Tools for producing formal specifications: a view of current architectures and future directions

Sunil Vadera[*] and Farid Meziane[†]

March 12, 2003

Abstract

During the last decade, one important contribution towards requirements engineering has been the advent of formal specification languages. They offer a well-defined notation that can improve consistency and avoid ambiguity in specifications.

However, the process of obtaining formal specifications that are consistent with the requirements is itself a difficult activity. Hence various researchers are developing systems that aid the transition from informal to formal specifications.

The kind of problems tackled and the contributions made by these proposed systems are very diverse. This paper brings these studies together to provide a vision for future architectures that aim to aid the transition from informal to formal specifications. The new architecture, which is based on the strengths of existing studies, tackles a number of key issues in requirements engineering such as identifying ambiguities, incompleteness, and reusability.

The paper concludes with a discussion of the research problems that need to be addressed in order to realise the proposed architecture.

[*]Depart. of Computer and Math. Sc., University of Salford, Salford M5 4WT, UK
[†]Inst. Software Technology, UNIMAS, 94300 Kota Samarahan, Sarawak, Malaysia

1 Introduction

Requirements analysis is a critical step of the software development process. Failure to produce a correct requirements document will result in the production of the wrong system. However, requirements analysis is a very difficult, tedious and error-prone task. Difficult, because it relies on a wide range of domains, which in most of cases, are unknown to the analysts. Tedious and error-prone, since there is a feedback loop between users, analysts and developers that is volatile with time and involves groups of people from different backgrounds, each favouring a possibly different representation.

During the last decade, a lot of effort has been devoted to improving the requirements and specification phases of the software development process. These efforts have been accompanied by the development of various tools and representations. One important contribution has been the development of tools to aid the control and management of requirements documents. Examples of such systems include the PSL/PSA system [36] and the SREM project [6].

Another important contribution has been the advent of formal specification languages (e.g., Z [35], VDM [18], and their extensions). The benefits of formal methods for the developments of systems are widely recognised [7, 25, 8]:

1. Formal methods provide a well-defined specification language that can enable automatic consistency and type checking.

2. It is possible to use a formal specification as an aid to identifying possible test cases and as a basis for animation.

3. When using a formal method, it is possible to prove that a program satisfies its specifications.

However, formal methods have two major weaknesses. First, formal specifications may not be consistent with the requirements written in natural language. Second, formal methods are based on mathematical logic, making them hard for some analysts to comprehend. In addition, one expects that some informality will always exist in software development [5, 28].

In an attempt to improve this problem, several tools have been proposed for obtaining formal specifications from informal and semi-formal specifications. Some of the tools concentrate primarily on integrating semi-formal notation (e.g. SSADM, data flow diagrams) with formal specification languages (e.g. [9, 27]). Other tools attempt to go further and aim to aid the production of formal specifications from informal specifications. This paper concentrates on this latter area since the integration of semi-formal and formal methods is already widely described (see [12] for a good recent survey).

Appendix A summarises the main proposals for obtaining formal specifications from informal specifications since 1977. As the summaries in the appendix suggest, the goals of the systems are quite different. None the less, each system contributes to the problem of obtaining formal specifications.

In this paper, we therefore examine the contribution made by each system and attempt to suggest an architecture that combines the strengths of the existing proposals.

The current systems can be divided into two broad categories: systems that are primarily knowledge based, and systems that are based on natural language analysis. Sections 2 and 3 summarise the main features of the systems in each category.[1] Section 4 draws together the lessons learned from these systems, and concludes with our view of the future direction of work on the formalisation of informal specifications.

2 Knowledge Based Identification of Formal Requirements

In this section, we summarise those systems we categorise as being knowledge based. That is, systems that make use of pre-defined domain knowledge as a basis for analysing requirements and producing formal specifications.

2.1 The SPECIFIER system

The SPECIFIER system can best be viewed as a case based system [19] whose architecture is summarised by figure 1.

It takes as input an informal specification of an operation where the pre and post-conditions are given as English sentences. The verbs in the sentences are used to identify the concepts. The identified concepts are then used

[1]For conciseness, we only present those systems that have influenced the proposed architecture.

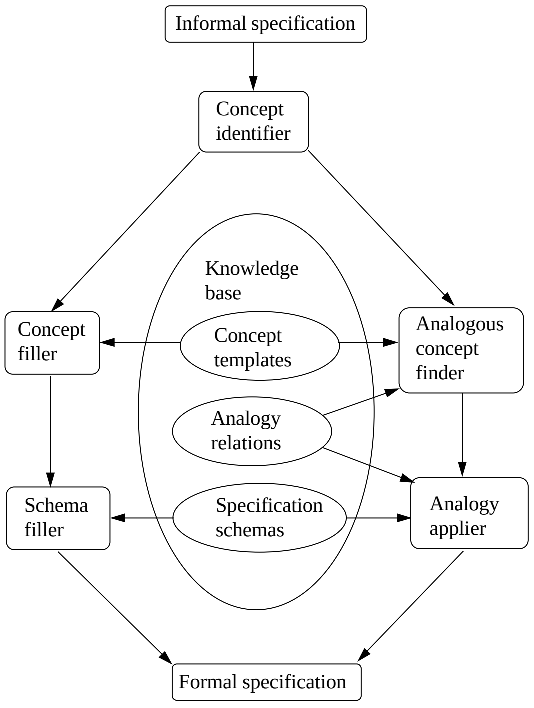

Figure 1: Architecture of the SPECIFIER System

to retrieve associated structure templates (represented as frames). These structure templates have slots that define the expected semantic form of the concepts and have associated rules that can be used to fill in the slots by using the informal specification. A set of rules is used to select specification schemas based on the identified concepts. The specification schemas are then filled by using the rules associated with the slots and the structures of the concepts. Once filled, the specification schemas produce formal specifications in a Larch-like language [14].

As an example, the authors define the “minimise” concept by a structure template that has three slots:

1. The function to be minimised (e.g. the cost, length, time, etc).

2. The objects over which the minimisation is done (e.g. distance, lists).

3. The form of the output or minimised object.

Given this definition, an informal post-condition such as:

> "the output l is a valid assignment that minimises the cost"

can provide the three slots as "cost", "valid assignments", "a valid assignment". This structure is, in turn, used to fill in the post-condition slot of a specification schema to give a formal post-condition.

When a concept does not have an associated specification schema, the SPECIFIER attempts to use analogy to obtain a specification. It first finds those concepts that have specification schemas and are related to the target concept by predefined "analogy" relations. It then identifies a concept, called the source concept, that has the most similar structure to the target concept in terms of the syntax and the semantic roles of the slots. Once a suitable analogous concept is identified, the structure trees are matched to obtain a mapping from the analogous concept to the target concept. This mapping together with the predefined relationships between the concepts are applied to the source specification in an attempt to obtain the target specification.

The matching process adopted by the SPECIFIER aims to find a one-to-one relationship between the source and target structures. However, if this is not possible, an approximate analogy heuristic is used that allows partial matching between the structure templates.

The use of approximate analogy enables the system to utilise parts of previous specifications. SPECIFIER achieves this by ignoring those parts of the source specification that are not associated when finding the analogy. It also identifies those concepts and structures in the target structure that were ignored and attempts to develop specifications for them.

The SPECIFIER system has been demonstrated on small, but interesting examples. The authors have shown how a specification of the knapsack problem can be obtained by direct analogy from a specification of the minimum spanning tree problem given an initial relationship between the concepts of minimising and maximising. The authors also give an interesting illustration of how the specification of the maximum plateau problem can be obtained by approximate analogy with the specification of the minimum spanning tree problem.

2.2 The Requirements Apprentice

The Requirements Apprentice (RA) [29, 30] is a part of the Programmer's Apprentice project [31] which deals with the requirements phase of the software life cycle. Its main aim is to assist an analyst in the creation and modification of requirements. The RA's architecture consists of three modules:

1. CAKE: a knowledge representation and reasoning system.
2. The executive: which offers a means of communication between the analyst and the RA.
3. The cliché library: a declarative repository of proto-typical information relevant to requirements in general and to the domain of specific interest.

The cliché library plays a central role in the RA system. It is organised as an inheritance hierarchy where each cliché is represented by a frame with associated constraints. Each frame defines what must be provided, what might be provided, as well as default information for instantiating a cliché. The associated constraints define the kind of invariants and restrictions one can expect to hold for a cliché. In addition, a special kind of constraint (called a pre-condition in the RA) defines when the RA can assume that a cliché is applicable. The top level of the inheritance hierarchy is divided into three kinds of cliché :

- The environment. This includes clichés for different types of domains and concepts in the domains.
- The needs. The needs contain clichés that express the desires and objectives of the end-user.
- The system. This contains clichés that define proto-typical requirements for different classes of systems. For example, the information systems cliché attempts to capture a range of systems like library systems, stock control systems, etc.

Given such a cliché library, the RA is able to:

- Allow reusability when an analyst's term coincides with a predefined cliché.
- Identify incompleteness by highlighting information in a cliché that is required but has not been provided.
- Identify ambiguous terminology when several different clichés are thought to represent an analyst's concept.
- Detect certain kinds of conflicts by using CAKE to demonstrate that a constraint has been violated.

The RA does not require that the clichés are filled in any particular order, or are completely and unambiguously filled before continuing with other requirements. Instead, it maintains a list of pending issues; thereby allowing the analyst to encode the requirements as they occur and evolve.

2.3 The ARIES system

ARIES (Acquisition of Requirements and Incremental Evolution of Specifications) is part of a more general tool known as the Knowledge-Based Software Assistant [13]. ARIES aims to provide the analyst with a tool that can help to evaluate system requirements and codify them into formal specifications. The developers of ARIES identify four major activities during the requirements analysis phase: acquisition, reasoning, evolution and presentation. These activities are supported by an architecture that consists of:

- A modularised central knowledge base of requirements information.
- A single internal knowledge representation of the requirements.
- An interface to enable an analyst to view the internal knowledge.
- A tools to manage, analyze and support the evolution of requirements.

The requirements knowledge can be shared by different systems or analysts during the requirements and specification phase, and therefore needs to be modularised. In ARIES, this modularisation is achieved by allowing the use of workspaces and folders.

Whenever an analyst works on a problem, it is in the context of a particular workspace. Each workspace consists of a set of folders. A folder consists of declarations of types, instances, relations, events and invariants about a particular aspect of the requirements.

A key feature of ARIES is its capability for presenting different views of the requirements as they are being developed. To give a flavour of its capabilities, we now summarise the example given by the authors. The example involves the specification of an advanced automation system (AAS) [15]. The AAS is composed of several functional areas such as radar data processing and flight plan processing. One particular functional area is the transfer of control of the aircraft between controllers and facilities. This process is called the "handoff". The requirements concerning "handoff" are recorded in several folders, the main one is called *handoff*. The *handoff* folder contains declarations of the relations and events. For example, relations such as *handoff in progress* and events such as the *initialisation of the handoff* are specified by presenting their types and their relationships with other events. Thus, in defining the *initialisation of the handoff* event, an analyst notes that it is related to the current controller and the receiving controller. The analyst can also provide pre-conditions and post-conditions for events.

Once some initial information is specified, ARIES allows one to view the knowledge in different forms. Thus, the event *initialisation of the handoff* can be viewed in four different modes:

1. The *describe-object presentation* view displays the event as an object with its associated objects and their values.

2. The *event taxonomy presentation* view displays the event as a graph of related objects.

3. The *English paraphrase presentation* outputs an informal description of the event.

4. The *formal presentation* outputs the formal specification of the event in the ERSLA specification language [39].

There are several problems in achieving these modes of presentation that are worth mentioning:

1. Intractable computations. Some automatic computations such as the state-transition presentation, are intractable in the ARIES system. This is particularly due to the difficulty of deriving pre-conditions and post-conditions for arbitrary events.

2. Incompatibility. Incompatibility arises when an internal representation makes use of constructs not present in the specialised presentation.

3. Ambiguity and incompleteness. Sometimes objects are introduced into the ARIES knowledge without specifying their category (i.e., type, class). This incompleteness is a problem for specialised categories. Two methods are used to resolve this problem depending on the specialised representation being employed. The first method is to omit from the presentation those object that are not completely known. This method is used in the type taxonomy representation. The second method is to assign default categories to the objects to enable their presentation.

When reviewing the requirements, an analyst may detect ambiguities, incompleteness or some missing requirements. The analyst can perform the appropriate modifications using the evolution transformations provided by ARIES. Once an analyst selects one of the 180 available transformations, ARIES guides the analyst through the changes in a manner that avoids introducing ambiguities or inconsistency.

3 Tools based on Natural Language Analysis

In this section, we describe some of the systems that are based primarily on the use of techniques from natural language understanding.

3.1 The NL2ACTL System

The system NL2ACTL aims to translate natural language sentences, written to express properties of a reactive system, to statements of an action based temporal logic. It takes each sentence, parses it, and attempts to complete it by identifying any implicit information that is required to produce a well-formed expression in the action based logic (called ACTL [26]). As an example, consider the following sentence (taken from [11]):

“It is always possible to insert a coin.”

Based on the presence of “always” in this sentence, NL2ACTL produces an ACTL expression that is equivalent to the following completed or extended sentence:

> “For all states there exists a computation path starting with the action coin.”

In general, a sentence may not have enough information to produce a well-formed ACTL expression. Thus, a sentence may omit information about when an event occurs, or about the future truth of an expression. When it is unable to identify such information, which may be present implicitly, NL2ACTL requests the information from the user. As an example, the authors give the following sentence:

“It is possible to insert a coin.”

When faced with this sentence, the system responds by asking the user to specify whether this will be “soon” or “eventually”.

There are two important contributions made by the work on NL2ACTL. First, NL2ACTL shows that it is possible to utilise existing natural language processing tools (a system known as PGDE [22]) to develop grammars that are useful for analysing English sentences and for producing formal specifications in a given domain. Second, NL2ACTL demonstrates that when there is a specific application domain and target formal specification language in mind, one can develop a system that can help to identify incompleteness at a detailed level.

The main weakness of the work on NL2ACTL is that the grammar developed may be too specific to the domain of application. That is, the grammar used by NL2ACTL is not a general grammar for English nor is it based on any semantic theory of language. Instead, the grammar has been developed in a bottom up fashion based on the kind of sentences that the authors expect users to provide when describing reactive systems. In particular, the authors obtained their initial grammar by first examining the kind of sentences that arise in a vending machine application, and then manually translating the sentences to ACTL formulae. Based on this experience, the grammar was developed to predict the implicit information and includes attached semantic functions to obtain the ACTL expressions.

3.2 The FORSEN System

The aim of the FORSEN (**For**mal **S**pecifications from **En**glish) system [37] is to assist in the formalisation of informal specifications. It is based on the view that much of the work in the area of natural language understanding can be utilised to aid the production of formal specifications. In particular, much attention has focused on the problem of handling ambiguities and quantification. For example Warren and Pereira's [38] system for natural language queries for a geographical database and McCord's [23] work for a student database both tackle these problems. The architecture of the FORSEN system is illustrated by figure 2.

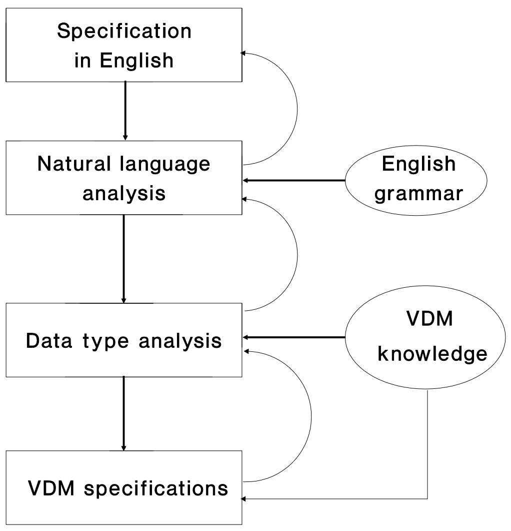

Figure 2: Architecture of the FORSEN system

As this figure shows, the FORSEN system is an interactive one. It does not assume that the original specification is complete or that it is consistent. The first phase of the FORSEN system is the analysis of the English requirements. This analysis is done sentence by sentence. If a sentence is ambiguous, it displays several alternative interpretations. The user is then required to select the intended meaning. At the end of the first phase, each sentence has a single associated meaning (represented in McCord's logical form language [23]).

The second phase of the FORSEN system identifies an entity relationship model. The entities and the relations are extracted from the nouns and verbs of the logical forms. The quantifiers defined by the logical forms are used to suggest default values for the degrees of the relations. This intermediate phase, between the informal specifications and the formal ones, is desirable for several reasons:

1. Entity relationship models offer a more natural visual view of the relationships than predicates nested in logical expressions. In particular, an analyst is used to identifying missing entities or relationships in entity relationship diagrams.

2. It enables the use of previous research that translates semi-formal representations to formal representations (e.g.[9, 27]).

The developed entity relationship model is then translated to a VDM data type. The last phase of the FORSEN system is the generation of VDM specifications. FORSEN generates specifications by filling in pre-defined schemas for a common range of operation specifications such as adding items, deleting items, and listing items that satisfy some conditions.

The main strengths of FORSEN are its ability to identify ambiguities in English specifications, and its capability for producing entity relationship models. Both these capabilities are produced by using general natural language analysis techniques and, unlike NL2ACTL, are not the result of a domain specific grammar. The main weakness of the FORSEN system is that it only generates a limited range of common specifications based on pre-defined schemas.

4 Discussion and Conclusion

As the above summaries suggest, each of the approaches has its own strengths and weaknesses. In this section we attempt to bring together the contributions made by each system and propose a more complete architecture for a system that aids the development of formal specifications.

First, section 4.1 begins by contrasting the contributions made by each of the systems. Then, section 4.2 proposes an architecture that aims to build upon the strengths of the existing systems and attempts to provide more complete support for the development of formal specifications.

4.1 Contributions made by the systems

In order to obtain a coherent picture of the current capabilities of the above systems, we present the main contributions of each system with respect to:

- The position of the system within the requirements life cycle.
- The identification of incompleteness, inconsistencies and ambiguities.
- The kind of specifications they produce.

4.1.1 Position within the requirements life cycle

Before positioning the various systems within the requirements life cycle, it is worth reviewing the main phases of the life cyle.[2] The cycle normally begins with the elicitation phase. This usually involves the use of interviews, questionnaires and consultation of the documentation. When 'enough' information is gathered, a document is produced that constitutes the requirements definition. This document usually represents the basis of a contract between the client and the analyst. Provided a system is considered feasible, a more detailed document, called the requirements specification, is written to define the expected services, constraints and limitations of the proposed system. This requirements specification provides a basis for developing a software specification that is meant to be an abstract description of the software design. Although, the software specification can be written in various notations, this paper focuses on systems for producing formal specifications.

The systems examined in this paper offer only a limited amount of support for the requirements elicitation phase. Thus, they do not offer much help in the acquisition of domain knowledge (to the same extent, as say KADS [33] does) or in the identification and maintenance of stakeholders' views. The knowledge based systems, do however, offer some help: when a new system is in the same category as a previously 'encoded' system, their attempt to reuse the structures of the previous system (e.g. via a cliché in RA) can lead to the elicitation of additional requirements. In contrast, the systems based on natural language processing only help with elicitation at a more detailed level. That is, the detection of ambiguities or lack of adequate information to obtain a well-formed logical statement can lead to prompts for further

[2]Requirements terminology differs in the literature. Here we adopt that used in [34].

information. For example, when FORSEN is presented with the following sentence [37]:

“The pilot draws the tracks of the route on the map”

It notices that the sentence is potentially ambiguous and asks the analyst to select an alternative, unambiguous form.

A number of systems aid the transition from an informal requirements specification to a formal specification. The SPECIFIER takes natural language descriptions of pre and post-conditions of operations and uses keyword matching to identify the concepts. The concepts then lead to the templates that result in the formal specifications (as described in section 2.1). Keyword matching, which includes ignoring quantifiers and prepositions, is not a proper basis for obtaining the intended semantics of the specification. That is, post-conditions with very different quantifiers could lead to the same specification. In contrast, the systems NL2ACTL and FORSEN, use a grammar as a basis for taking English sentences as input and are able to produce formal specifications. Both, however, are limited in the kind of requirements that can be formalised. The FORSEN system is limited by the problems of handling conjunctions and pronouns that it inherits from natural language analysis. The NL2ACTL system is limited by the fact that it uses a domain specific grammar instead of a general grammar based on linguistic theory.

4.1.2 Identification of incompleteness, inconsistencies and ambiguities

The systems reviewed adopt a rich variety of methods for identifying ambiguities and incompleteness. The FORSEN system uses natural language analysis techniques to identify ambiguities. Users may also detect some incompleteness by examining the generated ER model. For example, in a case study concerned with a flight planning databases system [37, p759], the FORSEN system analysed an English specification and produced an ER model. Although a majority of the ER model it produced was correct, it omitted a relationship between two entities: ‘waypoints’ and ‘route’. This omission was spotted on the ER model and an additional sentence expressing the relationship “a route is composed of waypoints” was added. FORSEN, however, adopts only one view of the requirements. In contrast, ARIES uses many views. These different views increase a user’s chance of spotting inconsistencies and incompleteness. In addition, ARIES is able to perform consistency checks that ensure a well-formed internal representation.

The SPECIFIER's use of analogy is perhaps the most novel and interesting approach to detecting incompleteness and redundancy. Concepts that are present in the analogous specification but missing in the target specification lead to possible additions to the target specification. Likewise, concepts that are present in the target specification but missing in an analogous specification may be redundant.

In general, all the systems surveyed are weak at detecting inconsistencies that are deeper than simple argument and type checking. Only the RA system uses automatic proof procedures in an attempt to identify constraints that are violated. None of them, however, attempt to obtain logical consistency across views by using verification techniques.

4.1.3 Kind of specifications generated

The systems reviewed generate the specifications in a wide range of languages. The FORSEN system generates VDM specifications, the NL2ACTL system produces expressions in an action based temporal logic (ACTL), and the SPECIFIER produces specifications in a Larch-like formal language.

Semi-formal specifications are also recognised as important views of requirements. The FORSEN system produces entity relationship models and the ARIES system produces graphs of related objects (the event taxonomy presentation).

The range of problems that can be specified are, however, limited. The FORSEN system only generates a pre-defined range of common specifications while the NL2ACTL system is limited by the domain specific grammar that it adopts. Fortunately, the SPECIFIER system offers some hope in that the range of specifications generated may broaden as the base of analogous problems increases.

4.2 A new architecture for obtaining formal specifications

Given the strengths and weaknesses of the reviewed systems, what sort of system seems possible at present? Figure 3 proposes an architecture that is based on the strengths of the reviewed systems.[3]

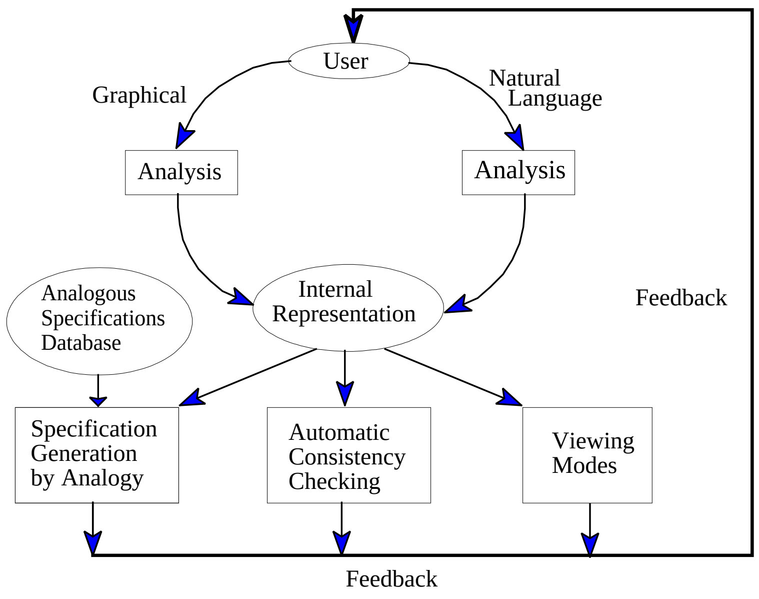
User
Graphical
Natural Language
Analysis
Analysis
Analogous Specifications Database
Internal Representation
Feedback
Specification Generation by Analogy
Automatic Consistency Checking
Viewing Modes
Feedback

Figure 3: Architecture of Future Systems

We believe that such a system would offer more complete support for developing formal specifications. First, an analyst could provide a range of input, from unstructured natural language to highly structured graphical objects. These different views of the functional requirements are mapped to a single internal representation. The process of mapping to a single consistent representation should lead to the identification of inconsistencies within the graphical and natural language components as well as between them. One can expect this because the different views often model parts that overlap and can therefore be used to cross-check each other once mapped to a common representation. In practice, appropriate general invariants, and heuristics for carrying out proofs need to be developed so that today's theorem provers can carry out automatic consistency checks. Failure to formally show or derive a common consistent view may also lead to the identification of incompleteness

[3]Given the focus of the paper, the figure omits other important aspects of requirements engineering such as history and viewpoint maintenance that are addressed by other authors.

or conflicts between the initial views.

The proposed architecture is not meant to implicitly suggest that all requirements can be mapped to a common representation, or that all aspects of the different views can be mapped to a common representation. Instead, given that we are concentrating on a subset of the requirements, most of which will eventually map to a single language (the formal language or eventually the implementation language), the architecture attempts to exploit the common ground covered by the different views of requirements that are available.

The use of analogy in the architecture has several merits (as demonstrated by the SPECIFIER system):

- Incomplete and redundant parts of formal specifications may be identified.
- Reusability of specifications should be increased.
- The range of specifications covered would improve incrementally as new specifications are added to the system.

To facilitate the use of such an architecture for large specifications, both the database of analogous specifications and the developing specification need to be organised in abstract hierarchies. This should be possible since one of the claimed benefits of formal methods is that if abstract, implicit specifications are adopted, one can obtain specifications that are more concise. These specifications can then be used as a basis for more detailed designs, which in turn, eventually lead to code. Thus, the architecture should inherit the kind of scalability and modularity benefits of formal specifications that have been demonstrated by the use of box methods in the cleanroom model [20].

This architecture is, of course, our vision of a system for aiding the development of formal specifications. However, this still begs the question:

Is it feasible ?

The architecture is based primarily on individual components borrowed from other systems that have been demonstrated by prototypes. So there is at least some evidence that the individual components are feasible. Thus graphical representations together with their analysis are used in ARIES, natural language techniques are adopted in FORSEN, and the use of analogy was demonstrated in the SPECIFIER. The extent to which the components can be combined as envisioned, depends on the feasibility of the interfaces between the different views and representations. Although the work on ARIES and FORSEN provide some evidence that both graphical and natural language can be mapped to a suitable logical representation, there is no practical evidence that one can proceed further and identify a consistent view from different views in a manner that can provide useful feedback for an analyst. There is, however, some research that shows how formal operation refinement principles can lead to proof obligations that help in identifying inconsistencies and incompleteness [1]. At a minimum, such work can be expected to lead to consistency checks that could be carried out across different views.

Finding a consistent view also requires reasoning about properties of objects that are represented in both the linguistic and the graphical domains. Bearing in mind that the different views may not correspond, there may be objects in one view that may not have related objects in another view. This necessitates an extended theory of reasoning that operates over domains whose objects are related by a partial mapping. Such theories are being developed by researchers interested in producing multi-modal explanations (e.g., [3]). One possible theory that is being considered for the architecture is an extension of Montague semantics [10] with a coordination structure that unifies the semantics of natural language and graphical expressions [32].

To conclude, this paper has attempted to bring together research aimed at developing tools for obtaining formal specifications from informal specifications. It proposes an architecture for aiding the development of formal specifications that builds on the strengths of previous systems. We hope that others can benefit by comparing their architectures with the one we envision and are developing.

Acknowledgements

We are grateful to the anonymous referees, whose comments have improved the discussion section of the paper.

APPENDIX A

The following summarises are meant to give a very brief indication of past and present research on systems that take informal specifications as input and attempt to aid the transition to a more precise specification. Readers should consult the given references for more details.

SAFE [5, 4]

Historically, SAFE was one of the first systems to attempt to produce formal specifications from informal specifications. It took as input, a procedure written in natural language and attempted to produce a formal and complete version of the procedure in a language known as AP2. SAFE was developed at a time when both formal methods and natural language analysis techniques were in their infancy.

The input therefore consisted of pre-parsed sentences and the output consisted of operational procedures instead of the kind of implicit specifications encouraged by today's formal methods.

SPAN [28]

The SPAN system aims to analyse functional specifications written in natural language in order to identify any ambiguities, inconsistencies and incompleteness.The mechanisms used to detect ambiguities are the use of a dictionary, a grammar and a set of specific rules based on experience. However, the system appears to rely heavily on the rules to propose inconsistencies and ambiguities. For example, one rule states that each sentence containing a preposition is ambiguous. The output produced by SPAN consists of case frames.

SPECIFIER [24]

The SPECIFIER takes a user's informal specification and identifies the concepts in the natural language text. The identified concepts are then used to retrieve associated structure templates and specification schemas. The specification schemas are filled to produce the formal specifications. If no pre-defined specification schemas are identified, it attempts to uses analogy to obtain specifications.

Requirements Apprentice [30]

The Requirements Apprentice (RA) uses the notion of cliché to define proto-typical knowledge and expectations for domains, needs, and types of systems. The RA's attempts at filling appropriate cliché's result in the identification of missing information, identification of ambiguities, and consistency checking. The adoption a list of pending issues enables an analyst to add requirements as they evolve. The RA also adopts a reasoning component that enables the identification of some types of conflicts.

ARIES [17, 16]

ARIES allows a user to represent information using different representations. It enables a user to view the knowledge using different views and therefore enables users to detect ambiguities, inconsistency, and incompleteness based on the different views.

NL2ACTL [11]

NL2ACTL is developed for reactive system. It takes informal English sentences as input and uses a domain specific grammar to generate statements in an action based specification language known as ACTL. The grammar is developed so as to enable the completion of "common" user sentences so that well-formed ACTL formulae are produced.

FORSEN [37]

FORSEN takes restricted English sentences as input and uses natural language understanding techniques in an attempt to identifies ambiguities and obtain a logical interpretation of the sentences. After the sentence have been analysed and their meaning established, it produces an entity relationship model. The entity relationship model is then used as a basis for producing VDM data types. These data types can then be used to instantiate a number of pre-defined schemas to produce a limited range of VDM specifications.

Macias and Pulmans' Approach [21]

Macias and Pulman propose an approach that encourages the production of natural language specifications that are simpler and more structured. They provide a structure-editor like system that enables one to provide sentences depending on key constructs such as “before”, “if”, “when”, etc. The sentences are then translated to a logical form using the natural language processing system known as CLARE [2]. As with the FORSEN system, when ambiguous sentences are given, several logical interpretations result and are presented to the user. Although, the system does not generate formal specifications, this research presents an interesting approach at bridging the gap between informal and formal specifications.

References

[1] M. Ainsworth, S. Riddle, and P. Wallis. Formal validation of viewpoint specifications. *Software Engineering Journal*, 11:58–66, January 1996.

[2] H. Alshawi et al. *CLARE: a Contextual Reasoning and Cooperative Response Framework for the Core Language Engine.* SRI International, December 1992.

[3] E. André and T. Rist. Referring to world objects with text and pictures. In *Proc. International Conference on Computational Linguistics*, pages 530–534, Kyoto, Japan, 1994. Association of Computational Linguistics.

[4] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, 1985.

[5] R. Balzer, N. Goldman, and D. Wile. Informality in program specification. *IEEE Transactions on Software Engineering*, SE-4(2):94–103, 1978.

[6] T.E. Bell, D.C. Bixler, and M.E. Dyer. An extendable approach to computer-aided software requirements. *IEEE Transactions on Software Engineering*, SE-3(1):49–60, 1977.

[7] B. Cohen. Justification of formal methods for system specification. *Software Engineering Journal*, 4(1):26–35, January 1989.

[8] D. Craigen, S. Gerhart, and T. Ralston. *An International Survey of Industrial Applications of Formal Methods, Vols 1 and 2.* National In-

stitute of Standards and Technology, U.S. Department of Commerce, March 1993.

[9] J. Dick and J. Loubersac. Integrating structured and formal methods: A visual approach to VDM. In *Third European Software Engineering Conference, LNCS 550*, pages 37–59, 1991.

[10] D. R. Dowty, R. E. Wall, and S. Peters. *Introduction to Montague semantics*, volume 11 of *Studies in Linguistics and Philosophy.* D. Reidel Publishing Company, P.O.Box 17, 3300 AA Dordrecht, Holland, 1981.

[11] A. Fantechi et al. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4(3):243–263, 1994.

[12] M.D. Fraser, K. Kumar, and V.K. Vaishnavi. Strategies for incorporating formal specifications in software development. *CACM*, 37:74–86, 1994.

[13] C. Green et al. Report on a knowledge-based software assistant. In C. Richard and R. Waters, editors, *Reading in Artificial Intelligence and Software Engineering.* Los Altos, CA: Morgan Kaufmann, 1986.

[14] J.V. Guttag, J.J. Horning, and J.M.Wing. *Larch in five easy pieces.* Digitial Equipment Corporation, Palo Alto, CA, 1985.

[15] V. Hunt and A. Zellweger. The FAA's advanced automation system: Strategies for future air traffic control systems. *IEEE Computer*, 20:19–32, February 1987.

[16] W.L. Johnson, K.M Benner, and D.R. Harris. Developing formal specifications from informal requirements. *IEEE Expert*, pages 82–90, August 1993.

[17] W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, 1992.

[18] C.B. Jones. *Systematic Software development using VDM.* Prentice Hall International, 1990.

[19] J. Kolodner. *Case-based reasoning.* Morgan Kaufmann Publishers, Inc., U.S.A, 1993.

[20] R.C. Linger. Cleanroom process model. *IEEE Software*, 11:50–58, March 1994.

[21] B. Macias and S.G. Pulman. A method for controlling the production of specifications in natural language. *The Computer Journal*, 38(4):310–318, 1995.

[22] M. Marino. PGDE: Process grammar development environments, user manual. Technical Report AITech TR1/92-PGDEUM, Pisa, 1992.

[23] M. McCord. Natural language processing in Prolog. In A.Walker, editor, *A logical approach to expert systems and natural language processing Knowledge systems and Prolog*, pages 391–402. Addison-Wesley Publishing company, 1990.

[24] K. Miriyala and M.T. Harandi. Automatic derivation of formal software specifications from informal descriptions. *IEEE Transactions on Software Engineering*, 17(10):1126–1142, 1991.

[25] P. Nico, J.V. Katwijk, and T. Hans. Applications and benefits of formal methods in software development. *Software Engineering Journal*, 7(5):335–346, September 1992.

[26] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transmission systems. In *Lecturere Notes in Computer Science 469*, pages 407–419. Springer-Verlag, 1991.

[27] F. Polack, M. Whiston, and K. Mander. The SAZ project: Integrating SSADM and Z. *Lecture Notes in Computer Science*, 670:541–557, 1993.

[28] S.G. Presland. *The analysis of natural language requirements documents.* PhD thesis, University of Liverpool, 1986.

[29] H.B. Reubenstein. *Automatic acquisition of evolving informal description.* PhD thesis, MIT Artificial Intelligence Laboratory, 1990.

[30] H.B. Reubenstein and R.C. Walters. The requirements apprentice: automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, 1991.

[31] C. Rich and R.C. Walters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, November 1988.

[32] J. S. Santana. Coordinated linguistic and graphical semantic units. Technical Report IIE/Salford-SS-19, University of Salford, Department of Computer and Mathematical Sciences, University of Salford, UK, May 1996.

[33] G. Schreiber, B.Wielinga, and J. Breuker. *KADS: A Principled Approach to Knowledge-Based Systems Developement.* Kluwer, 1993.

[34] I. Sommerville. *Software Engineering.* Addison-Wesley Publishing Company, 1992.

[35] J.M. Spivey. *The Z Notation: a reference manual.* Prentice Hall London, 1989.

[36] D. Teichroew and E.A. Hershey. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing system. *IEEE Transactions on Software Engineering*, SE-3(1):41–48, 1977.

[37] S. Vadera and F. Meziane. From English To Formal Specifications. *The Computer Journal*, 37(9):753–763, 1994.

[38] D.H.D. Warren and F.C.N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, 1982.

[39] G.B. Williams and J.J. Myers. Exploiting metamodel correspondences to provide paraphrasing capabilities for the KBSA concept demonstration project. In *Proceedings of the fifth annual RADC Knowledge-Based Software Assistant (KBSA) Conference*, pages 331–345, 1990.