e.g., the documents can be "toggled" between presenting mathematical expressions as structured English statements or algebraic terms by the choice of appropriate presentation style.

We introduce the term *presentation* to denote the fundamental data required to present a given element: i.e., for a given element name, the production and the style to which it is assigned.

2.3 Reference for consistency

The consistent correlation between documents is built up through the use of *references* to a central fact repository. Thus, the repository stores a table of elements and a table of presentations. In use — such as in the example of Figure 1 — the elements are instantiated *by reference* to the repository. In a document, a given element reference includes the name of the desired element and the desired presentation style. The document is then rendered to the screen with the element reference replaced by the corresponding production of that style.

Element references provide the skeleton of structured text as described in Section 2.1 above. They are actually an instruction for constructing the syntax constructor corresponding to a given element using data from various tables in the repository. We will also extend the structured text to incorporate *direct* reference to entries in repository tables. We mentioned just two such tables in the previous paragraph, but in fact many tables are required to support the basic Writer functionality – and there will be many more as the HiVe is developed. The different tables store different classes of *facts* about elements. Note, for example, that a presentation can be considered to be a fact about a given element: the fact that in the given style the element will be presented by the given production. The individual chunks of data – the style, or the production – will be referred to as fact *attributes*.

Thus, these direct references will be referred to as *attribute references*.

We have already introduced facts about elements other than presentations. There is a table that records the ordering on styles. We also have a table that records a similar ordering on sorts – an ordering that introduces subsorting to the language.

2.4 Structured text in model-based documentation

The overall user goal in the HiVe is to create a *project*: a collection of related analysis constructs and documentation for a given activity. The concept of project encompasses a wide span of activities: from writing a technical paper through to a major system design and beyond. Documents required in a project will often be prescibed for a given activity, but may be extended to include any view of the project – say, pedagogical or summary form, pictorial or slideshow – that the user desires to construct.

The user does not start each project from scratch: a given project is built as an extension of existing projects, which are said to be *included*. In particular, the user will have a collection of analysis tools available, each of which can be controlled through the HiVe Writer. That is, each tool will have a plug-in that defines an interface to the Writer and a corresponding collection of elements, facts and tables needed to model the interactions with the tool. We call this combination a HiVe *module*. The developers of a HiVe module must construct the base element algebra for recording tool interactions and outputs. Modules may also add additional User Interface structures – such as buttons and menu items – to allow tool-specific commands to be run from within the HiVe. At the bottom of the project hierarchy is the core Writer "module" that includes the repository structure discussed above and certain native tool mechanisms discussed below.

The HiVe inextricably mixes domain modelling with document construction. The user selects the HiVe modules (that is a standard *vocabulary*) relevant for the given activity and includes the corresponding projects in the current project. This give them a powerful, but highly structured framework for expressing their ideas. The user then introduces the specific elements and facts required to develop a model of the activity at hand to the desired level of detail. The grammar of the document language is derived from the total collection of elements. During final document preparation, the repository provides easy access to all elements and all facts about those elements defined in the current project as well as those in included projects. This adds to the technical quality of the product by enforcing consistency in vocabulary and facts. The user is able to tailor the presentation of the various elements (and hence facts) to different audiences by defining appropriate productions, thus adding to communicative effectiveness of the documents produced.

We can now summarise the different ingredients of what is meant by the rubric "structured text".

- Element references that provide the skeleton of structured text.
- Attribute references that allow re-use of pre-defined fragments of structured text.

- The dummy box that indicates a placeholder for structured text entry.
- Literals that allow "keyboard" entry of strings, numbers, *et cetera*.

The on-screen (and on-paper) presentation of structured text is controlled through the setting of standard text attributes such as type face, weight, shape, but also through an additional text attribute called presentation style that controls the actual characters used to represent the various elements. Text attributes may be inherited through the structured text hierarchy or set locally.

2.5 Supported mechanisms

The HiVe Writer provides native support for building tables. A functional view is given in the next section. Here we just point out two mechanisms which derive from the constructs developed above.

(i) **Syntax-direction:** The sort-direction explicit in the element algebra can be directly harnessed for syntax-directed editing capabilities. Observe first how Figure 1 can be read "downwards" as a temporal record of the construction of the mathematical expression. At each stage the user selects a dummy box and inserts either an element reference (with further dummy boxes for any arguments), or alphanumeric data from the keyboard. By including the dummy box as part of structured text we ensure that each stage in the construction is a legal expression.

The Writer provides a palette of syntax constructors for data-entry of element references, and a view of the tables that can be used to enter attribute references. Both windows provide syntax-direction by reflecting the allowed sort at the insertion point: data with the wrong sort are "greyed-out" and cannot be selected.

(ii) **Intelligent change management:** Just one important consequence of the referencing structure is to provide a straightforward facility for intelligent propagation of changes throughout the documents of a given project. The change is made once at a central point – wherever the data is entered into the repository. Since all other instances throughout the documentation are references to facts in that repository, the change is immediately propagated. We provide a simple illustration of this point in Section 4.

3 The Writer: an application overview

As discussed, the HiVe Writer supports the preparation of structured technical documents. Thereto, it has three principal components.

(i) The *Document* editor which is for producing and updating structured documents.

(ii) The *Datastore* which is the combination of fact repository and syntax-directed data-entry mechanism discussed earlier. It provides the central design model for the project. All documents created within the project will be consistent

with this design model and therefore with each other.

(iii) The *Tool Interface* which is an interface, with a flexible plug-in architecture, between the Datastore and the tools used to process the design constructs.

The user controls tool interactions through a distinguished document, called the *Normative Design Document* (NDD). The NDD is strongly coupled to the Datastore: it provides the declaration point for all entries in the Datastore from the current project. There is exactly one Datastore, and so exactly one NDD, in any given project.

3.1 Tool control in the NDD

The NDD is a structured document in which the user records the epistemic narrative of the project development. It is special, however, as we will now explain: it is a (literately programmed) script through which the user controls all interactions with tools.

The HiVe can potentially interface to a broad range of external tools, such as: theorem provers and model checkers; algebraic analysis and numerical analysis; system simulation; programming support; drawing; project management; requirements analysis; and version control. Moreover, the population of the Datastore tables is considered an interaction with the core Writer tool. The only assumption of the HiVe is that a supported tool must apply some function to given input data, and produce diagnostics and output data.

The uniformity of the tool interaction is supported by the concept of *commands*. Commands are collected in the DataStore as a special sort of element in the algebra of structured text. The commands required for interaction with a particular tool are defined in the module that supports that tool along with the code necessary to enact the interaction.

Each tool interaction proceeds as follows.

The syntax constructor for the command is inserted in the NDD, and the input arguments filled in, using standard Datastore/keyboard data-entry mechanisms. The input arguments encode the user input required for a given tool action. Some arguments to a command may be reserved for presenting the results of a tool interaction and the user is prevented from editing them. *Every* tool command (such as a proof step in a theorem prover tool) is constructed similarly.

The user then instructs the HiVe to *process* the command. Control is passed to the relevant module which then enacts the desired transaction with the tool. Results from the tool transaction may be presented in any output arguments of the command and one or more facts are entered into the Datastore. Error messages are handled and displayed appropriately to the user.

If the user is unhappy with the results, the command may be unprocessed to allow editing of command input and then re-processed. The updated results are then entered into the Datastore.

The NDD is a *linear* script, and thus processing a region of commands is simply sequencing the intermediate steps. Linearity allows the user to keep contextual focus

8.2.2 The HAZARD ANALYSIS REPORT shall consist of:

1. **A description of the OPERATIONAL CONTEXT for the SYSTEM;**
2. **A SYSTEM DESCRIPTION;**
3. **A list of potential ACCIDENTS with corresponding ACCIDENT SEVERITIES;**

Fig. 2. A snapshot of Def (Aust) 5679.

whilst controlling several tools. Moreover, the script *must* be constructed in a well ordered fashion as determined by the execution logic of the tools it is controlling: for example, a theorem prover tool will not accept constants that have not been declared at an earlier stage.

Since commands are easily recognised in structured text, the command script can be embedded in an arbitrary pedagogical development of structured text. Ultimately, the universality of the command concept leads to the tight correlation between the Datastore tables and the NDD: modulo the pedagogical text, the data of the processed NDD command script is precisely equivalent to the current project's Datastore entries. Thus, indeed, the NDD is a literate exposition of the entire project construction.

4 Example: The Def (Aust) 5679 Safety Standard

To illustrate the HIVE Writer in use we consider the recent revision of the Australian Defence Standard Def (Aust) 5679 [7]. This Standard provides detailed requirements and guidance on the structure of the *Safety Case* for a safety-critical system. It focuses on *assurance* activities: these are system development and analysis activities that provide evidence that the system meets its safety requirements.

Our example is based on Chapter 8 (*Hazard Analysis*). Hazard Analysis is the first phase of Safety Case Development. Paragraph 8.2.2, shown before revision in Figure 2, highlights a frequent problem in the drafting of requirement documents: that of ensuring the assignment of clear lines of responsibility while maximising readability. The requirement, as stated, places an obligation on a document, the Hazard Analysis Report. This is hard to enforce. On the other hand, clarifying the responsible agent (in this case the Supplier) *in situ* would result in an overly complex statement of the requirement.

The solution adopted by the authors of Def (Aust) 5679 is a two stage assignment of responsibility: firstly, responsibility for a document (or a process) is assigned to an appropriate agent; then requirements are placed on the document itself. This allows clear assignment of responsibility while ensuring the requirements are stated in a natural way. The resulting revision is shown in Figure 3.

The following describes a simple HIVE module that we have developed to support this discipline by enforcing the use of a constrained requirements language.

8.2.2 The SUPPLIER is responsible for compiling the HAZARD ANALYSIS REPORT.

8.2.3 The HAZARD ANALYSIS REPORT must include:

1. **A description of the OPERATIONAL CONTEXT for the SYSTEM;**
2. **A SYSTEM DESCRIPTION;**
3. **A list of potential ACCIDENTS with corresponding ACCIDENT SEVERITIES;**

Fig. 3. The revised Safety Standard document

4.1 The Requirements Module

The Requirements module consists of a plug-in that provides special commands for constructing requirements and guidance paragraphs. The requirements command enforces a discipline so that

- each paragraph constructed in this manner is labelled uniquely; and
- the requirement text can only be formed with provided syntax constructors.

The latter is achieved through the Writer's sort-driven editing via a model for the requirements gathering process based on Figure 4. The boxes hold the sorts that distinguish the entities of the process, whilst the diamonds hold the syntax constructors that express the allowed relationships between entities.

As we will elaborate below, the resultant language cannot express the requirement in Figure 2, but *can* express the two-stage assignment of Figure 3. The plug-in also provides more subtle support. The user is protected from badly implementing the two stages: i.e., if the plug-in processes a 'must' requirement for a given Target then it also checks that a corresponding responsible Agent has been assigned – via an 'is responsible for' requirement – before allowing the document to be processed.

4.2 Using the HiVe *on Def (Aust) 5679*

A typical session for the Def (Aust) 5679 project in the HiVe Writer is shown in Figure 5. The top left hand window is the Project Navigator which provides

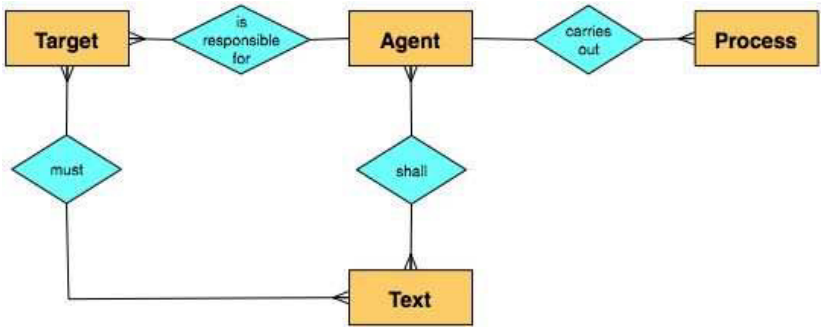

Fig. 4. Entity Relationship diagram of the requirements model.

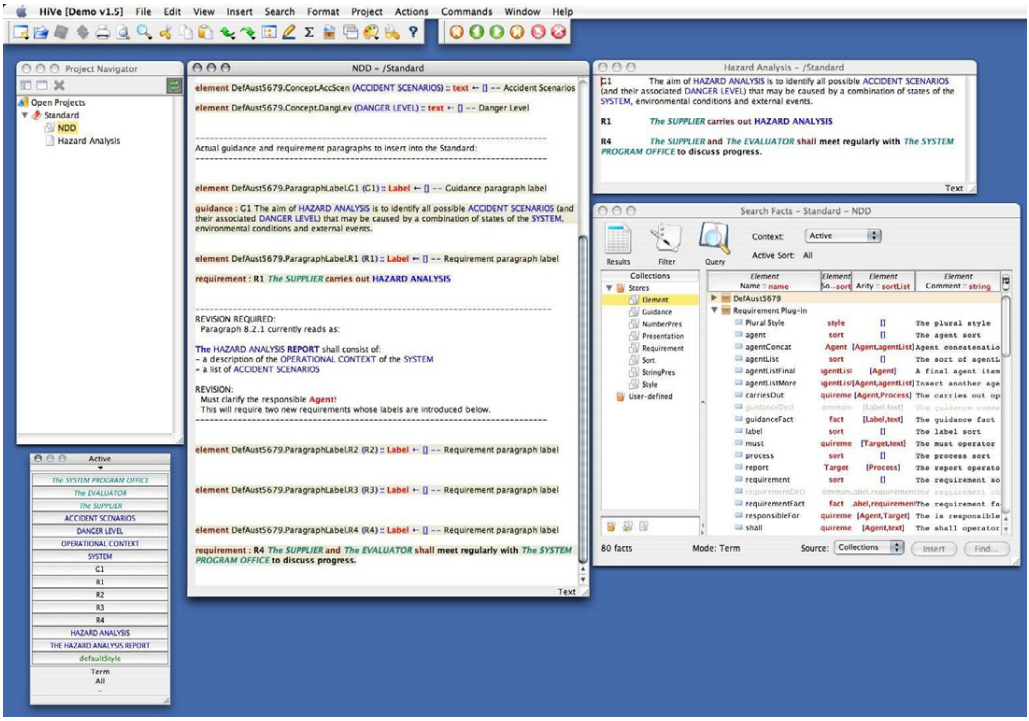

Fig. 5. A view of the HiVe in action

an easy mechanism for moving between the different documents in different open projects. The middle window is the NDD. It begins by instantiating the particular Agents (*Acquirer*, *Supplier*, *Auditor* and *Evaluator*), Processes (*Hazard Analysis*) and Targets (*Hazard Analysis Report*) used by Def (Aust) 5679 Chapter 8. It also contains some informal text flagging the need for revision of the badly-worded requirement. The top right hand window shows the Hazard Analysis document at the current stage of revision.

The Search Facts window, bottom right, is a view on the Datastore currently opened to show the Requirement plug-in elements. Finally, at the bottom left, is a palette derived by filtering a different Datastore view. It contains all the syntax constructors defined to date Def (Aust) 5679 in the project. In each project session any number of independent Datastore views (and syntax palettes) can be open, allowing multiple views into different aspects of the project data.

4.3 Revising the requirement 8.2.2

The requirement 8.2.2 must be replaced by a two-stage assignment as discussed earlier. For, suppose we were to attempt to introduce the previous requirement. Thereto, we insert the requirement command, provide a label (R2) for this command, and insert the 'shall' constructor to arrive at the situation shown in Figure 6. However, as seen there, an *Agent* must be supplied in the first argument (currently

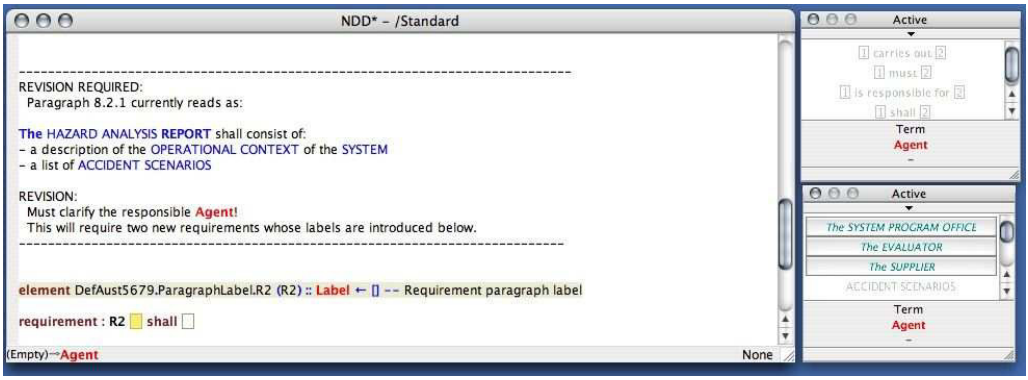

Fig. 6. The 'shall' constructor requiring an element of sort **Agent**

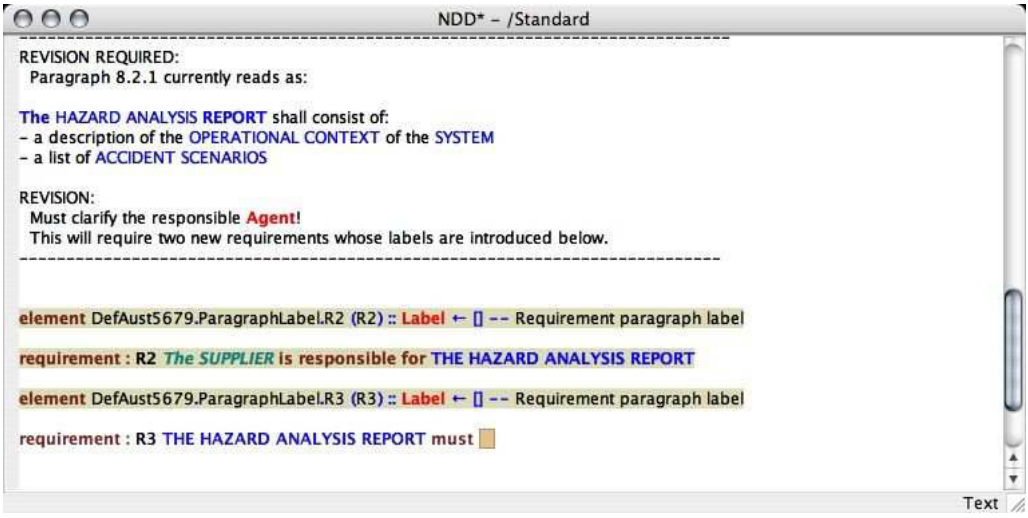

Fig. 7. Completing the text of the final requirement in the Standard.

selected) of the constructor:

- the status bar (bottom left) displays the sort required to fill the selected dummy box;
- the Writer has 'greyed-out' in the palettes all syntax constructors that do not result in the Agent sort.

This feedback allows the user to be contextually aware at all stages in the construction.

The proper application of two-stage assignment needs *two* requirements to replace the current one, as seen in Figure 7. The first requirement establishes the responsible Agent using the 'is responsible for' constructor – we choose The Supplier as the responsible Agent and The Hazard Analysis Report as the corresponding Target. The second requirement establishes the contents of The Hazard Analysis Report using the 'must' constructor. Figure 7 shows the NDD waiting for the text stating the contents. On processing, the requirements are added to the DataStore,

Fig. 8. The two-stage assignment added to the Datastore.

as shown in Figure 8. Finally, these requirements are added by reference the Safety Standard document leaving the result shown in Figure 3.

5 Conclusion

In this paper, we have given an overview of The HiVe Writer and shown how it can be applied in the case of a very simple example: a Safety Standard. We have shown how the production of the Safety Standard was supported through the construction of a light-weight model of the system being described. This model consisted of elements such as the Supplier and the Hazard Analysis Report as well as a structured language for describing the requirements placed on them by the Safety Standard (see Figure 4) The use of this model, along with the syntax-directed editing capability of The HiVe, ensures that users are constrained to write only paragraphs that conform with the model.

The Writer is still under development. Informing its development are a number of security devices and safety-critical systems. These examples are of great value in determining the best mode of user interaction with The Writer.

Future work will build upon the Writer: first of all, the Prover will incorporate the Isabelle theorem prover as a plug-in, extending the Writer by adding the ability to carry out literate theorem proving in Higher-Order Logic (a powerful and widely used framework for formal reasoning). The Modeller will enhance the Writer's functionality to allow for fragments of design (i.e. dataflow diagrams and state machines) to be stored in the datastore and referenced in documents. The Modeller will thereby exploit the Prover to support system modelling and verification.

References

[1] Australian Government Department of Defence. *Def (Aust) 5679, Issue 2: Safety Engineering for Defence Systems*, to appear in 2007.

[2] Tony Cant, Jim McCarthy, and Robyn Stanley. Tools for requirements management: a comparison of telelogic doors and the HiVE. General Document DSTO–GD–2006–0466, Defence Science and Technology Organistion, 2006.

[3] E.M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science, 2001.

[4] Common Criteria Project Sponsoring Organizations. *Common Criteria for Information Technology Security Evaluation*, version 2.1 edition, 1999.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[6] ISO/IEC 13568. *Information techonolgy – Z formal specification notation – Syntax, type system and semantics*, first edition, July 2002.

[7] Land Engineering Agency, Australian Government Department of Defence. *Def (Aust) 5679: Procurement of Computer-Based Safety Critical Systems*, 1998.

[8] The MathWorks. *Simulink 6 Reference*, 2007. http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/slref.pdf.

[9] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[10] Object Management Group. *UML Superstructure, OMG document formal/07-02-05*, v2.1.1 edition, 2007. http://www.omg.org/cgi-bin/doc?formal/07-02-05.

[11] J. M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, Germany, 2001.