

Formal Specification of Static Configuration Data

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science (Computer Science)

by

Patrick John Graydon

August 2006

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the
requirements for the degree of
Master of Science (Computer Science)

This thesis has been read and approved by the Examining Committee:

Thesis Advisor

Committee Chairman

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

August 2006

Abstract

A growing number of systems use static configuration data to adapt a standard package of hardware and software to the site at which it is deployed. When these systems are safety-related, errors in the configuration data can cause harm to humans or the environment. As part of a comprehensive effort to forestall such harm, static configuration data should be treated as an independent system component and subject to a complete lifecycle. This effort will include the production of a specification that serves as a contract between those who supply the data and those who design the applications that will consume it and as a guide for verification and validation activities.

In this thesis we enumerate the stakeholders in static configuration data, their needs, and the goals for static configuration data component specifications that come from these needs. We propose a general technique for writing formal specifications for static configuration data components. Such formal specifications are practical, and they enable the development and use of a valuable set of tools, including tools for editing, verifying, and validating data and building applications that consume or process the data.

To evaluate the practicality of formal specifications for static configuration data components and the power of the tools such specifications make possible, we created a concrete specification language for use with our technique, re-wrote a specification for an existing safety-related static configuration data component in our language, constructed a mechanical verification tool based on our language, and used this tool to find errors in a real, well-validated instance document for errors. Our evaluation efforts demonstrate that formal specifications for static configuration data are practical and permit the development of at least one valuable tool, the mechanical verification tool.

Acknowledgments

First, I thank John Knight and Elisabeth Strunk for their efforts to teach me how to do research and write about it, and for providing the foundations upon which this work was built. They have taught me what it is to work with bright people who deeply understand software engineering. Without their mentoring and assistance this work would not have been possible.

I thank William Greenwell for his help in coming to understand the MSAW system in context. I would also like to thank Westley Weimer and C. Michael Holloway for their help with background.

Finally, I thank my family for supporting me from afar while I focused on this work. Without their well-wishes, encouragement, and understanding I would not have completed this thesis.

Table of Contents

1. Introduction.....	1
1.1. The need to specify configuration data.....	1
1.2. What we must specify.....	3
1.2.1. A description of form.....	3
1.2.2. A description of required properties.....	3
1.2.3. A description of semantics.....	4
1.3. MSAW as an example of static configuration data.....	5
1.4. Summary of the approach.....	6
1.5. Thesis organization.....	8
2. Goals for data specifications.....	9
2.1. The stakeholders.....	9
2.1.1. The instance developer.....	9
2.1.2. Third party data providers.....	11
2.1.3. Instance verifiers.....	11
2.1.4. Instance validators.....	12
2.1.5. Regulatory agents.....	13
2.1.6. Application designers.....	13
2.1.7. Specification developers.....	15
2.1.8. Specification validators.....	16
2.2. Goals for static configuration data component specifications.....	16
3. Data specifications.....	21
3.1. The form of a specification.....	21
3.2. A description of form of the data.....	22
3.3. A description of the required properties of the data.....	23
3.4. Semantic description of the data.....	25
3.4.1. Ad-hoc natural language.....	25
3.4.2. Precise definitions.....	27
3.4.3. The CLEAR process.....	28
3.4.4. CLEAR and static configuration data component specifications.....	30
4. Text data specifications.....	33
4.1. Lexical specification.....	34
4.1.1. The importance of lexical structure specification.....	34
4.1.2. Describing lexical structure by describing a notional lexer.....	35
4.1.3. Character encoding declaration.....	36
4.1.4. Lexer states.....	37
4.1.5. Token declarations.....	40
4.1.6. Match declarations.....	40
4.1.7. Match precedence rules.....	41
4.1.8. Comment and white space declarations.....	43
4.2. Context-free syntax specification.....	44
4.3. Non-context-free syntax specification.....	45
4.3.1. Variables and types.....	48
4.3.2. Lists.....	48

4.3.3.	Characters and strings	49
4.3.4.	Arithmetic	50
4.3.5.	Built-in predicates	51
4.3.6.	User-defined predicates	52
4.3.7.	Unchecked prohibitions	53
4.4.	Designations and context	53
4.4.1.	Non-terminals are designated	54
4.4.2.	Designations	55
4.4.3.	The context specification	56
4.5.	Specification goals, revisited	57
5.	Tools for text data	59
5.1.	The tools we envision	59
5.1.1.	An instance document editing tool	59
5.1.2.	Tools for automatically creating instance documents	61
5.1.3.	A mechanical verification tool	62
5.1.4.	An instance document parser generator	63
5.1.5.	Specification checking tool	64
5.2.	Prototype toolset	65
5.2.1.	Mechanical verification tool	66
5.2.2.	Instance document parser generator	71
6.	Evaluation	74
6.1.	Choice of subject	74
6.2.	Evaluation goals	80
6.3.	Evaluation method	81
6.4.	Observations	82
6.4.1.	Lexical structure	83
6.4.2.	Context-free syntax	84
6.4.3.	Non-context-free syntax features	85
6.4.4.	Constraints on groups of values	87
6.4.5.	Non-syntactic requirements	88
6.4.6.	The form of the data	88
6.4.7.	Natural-language semantic descriptions	89
6.4.8.	Systematic specification	91
6.4.9.	Errors found	92
6.5.	Discussion	94
6.5.1.	Limited evaluation of the semantic portion of the specification	95
6.5.2.	The benefits of a formal specification	96
6.5.3.	External validity	98
7.	Related work	101
7.1.	Data as a separate system component	101
7.2.	The Role of Natural Language in a Software Product	102
7.3.	Specification and Analysis of Data for Safety-Critical Systems	103
7.4.	ASN.1	105
7.5.	XML and XML Schemas	107
7.6.	Constraint languages	108
8.	Conclusion	111

References.....	115
Appendix A. Specification language syntax.....	118
A.1. Specification language lexical structure	119
A.2. Specification language grammar.....	121
A.2.1. Lexical specification	121
A.2.2. Context-free syntax specification	123
A.2.3. Non-context-free syntax specification	125
A.2.4. Context specification	126
A.2.5. Regular expression.....	126
A.2.6. Predicate expression.....	127
A.2.7. Mixed string expression.....	129
Appendix B. Rewritten specification for MSAW site data files	130

List of Figures

Figure 1 - Configuration data, its specification, and consuming components.....	2
Figure 2 - The MSAW system’s site data.....	6
Figure 3 - A general technique and specialized languages	7
Figure 4 - Evaluation strategy.....	8
Figure 5 - Instance developers, verifiers, and validators	10
Figure 6 - Regulatory agent; application designer; spec. developer and validator	14
Figure 7 - The three components of a specification.....	22
Figure 8 - A specification for text-based data	33
Figure 9 - Example character encoding declaration	37
Figure 10 - Example of lexer state declaration and lexer state use.....	38
Figure 11 - Identifier declaration relying upon precedence rules	42
Figure 12 - Identifier declared unambiguously.....	42
Figure 13 - Example context-free syntax specification	44
Figure 14 - Example of a non-context-free syntax property.....	46
Figure 15 - Example predicate and prohibition	47
Figure 16 - Example positive existential constraint.....	47
Figure 17 - List notation and the “is” operator	48
Figure 18 - Example of built-in predicates	51
Figure 19 - Example of a predicate used to define a virtual value	52
Figure 20 - Example designation	56
Figure 21 - An instance document editor	60
Figure 22 - A data processing tool	61

Figure 23 - An instance document verification tool.....	63
Figure 24 - Components of the instance document verifier.....	67
Figure 25 - The non-context-free syntax checker	69
Figure 26 - Hidden existential quantification in Prolog.....	70
Figure 27 - Example MSAW table.....	76
Figure 28 - 3-range, 3-azimuth shape	79
Figure 29 - The octal literal non-terminal from our re-written specification.....	84

1. Introduction

1.1. The need to specify configuration data

A growing number of systems make use of static configuration data to adapt a standard package of hardware and software to the site at which it is deployed [1]. An example of such a system is the Federal Aviation Administration (FAA) Minimum Safe Altitude Warning (MSAW) system, which is deployed in air traffic control facilities to warn air traffic controllers when aircraft are, or soon will be, flying too close to terrain for safety. At each facility a *site data* file is used to adapt a standard software package to the site. The site data file includes measurements of the elevation of the terrain in the area served by the site, the name and position of airports in that area, the approach paths to those airports, and the configuration of the hardware on which MSAW will run.

When systems making use of static configuration data are safety-related, faults within the data might lead to system failures that endanger human life or the environment. In the Korean Air flight 801 accident, for example, an MSAW site data setting inhibited an alarm that would otherwise have warned the tower crew—and through them the flight crew—in time to prevent the crash of flight 801 and the deaths of 228 people [2, 3]. Because faults in configuration data have just as much potential to cause system failures as faults in the software do, we must treat the provision of configuration data with just as much care as we treat the construction of software. Unfortunately, there is evidence that developers do not treat configuration data with the same care as they treat software [4].

Developers of systems that rely on static configuration data frequently assume that since each deployment differs from previous deployments only by configuration data,

only these data need separate validation [4]. Despite this assumption, and despite the fact that the suppliers of configuration data are often not the developers who were responsible for building the software that will consume it, developers rarely define requirements for configuration data [4].

Storey and Faulkner have suggested that static configuration data should be treated as a separate system component and has proposed a lifecycle for its specification, design, creation, testing, and validation [5]. Treating data in this fashion, he argues, will help to ensure that data hazards are identified and appropriately dealt with. The lifecycle Storey proposes includes both the elicitation of safety requirements and the design of the data component. These constraints will need to be captured in a static configuration data component specification.

In this thesis we describe a technique that can be used to specify static configuration data. To those who will create configuration data component instances, this specification describes the assumptions that each instance must satisfy as shown in Figure 1. To those designing, constructing, verifying, or validating components that consume the data, on the other hand, this specification describes the assumptions about the data that their components can and cannot make. Since the configuration data component's specification

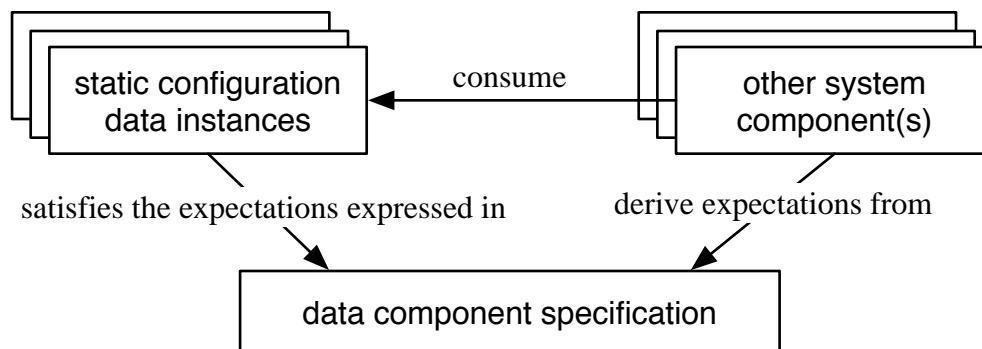


Figure 1 - Configuration data, its specification, and consuming components

serves as the reference for both configuration data component creators and the creators of components that consume that data, it must describe all of the attributes of the configuration data that matter to either party.

1.2. What we must specify

In order to serve as a contract between the providers and consumers of static configuration data, a specification must include a description of form, a description of the required properties of the data, and a description of the semantics of the data.

1.2.1. A description of form

A specification for a configuration data component must precisely describe the form that the data are to take. While a measured value, such as the altitude of terrain at a given coordinate, includes a measurement error that can be either tolerated or compensated for by the software, the form of configuration data must exactly match the software's expectations. If a 16-bit integer is given where a 32-bit floating-point number is expected, for example, the result would almost certainly be an error.

1.2.2. A description of required properties

A specification for a configuration data component should also describe any properties that consuming components will assume are satisfied. A runway length, for example, may need to be accurate to within one meter if the system is to perform as intended.

Some properties of interest are internal properties of the data. The MSAW site data file, for example, contains a set of airports in the site area that must be disjoint from the set of fix designations it also contains. Since all of the data needed to determine whether a given configuration data instance satisfies an internal property are contained

within that instance, mechanical verification tools can be used to check these properties. In order to enable such mechanical checking, and forestall misunderstandings, internal properties should be stated formally where possible.

Unfortunately, some of the properties we must specify will not be internal properties. The runway length property mentioned above, for example, constrains configuration data instances with respect to a real-world attribute of the entities they describe. These non-internal properties cannot be checked by a mechanical verification tool that has access only to a configuration data instance and the specification that it purports to adhere to. The developer of the configuration data instance must use a different technique to verify that these properties hold. If data used in a configuration data instance comes from a third-party supplier and no benchmark is available, for instance, the configuration data instance developer may need to acquire evidence from the supplier of the data. Although these non-internal properties cannot be checked mechanically, it is still vital that they be recorded, as the developers of configuration data instances will need to know that these properties must be satisfied.

1.2.3. A description of semantics

Descriptions of the form and required properties alone are insufficient to specify a static configuration data component. It is also imperative that readers of the specification come away with an adequate understanding of the meaning of each datum.

Suppose a static configuration component included the set of airports in a given region, and that it was necessary to have high confidence that no airport was omitted or non-airport included. Someone preparing an instance of that component would need a precise definition of “airport” in order to determine whether a given facility in the area

should be included. While such a definition may seem obvious, different systems may use distinct domain-specific definitions. Hospital helicopter pads and private airstrips both serve as the terminus of air flights, and one can imagine systems in the context of which these would be “airports” and systems in the context of which they would not be. An ideal specification language for describing text-based static configuration data would include features, guidance, or both for helping engineers to describe the problem-world concepts to which data correspond in a manner that helps to ensure the integrity of the description.

1.3. MSAW as an example of static configuration data

In order to illustrate the specification technique we present in this thesis we give examples from the FAA’s Minimum Safe Altitude Warning System. We have chosen MSAW as an example because it is a safety-related system that uses a substantial quantity of static configuration data. Though the static configuration data component of this system has been systematically documented, we were able to find several instances of information that needed to be in the specification but were not. While instances of this component are carefully prepared, we examined a sample instance and found violations of the properties expressed in the specification.

The MSAW system is intended to help avert a type of accident known as a Controlled Flight Into Terrain (CFIT) accident [2]. In such an accident, a qualified air crew flies a working aircraft into the ground, presumably unaware that they are doing so. The MSAW system, installed in air traffic control towers across the US and elsewhere, monitors aircraft and warns controllers when an aircraft is, or is predicted soon to be, flying too close to terrain for safety.

In order to perform its function, the MSAW software needs to know several things. The position of the aircraft is obviously needed, as is the altitude of the terrain. The locations of airports and their runways, and the approach paths to these, are also necessary, in order to avoid issuing false alerts for aircraft making a safe landing. Some details of the hardware on which the system will be running are needed, as these can vary from site to site. Some of these data, like the positions of the aircraft, are dynamic and are obtained from sensors at runtime. Others, like the terrain data, airport geometry, and hardware configuration are static. For each site, the FAA prepares a site data file that encodes the needed static configuration data.

1.4. Summary of the approach

In this thesis we examine the stakeholders in static configuration data and their needs and propose a general technique for writing static configuration data component specifications. Rather than calling for the use of one specification language to specify all static

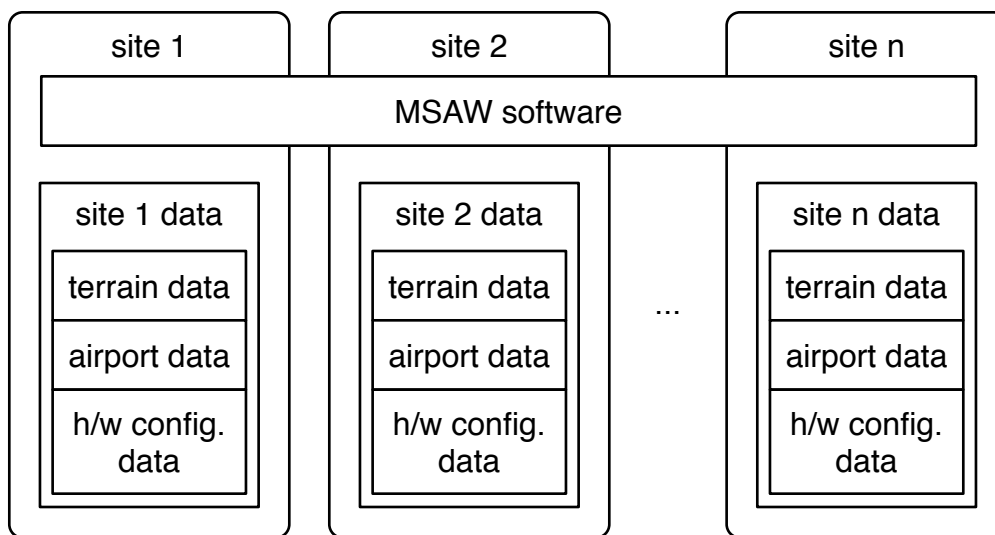


Figure 2 - The MSAW system's site data

configuration data components, our technique calls for writing the specification for each static configuration data component in a language specific to the general form of the data in that component as shown in Figure 3. Because the MSAW system that we will use as an example reads its static configuration data from a text file, we have created a specification language that specializes our technique to text-based data.

We claim that formal specification of static configuration data components is practical and that such specifications enable the development and use of a valuable set of tools. In order to evaluate the practicality of our specification technique and the power of the tools it makes possible, we re-wrote the specification of MSAW site data files using our specification language for text-based data, constructed a mechanical verification tool based on that language, and used this tool to check a sample site data file for errors. This approach evaluates our general specification technique indirectly through its text-specific instantiation as shown in Figure 4.

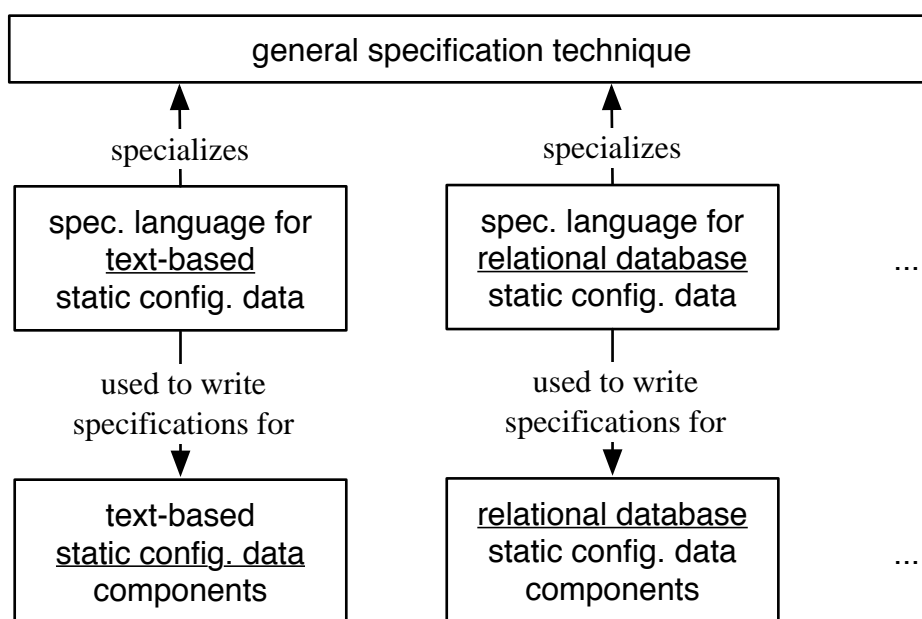


Figure 3 - A general technique and specialized languages

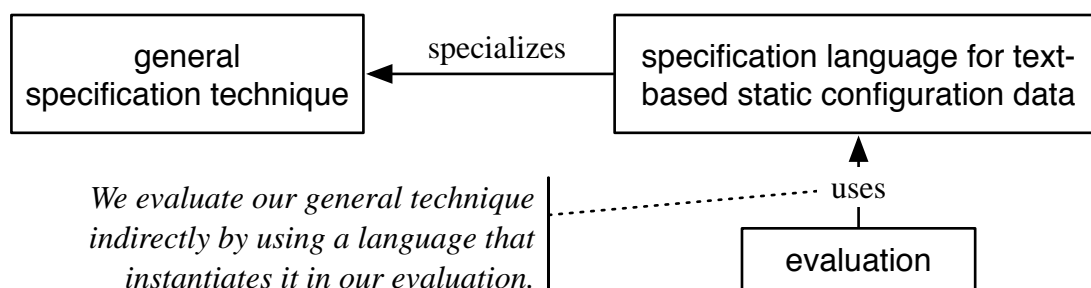


Figure 4 - Evaluation strategy

1.5. Thesis organization

In Chapter 2 we describe the goals we have for static configuration data component specifications. In Chapter 3 we elaborate our general technique for specifying such components. In Chapter 4 we discuss how we adapted our general technique to the specific problem of specifying a text-based static configuration data component. In Chapter 5 we discuss our vision for a tool set surrounding static configuration data components and present the tools that we have built at part of this work. In Chapter 6 we discuss our effort to evaluate our technique by using it to re-specify the MSAW site data file component and using our prototype tools with this specification to check an instance document for errors and comment on our findings. We discuss related work in Chapter 7 and conclude the thesis in Chapter 8.

2. Goals for data specifications

In this chapter we describe the goals that a static configuration data component specification should meet. In order to motivate these goals, we enumerate the stakeholders in the design, creation, verification, and validation of static configuration data components and their specifications. We examine the needs and motivation of each stakeholder, using examples from MSAW for illustration.

2.1. The stakeholders

There are several stakeholders with an interest in the design, creation, verification, and validation of static data components. These stakeholders are depicted in figures 5 and 6 and described in the following sections.

2.1.1. The instance developer

Instance developers are responsible for creating *instance documents*, each representing an instance of a static configuration data component. A team of FAA employees, for example, create MSAW site data files. To create an instance document, an instance developer must:

1. Understand what data are wanted and acquire them;
2. understand the properties that the data must have and obtain evidence that the data do in fact have these properties; and
3. Understand the form that the data must take and encode the data in that form.

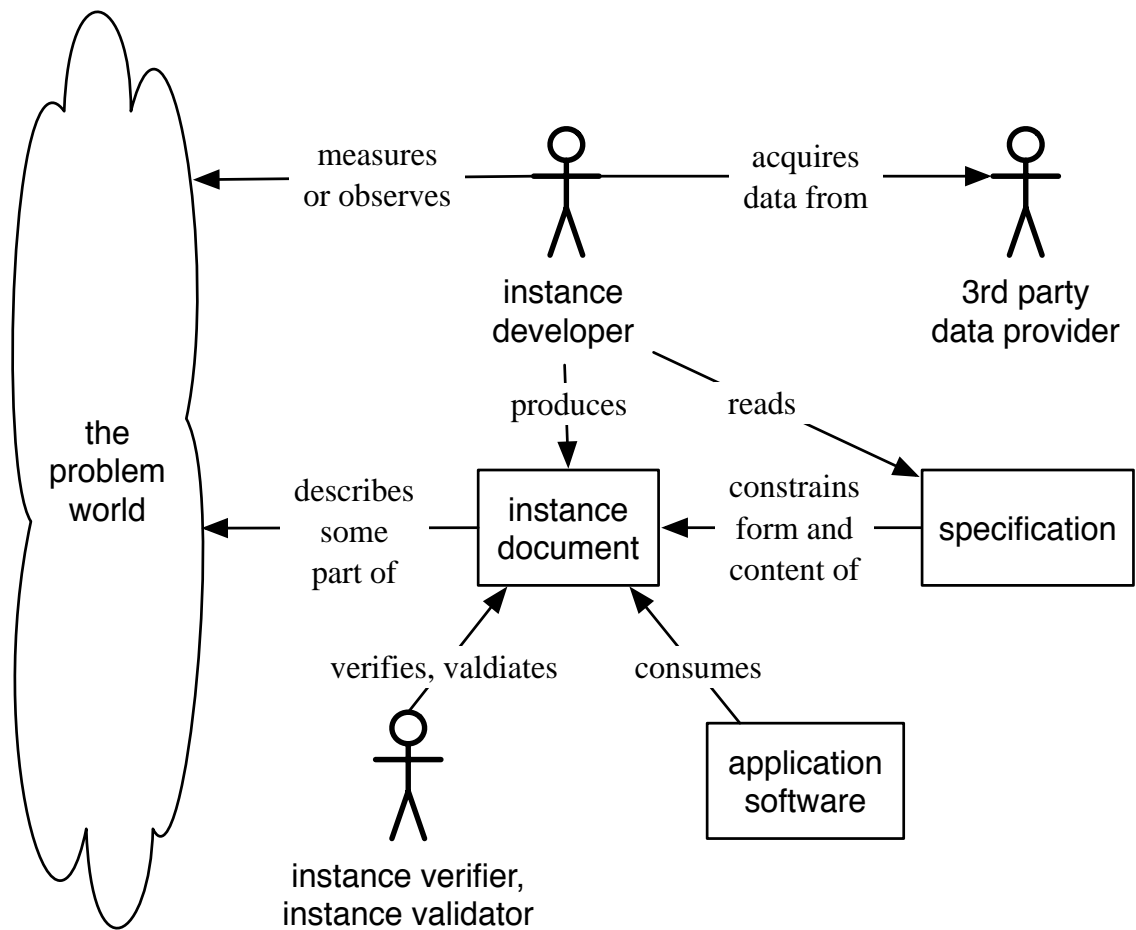


Figure 5 - Instance developers, verifiers, and validators

It is likely that the instance developer was not a part of the team that created the software [5] that will consume the instance document. This complicates the above procedure, as he or she must be taught the structure of the instance document to be produced, the properties it must have, and the correspondence of the individual data items in it to problem world entities. We expect the specification of the instance document format to be the primary reference for this information.

2.1.2. Third party data providers

In some cases instance developers will acquire data for an instance document by measuring or observing the problem world. An instance developer may, for example, create an instance document describing the configuration of existing hardware components by examining the configuration and keying his or her observations into a file. In other cases, however, instance developers will acquire data from *third party data providers*. The terrain data in MSAW site files, for example, is obtained from National Oceanographic and Atmospheric Administration (NOAA) surveys.

When the data an instance developer encodes is produced by another system, or perhaps by another organization (that might itself acquire data from yet another organization), verification and validation of the data might be complicated by lack of any standard against which to judge it [1]. In such cases, a developer may be forced to rely upon guarantees made by the provider of those data. We expect that each *source document* supplied by a third party data provider is described and constrained by a specification that will serve to document the form of the data, the provider's guarantees, and the meaning of the data.

2.1.3. Instance verifiers

Once an instance document is produced, it must be verified and validated before it can be placed into service, just as a software component must be. *Instance verifiers* are charged with ensuring that instance documents satisfy all of the checkable properties expressed in the specifications they purport to adhere to.

Mechanical verification tools should be the primary means of verifying that instance documents are encoded in the required form and possess all of the required inter-

nal properties. Just as theorem provers, static code analyzers, and model checkers can help software engineers find subtle defects that are easily missed, mechanical verification tools for instance documents can help instance verifiers find subtle faults in instance documents. A mechanical verification tool helped us to find a comment in a sample MSAW site data file that violated the specified comment syntax, for example. It is unlikely that a human reader would find such a fault unless he or she was deliberately looking for it, as the offending text begins with a character that is used to introduce comments and is clearly intended as a comment. However subtle, this fault has the potential to cause an error if the site data file is used with software that assumes it is well-formed.

Some other kinds of properties may be ‘tested’ by comparing them to measurements or observations of the problem world. An instance verifier could check the portion of an instance document describing which sensors are plugged into which ports of a computer, for example, by observing these connections and comparing them to what is encoded in the document.

2.1.4. Instance validators

Instance validators are charged with ensuring that given instance documents are valid for use in given system deployments. Validation activities will ensure, to the degree practicable, that instance documents meet all of their requirements.

Some of these requirements will be things that cannot be mechanically checked and cannot be reasonably tested. A property such as the accuracy of terrain data, for example, cannot be tested by comparison of the instance document to a table of measurements known to be accurate enough, because if measurements known to be accurate enough were available there would be no need for testing. In such cases the validators

may instead obtain evidence showing that the data were acquired in a manner that guarantees the required accuracy.

Inspection, review, and simulation activities may also be a part of validation. A reviewer making use of visualization tools to examine the MSAW site data file implicated in the Korean Air 801 accident, for example, may well have questioned settings that inhibited alerts in all but a 1 nautical mile ring around Guam [2, 3].

2.1.5. Regulatory agents

In some cases, those who deploy the system of which the static data component is a part will need to convince *regulatory agents* that the system they have deployed complies with applicable regulations. Regulations may require adherence to a process, in which case the instance developer, instance validator, and instance verifier will need to show that they have followed the appropriate processes. In other cases, the deployers may be required to generate a system assurance argument. The instance validator would prepare this argument by bringing together evidence from the system's construction and verification activities and artifacts to form a convincing informal argument that the system meets its safety or other regulated goals.

2.1.6. Application designers

Application designers are responsible for designing the software components that will form part of the system consuming the static configuration data component. Since their software will consume static configuration data, they must understand the form that the data will take. Since their software will attempt to deliver the system's functionality while adequately safeguarding humans and the environment, they must understand the require-

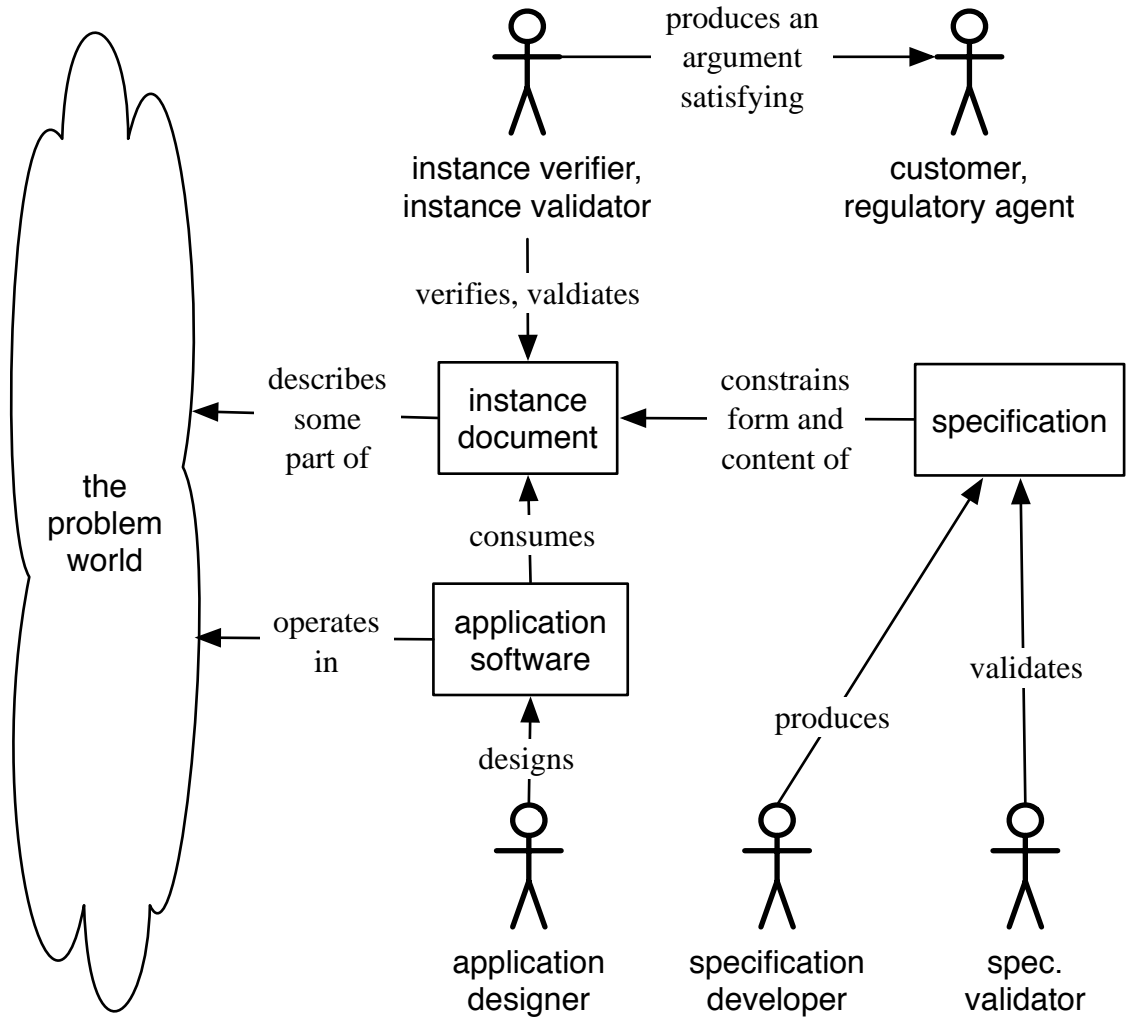


Figure 6 - Regulatory agent; application designer; spec. developer and validator

ments the data are required to satisfy so that they know what it is safe to assume about data instances.

The task of application designers is complicated by the fact that the software they create must be compatible with instance documents that have yet to be produced. It is not enough for the developers of application software to test their software with one—or even many—instance documents, as these documents may by chance happen to have a property that future instance documents are not guaranteed to have. Application designers must

design software that relies only on the properties of a static configuration data component that are guaranteed by that component's specification.

While application designers do not write static configuration data component specifications, they understand the needs of the software components they create, and so we expect their input to be invaluable to those who will. Application designers and instance developers will in many cases have conflicting needs that must be balanced in the specification: instance developers will want to guarantee as little as possible and avoid making promises that will be difficult or impossible to keep, while application designers will want guarantees that decrease the complexity of their software.

Consider, for example, a static configuration data component containing what its specification describes as a “table of the names of airports to which pilots can make instrument landing approaches.” Stated in this way, the specification is unimplementable. The instance developer cannot guarantee that any given airport will never be temporarily or permanently unavailable. On the other hand, if the description is changed to state that the table is a “table of the names of airports to which a pilot can *request to* make an instrument landing approach,” the application will need to include features to account for the case in which a pilot requests a landing at an airport that will not be available.

2.1.7. Specification developers

Specification developers are responsible for understanding the needs of the application designers and instance developers and crafting a specification for static configuration data components that can guarantee the properties that applications need them to guarantee and yet can be produced by instance developers. We expect that in most cases specification developers will be responsible for designing the form that data are to be encoded in.

2.1.8. Specification validators

Specification validators are responsible for ensuring that static configuration data component specifications meet the needs of the application and yet do not so constrain instance documents that they are impossible or impractical to produce. It seems likely that traditional software verification and validation practices such as reviews and inspections will form part of the specification validator's activities.

2.2. Goals for static configuration data component specifications

It is imperative that a specification for a static configuration data component capture all of that component's relevant requirements. If specification for a data component is to serve as the primary reference for instance developers, they must be able to rely on it being the most complete description available of what the instance documents they produce must be. A complete description is also essential if the instance verifier and instance validator are to use the specification as a checklist of requirements whose fulfillment must be demonstrated. At the same time, it is imperative that such a specification permit the use of mechanical verification tools. Such tools will be invaluable in helping instance verifiers to find subtle faults that might otherwise be missed. Considering the needs of the stakeholders described above, specification developers should:

- Write specifications formally,
- identify the problem world entities described by the data,
- describe the form of data in an appropriate manner,
- describe form and content separately,
- describe the instance document rather than how to read or write it, and
- enable the development of valuable tools.

We elaborate these goals in the following sections.

Write specifications formally. The specification language should allow and encourage the specification developer to describe the data as formally as is practicable. While natural language descriptions will be needed for some elements of the specification, such as identifying the real-world elements that data items correspond to, many others can be expressed formally. Doing so both eliminates the potential for ambiguity found in natural-language and even semi-formal descriptions and enables the use of mechanical verification tools. Researchers at the Naval Research Laboratory formalized a carefully prepared and reviewed semi-formal specification for the Operational Flight Program of the Navy's A-7, subjected it to mechanical analysis, and found 17 errors in the specification, illustrating the ability of formal languages and mechanical tools to find defects that human reviewers miss [6].

Identify the problem world entities described by the data. Instance documents encode information about the problem world. Terrain data is an obvious example, but even configuration values that control the mode of operation of software are fundamentally about the problem world, since they control the effect on the problem world that the system will ultimately have.

Instance developers must acquire the data to be encoded in the instance documents they create. In order to do this, they must know what data are needed, and where in an instance document it should be placed. Thus it is essential that a specification for a static configuration data component identify the entity in the problem world each field in an instance document describes. Moreover, the specification must communicate this information clearly, as misunderstandings can have disastrous consequences.

Describe the form of data in an appropriate manner. While it is possible to create a specification language with features that can specify any kind of data, specification languages tailored to particular kinds of data should allow the specification developer to more clearly and concisely describe the form of the data. A specification language geared to general binary files, for example, would require a specification developer describing an eXtensible Markup Language (XML) document to write or refer to a description of the general form of XML documents. A specification language for XML documents, in contrast, could permit the specification developer to describe the form of a document in terms of familiar XML features such as elements and attributes. Because a simpler, clearer description of form should decrease the likelihood of a reader misunderstanding it, specifications should be written in languages that allow specification developers to describe the form of particular instance documents in a manner appropriate for the general form of data in those documents.

Describe form and content separately. Part of what a tool used to prepare an instance document does is to transform data from one format to another. Developers writing such transformations must assure the users of their tools that the format of information has been changed appropriately while its meaning has remained the same. A facility for automatically or semi-automatically creating code to accomplish such transformations would increase this confidence. In order to make such tools possible, we believe that static configuration data component specifications should separate their descriptions of form and content to the extent possible.

Describe the instance document, not how to read or write it. The specification of a static configuration data component describes the form of the data that component con-

tains, and so defines what consumers of the data are expected to parse. Just as software developers are told to specify what and not how, writers of static configuration data component specifications should specify what the form of the data must be, not how to parse it. Specifying how the data is to be read would both unduly constrain the implementation choices available to application developers and potentially confuse instance developers, to whom it will be presented one further step removed. While developers of data processing tools may not find the additional constraints problematic, such implementation choice restrictions could pose problems for developers of embedded systems software that will consume instance documents, as such systems must often meet strict time, power, code size, and memory use constraints.

Enable the development of valuable tools. Just as proper tool support is essential in making software engineering practices practical for real-world use, tool support will be needed to support a static configuration data development process. A mechanical verification tool is an obvious and essential tool for instance verifiers, but there are other valuable tools made possible by formal specifications of static configuration data components. Instance developers, for example, could be supported by instance document editors equipped with a context-sensitive specification reference, check-as-you-type technology, visualization plug-ins, and the like. In cases where third party data is used, instance developers could be supported with data processing tools that automatically extract the required data from source documents and reformat it as needed. Application developers could be supported by automatic parser generators that take their input directly from the specification. Specification developers could even be supported by tools for creating, verifying,

and validating the specifications that they write. We will describe our vision for these tools in more detail in Chapter 5.

Developers of these tools will use static configuration data component specifications as primary references to the form and content of the instance documents their tools will work with. Because some of the tools must be relied upon to produce output with certain characteristics, tool developers will have many of the same needs as application designers.

Because many static configuration data components will make use of one of a small number of common general forms, it should be possible to develop tools for these general forms that read properly-formatted specifications and customize their behavior accordingly. While our primary concern is safety, decreasing the cost of valuable tools would clearly be beneficial for developers of static configuration data components and instances.

3. Data specifications

In this chapter we describe our technique for specifying static configuration data components. We describe the general form of a specification, including its provisions for describing form, required properties, and semantics. We then conclude with a discussion of how new techniques for improving the fidelity of communication during requirements capture can be applied to static configuration data component specification in order to improve the fidelity of descriptions of the meaning of the described data.

3.1. The form of a specification

In order to achieve the goals described in section 2.2, we propose a technique for specifying static configuration data components. Specifications made using our technique will contain three major components:

1. A formal specification of the form in which the data should be encoded.
2. A formal specification of the properties the data must have (to the extent that this is possible).
3. A semantic description of the data, in the form of a rigorous description of which problem-world entity each item of data represents.

This specification form is shown in Figure 7. The following sections describe each of these components in greater detail.

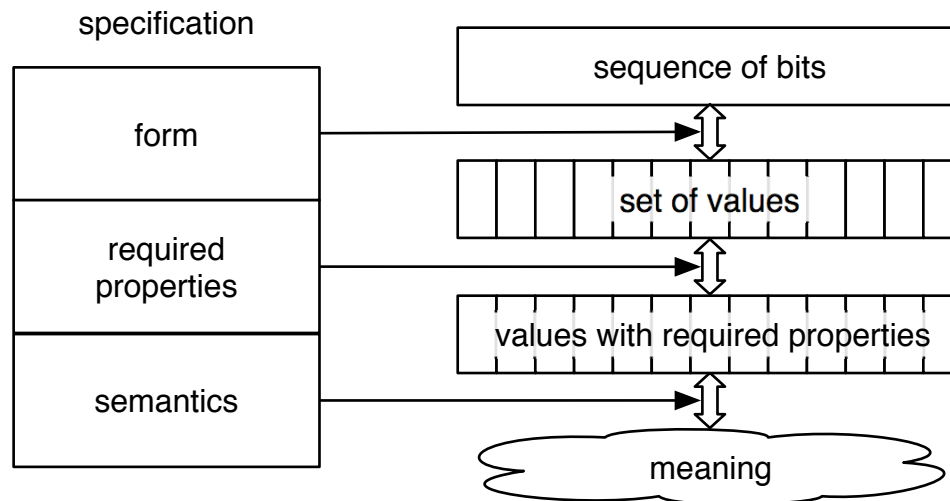


Figure 7 - The three components of a specification

3.2. A description of form of the data

A description of form in our technique is a formal statement of the relationship between the abstract values represented by a static configuration data component and the sequence of bits that these values are encoded in. If producers and consumers of static configuration data were to disagree on the format of that data, a system error might result. Suppose, for instance, that an instance developer assumed that rows in a “comma-separated value” table were to be separated by line-feed characters while an application developer assumed that they were separated by carriage-return and line-feed pairs: the consuming application may read a multi-row data instance as if all of the data in it was part of one long row. Describing the form of data formally will forestall such disagreements.

While it may be intuitively desirable to create one general-purpose specification language, it is not possible to create a language that can be used to specify data in all general forms. Consider a static configuration data component implemented using a commercial Relational DataBase Management System (RDBMS). The RDBMS vendor may not

share knowledge of how the fields and records in a database map to bits on disk with its customers if it feels that information constitutes a trade secret. In such a case, however, the only description of form needed may be the database schema and perhaps the version number of the particular version of the RDBMS being used. A form of our specification technique tailored to RDBMSs in general (or perhaps to the RDBMS in question) should be able to capture this information unambiguously and concisely.

A description of the form of a particular instance document should be written in a language with features appropriate for describing data of the general form used in that instance document. An instance document encoded in a form that is based on XML, for example, should be described using a language tailored to describing XML-based static configuration data components. While more general languages can be created, using specific languages reduces the complexity of the description of form. Because a simpler, clearer description of form should decrease the likelihood of a reader misunderstanding it, our specification technique should be adopted to use with general forms of interest through the provision of languages tailored to the task of describing specializations of those forms. In Chapter 4, we describe how our technique can be adapted to general text files.

3.3. A description of the required properties of the data

In addition to describing the form in which data should be encoded, a specification in our technique should include a description of the properties that must hold over each data instance. Some of these properties, like the description of form, address the syntax of the data. That is, they constitute part of a description of which bit-patterns would be considered well-formed instance documents and which wouldn't. An MSAW site data file, for

example, contains both a table of sensor names and a table of sensor locations and requires that the location of each sensor declared in the names table be defined in the locations table. This property is outside the scope of a discussion of the form in which the MSAW site file is encoded—although one could imagine a hierarchical rather than tabular form that would guarantee it—but is part of a comprehensive definition of the bit-patterns that constitute well-formed MSAW site files.

Other properties go beyond syntax to describe the set of well-formed instance documents that would be acceptable encodings of a given set of problem-world values. The existing MSAW site data file documentation, for example, requires that the table of sensors “include all the sensors defined for the site.” This restriction, like restrictions on measurement accuracy, timeliness, and so on, is not a matter of syntax: it would not be possible to determine whether a given site data file upheld or violated this restriction by examining the bits of the file in isolation. Yet presumably this property is described in the MSAW site data file documentation because the author of that documentation considered it important to the correct operation of the system. Because non-syntactic property requirements support assumptions that may be made by the systems that consume static configuration data, it is vital that they be recorded.

Some of the non-syntactic required properties of a static configuration data component may be difficult or impossible to express formally. We encourage specification developers to describe properties formally wherever doing so is practical. Because there will be cases in which a formal description is not practical, our specification technique permits informal descriptions and developers are encouraged to write these in natural language rather than let a property go unstated.

3.4. Semantic description of the data

A specification constructed using our technique must identify the problem-world entities to which data items in the described instance document correspond. Unfortunately, formal descriptions are constructed using primitive terms, such as variables, that have no inherent formal meaning [7, 8, 9]. A formal specification constrains relationships between the entities described by its formal terms, but cannot formally define the meaning of these terms. Consequently, a specification for a static configuration data component must include informal descriptions of the problem-world entities to which the primitive terms used in its formal descriptions correspond.

Specifications developed using our technique will encode these descriptions as a series of *designations*. These designations are similar in form and identical in purpose to the designations Zave and Jackson have proposed for requirements engineering in general: they are natural language statements identifying the part of the problem world that is represented by a given identifier in the specification [7, 10, 11].

3.4.1. Ad-hoc natural language

It is critical that these designations cause all readers of a specification to identify the problem world entities that the specification developer intended. Suppose, for example, that a specification designated a field as “the bearing, in degrees relative to North” from some point to another point. If an application designer interpreted “North” as magnetic North and an instance developer interpreted it as true North, or vice-versa, the bearing as read by the software might be considerably different than the instance developer intended, particularly in the northern latitudes. If the system depended upon the accuracy of the bearing in

order to meet its safety goals, the consequences of this misunderstanding could be disastrous.

Precise communication may be frustrated by the use of application-specific or domain-specific terminology in designations. Because a reader who is unfamiliar with a given term will know that he or she needs to seek an explanation of its meaning, the danger is mainly from terms that have meanings other than the intended meanings that readers may already be familiar with [8]. A reader, upon reading such a term, might associate with it a meaning that is different enough from the intended meaning to matter, yet similar enough not to alert the reader to the problem.

Consider, for example, the use of the word “airport” in the MSAW system, where it is intended to refer to a facility to which a pilot could declare an instrument landing approach. This meaning, while similar to the common meaning of a facility from which commercial passenger air-flight services operate, is subtly different.

If safety is threatened by incorrect values in a static configuration data instance, those values will need to be supplied with a high level of assurance. Because the correctness of the values in the context of their use in a system depends upon the instance developer, instance validator, and application designer sharing a common understanding of their meaning, the designations for them must reliably give rise to the same meaning in the minds of all readers. Unfortunately, ordinary language skills cannot reliably provide precise and accurate communication across domain boundaries [8]. Because ad-hoc natural language is not sufficient to ensure high-fidelity communication of the identity of a problem-world entity to which an instance document field corresponds, specification developers must be provided with aid that addresses this deficiency.

3.4.2. Precise definitions

Research by Wasson [8, 12, 13, 14, 15] addresses the problem of misunderstandings caused by the writer and reader of a natural-language software requirements statement having different conceptions of the meanings of the terms in which it is written. Some writers understand that the use of complex or domain-specific terminology can hinder communication with readers unfamiliar with the meanings of the terms in the problem domain and so provide a dictionary of these terms for use with their documents. Unfortunately, there is little guidance on how to prepare such dictionaries so as to minimize the probability that a reader will misunderstand the intended meaning of a term, and so these dictionaries are prepared in an ad-hoc manner. Wasson's work has identified the following classes of defects found in such ad hoc definitions:

- **Obscurity.** When a term is defined in terms of concepts more complicated than itself, the definition is not likely to give rise to the correct meaning in the mind of the reader. Wasson has cited the definition of "air" given in [8] as an example of this. It states that air is "the mixture of invisible odorless tasteless gasses (as nitrogen and oxygen) that surrounds the earth." While this statement is true, anyone needing a definition of air would likely not understand the terms being used to define it, and so would find this definition of little value.
- **Circularity.** When a term is defined in terms of other terms that are themselves defined in terms of the first term, the reader is given no way to come to understand one of the terms in the circle and so gain a starting point for understanding the remainder.
- **Otherwise non-predictiveness.** When a term is defined in terms of examples introduced with phrases such as "may", "might", "such-as", "especially" and the like, or

completed with blanket terms such as “etc.”, the reader is left with an incomplete understanding of the category of concepts associated with the term. With no explicit limits on which things are or are not included in the category of things indicated by a term, readers may have trouble determining whether some atypical items are or are not included in the category.

- **The provision of too much information.** When the provider of a definition includes too much information, he or she risks over-constraining the category of concepts to which the term refers.
- **The provision of too little information.** When the provider of a definition includes too little information, he or she risks under-constraining the category of concepts to which the term refers.
- **The provision of erroneous information.** When the provider of a definition includes erroneous information, he or she risks causing the reader to associate the defined term with the wrong category of concepts.

3.4.3. The CLEAR process

To improve the fidelity of requirements capture by addressing misunderstanding of terminology, Wasson has created the Cognitive Linguistic Elicitation And Representation (CLEAR) process. The CLEAR process guides the analyst through identifying phrases that need attention, gathering information about those phrases, and creating *explications* that will create in the reader an association between the explicated term and the category of concepts it is meant to represent. The process has these four phases:

1. **Selection.** In the selection phase, the analyst gathers the set of terms that will be investigated and possibly explicated by later phases. CLEAR is designed to operate on a corpus of existing written materials, so this phase involves a combination of automatic and manual inspection of those documents in order to isolate phrases that may need attention from those that need no treatment.
2. **Elicitation.** In the elicitation phase, the analyst gathers information about the phrases identified in the selection phase. To do this, the analyst asks domain experts to complete a collection form. The collection form solicits information about the attributes that identify members of the category represented by the target phrase, the relationship of that category to others, and other aspects of the category to be communicated. The analyst reads the input from domain experts contained on these collection forms and fills out a category record recording the information and insight gained from the collection and examination of the source documents.
3. **Representation.** In the representation phase, the analyst uses Wasson's Partial Reductive Paraphrase (PRP) process to create explications of the target phrases. Unlike traditional ad hoc definitions, the explications produced by the PRP process are by nature free from obscurity, circularity, and otherwise non-predictiveness. An analyst using the PRP process starts with a candidate definition written without the use of hedge words. He or she then progressively re-writes it using simpler terms until the explication uses only *relatively primitive* terms, whose meanings are likely to be agreed upon by all readers, or explicated phrases that are not themselves explicated, directly or indirectly, in terms of the phrase being explicated.

4. **Integration.** In the integration phase, the analyst combines the results from the preceding phases into a CLEAR Knowledge Base (CKB). The CKB contains the target phrases, their explications, relationships between the categories they identify, and additional notes made by the analyst. In addition to serving as a repository, the CKB permits the analyst to make observations about the terms and their meaning not recorded in the explication. Analysts may, for example, use the information in the CKB to compute a *risk index* for each term that quantifies the risk to understanding of the source material that would arise from a misunderstanding of that term.

3.4.4. CLEAR and static configuration data component specifications

In order to help ensure that all readers of a static configuration data component specification adequately understand the set of real-world entities that designations in that specification are intended to describe, we will use the CLEAR process in order to address lexical shortcomings in those designations. The input to the CLEAR process is a set of project documents written in natural language, so in this section we identify the text to be examined. The output of clear is a CKB, so in this section we will also describe how the contents of this CKB are to be used in a specification.

Each static configuration data component is designed by a specification developer to contain information provided by instance developers and consumed by one or more applications. Its requirements, then, are derived mainly from input from application designers, who know what information their software needs to have in order to satisfy its requirements, and in part from input from instance developers, who know what information can be obtained by a reasonable effort. Since the specification developer must use natural language to convey the identity of the problem-world entities to which the fields in

an instance document correspond, the terms that he or she must analyze and possibly treat are the names of those entities, if they are named, and the terms needed to describe them.

If the CLEAR process is used during elicitation of a system's requirements, the application designer will already possess a CKB containing some or all of the information needed to write designations for the fields of a static configuration data component. In such cases we expect that application developer's request will be phrased in terms of relatively primitive terms, terms that have already been explicated, or terms that can be investigated and explicated as requirements for the static configuration data component are gathered. The application developer will present the explications, and perhaps other details from the CKB, to the specification developer along with the request, giving the specification developer enough information to craft designations that are free of the kinds of lexical shortcomings that CLEAR was created to address.

If CLEAR has not been used as part of the application project, the specification developer will obtain a description of the required information in natural language. The specification developer must then analyze these natural-language requirements for terms that may need attention, investigate these terms, and craft explications for them as necessary, just as a requirements engineer making use of CLEAR would.

Our static data component specification need not include a CKB, although it will be informed by one. Since we are employing the CLEAR process in order to clarify the terms used to describe problem-world entities, the specification will need to capture the explications for these terms. In addition to designations, then, a specification should include explications of the terms used in those designations that are likely to be misunderstood. In order to enable tools such as an instance document editor to make these designa-

tions easily accessible by the instance developer, uses of an explicated term, either in a designation, informal natural-language requirement, or explication of another term, should be encoded as links to the designation of the term.

4. Text data specifications

As part of our evaluation of the specification technique described in Chapter 3, we have created a specification language based on this technique that is suitable for specifying text-based static configuration data components. In this chapter, we present our specification language in order to illustrate the technique. We describe some of the decisions we made during the design of our language and how these helped us to achieve the goals we enumerated in Chapter 2.

Specifications created using our technique contain a description of form, a description of required properties, and a semantic description of the data. In order to permit the specification developer to describe form, our specification language for text-based data includes a lexical specification and a context-free syntax specification in forms similar to

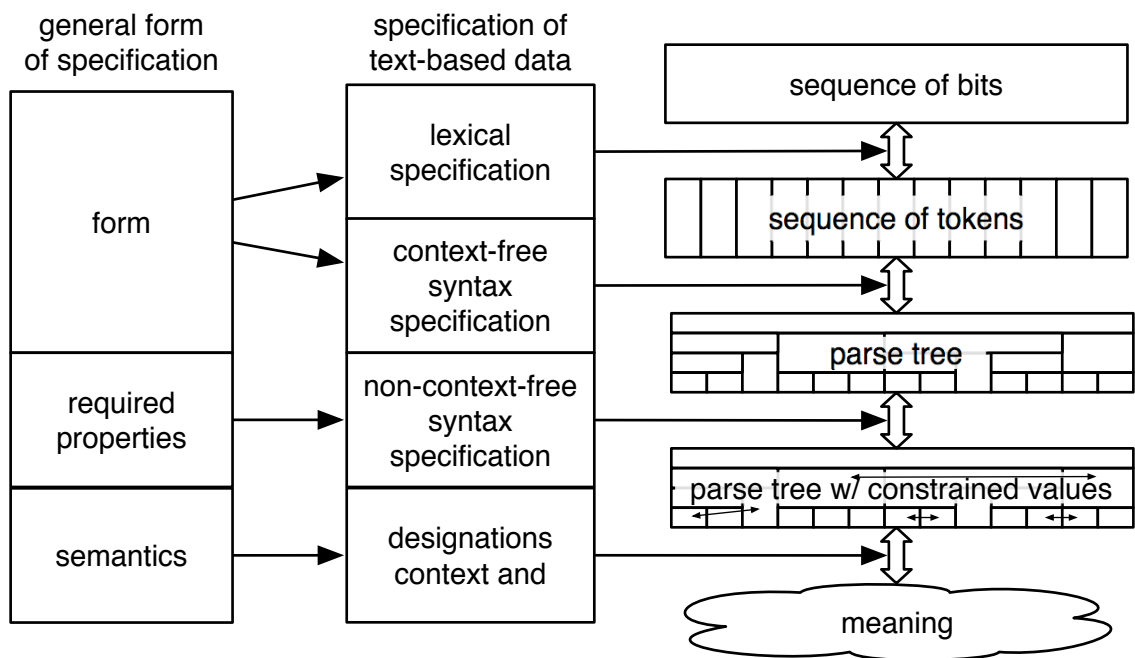


Figure 8 - A specification for text-based data

those given to lexer and parser generators. For describing the required properties of the data, we have a non-context-free syntax specification in a logic language inspired by Prolog. Finally, to give a semantic description of the data, we include a context description and a set of designations indicating which elements of the data correspond with which elements of the problem world.

4.1. Lexical specification

The *lexical specification* portion of our text-based static configuration data specification describes how the sequence of bits that comprise a given instance document should be converted into logical characters, which characters in that instance document represent comments or white space and should be ignored, and how the remainder of that instance document should be broken up into tokens.

4.1.1. The importance of lexical structure specification

It is imperative that instance documents strictly adhere to a precisely specified lexical structure. Lexical errors in instance documents are likely to send lexical analyzers or parsers attempting to read an instance document off course, thus causing system errors. Despite the importance of lexical specification, lexical structure is not always given adequate attention by specification developers.

In the original specification for MSAW site data files, for example, string literal tokens are described as sequences of “printable” ASCII characters. While this term is probably intended to refer to characters with values in the range [32, 126], neither the specification nor the X3.4 (ASCII) standard define this term [16]. While our attention to the use of the term printable may seem pedantic, there may be some cause for concern in disagreements as to whether character 32, the space, ought to be considered printable, as

its glyph is empty and its purpose is to advance the position of the following character. (Perhaps reflecting this disagreement, the X3.4 standard defines the space as a both a *graphic character* and a *control character*.)

The original MSAW specification also states that string literals may appear without quotes if they contain no spaces or semicolons. It is not clear how an instance document developer would encode a string containing a double quote character, as the specification does not mention the availability of any escape sequences.

In some cases, the original MSAW specification attempted to more formally define parts of the lexical structure. Longitude literals, for example, are described as being of the form `[DD]D:[M]M:[S]S[.S][C]`. Even this description is problematic, however, as it is not clear whether the hours portion of the longitude may be written with two digits. This distinction is important, as the sample MSAW site data file we examined contained some longitudes written with two-digit hours.

Given the deceptively simple appearance of the problem of lexical specification and the vital importance of getting it right, we have considered our specification language's features for lexical specification very carefully. These features, described in the sections below, are meant to allow specification developers to describe lexical structure precisely, simply, and in a manner suited to the automatic creation of lexical analyzers.

4.1.2. Describing lexical structure by describing a notional lexer

In order to describe the lexical structure of an instance document, a specification developer using our language would instead describe, in a form similar to the input given to automatic lexer generators, a notional lexer that would break an instance document into the appropriate tokens. Specifying the lexical structure in this form brings the risk that

developers of applications that consume instance documents will feel compelled to use a lexer matching the specification at the expense of one that better suits their needs. However, describing lexical structure indirectly by describing lexical analysis enables specification developers to reduce the complexity of the specification through lexer state declarations (as shown in section 4.1.4) and match precedence rules (as shown in section 4.1.7).

4.1.3. Character encoding declaration

While there are many text-based documents that use the ASCII character set, this character set is not universal. Languages such as Chinese and Japanese contain characters that do not appear in the ASCII character set. In order to support the specification of instance documents containing text in languages not supported by the ASCII character set our specification language describes instance documents as if composed of Unicode characters. While Unicode is not quite universal—it contains no glyphs for writing in Egyptian hieroglyphics, for example—it supports the majority of the world’s modern languages and is very widely used [17].

There are many ways to encode Unicode characters into bits. For example:

- The ASCII subset of Unicode can be encoded as ASCII text.
- The UTF-8 encoding maps each character to a sequence of one or more bytes, and is frequently used when the majority of characters are expected to be from the ASCII subset.
- The UTF-16 encoding maps 16-bit values to Unicode characters, and is frequently used for Asian text.

In our specification language the lexical specification begins with a *character encoding declaration* that specifies the set of character encodings that can be used to encode instance documents that comply to the specification. We expect that each instance document will be encoded using one encoding, and that consumers of the data will know in which encoding a particular instance document is coded before they attempt to read it. Where possible, static configuration data components should be constrained to exactly one encoding in order to simplify consuming software, but where necessary our specification language permits specifying multiple possible encodings.

4.1.4. Lexer states

The notional lexer described by our specification language is controlled by a state machine, just as the lexers generated by javacc and other tools are. The state of this machine determines which string patterns can be matched as which tokens, and can change after each match. While descriptions of multi-state lexers are arguably more difficult to write and understand than descriptions of single-state lexers, including a notion of lexer state in our specification language permits specification developers to more conveniently describe some lexical structures that occur in practice.

In the MSAW configuration file, for example, both the ‘#’ and ‘%’ characters introduce comments. The ‘#’ comments are user comments that are preserved by tools processing the file, while ‘%’ comments are inserted into the file by these tools. While both

```
# All instance documents compliant with this specification will
# be encoded in either UTF-8 or UTF-16. Software reading these
# instance documents will be assumed to accept documents encoded
# in either encoding unless otherwise specified.
character encoding is in { "UTF-8", "UTF-16" }.
```

Figure 9 - Example character encoding declaration

comments continue until the end of the line, the original MSAW site file specification indicates that comments introduced with ‘%’ may only appear at the beginning of the line. These comments can be described succinctly in our notation, as shown in Figure 10. Without lexer states, we would have had to describe these comments in the context-free syntax portion of our specification, greatly cluttering the resulting grammar.

Interestingly, while the original MSAW site file specification requires ‘%’ comments to begin a line, the sample site file we examined contained several comments violating this restriction. We found this fault while testing a prototype mechanical verification tool with our re-written specification, illustrating the power of the combination of a formal specification of form and mechanical verification tools. Presumably, since the sample file has been deployed and used, the MSAW system software recognizes the percent symbol in any column as indicating the start of a comment, and so the fault lies with the specification and not with the sample instance document. Regardless of where the fault lies, it is still important to find and correct such faults: if new software were written or existing software were modified to expect only the comment form described in the specification, an error would likely result.

```

states are { at_bol, not_at_bol }.
initial state is at_bol.

begin definition of token whitespace
  upon matching [ \t] state becomes not_at_bol.
  upon matching "\n" | "\r\n" state becomes at_bol.
end

begin definition of token comments
  upon matching "#" [^\r\n]* "\r"? "\n" state becomes at_bol.
  in { at_bol } upon matching "%" [^\r\n]* "\r"? "\n"
    state becomes at_bol.
end

```

Figure 10 - Example of lexer state declaration and lexer state use

It is very important for specification developers to avoid defining lexical forms which are so complicated that lexer states are required to describe them. Such forms require a human reader attempting to understand how an instance document will be broken up into tokens to understand the effect of lexer state on how the notional lexer operates, and so may lead to misunderstandings. A human reader attempting to understand how part of an MSAW site file will be broken up into tokens, for example, must consider the effect of previous parts of the file on lexer state in order to understand what the specification means with regard to the part in question. Simple changes to the form of MSAW site files, including permitting ‘%’ comments to begin on any column, could eliminate the need to use lexical states and thus remove some potential for misunderstanding. Our language provides lexer states, despite the dangers presented by forms complicated enough to require them, because we recognize that some specification developers will be describing instance document forms over which they have no control.

The lexer state declarations in our specification format require that the specification developer provide a list of the identifiers that will be used to name lexer states and identify the lexer’s initial state. The lexer’s state transitions are described alongside the patterns for the tokens the lexer recognizes. While it would be possible to infer that an identifier used where a lexer state is expected is a lexer state identifier, we require that specification developers explicitly list the names of the lexer states they will refer to in order to forestall subtle errors caused by misspellings. The form of identifiers is described in section A.1.

4.1.5. Token declarations

For each of the tokens the notional lexer will recognize, the lexer specification contains a token declaration block of the form shown in Figure 10. Each token declaration block contains one or more match declarations.

4.1.6. Match declarations

Each match declaration in a given token declaration describes a pattern of characters that the lexer could recognize as that token in one or more states as shown in Figure 10. The pattern of characters is specified in a regular expression notation similar to that used by lexer generators. (See section A.2.5 for the syntax of our notation.) In order to allow specification developers to arrange their specifications visually in a way that is easy for human readers to parse, we permit any amount of white space around the operators and literals in our regular expressions.

Sequences of literal characters can be expressed as literal strings encased in double quotes, and we provide an escape sequence mechanism to permit the writing of literal strings that contain characters such as double quote and newline. For a description of the permitted escape sequences, see section A.2.1.

Character classes are written in a notation similar to that used by Perl: (square) brackets surround the set of characters and ranges in the class. We provide an escape sequence mechanism for specifying characters such as right bracket and newline. We deliberately do not support the Perl's meta-character escape sequences—such as `\d` for ‘any digit’ and `\s` for ‘any white space character’—in order to force developers to explicitly list the characters they intend to include in the character class. As in Perl, a leading ‘`^`’

character negates the character class. For a more detailed description of character classes in our language, see section A.1.

If a particular pattern is only to be matched in some lexer states, the specification developer can precede a match declaration with a set of states in which the match can be made. If this set is omitted, the match is understood to be applicable in all states. If the lexer is to transition to a different state following a match, the specification developer may follow a match declaration with the name of the state to which the lexer should transition. If not, the developer must specify `state remains unchanged`.

4.1.7. Match precedence rules

Our language, like the input language used by many lexer generators, includes match precedence rules that disambiguate tokens declared so as to match the same strings. Using these rules, one could specify integer and floating-point literal tokens so that the string “123” would match either, and rely on the rules to note that strings matching the patterns of both tokens should always be considered integers.

Specification developers should avoid relying upon match precedence rules wherever practical, as specifications that rely on these rules might give rise to misunderstandings. A human reader trying to determine whether “123” is a floating-point literal might, for example, consult the definition of floating-point literal, fail to notice that a definition of integer literals also matches that string, and erroneously conclude that the string is a floating-point literal. A specification developer free to choose the form of an instance document could avoid this potential for confusion by requiring floating-point literals to include a decimal point so that no string matches the pattern for both tokens.

Although reliance upon match precedence rules should be avoided, it may not be possible for specification developers who do not have control over the form of an instance document to avoid doing so without severely complicating the match expressions. In Figure 12, for example, we show how a description of identifier tokens is complicated by an attempt to keep it from matching program keywords. In order to help the specification developer describe clearly and concisely lexical structures that cannot be specified practically without resort to match precedence rules, our specification language for text-based static configuration data components includes the following match precedence rules:

- When the same input could be matched as more than one token, the token that will be used matches the longest possible string. If, for example, the input contained “123.4;”, the lexer specification included match declarations that matched “123” to integer and “123.4” to floating-point tokens, no other match declaration matched

```
begin definition of token t_int
  upon matching "int" state remains unchanged.
end

begin definition of token t_identifier
  # Won't match "int", since t_int was declared first.
  upon matching [a-z]+ state remains unchanged.
end
```

Figure 11 - Identifier declaration relying upon precedence rules

```
begin definition of token t_identifier
  # Imagine this complicated by the need not to match
  # several longer keywords.
  upon matching ( ([a-hj-z]) or ("i" [a-mo-z]) or
    ("in" [a-su-z]) or ("int" [a-z]) ) [a-z]*
    state remains unchanged.
end
```

Figure 12 - Identifier declared unambiguously

- “123.4”, and no match declaration matched “123.4;”, the string “123.4” would be matched as a floating-point token.
- When two tokens *X* and *Y* can match a single string *S*, and *X* is declared before *Y* in the specification, then string *S* will never be matched as token *Y*. If, for example, the input contained “123;”, the lexer specification included match declarations that matched “123” to both integer and floating-point tokens, no match declarations matched “123;”, and the match declaration for the integer token preceded that for floating-point token, “123” would be matched as an integer.

4.1.8. Comment and white space declarations

The optional comment and white space declarations both identify sequences of characters that can safely be ignored in the discussion of context-free syntax and non-context-free syntax. When given, both declarations take the form of token declarations similar to that Figure 10, except with special names and some slight restrictions as described in section A.2.1. The notional lexer described by our specification will throw away any characters it matches to these patterns in the same manner that the lexer for a programming language such as C does. A specification without a comment declaration indicates that no strings are to be considered comments. Likewise, a specification without a white space declaration considers no strings to be white space.

While comments and white space in a document are both irrelevant to the discussion of higher-level syntax restrictions, they are not completely identical. An instance document editor that features syntax highlighting, for example, may color-code comments but not white space.

4.2. Context-free syntax specification

Our specification format permits the specification developer to describe the context-free syntax of the instance documents he or she is specifying as an $LL(k)$ grammar in Extended Backus-Naur Form (EBNF) as shown in Figure 13. If k is greater than 1, developers can specify the needed lookahead. Specification developers are encouraged to write $LL(1)$ grammars where possible, as tools that consume specifications are likely to make use of automatic parser generators and not all parser generators support lookahead. Those that do may not be able to generate reasonable parsers for any but low single-digit values of k .

While restricting the grammar to $LL(k)$ makes it easy for developers to write tools to process instance documents, it disallows the specification of many grammars, including some unambiguous grammars that are in LALR or $LR(0)$. Specification developers wishing to express such a grammar must instead specify a $LL(k)$ grammar that produces more strings than they are prepared to accept and then add prohibitions to the non-context-free syntax section of their specifications to disallow these strings.

```
begin context-free syntax specification
  lookahead by 1. # Grammar is  $LL(1)$ .
  start symbol is nt_document.

  # ...

  symbol nt_apt_definitions_table is begin
    derives t_apt_definitions nt_apt_definition_row+.
    designation is "An " (nt_apt_definitions_table) " represents "
      "the set of " (airport as "airports") " in the "
      (operating_area as "operating area") " that can be used to "
      " designate a " (known_airport_status as
      "known airport status") ".".
  end

  # ...
end
```

Figure 13 - Example context-free syntax specification

Compiler generators typically accept their input grammar in some form of BNF. Many compiler generators (including yacc and javacc) require the programmer to weave parse actions into the specification to specify what the program should do at each step in the parsing process. Others (including sablecc) instead assemble a parse tree in memory during a parse and require the programmer to write code to traverse the parse tree and act upon what is found in it. Our context-free-syntax specification is intended to allow a specification developer to express the form of an instance document, not how it should be parsed, and so has no need of a feature such as parse actions. However, while specification developers are not describing software that makes use of parsed values, they will be describing non-context-free portions of the syntax. Consequently, it is necessary for our specification language to provide a mechanism for binding parsed symbols and tokens to identifiers that can be used in the non-context-free syntax specification. We accomplish this through built-in predicates, which are described in section 4.3.5.

4.3. Non-context-free syntax specification

It is likely that specification developers will want to specify properties of data that cannot be expressed with an $LL(k)$ grammar, or, for that matter, in any context-free grammar. MSAW site data files, for example, contain a table defining the airports in the region and a table defining the fixes in the region, as shown in Figure 14. The three-letter names of the airports and fixes may appear in any order, but it is illegal for an airport name to appear in

the fix table and vice-versa. This disjointness property cannot be specified in a context-free grammar.

We could elect to express such constraints informally as comments, but relying upon human developers to manually verify instance document compliance with such constraints introduces the potential for human error, both in the understanding of precisely what the constraint is and in determining that a given document meets it. Indeed, the sample MSAW site file we examined contained violations of this property, despite the property being specified as a comment above both tables.

In order to permit the specification developer to formally express constraints that cannot be encoded in a context-free grammar, our specification language includes a non-context-free syntax specification. Others might call this non-context-free syntax ‘semantics’, but we reserve that term for describing the problem-world meaning of data, rather than which patterns of bits are legal and which are not.

In our specification language the non-context-free syntax of a language is defined by prohibition declarations. A prohibition declaration takes the form of a negative existential assertion across instance documents and is written in predicate logic as shown in

The "fix definitions" and "airport definitions" tables must not both contain the same name.

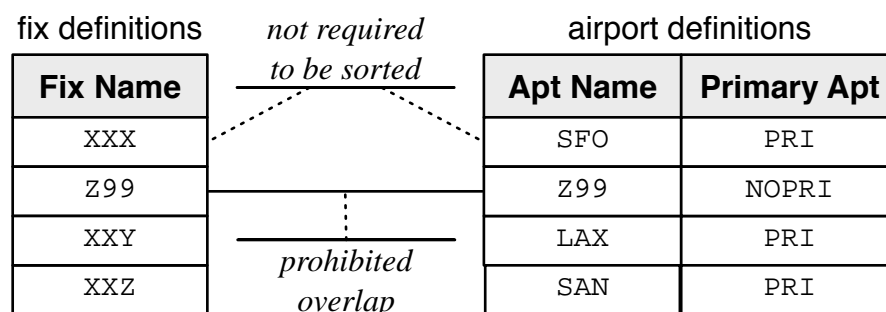


Figure 14 - Example of a non-context-free syntax property

Figure 15. While it is possible to write positive existential constraints (and thus negative universal constraints) as shown in Figure 16, specification developers should write negative existential constraints wherever possible. A verification tool such as the one we described in section 5.1.3 could only inform the user that a positive existential constraint has been validated, whereas for a negative existential constraint it could provide a counter-example from the instance document to ease debugging.

```

predicate fix_definition(ID, fixName) is begin
  there exists fnid, simid, fnsid such that
    nt_fix_definition_row(id, { fnid, simid }) and
    nt_fixDefinition_fixName(fnid, { fnsid }) and
    alphanum_t(fnsid, fixName)
end

prohibition no_invalid_fix_definitions is begin
  there exists id1, fn1 such that
    fix_definition(id1, fn1) and
    (
      (
        there exists id2 such that
          fix_definition(id2, fn1) and
          id1 <> id2
      ) or
      (
        there exists id2, ip2 such that
          apt_definition(id2, fn1, ip2)
      ) or
      (
        not
          (
            string_length_is(fn1, 3)
          )
      )
    )
end

```

Figure 15 - Example predicate and prohibition

```

prohibition require_at_least_one_fix is begin
  # Prohibit "not there exists ID and FN such that
  # fix_definition(ID, FN)" is equivalent to "require
  # the existence of some ID and FN such that
  # fix_definition(ID, FN)".
  not
    (
      there exists ID, FN such that
        fix_definition(ID, FN)
    )
end

```

Figure 16 - Example positive existential constraint

4.3.1. Variables and types

Our logic language supports variables that are either integers, floats, or lists, and provides some basic operations on these types. While it would be possible to describe all of the basic operators for these types using predicates, we think that specification developers will find specialized operators easier to read and understand, and so we have incorporated a set of operators borrowed from the Prolog logic language. Section 4.3.2 describes our language's features for describing lists, section 4.3.3 describes our language's features for describing characters and strings, and section 4.3.4 describes our language's arithmetic features.

4.3.2. Lists

Lists are introduced with (curly) braces, as shown in Figure 17, and may be composed hierarchically. While our list mechanism is intended to mirror that of Prolog, we use braces rather than (square) brackets in order to avoid the complication of the lexical structure of our language that would arise from using brackets for both character classes and lists. As in Prolog, the vertical line operator may appear before the last item in a list, signifying that the item represents the remainder of the list.

```

predicate list_length(list, length) is begin
    list_length_acc(list, 0, length)
end

predicate list_length_acc(list, acc, length) is begin
    there exists a, t, i such that
    ( list = {} and
      length = acc
    ) or
    ( list = { a | t } and
      i is (acc + 1) and
      list_length_acc(t, i, length)
    )
end

```

Figure 17 - List notation and the “is” operator

4.3.3. Characters and strings

In our specification language, characters are represented as integers with a value equal to the character's Unicode number, and strings are represented as lists of integers. We do not include a built-in string type or predicates for comparing strings because doing so would complicate the language and could lead to subtle errors.

Consider, for example, the complexity involved in specifying a predicate that would indicate that string *A* should be sorted after string *B*: the sort order of strings varies by language and culture. A similar difficulty arises with numbers, as some languages use ‘.’ to separate the whole and fractional parts of a number and ‘,’ to indicate the thousands place, millions place, and so on, while other languages use ‘,’ to separate the whole and fractional parts and ‘.’ to indicate places.

To account for these language and culture specific differences, our specification language would need to include a representation of language and culture. Such a specification language accounting for language and culture should have semantics compatible with the semantics of the programming language used by application developers, as programmers frequently use conversion functions like the C standard library's `atoi` without fully considering the implications of their choice.

A carefully thought out language and culture mechanism would, unfortunately, complicate the specification language. Rather than include such a feature in our language, we instead call upon specification developers to create data forms that are invariant across language and culture wherever practicable. Lists of strings, for example, should either be unordered or else ordered in a way that can easily be specified formally, such as ascending lexical order according to Unicode character value (which may not be dictionary order in

all languages). In no case should a list of strings be required to be sorted by an informally-defined order, as different readers might make different assumptions about what is meant.

4.3.4. Arithmetic

We support integer arithmetic using the keyword `is`, which we have also adopted from Prolog. The `is` keyword introduces an algebraic expression that gives the value of a variable as shown in Figure 17. We support the addition, subtraction, multiplication, division, and exponentiation operators.

The integer and floating-point types in our language are not bounded. That is, a specification developer can write arithmetic expressions as if integer variables could represent any variable in \mathbf{Z} and floating-point variables any variable in \mathbf{R} . Literals are, of course, restricted to a subset of \mathbf{Z} and \mathbf{Q} , respectively, as it is not possible for specifications of finite length to be able to encode all values from \mathbf{Z} and \mathbf{R} . Likewise, no verification tool could perform the computations necessary to check properties over all of \mathbf{Z} or \mathbf{R} on a machine with finite memory. We expect that mechanical verification tools will either use typical fixed-size representations of integer and floating-point types and fail to verify properties that cause an over- or under-flow during checking, or else use variable-length representations and fail if checking causes the available memory to be exhausted.

While we provide some floating-point support, we do not guarantee precise floating-point semantics. Our specification language is not a programming language, and so does not specify an order of operations for performing the calculations necessary to check properties. Because the result of a floating-point calculation may vary as the order of its operations varies in ways that would not affect true real-number operations, floating-point operators in our language should only be considered approximate. We provide imprecise

floating-point support because it may be useful for expressing reasonableness properties. Specification developers requiring more precision must instead use fixed-point data forms and express relationships over these using integer arithmetic expressions.

4.3.5. Built-in predicates

In our specification language there are built-in predicates that can be referred to in prohibition declarations and user-defined predicate declarations (described in the next section) that are provided automatically by the system. For each non-terminal named x in the grammar specified in the context-free syntax section of the specification there is a corresponding built-in predicate $x(id, productKeyList)$ as shown in Figure 18. This predicate is true with respect to a given instance document if and only if id corresponds to a portion of the instance document represented by the non-terminal x that is composed of a sequence of terminals and non-terminals whose identifiers match those in $productKeyList$.

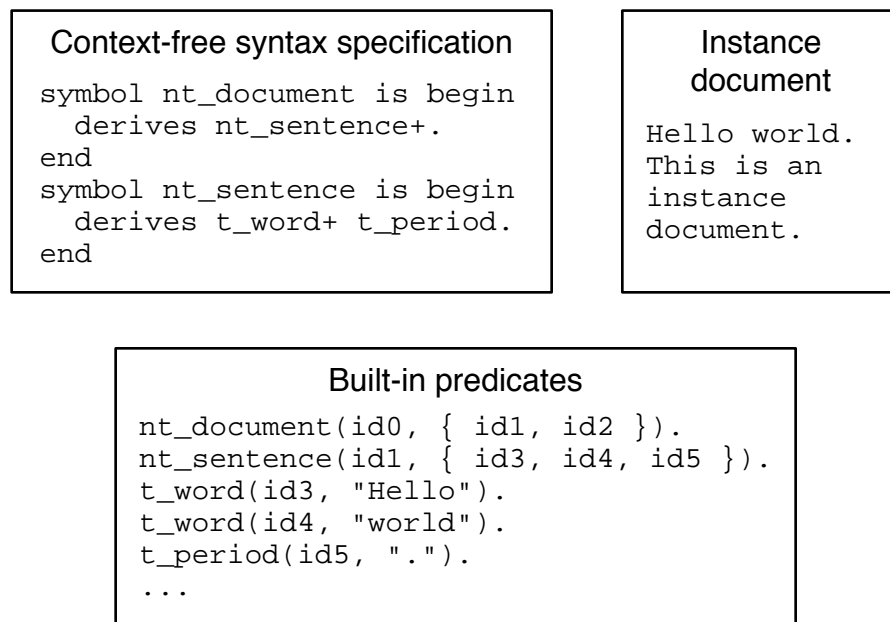


Figure 18 - Example of built-in predicates

4.3.6. User-defined predicates

User-defined predicates in our specification notation serve the same function as text macros served in Heninger’s SCR specification notation [18]. That is, they replace bulky expressions that may be repeated in multiple prohibition declarations with short names, and, when change is necessary, allow the change to be made in one place.

One use for user-defined predicates that we expect to be common is in defining virtual values and value groups within the parse tree of an instance document. Figure 19, for example, shows a predicate from our re-written MSAW specification. If this predicate is true for a given `id`, `aptName`, and `isPrimary`, then there is a row in the airport definitions table with values matching those arguments. Including this user-defined predicate in our specification allows us to write prohibitions involving airports by referring to `apt_definition` rather than to the built-in predicates representing the specific symbols and tokens that comprise the table. In addition to simplifying the prohibitions, using user-defined predicates in this way makes the specification more modular: specification developers who are defining the form of an instance document while they write the specification will find that their prohibitions are insulated from the change they make to the lexical structure and grammar of the instance document.

Another use for user-defined predicates is specifying constants. In the original MSAW site data file specification, for example, limits on the number of rows that could

```

predicate apt_definition(id, aptName, isPrimary) is begin
  there exists anid, ipid, simid, ansid, ipsid such that
    nt_apt_definition_row(id, { anid, ipid, simid }) and
    nt_aptDefinition_aptName(anid, { ansid }) and
    nt_aptDefinition_isPrimary(ipid, { ipsid }) and
    alphanum_t(apsid, aptName) and
    string_literal(ipsid, isPrimary)
end

```

Figure 19 - Example of a predicate used to define a virtual value

appear in each table were expressed in terms of system constants, each representing the maximum number of some kind of item that the system could handle. In our re-written specification, we used user-defined predicates to define these constants. Just as software engineering best practices forbid programmers from employing magic numbers in their code, best practices should require specification developers to define constants in this way in order to centralize the representation of design decisions and to make the meanings of the constant values clearer.

4.3.7. Unchecked prohibitions

Our logic language does not allow us to express all of the properties that may be required of a instance document. We could not, for example, specify the required accuracy of a runway length field. In order to accommodate such requirements, our specification language permits specification developers to express *unchecked prohibitions* in the form of natural-language sentences.

4.4. Designations and context

Our text-based specification language requires specification developers to write designations identifying the problem-world entities corresponding to each part of an instance document. These designations serve to situate the specification in the problem world, making the specification a description of problem-world values and how these are to be encoded into a sequence of bits. In this section we describe which elements of a specification need designation and how to encode these designations.

4.4.1. Non-terminals are designated

There appears to be no need to designate tokens, as individual tokens are frequently used to represent more than one problem-world entity. A floating-point literal token, for example, may appear in one part of an instance document to encode a runway length, and in another part to encode the altitude of a terrain feature. Our specification format does not permit terminals to be designated. If such a designation is needed, the specification developer should instead create a non-terminal that derives the terminal to be designated and designate this non-terminal instead.

There appears to be no need to designate predicates either. Predicates representing the values from subtrees of the parse tree of an instance document need not be designated, as the root non-terminal of that subtree could be designated instead. Predicates representing values from disparate parts of an instance document, on the other hand, describe high-level properties of the instance document rather than fields within it, and so again do not seem to warrant designation.

While some symbols in the context-free syntax specification may be at too high or low of a level to designate, we think that it is possible to write the context-free syntax specification for any instance document so that each part of that document that a developer might think of as an individual “field” can be associated with a particular non-terminal. Designating only non-terminals reduces the risk that a specification will be subtly self-contradictory because a region of it is designated to be one thing at the context-free syntax level and a different thing at the non-context-free syntax level. Such a self-contradiction would be impossible to detect mechanically and could lead to misunderstandings

and errors if different readers, each assuming that a specification is not self-contradictory, read only the designations at different levels.

In some cases, a specification developer may need to designate a non-terminal that produces (possibly among other things) another designated non-terminal. In our re-written specification for MSAW site data files, for example, we have designated both rows in the airport definition table and the airport name cell of each row. The airport row has a meaning: each row represents one airport in the region. The airport name cell also has a meaning: each airport name cell represents the three-letter name of the airport corresponding to the row it appears in. Since it will be desirable in some cases to designate such symbols, and since we want to ensure that all fields in an instance document are designated, we require specification developers to designate all non-terminals that correspond to problem-world entities and require that all paths from the start non-terminal to a terminal through any abstract syntax tree representing a parsed instance document must go through at least one designated non-terminal.

4.4.2. Designations

Designations appear as part of the declaration of the designated non-terminal as shown in Figure 13. Designations may refer to non-terminals or designated terms appearing in the context section. Accordingly, the form of a designation is a sequence of literal strings interspersed with references to other entities in the form `(identifier as "text form")`, where *identifier* is the name of the non-terminal or designated term and the optional `as "text form"` clause allows the developer to change the appearance of the reference as necessary to account for capitalization, conjugation, and the like. An example designation is shown in Figure 20.

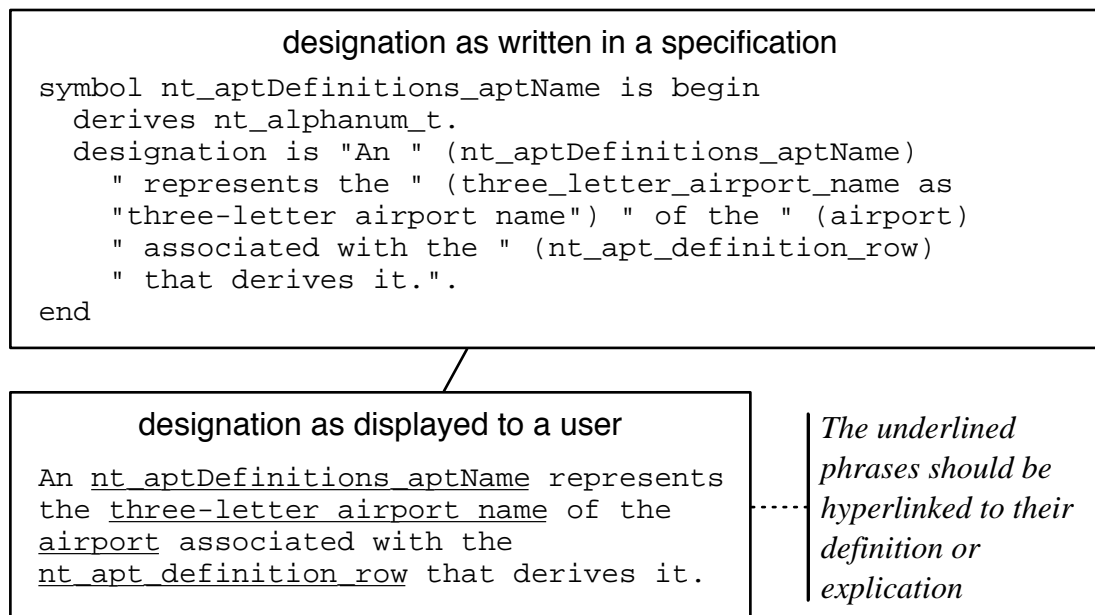


Figure 20 - Example designation

Identifier references link the use of a term in a designation to its definition (in the case of a non-terminal reference) or explication (in the case of an explicated term reference). Tools that preset this information to the user should take advantage of this link structure in their displays in order to make the referenced information more accessible to the user. An instance document editor, for example, could display the referenced terms as hyperlinks, allowing the user to quickly navigate to material that will explain an unfamiliar terms.

4.4.3. The context specification

The designations of non-terminals will likely make use of domain-specific terminology that warrants explication. The context section of a specification collects these terms and their explications. The form of these explications is similar to that of the designations. The identifiers referenced in explications, however, are restricted to terms defined in the context specification. Furthermore, because designations must, by definition, be free of

circularity, tools that perform mechanical verification of specifications should examine the link structure of these references to ensure that no cycles exist.

4.5. Specification goals, revisited

In Chapter 2, we listed a number of goals for specifications. In this chapter, we presented a language specializing the specification technique we described in Chapter 3 to text-based data. Our specification language meets many of these goals by design:

- **Specification developers should write specifications formally.** Our specification language permits a formal specification of syntax. We describe our assessment of the practicality of such specifications and the benefits they bring in Chapter 6.
- **Specification developers should identify the problem world entities described by the data.** The designation mechanism described in section 4.4 permits developers to describe fields in terms of the problem-world entities they represent.
- **Specification developers should describe the form of data in an appropriate manner.** Our specification language includes lexical and context-free syntax sections that permit specification developers to describe the form of text-based data in a way that should be familiar to any software developer who has used a parser generator.
- **Specification developers should describe the form and content of static configuration data separately.** Our specification language does not permit specification developers to completely separate descriptions of form and content. Designations, for example, are associated with symbols declared in the context-free syntax portion of a specification. However, every language specializing our technique to text-based data must include a mechanism for identifying the content being described. We feel that

simplicity of the specification language is more important than achieving the goal of allowing developers to specify form and content separately so as to simplify the translation of data from one form to another, and are satisfied with the compromise our language for text-based data makes in this regard.

- **Specification developers should describe the instance document rather than how to read or write it.** Our specification language's lexical and context-free syntax specifications force specification developers to describe the form of instance documents in a manner similar to the way application developers would use a parser generator. While we risk application developers concluding that instance documents must be read by an automatically-generated parser, we feel that this risk is warranted given the advantage of a notation that should be familiar to anyone who has used a parser generator and the ability of tools to automatically generate an instance document parser from the specification as described in Chapter 5.2.2.
- **Specification developers should write specification in a manner that enables the development of valuable tools.** Our specification language was written so as to enable the development and use of a valuable set of tools. In Chapter 5, we describe our vision of tool support for static configuration data components and the mechanical verification tool we built as part of our evaluation. In Chapter 6 we describe our assessment of the value of that verification tool.

5. Tools for text data

Tool support will be just as essential for those creating, verifying, validating, and consuming static configuration data component instances as it is for software development in general. In this section, we describe the tools we envision for verifying, creating, and parsing text-based static configuration data instance documents. We then describe be portions of this toolset that we have created as part of our evaluation.

5.1. The tools we envision

In order to further illustrate our vision of the process of specifying, creating, verifying, and validating static configuration data components and instances, the following sections describe briefly some of the tools that instance developers, instance verifiers, application designers, and specification developers will need.

5.1.1. An instance document editing tool

Instance developers who are manually creating or editing instance documents should use an editing tool geared to the task. Experience with the Eclipse development environment suggests that syntax highlighting, automatic statement completion, and check-as-you-type technologies help software developers find their errors more quickly [19]. These technologies will bring similar benefits to instance developers. If the specifications are formatted to allow programmatic extraction of relevant information, it should be possible to create an editor for instance documents that will:

1. Help the instance developer to understand what information should go in the instance document by presenting a context-sensitive description of the problem world entities

- related to the portion of the instance document at the insertion point. For example, if the instance developer is editing a portion of an instance document that describes the length of a runway, the editor should be able to quickly display a definition of runway length, a description of the required measurement accuracy, and so on. When the specification uses potentially ambiguous or domain-specific terms to present this information, the editor could provide hypertext links to explanations of these terms.
2. Help the instance developer to format the data in a syntactically valid way by offering a description of the required syntax, color and auto-completion clues as to which syntactic elements are valid at which points in the document, and rapid feedback about syntactic errors.
 3. Help the instance developer to visualize the data in the instance document. Instance documents do not always encode information in ways that are easy for humans to understand. An instance document describing the topology of terrain surrounding an airport, for example, may be encoded as a table of numbers. By presenting the information in a more natural way, such as a three-dimensional graphic in the case of terrain data, a visualization tool may help the instance document developer to more easily find errors in the instance document. The FAA has created visualization tools for the

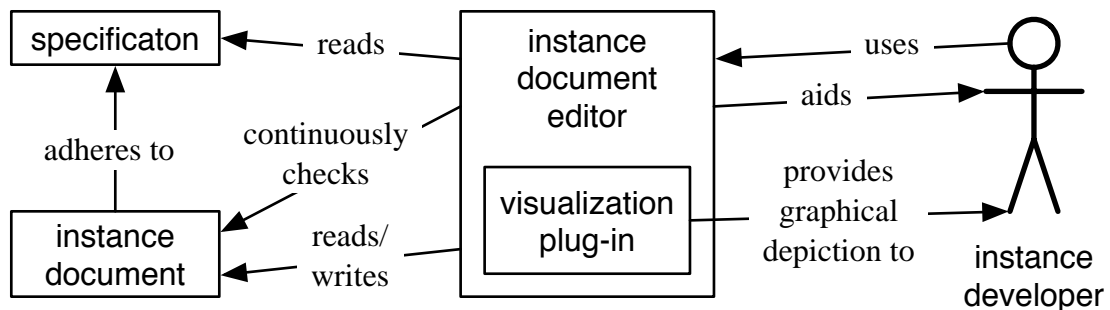


Figure 21 - An instance document editor

MSAW data, presumably for these reasons. We do not expect static configuration data component specifications to contain all of the information necessary to automatically generate such visualizations, but an editor could support a plug-in structure to make developing and using them easier.

5.1.2. Tools for automatically creating instance documents

When instance developers are tasked with creating instance documents that contain information available to the instance developer in machine-readable form, they should use automatic tools to extract the relevant portions of those source documents and combine and reformat these as necessary to create an instance document. Use of a carefully created, verified, and validated automatic tool, rather hand processing, will give greater confidence that the resulting instance documents are fit for use.

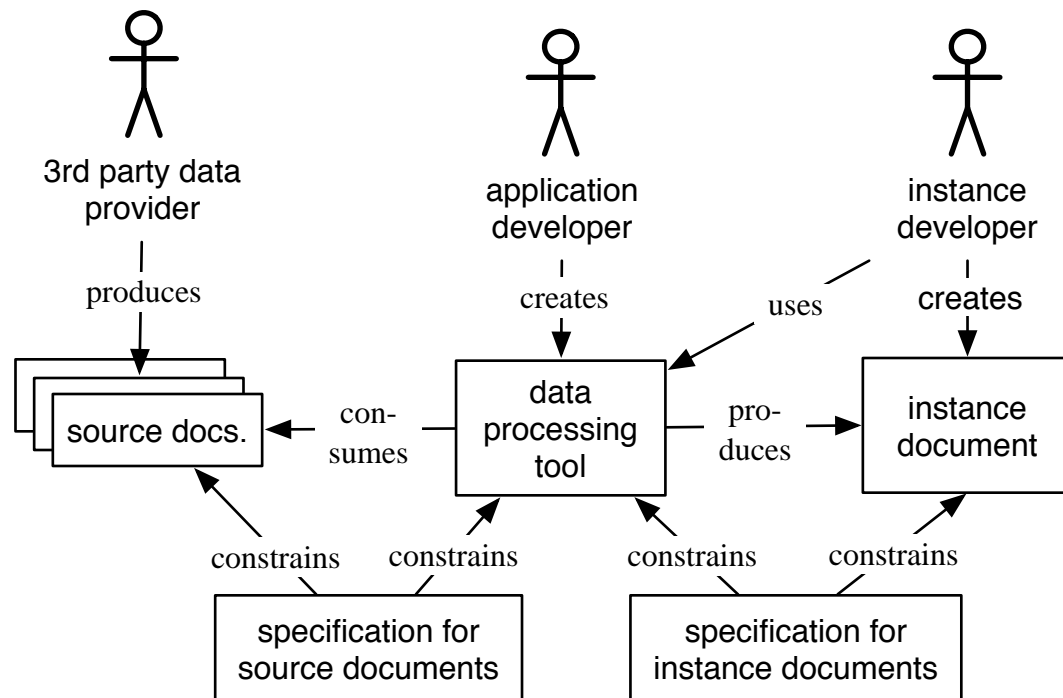


Figure 22 - A data processing tool

A developer creating such a tool will need to argue convincingly that the tool he or she produces will produce valid output if given valid input. The specification of the instance document describes what the tool must produce. A parameter to the tool—the paths to the source documents, for example—will describe which data the tool should include in the produced instance document. Specifications for the source documents describe the assumptions that can be made about them. The tool developer will argue that, given valid source documents, the tool will create an instance document that contains the requested information and adheres to the proper specification.

5.1.3. A mechanical verification tool

Some kinds of defects in specification documents—such as the use of a value in the wrong units of measurement when that value is not grossly out of range—may not be detectable by mechanical analysis. Internal consistency errors, however, are. The sample MSAW site file that we examined, for example, contains a pair of tables that are supposed to be disjoint but aren't. We caught this error while developing the mechanical verification tool described in section 5.2.1.

A mechanical verification tool should accept an instance document and its specification and produce an error report as shown in Figure 23. To do this, it must read the specification, parse the instance document in accordance with the specification's description of form, and search the contents of the instance document for violations of the required properties expressed in the specification.

Instance verifiers should use mechanical verification tools to check for violations of internal consistency, including syntax errors. The tools should be automatically derived from specifications, both to eliminate the cost of hand-writing them and to increase confi-

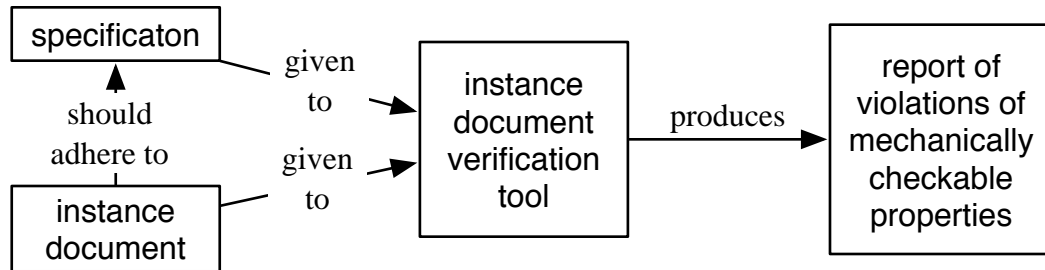


Figure 23 - An instance document verification tool

dence that the tools check precisely those properties expressed in the specification. Whenever possible the tools should operate without human guidance, to reduce the cost of using them and to increase confidence that they are used correctly.

In some cases, software consuming an instance document will check some or all of the mechanically-checkable required properties of the data before using it. Indeed, we suggest such checks where they are practical. In some cases, however, time, memory, and code-space limitations will preclude some or all of such checks. In other cases, the system may be required to be operational to guarantee safety, denying developers the luxury of outputting an error message and halting when a property violation in the input data is detected. In such cases it is essential that mechanically-checkable properties are checked as the instance documents are produced and configuration control, check sums, and the like are used to ensure that the instance documents are not altered before use as described in [20].

5.1.4. An instance document parser generator

When developers create software that consumes an instance document, they will need to construct parsers that read the instance document, extract the needed data items from it, and store these in memory in a manner suited to the programming language employed by the application developer. While it is possible to write such parsers by hand, and in some

case it may be necessary to do so, automatic generation of these parsers could reduce the possibility for code errors.

Some programming languages, such as the SPARK subset of Ada developed by Praxis High-Integrity Systems, support annotations that the developer can use to express the required behavior of functions and modules. It may be possible in some cases to write annotations reflecting some of the guarantees expressed in the data specification and to use these annotations with automatic verification tools to prove that the application will meet its mechanically-checkable requirements provided that the data it is given is valid. An automatic parser generator for such languages may be able to generate these annotations automatically, thus reducing the potential for developer error.

5.1.5. Specification checking tool

Just as instance documents can suffer from mechanically detectable errors and so benefit from mechanical analysis, specifications can contain errors that would be caught by a mechanical verification tool for specifications. Specifications of the form we describe in Chapter 4, for example, contain numerous identifiers that are used to reference elements declared in other parts of a specification: terminal identifiers declared in the lexer specification are referenced in the context-free syntax section, non-terminal identifiers declared in the context-free syntax section are referenced in the non-context-free syntax section, and so on, each of which could be spelled incorrectly or referenced in an inappropriate context. Mechanical analysis can catch these kinds of syntactic errors. Indeed, we expect that many of the tools that read specifications will check for such errors lest the errors cause the tool to fail gracelessly.

While automatic mechanical analysis can find internal inconsistencies and syntax violations, it cannot find instances where the specification as written does not mean what the specification developer intends. To check for these, a different kind of tool might be helpful. Such a tool may take on the form of a putative theorem prover, allowing the specification developer to state a property that he or she expects all legal instance documents to have and proving that the specification, as written, enforces compliance with that property.

While no automatic tool can check the correctness of the natural-language components of a specification, a checking tool could examine these portions of a specification, collect indicators such as the CLEAR Risk Index [8], and possibly even identify potential problem areas heuristically. A specification validator could use the output of such a tool to guide and focus inspections and other human verification and validation activities.

5.2. Prototype toolset

In order to demonstrate the value of the tools made possible by our specification technique, we have created a prototype implementation of parts of the toolset. Due to limited resources we were not able to prototype all of the tools we envision. We started with the mechanical verification tool because, in our opinion, it is the most important and because we could use it as part of the evaluation of our specification technique. The instance document editor is primarily intended to reduce the cost of development, which is less important to us than ensuring high quality. Data processing tools are application-specific and help only to produce instance documents and so evaluating them would require active participation in the process of producing instance documents, which was beyond the scope of

our evaluation activities. The mechanical verification tool also contains an instance document parser generator, demonstrating the feasibility of creating that tool.

5.2.1. Mechanical verification tool

Our mechanical verification tool is designed to accept a specification and an instance document and produce a report listing any violations of mechanically-checkable properties expressed in the specification as shown in Figure 23. A block diagram of the tool is shown in Figure 24. The tool is written in Java 1.4 and is designed to run on Mac OS 10.4 systems that have the SWI Prolog 5.4.7 and javacc 3.2 parser generator installed. The tool is composed of a specification parser, which parses the given specification and verifies that it is well-formed; an instance document parser generator, which automatically generates a parser for instance documents compliant with the given specification; and a non-context-free syntax checker.

Specification parser. The specification parser reads a specification for a static configuration data component and checks it for syntax errors. If any errors are present, the tool halts and outputs them. Since the mechanical verification tool is intended for use in the verification of instance documents, not specifications, it makes no attempt to check other properties of the specification. Although we do not explicitly attempt to verify the specification, the process of verifying an instance document will have the side effect of verifying that the mechanically-checkable properties expressed in the specification are not self-contradictory. If they were, the verification tool would not accept any instance documents.

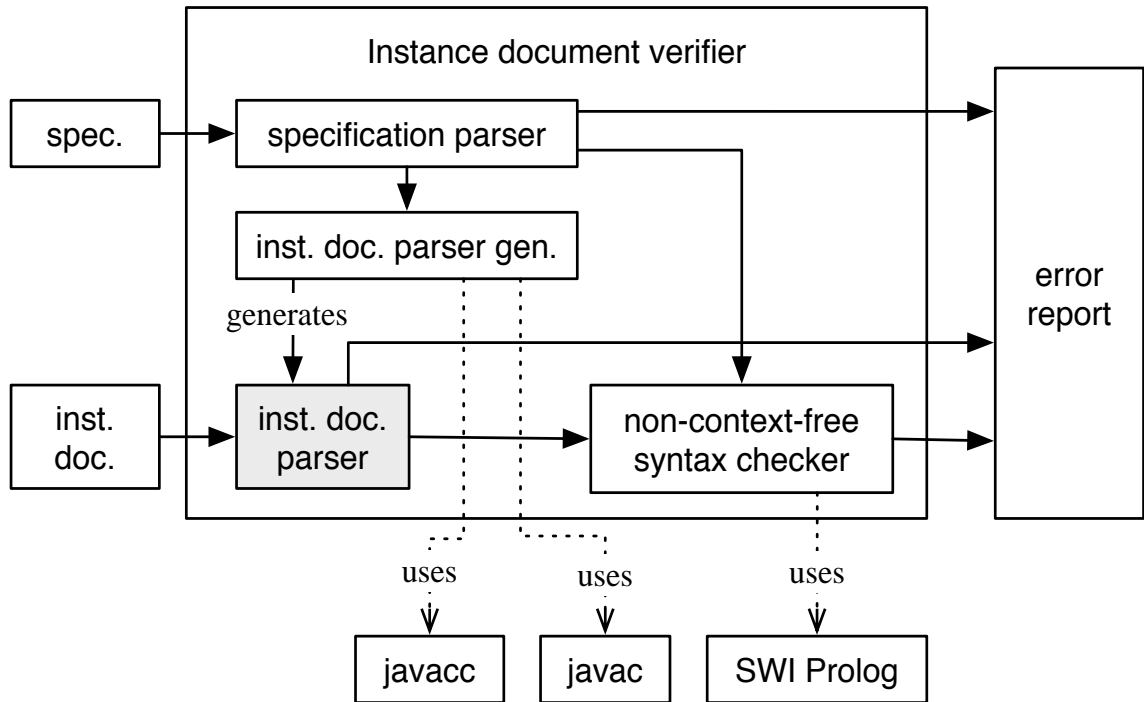


Figure 24 - Components of the instance document verifier

Instance document parser generator. Once the verification tool has parsed the given specification, it creates a parser to parse the given instance document and then uses this parser to generate the built-in predicates representing the instance document's contents as shown in Figure 24.

The instance document parser generator begins by generating a parser specification in the input language of the javacc tool. Since the javacc tool's language is similar to our specification language's lexer specification and context-free-grammar sections, this is fairly straightforward. To each non-terminal declaration in our grammar we add parser actions that cause the generated parser to emit definitions for the built-in predicates described in section 4.3.5.

Once the parser specification has been generated, the verification tool invokes the javacc parser generator to generate Java source code for the parser. This source code is

then compiled by the Java compiler into Java byte code, loaded into the Java virtual machine running the verification tool, and invoked to parse the given instance document. If the instance document contains any lexical or context-free syntax errors, the verification tool will report them and halt.

Non-context-free syntax checker. The non-context-free syntax checker is designed to consume the list of built-in predicate definitions generated by the instance document parser, combine these with the user-defined predicate definitions from the specification, and attempt to find counterexamples for the prohibitions declared in the specification. To do the work of searching for counterexamples we will use the SWI Prolog system as shown in Figure 25.

First, the non-context-free syntax checker re-writes the user-defined predicates in the Prolog notation. Next, the non-context-free syntax checker translates the prohibitions declared in the specification into Prolog queries. One by one it issues these queries to the Prolog system, which attempts to find values to bind to the variables in the expression to make it true. If it succeeds it has found a counterexample, which is collected and reported to the user.

In many ways this is straightforward, as the design of the non-context-free portion of our specification was heavily influenced by the Prolog language. However, there are a few difficulties.

Hidden existential quantification in Prolog. Unlike the non-context-free portion of our specification language, the Prolog language neither permits nor requires explicit existen-

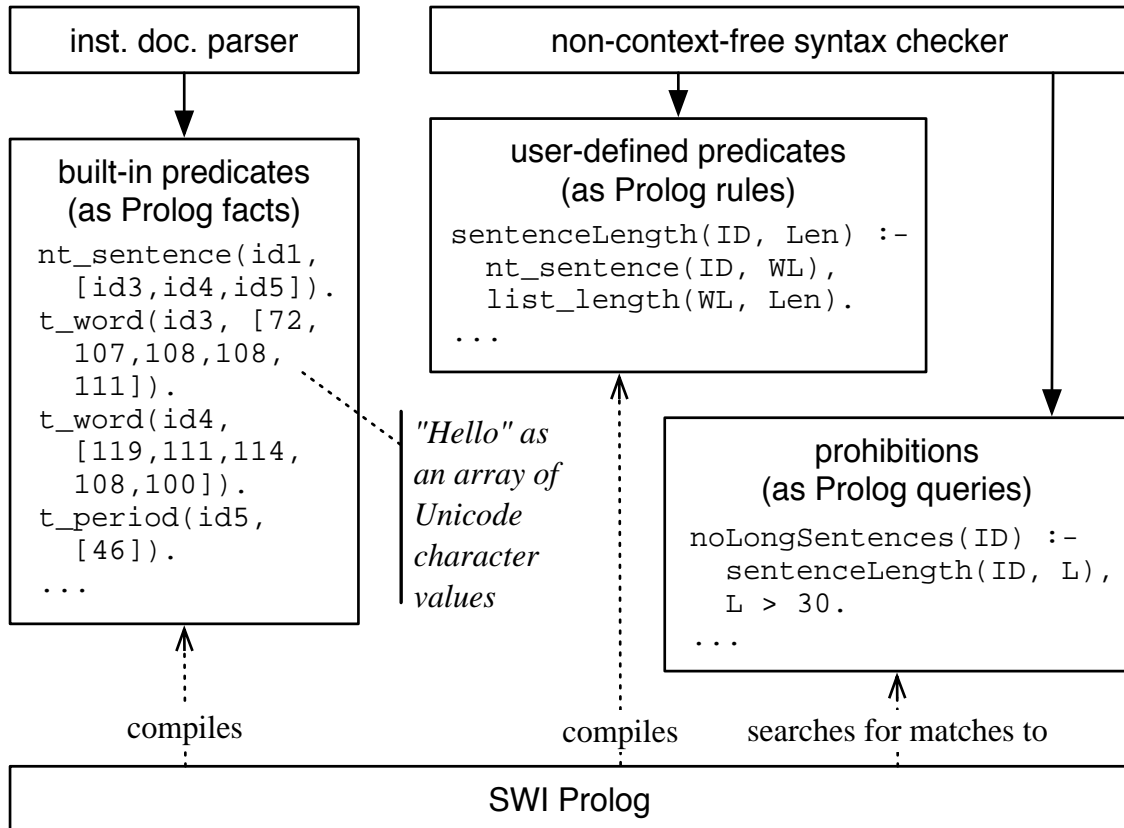


Figure 25 - The non-context-free syntax checker

tial quantification. Because existential quantification is implied in Prolog queries, the order of a query's elements must be considered carefully if the computation is to have the desired effect. Suppose, for example, that the facts `foo(1)` and `foo(13)` and the rule `bar(X) :- foo(X), X < 2` were asserted. The query `not(bar(X))` yields no matches, despite the fact that the query `not(bar(13))` yields Yes. The query `not(bar(X))` causes the system to attempt to prove the goal `bar(X)`, which causes it to bind `x=1`. Since `bar(X)` then succeeds, `not(bar(X))` fails and `x` is unbound.

The effect of Prolog's semantics is that there is a hidden $\exists x_1, x_2, \dots, x_n$, where $x_1 \dots x_n$ are the unbound variables referenced inside the expression to be negated, just after the opening parenthesis following each `not`. Where this hidden existential quantification is undesirable, we can insert a goal before the `not` operator that will bind each x_i to all possi-

ble values in sequence as the system attempts to satisfy the goal. Suppose, for example, that in our earlier example the fact `quux(X)` was asserted for all values of `X` in which we had an interest, including `13`. Then `quux(X), not(bar(X))` would bind `X=13` and succeed as expected.

Since the only values of interest when checking an instance document are values that appear in that document, it is always possible to create the effect of existential quantification by inserting references to built-in predicates into the Prolog code. Our specification language for text-based static configuration data requires explicit existential quantifiers, so our mechanical validation tool knows where the specification developer intends existential quantification. However, since our language lacks a mechanism for specifying the types of variables appearing in predicates and prohibitions, it is not possible for our verification tool to automatically insert the Prolog goals required to implement a check for the specified property. Our tool will instead detect cases where it is necessary to insert those goals and output an error message, prompting the specification developer to re-write prohibitions and user-defined predicates to insert sub-expressions that will be translated into the needed Prolog goals.

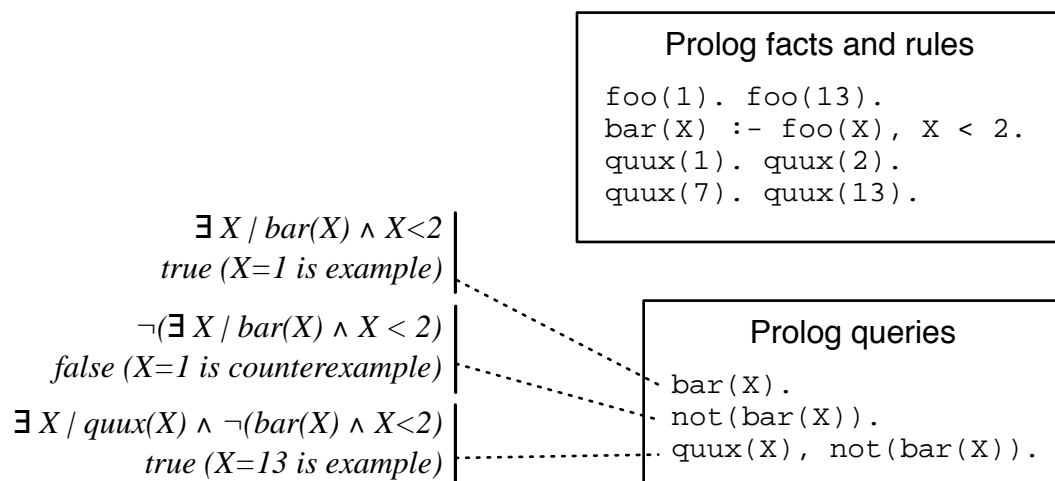


Figure 26 - Hidden existential quantification in Prolog

No support for Unicode in Prolog. The Prolog system and its standard libraries do not support Unicode. This is not as much of a handicap as it might at first appear. Prolog treats strings as lists of integers, with each integer representing the value of one character in turn. The system automatically turns quoted ASCII string literals into the appropriate lists, and is capable of writing ASCII strings to the console and reading ASCII strings from it. While the built-in string manipulation features of the language do not support Unicode, we can still use Unicode with Prolog. All strings generated by the non-context-free syntax checker and passed to the Prolog system are passed as lists of integers with each integer giving the character value of one Unicode character in turn as shown in Figure 25.

Recall that our language does not include built-in predicates for common string relationships because of the complexities involved. Describing which integer a string containing an integer relates to, for example, requires accounting for language and culture differences: where an American might write 1,234.5, a French person would write 1.234,5. Because our language requires specification designers to explicitly codify the rules for the language and culture the static configuration data instances are targeted to, there is no need to rely upon Prolog's built-in string handling routines.

5.2.2. Instance document parser generator

The software components that consume instances of static configuration data components will need to include parsers for those components. As discussed in section 5.2.1, automatic generation of these parsers could reduce the possibility of error. The automatic instance document parser generator we constructed as part of our mechanical verification tool is not a complete generating parsers for use in the consuming system. The verifica-

tion tool's parser generator targets only Java. Furthermore, rather than populate in-memory data structures representing the contents of the document, the parser generated by the verification tool's parser generator writes a disk file containing the contents of the instance document re-cast as Prolog facts.

An instance document parser generator for application components must in general generate a parser in the programming language used to construct those application components. In some cases it may be possible for the parser generator to produce code in a different programming language, as long as the parser can be reasonably built as a separate module, it is reasonably easy for application programmers to make calls to that module, and the overhead of cross-module calls does not pose a performance problem.

An instance document parser generator must also generate a parser that makes the parsed data available to the application developer in a form that is easier to consume than the instance document. A generated parser could create a series of objects in memory representing the data it has read, and require the application developer to write code to traverse that data structure. Alternatively, an instance document parser generator could be equipped with an input language allowing the user to specify parse actions to be called as the parse progresses. The latter method might be preferable in cases where the system must run in a fixed amount of memory and it is necessary to guarantee at design time that memory will never be exhausted at run time. This does not require the generated parser to either use dynamic memory or know the maximum number of each element that an instance document could contain.

In any case the challenges of interfacing a generated parser to a program that consumes the parsed data have been faced before. While the instance document parser gener-

ator we built as part of the mechanical verification tool will not serve the needs of users needing to create their own parsers, it demonstrates that the specification language we have created contains the information about the instance document that such a parser generator would need.

6. Evaluation

In this chapter we describe our evaluation of the specification technique we have described. The evaluation approach we used focused on re-writing a specification for a static configuration data component used in a real safety-critical application using our technique. In order to demonstrate the power of formal static configuration data component specifications and of the tools they enable, we constructed a prototype mechanical verification tool and used this tool to check a real instance document for mechanically detectable errors. We discuss our observations and how these support our claims that our specification technique is practical, that it permits more precise communication than existing methods, and that it enables the development and use of a valuable set of specification-related tools.

6.1. Choice of subject

We chose MSAW site data as an example static configuration data component because it satisfied the following goals:

- **Part of a substantial system deployed in the real world.** The MSAW system has been deployed and in operation for many years, giving us confidence that it is a better example of the kinds of systems to which our technique is meant to apply than a toy example would be.
- **Responsible for satisfying stringent dependability goals.** The accidents and incidents in which MSAW site data file contents were implicated demonstrate the potential safety consequences of erroneous site data. Again, we could not be confident that

a toy example would adequately represent the kinds of static configuration data components we are concerned with.

- **Contain a wide variety of kinds of data.** MSAW site data files contain a wide variety of kinds of information. Terrain data, for example, is real-valued in the problem world and is measured by a third party (NOAA), converted to the MSAW format, and recorded in the instance document. One of the tables in a site data file describes the computer ports to which some accessories are connected; this information is already discrete in the problem world and can be easily observed. Yet other fields configure the displays used by air traffic controllers. There is even a field that is apparently used to control the process of converting and updating the terrain data. With such a wide variety of kinds of data—data obtained from 3rd-party sources and data that instance developers probably measure directly, data that is real-valued and data that is discrete, data describing features of the problem world and data describing the effect the software is to have—we can have some confidence that if our technique was not a practical way to specify an entire class of data, an assessment of our technique using MSAW site data files as an example would reveal the problem.
- **Already have a specification.** Since our efforts focus on capturing, not eliciting, the requirements of static configuration data components, we sought an example for which elicitation had already been done. There is an existing specification for MSAW site data files. This specification is largely expressed as a set of system comments of the form shown in Figure 27. These comments are inserted into MSAW site data files by tools used to process these files. They are supplemented by an FAA document enti-

```

%*****
%
%      Table:  CAM_AURAL_TOWER = Tower Aural Alarm Areas
%
%
%      MAX Rows =    MAX_REM_ALARMS
%      MIN Rows =    0
%
%/*****/
%COL HEADING:          Enable MSAW
%COL LONG NAME:        MSAW Tower Aural Alarm Enable
% C_AUR_LA_CA_REM_1 = TRUE    ( TRUE, FALSE )
% DATA TYPE = LIST_T
%DESCRIPTION: When TRUE, specifies that a tower aural alarm enable volume is
% applicable for MSAW.  Otherwise, FALSE.
%
%COL HEADING:          Enable CA
%COL LONG NAME:        CA Tower Aural Alarm Enable
% C_AUR_LA_CA_REM_2 = TRUE    ( TRUE, FALSE )
% DATA TYPE = LIST_T
%DESCRIPTION: When TRUE, specifies that a tower aural alarm enable volume is
% applicable for Conflict Alert.  Otherwise, FALSE.
%
%COL HEADING:          Region Name
%COL LONG NAME:        CA/MSAW Tower Aural Alarm Enable Region Name
% REGION_NAME_12 = REM1    ( 1 to MAX_CHARS_REG_NAME ) displayable character(s)
% DATA TYPE = DISP_CHAR_T
%DESCRIPTION: Region name for the CA/MSAW Tower Aural Alarm Enable Region.  If
% a track controlled at a tower display is found to be in either MSAW or
% Conflict Alert, an aural alarm at that display is sounded if the track
% (either track of the pair for Conflict Alert) is within a Tower Alarm Enable
% Region.  Thus if no such regions are defined, the Tower alarm will never
% sound.  A tower display is defined in column C_DISP_TYPE as the following:
% 16DB, 12DB, or a RACD with column CHASSIS_TYPE1 defined a RACD or LACD (not
% TMU).  Notice these regions define areas within which the alarm should be
% sounded as opposed to the IFR Alarm Inhibit Regions which define regions
% within which the alarm should not be sounded.  The Region Name is used in a
% keyboard entry (F10) to specify a particular geographic region for display.
%
%COL HEADING:          Region Text
%COL LONG NAME:        CA/MSAW Tower Aural Alarm Enable Region Text
% REGION_TEXT_12 = REM1 AUR ALARM    ( 1 to MAX_CHARS_REG_TEXT ) displayable character(s)
% DATA TYPE = DISP_CHAR_T
%DESCRIPTION: Region text for the CA/MSAW Tower Aural Alarm Enable Region.
% Must be displayable characters.  The region text is displayed in the Region
% Name Tabular List (resulting from an F10 keyboard entry).
%
%COL HEADING:          Shape
%COL LONG NAME:        CA/MSAW Tower Aural Alarm Enable Region Shape
% REGION_SHAPE_12 =          ( POLY, RECT, CIRC, RNG3, RNG2, RNG6 )
% DATA TYPE = LIST_T
%DESCRIPTION: The two-dimensional shape of the region.  See NAS-MD-643
% Section 1.1 for more detailed information on region shape.
%
%COL HEADING:          Point Type
%COL LONG NAME:        CA/MSAW Tower Aural Alarm Enable Region Point Type
% REGION_POINT_TYPE_12 =          ( POLAR, CART, LATLON )
% DATA TYPE = LIST_T
%DESCRIPTION: A region uses one or more points depending on its shape.  A
% point defines a location in a 2D coordinate system.  The type of coordinates
% defining the various points for this region are as follows: POLAR, CART,

```

Figure 27 - Example MSAW table

```

% LATLON. Range/azimuth are entered for POLAR coordinates. X/Y are entered
% for CART (cartesian). Latitude/Longitude are entered for LATLON
% (geographic) coordinates.
%
%COL HEADING:          Geo Data
%COL LONG NAME:        CA/MSAW Tower Aural Alarm Enable Region Geometry
% REGION_GEOMETRY_12 =
% DATA TYPE = REG_T
%DESCRIPTION: The geometrical description of the region. For this region,
% minimum and maximum altitudes may be specified. See NAS-MD-643 Section 1.1
% for a description of region geometry.
%
begin CAM_AURAL_TOWER

% Enable  Enable  Region Name      Region Text      Shape  Point  Geo
% MSAW    CA
%
TRUE     TRUE     MLU                "AURAL REM "     CIRC   CART
0        10000
MLU      0        0                    % MIN & MAX ALTITUDE
% SENSOR NAME AND CENTER POINT
% (XXX, NM, NM)
10
% RADIUS IN NM
;
end;

```

Figure 27 - Example MSAW table (continued)

tled *Standards and Guidelines to Define and Adapt Values for Minimum Safe Altitude Warning and Conflict Alert Site Variables*, which contains, among other things, the values of constants such as `MAX_REM_ALARMS` that are used in the system comments to describe the maximum number of rows in each table [21].

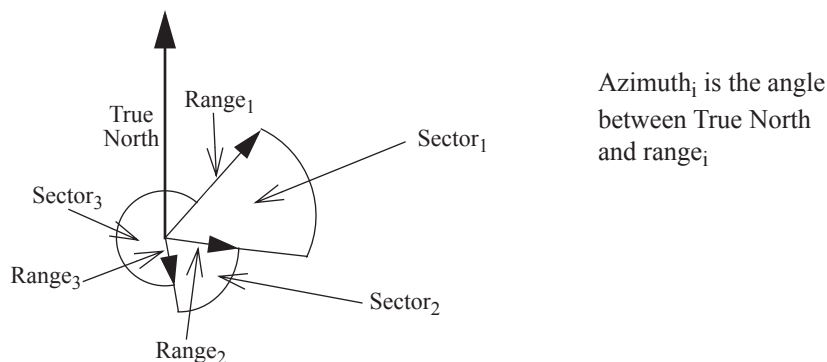
- **Available instance documents.** Since specifications lie between components and their instance documents, an assessment of our technique or language requires both an example component and an example instance document. The FAA kindly provided us with a example instance document representing a typical regional airport. This example instance document meets many of the goals we established for our sample component: it is substantial, as shown in table 1; it is a real example that has been used in the real world; and it contains data (rather than empty fields) of many kinds.

Table 1: Sample site data file characteristics

ASCII characters	1,167,344
Lines	26,530
Blank lines	985
Comment lines	17,255
Tables	223
Empty tables	105
Rows in largest table	6,806

As defined by the FAA, MSAW site data files are ASCII text files organized into a sequence of tables similar to the one shown in Figure 27. There are 223 tables in each site data file. Of these, 49 tables are limited to at most one row and are used to encode single values within a document whose overall form is that of a set of tables. Cells in rows are separated by white space, and rows are separated by semicolons. The type of each cell is one of:

- **Integer** There are decimal, octal, and hexadecimal integers.
- **Floating-point** Floating-point numbers must be written in decimal notation.
- **String** There are several sub-types, each of which permits a different range of characters. Strings may appear quoted or unquoted.
- **IP address** IP addresses must be written in dotted-decimal notation.
- **Latitude** Latitudes are expressed in hours, minutes, seconds, and tenths of seconds, north or south.
- **Longitude** Longitudes are expressed in hours, minutes, seconds, and tenths of seconds, east or west.
- **Region** A region may be specified in one of 24 different ways. The shape may be specified using polar, Cartesian, or latitude/longitude coordinates as an n-sided polygon; a circle; a 3-range, a 3-azimuth shape (shown in Figure 28); a 6-range, 3-azimuth shape; or as a 2-range, 2-azimuth shape. Like cells in a row, the fields describing a region are separated by white space, which makes some tables in



Azimuth_{*i*} is the angle between True North and range_{*i*}

Figure 28 - 3-range, 3-azimuth shape

(From *Specification and Analysis of Data for Safety-Critical Systems* [25]. Used with permission.)

our sample instance document appear to have more fields than they really do.

Unfortunately, while the choice of MSAW as an example has many benefits, it also has a few drawbacks. Chief among these is that the existing specification makes no attempt to supply the reader with the domain knowledge necessary to determine precisely to which real-world concept each field corresponds. Domain and application specific terms are frequently used without being defined. Furthermore, while the specification contains a comment for each column in each table, these comments frequently refer to specific features of the MSAW software, rather than directly to problem-world entities.

We note that the original developers of MSAW are careful and conscientious people. We do not feel that the defects we point out reflect any lack of skill or professionalism on their part. Instead, we feel that these defects illustrate that even a specification created by highly competent and dedicated engineers can suffer from some of the problems that our specification technique is meant to address.

6.2. Evaluation goals

In Chapter 2 we enumerated our goals for data sections, and in Chapter 4 we argued that many of these goals were met by the design of our specification language for static configuration data. In this chapter we describe an evaluation meant to assess the practicality of our technique and our satisfaction of the remaining goals. Our evaluation activities also helped us to fine-tune the design of our language for text-based static configuration data component specifications. The goals left to be addressed by our evaluation activities plus additional goals related to practicality were:

1. **Assess the practicality of formally specifying data.** A technique for formally specifying static configuration data components is useless if using it to specify real components is impractical.
2. **Assess the practicality of our text-specific language.** As with specification techniques, specification languages are useless if it is not practical to write specifications for real components in them.
3. **To assess the precision benefits of formal specifications.** Before we can convince developers of static configuration data components to switch from ad-hoc specifications to formal ones, we must first convince them that doing so will bring desirable benefits.
4. **To assess the value of tools made possible by formal specification.** One of the benefits that we claim users will gain from a formal specification is a valuable set of tools made possible by that specification. In order to demonstrate this, we must demon-

strate that our specification technique and language make a set of tools possible and that those tools are valuable.

5. **To refine our technique and language.** While we were confident in both the overall specification technique we propose and the general form of its text-specific specification language embodiment, we sought experience in using the specification technique to help refine details of the language such as whether multiple lexer states should be used and how the names and types of variables in non-context-free prohibitions should be declared.

6.3. Evaluation method

In order to evaluate our specification technique, we used it to re-write a specification for the site data files used to configure the FAA's MSAW system and then used that specification to check our sample instance document for errors. More precisely:

1. **We specialized our technique to text.** Since the form of MSAW site data files is based on ASCII text and since a language and tool set specializing our technique to text were not already available, we created such a language and a prototype mechanical verification tool.
2. **We translated the existing specification to our language.** We re-wrote the existing specification for MSAW site data files in our language in order to assess the practicality of formal specification of static configuration data in general and specification using our language in specific. During translation, we made note of aspects and particular elements of the original specification that were contradictory or suspicious or that we could not understand well enough to be confident in the accuracy of our trans-

lation. In the latter cases we used our best guess in the translated specification. We translated the entirety of the original lexical structure and the grammar for all of the tables that are populated in our sample instance document. Due to time constraints, we fully translated the syntactic specifications of 19 of the 118 populated tables. We selected these 19 tables because there are no constraints limiting the values in these based on a value in any other table and because they cover a wide variety of kinds of information and data types.

3. **We checked a sample instance document for errors.** We then used our prototype mechanical verification tool to check our sample instance document for inconsistencies with the specification. We did this both in order to assess both the precision benefits of formal specification and the power of the tools made possible by our technique. We note that errors reported by the tool represent either errors in the translation, errors in the specification, or errors in the instance document. In order to ensure that we were not counting errors in our translation as errors discovered by our mechanical verification tool, we manually evaluated each violation reported by the tool with respect to the original natural language specification. Some of the errors we discovered prevented our tool from running to completion. In these cases we noted the error and then weakened the translated specification so that the tool could continue.

6.4. Observations

During the course of performing the evaluation, we made several observations. These observations are described in the following sections.

6.4.1. Lexical structure

The original specification’s description of lexical structure was contained in a system comment at the beginning of the sample instance document. This comment describes the overall organization of the file, including the character encoding, general form of a table, use of semicolons to separate rows, and so on. It also describes the lexical representation of values for each of the basic data types.

The original description was not, however, organized as a description of the lexical structure of the document independent of more complicated syntactic concerns. Thus some parts of the comment block containing the original lexical specification were actually translated into context-free and non-context-free syntax elements. The description of the type `OCT_T` given in the original, for example, is that `OCT_T` values are strings of the characters representing digits 0 through 7. While this is clear, it could not be translated directly into the declaration of an octal literal token, as such strings are also valid decimal integer literals. In our re-written specification, we created a non-terminal for each of the described data types and a set of more primitive terminals that are used in the definition of those non-terminals. Our decimal integer literal non-terminal, for example, derives both a terminal representing a string of digits and a terminal representing the two double quote characters, because the pair of quote characters indicates a ‘null’ value of any type. Our octal integer literal non-terminal is identical as shown in Figure 29; we use a prohibition to enforce the absence of 8 and 9 characters in octal literals.

While written principally in natural language, the original description of lexical structure included some formally defined elements. The description of longitude literals, for example, states that they match “[DD]D: [M]M: [S]S [.S] [C]” and describes the mean-

```

symbol nt_oct_literal is begin
  derives
    t_empty_quoted_string
    | t_int_literal
  .
end

```

Figure 29 - The octal literal non-terminal from our re-written specification

ings of the components identified by the different characters using natural language. Interestingly, we found that particular description ambiguous as we did not know the semantics of the square braces: if they enclose components that are individually optional, then 1, 2 or 3 digits of degrees are permissible; if they enclose groups of components that are optional as a group, then 1 or 3 digits are permitted but not 2.

6.4.2. Context-free syntax

The original specification's description of context-free syntax is carried in part in the same block of comments that defines the lexical structure and in part in the comments at each table that describe the table's columns. Once again we found that the original descriptions, while in natural language, were quite clear and that the translation process was very straightforward. For every table we created a non-terminal representing the table, a non-terminal representing each row, and a non-terminal matching each column. In order to prevent name collisions, we gave non-terminals representing columns names that identified both the table and column: `nt_apptDefinitions_apptName`, for example. Each of the column non-terminals derives the non-terminal representing that column's type.

One major element of the original context-free syntax description is the description of the table structure used in the file. Since we created a non-terminal for each table, we needed to create a start non-terminal that derived each of these tables. Here again we faced a lack of information from the original specification, which does not say whether the

223 tables in our sample instance document must appear in the same order in other instance documents, or indeed whether or not other instance documents may have more or fewer tables than the one we examined. We elected to write our specification assuming that all instance documents have the same tables in the same order. In order to see if this choice would affect the practicality of specification we also attempted to write the specification as if all instance documents must contain the same tables but in arbitrary order. The main difference between the approaches is that in the latter we employ a generic table non-terminal that derives any one of the table non-terminals, define the complete document to be one or more of these, and include prohibition rule per table prohibiting instance documents containing anything other than one instance of that table.

6.4.3. Non-context-free syntax features

Our specification's features for expressing non-context-free syntax requirements were well-suited to expressing the requirements in the original MSAW site data file specification. The non-context-free syntactic requirements expressed in the original specification were¹:

Range limits added to a base type (64%). Columns frequently contained values described as strings of a particular length or numbers in a given range. Range limits become prohibitions against any instance of a given non-terminal representing a value out of the indicated range, and are easily specified in our logic language.

1. We arrived at these figures by examining the original natural-language requirements for each table (giving us the number of table length restrictions) and for each column in each examined table (giving the remainder of the counts). A column described as containing a string of a certain length that must match one of the strings in a given column in a different table would have been counted as one range limit and one inter-table constraint. The figures given cover all observations but do not add up to 100% because they have been rounded to the nearest whole number percentage.

Table length limits (15%). Table length restrictions are easily expressed in predicate logic in terms of the number of row non-terminals composing a given table non-terminal.

Foreign key defined (10%). In several cases the original specification explicitly constrains values in a cell to be one of a set of values defined by another table. While the original specification does not identify primary or foreign keys, we observed several cases where the description seemed to suggest that a column contained a foreign key for which no explicit referential integrity constraint was described, which in turn suggests that there are more restrictions of this type than we actually observed. In our language these restrictions become prohibitions on rows containing a foreign key for which no matching row in the referred-to table exists.

Uniqueness (5%). The original requirements for some table columns require each cell to contain a unique value. In a third of the uniqueness constraints we examined, values were required to be unique across tables. Values appearing in the `Apt Name` column of the airport definitions table, for example, are prohibited from appearing in the fix definitions table; the two columns taken together must form a set. Our intuition suggests that there are many more uniqueness constraints than we actually observed. In our language, these restrictions become prohibitions on the existence of two rows giving the same value.

Row order (2%). Some of the tables we examined required that rows appear in a particular order. Since the built-in predicates representing non-terminals provide the identifiers of the matched terminals and non-terminals as an (ordered) list, such restrictions are easily implemented as prohibitions on tables containing out of sequence rows.

Other intra-table constraints (1%). The original MSAW specification constrained one table that we examined so that each of its rows had a value in one but not both of a pair of columns. This was used in order to provide the equivalent of a variant record. We incorporated this constraint into a prohibition on rows with illegal values by specifying a row with legal values either contained null in the first of the columns but not the second or the reverse.

Other inter-table constraints (1%). The original MSAW specification requires that some cells take on certain values depending upon a value looked up via a foreign key. As with the intra-table constraints, this was easily added to our prohibition on illegal rows.

6.4.4. Constraints on groups of values

There were no restrictions on combinations of values other than those mentioned above. We find it curious that the original specification contained so few requirements constraining groups of values. It seems, for example, that the description of regions of low-altitude airspace that aircraft are expected to occupy during normal takeoff, landing, and taxiing ought not to include a region that is completely contained in another. Most exclusion regions should probably be within a few nautical miles of a runway. If this is the case, it may be advisable to include a requirement that all exclusion regions should be close to an airport declared in the site data file unless a special override flag is set. Such a requirement, coupled with mechanical analysis, could have raised a configuration such as the one that contributed to the crash of Korean Air 801 to the attention of the instance document verifier. Although alerting the instance document verifier may not be sufficient to prevent accidents—the MSAW system was deliberately inhibited at the time of the flight 801 acci-

dent to reduce nuisance alarms [25]—it is part of a systematic approach to ensuring safety that would also include mandatory review and justification of any abnormalities.

6.4.5. Non-syntactic requirements

The original specification is almost completely devoid of non-syntactic requirements. We expect that static configuration data components containing measured values will express requirements on the accuracy and precision of those values, amongst other things. We find it odd that, aside from a few paragraphs describing the required geometry model for terrain altitude data, the original MSAW site data file specification lacks such requirements. It may be that such requirements are expressed elsewhere in the system documentation, or that the provided data is assumed to adhere to an industry standard such as RTCA DO-201A. However, as new instance developers brought on to the project may be as ignorant of what the remainder of the system documentation has to say about measurement accuracy, we think that it is important that the specification repeat or reference these requirements, as it is the instance developer's primary reference document.

6.4.6. The form of the data

The MSAW site data file's form could be made easier to specify and parse with a few minor changes. In addition to requiring the use of lexer states to describe correct system comments and table end keywords, the original specification allowed string literals to appear un-quoted within table bodies and so required us to use a system of non-terminals to allow keywords to be recognized as string literals. With some very minor changes to the form—allowing comments and table endings to appear in any column, requiring strings to be quoted, etc.—the form could be modified to something that would be easier to formally specify and, by extension, easier for automated tools to automatically generate

parsers from. This observation illustrates the point that all forms are not equal; specification developers who have control over the form of the components they are describing should take care to choose forms that lend themselves to easily understood description and automatic processing.

6.4.7. Natural-language semantic descriptions

The original MSAW site data file specification includes a description of the meaning of each column in each table. The meaning of the table as a whole and of each row in it were not explicitly described in the original specification; the reader is left to infer these meanings from the table name and the column descriptions. The table containing airport definitions, for example, contains an `Apt Name` column for which the following description is given:

Three-character airport names, (e.g., JFK), for the airports that have auto associate/dissociate regions. The airports defined in this table can be used to designate a known airport status (e.g., JFK arrival instead of unknown arrival, UNAR). Cannot be same as a fix name in the Airports and Fixes table (FIX_DEFINITIONS).

The other column in this table, `Primary Apt`, is described by the following:

Specifies whether each airport is a primary airport (PRI) or not (NOPRI). Used by parameter `SCRPAD_PRI_APT` to determine whether to display a primary airport exit fix in scratchpad.

These descriptions commingle descriptions of the table as a whole, the columns in it, the rows in it, and the cells in those. The sentence beginning with “the airports in this table,” for example, describes the table as a whole; the sentence beginning with “three-character airport names” describes a column; and the sentence beginning with “specifies whether each airport” describes the meaning of a particular cells. Note that the original descriptions include examples and statements of what the software does with the value,

neither of which form part of a well-crafted rule for recognizing the described entity.

From these descriptions we created the following designations:

- The nt apt definitions table represents the set of airports in the operating area that have auto associate/dissociate regions.
- An nt apt definitions row represents an airport in the operating area that has an auto associate/dissociate region.
- An nt aptDefinitions aptName represents the three-letter airport name of the airport represented by the nt apt definitions row that derives it.
- An nt aptDefinitions isPrimary represents whether the airport represented by the nt apt definitions row that derives it is a primary airport (indicated by the value “PRI”) or not (indicated by the value “NOPRI”).

Some of the underlined words and phrases are formally-defined entities such as non-terminals. The remainder are informally defined terms that represent concepts that we think require clarification and should be explicated in the context section. If we had access to domain experts, we would have crafted these explications. Since we had no such access and thus have no assurance as to the accuracy of our understanding of these fields, these designations serve only to demonstrate the form we think designations should take. We note that some of the information in our re-written descriptions (such as the limitation to the area in which the MSAW system making use of the site data file will be operating) is not actually present in the original descriptions. If the integrity of static configuration data is required to ensure the safe operation of the system, readers should not be forced to make such assumptions lest they make differing assumptions.

Like every other table we examined, the description of the airport definitions table involved numerous terms with domain specific meanings. “Three-character airport names,” for example, probably is intended to indicate the particular short airport names used by agreement in the aviation industry rather than any name for an airport that happens to be three characters long. While we might assume that anyone who reads a specifi-

cation for MSAW site data files will have at least some background in aviation, and so will understand the meanings of such terms, the original MSAW specification also makes extensive use of undefined MSAW specific terms. One column in another table was given this description in the original:

For each airport, fix, or navaid, up to 37 (see note below) displayable characters of data. Intended to include such items as airport name, runway information, elevation, lights information, runway surface, UNICOM frequency, etc., in any format. Displayed in Readout Area in response to an Display (Lighted) Airport, Fix, or Navaid Data keyboard entry (*3-5 char apt/fix/navaid, SLEW), or an Emergency Airport Readout keyboard entry (*,SLEW,SLEW).

To understand what information he or she needs to provide, an instance document developer reading this description would need an understanding of how users use the MSAW system that consumes it and what these users need. This is not merely a problem arising from the use of MSAW-specific terms such as “Readout Area” and “Emergency Airport Readout keyboard entry.” Rather, the use of those terms follows from the decision to couch the description in terms of what it causes MSAW operators to see. In order to reduce the cognitive load on readers, we suggest either eliminating this field and instead automatically generating this string from fields defined in terms of the environment or else providing a set of instructions for generating suitable strings rather than a description of what they mean.

6.4.8. Systematic specification

The original specification included a comment block for each table giving a (very brief) description of the table and the minimum and maximum number of rows. For each column in each table, there is a comment block giving the shorthand and long names for the column, a (very brief) description of the meaning of the column, the basic type—integer,

floating-point, string, latitude, etc.—of the values in its cells, range restrictions to apply to the basic type (if any), and limitations on the values that can appear in the cell given the values appearing in other tables. The fact that this information has been specified for each table and each row leads us to believe that this specification was developed carefully and systematically. While the lack of explanation of domain and application terms makes it impossible for us to judge the clarity of table and column descriptions, we note that for the most part the intent of the syntactic restrictions was clear to us.

6.4.9. Errors found

Despite the care with which the original MSAW site data file specification seems to be written and the fact that our sample instance document has been verified and used, we found a number of errors in the specification and/or instance document.

1. **Hexidecimal literals using prohibited prefixes.** The original specification states that hexidecimal literals will begin with `0x`, but we found instances of hexidecimal literals beginning with `0X`.
2. **Latitude literals with too many digits.** The original specification states that the format of latitude literals “must be `[D]D:[M]M:[S]S[.S][C]` where ... `[.S]` is optional fractional seconds from `‘.0’` to `‘.9’`”, but we found a latitude literal with three digits of fractional seconds.
3. **Undefined constants.** The original specification refers to constants that are undefined. In some cases, we found constants with similar enough names that we could identify the constant that was meant. We assume that `MAX_CHARS_REG_NAME`, for

example, was meant to be `MAX_CHARS_REGION_NAME`. In two other cases, however, we were unable to determine the constant that was intended.

4. **Type errors.** The original specification states that null values for one column we examined should be treated as if they were zero, but the column is of a string type.
5. **References to nonexistent columns.** The original specification includes a restriction for one table we examined requiring values in that column to appear in one of two columns in other tables, but one of these tables does not have the specified column.
6. **Reference constraints not met.** One constraint in the original specification prohibits airports from also being declared as fixes, yet the sample instance document we examined contains some identifiers that are declared as both airports and fixes. Another table we examined required a particular value in a given column whenever the row referred to a fix or navaid, but we found an example of a row referring to a fix that did not take on that value.
7. **Invalid system comments.** The original specification requires system comments, which are introduced by the percent symbol, to begin a line, but we found several comments that did not.

As we had no way of determining whether the original specification, the instance document, or neither is correct when we discovered a defect, it is difficult to determine whether inconsistencies we found were due to specification errors or instance document errors. Because of this, it is difficult to quantify the number of errors we observed. For example, 81 strings of type `ALPHANUM_T` in the instance document we examined con-

tained spaces; we could not determine whether this observation should be counted as 81 errors in the instance document, or 1 error in the specification. We note, however, that of the 19 tables that we completely specified the syntax of, 8 were effected by one or more of the errors we detected. Considering the safety-related nature of the MSAW system, such a large number of defects, wether in the specification or the instance document, is alarming.

6.5. Discussion

The observations we made during the evaluation show that our specification technique is practical, that formal specification does indeed bring gains in precision, and that our specification technique makes a useful tool set possible. We make these claims for the following reasons:

- **The practicality of formally specifying data.** We did not find that re-writing the existing MSAW specification in our language took an exorbitant amount of time or effort. Indeed, we were able to re-write much of the original specification—which forms the bulk of the 17,255 comment lines in the original specification and refers to many more pages of supplemental documentation—in only a few days, and we note that the effort did not require specialized mathematics or computer science knowledge beyond what is typically covered in undergraduate discrete math, theory of computation, and compiler design courses.
- **The practicality of our text-specific language.** Again, the experience of re-writing the existing MSAW specification in our language shows that specification of text-based static configuration data in our language is practical.

- **The precision benefits of formal specification.** During the process of re-writing the original MSAW specification in our language we discovered a number of errors and ambiguities in the original specification. Our prototype mechanical verification tool reported several more. That these errors and ambiguities remained in a carefully prepared and validated component and instance document and were brought to light by either the process of writing a formal specification or the use of a tool made possible by such a specification illustrates the precision benefits brought by formal specification.
- **The value of tools made possible by formal specification.** We created a prototype mechanical verification tool that is able to read a specification in our specification language and search a given instance document for violations of the syntax properties expressed in the specification. The ability of this tool to quickly find a number of discrepancies between the re-written specification and the carefully prepared and validated instance document demonstrates its value.

While our evaluation illustrates that formal specification is practical and enables both gains in precision and the development of useful tools, the limited resources available forced us to make compromises in our evaluation. These compromises and their effects upon our argument need careful consideration and are as follows:

6.5.1. Limited evaluation of the semantic portion of the specification

As discussed in section 6.1, the original MSAW specification did not include enough domain and application information to allow us to understand fully the semantic meaning of the fields in each site data file. Because we did not have access to a domain expert who

could supply this information, we were unable to complete the semantic section of our re-written application.

Fitness to judge accessibility. While we were unable to make observations about the practicality of designations and the CLEAR process in the specification of static configuration data components, we were able to make observations about the failings in the original document that these are meant to address. Our ignorance of the workings of the MSAW system and its software makes us better suited to judging whether descriptions in the original specification were adequately accessible to instance developers, as knowledge of the MSAW system would have biased us. That so many of the descriptions in the original specification were inaccessible to us suggests that there are specifications with semantic sections needing improvement.

Experimental assessment. In addition to making first-hand observations about the practicality of designations and CLEAR, we would like to have been able to measure the effect of these techniques on the reliability with which readers gain the correct understanding of the problem-world entity each field corresponds with. Measuring this effect experimentally would have required completing the semantic portion of our re-written specification, validating this with MSAW experts, collaborating with MSAW efforts to devise a comprehension test, and obtaining an adequate sample of the target population on which to experiment. Unfortunately, those requirements were beyond the scope of our resources.

6.5.2. The benefits of a formal specification

As discussed in section 6.3, the process of re-writing the semantic portion of the original MSAW specification in formal terms caused us to find and required us to correct errors and ambiguities in the specification. While we are unable to claim that our corrected ver-

sion matches the specification developer's intent, we note that the extent to which these deficiencies were present in the original is indicative of the gain in precision brought by the use of a formal language.

Self-contradiction is mechanically detectable. The existence of self-contradictions in the original specification, such as the reference to a non-existent column, supports formal specifications in a different way. When the syntax specification is expressed in natural language, as the original MSAW site data file specification was, then such errors can only be detected by human beings. Since such errors require no information other than what is contained in the specification to detect, the use of a formal specification language makes it possible to mechanically check the specification itself for them. Indeed, while our mechanical verification tool is designed to check instance documents and not specifications, the basic checks it performs on the specification it reads would have caught this error.

Mechanical verification finds defects. The number of defects that our mechanical verification tool was able to discover in the carefully prepared specification and instance document stands as testimony to the power of such tools. When considered together with the fact that our tool operates fully automatically and took only 75 seconds to completely check our sizeable sample document, we think that this observation supports our advocacy for the use of mechanical verification tools. This in turn offers support for a formal description of syntax, as only with a formal syntax description can one build a mechanical verification tool that does not need to be hand-customized to type of static configuration data component.

6.5.3. External validity

All of the observations described in this chapter are observations about one sample static configuration data component and one instance of it. While we chose this sample because it is a real static configuration data component used in a real system with stringent dependability requirements, we cannot claim that this example is representative of all such components.

Generalizing from text-based data. We are making claims about both our specification technique in general and our specification language for text-based static configuration data in particular. However, because our method was to use our language to re-specify an existing component, our evaluation was an evaluation of the language; it was only indirectly an evaluation of the general technique. We have called this language a language for specifying text-based static configuration data components, but the only such component we have used it with to date is the MSAW site file.

Although we have not shown that our language is practical by evaluating its use on a wide sample of text-based data, we note that our experience with our single sample shows that there are text-based forms that can be specified practically using our language. In cases where the specification developer can modify the form as needed, it is possible to select a form that makes specification using our language, and thus our technique, practical. Note that this logic extends to our specification technique generally: because we were able to specify the form of this text-based static configuration component practically, there is at least one general form for which our technique can be practically used.

While we cannot make as strong a case when the specification developer has no control over form, we see little reason to believe that there will be a significant number of

reasonable forms that for which no formal specification can be practically written. We need only support forms that application developers could reasonably produce code to read. For text-based instance documents, application developers will very likely want to make use of parser generators rather than write parsers by hand. Since our text-specific language deliberately include many of the features that lexer and parser generators offer, it should allow the specification developer to specify most text-based forms that application developers could practically write applications to read.

This logic generalizes: if the application developer has a practical way to read instance documents, then there must be some practical way to formally specify the form. That practical way may constitute a different specification language, but so long as there is sufficiently broad interest in any given general form, there ought to be adequate justification for developing a language and tool set specific to that form.

Generalizing observations about ad-hoc natural language. While we were unable to make observations about the power of our formal specification to forestall misunderstanding, our observations of the original MSAW site data file specification show that in at least one case careful but otherwise unguided use of natural language applied systematically across the fields of a document was not sufficient to clearly communicate the required information to a member of the target audience. Unfortunately, the limits of our resources prevented an evaluation that would have demonstrated the degree to which this problem exists in other static configuration data component specifications. Yet the imprecision of natural-language communication is well-known and the difficulty of communicating application and domain-specific information to people unfamiliar with the domain and

application has been described elsewhere [8, 15]. These things suggest that strategies aimed at reducing the problem warrant attention and consideration.

Generalizing our experience with tools. Finally, our experience with the MSAW site data file and our mechanical verification tool shows that, at least for the types of requirements expressed in the MSAW site data file specification, there is at least one valuable tool made possible by formal specification. While experience with the kinds of requirements expressed in the original specification does not necessarily generalize to other kinds of requirements, we note that requirements like those in the original specification—requirements constraining the type of fields, including the ranges of their values and limitations on values based on the value of other fields—will almost certainly be present in any other specification. Since our mechanical verification tool was useful in finding violations of those requirements that were apparently missed by the humans validating the MSAW system and its deployments, similar tools will bring similar benefits to other specifications even if there are requirements of a kind unseen in MSAW that require additional tools or V&V procedures. Thus the formal specification of syntax called for by our technique does generally enable the development of a useful set of tools.

7. Related work

7.1. Data as a separate system component

Research by Storey and Faulkner [1, 4, 5, 22, 23, 24] has examined the use of data in safety-related systems, found significant shortcomings with current approaches, and proposed alternatives and directions for future research. Their research, including a survey of industry professionals whose work involves safety-related systems making extensive use of data, indicates amongst other things that:

- “Hazard analysis is often not applied to data in the same way that it is applied to other system components.”[4]
- “It is common not to apply integrity requirements to data.” [4]
- “The development process changes significantly with time ... later applications [of reusable components adapted by data] will not have the benefit of the input from those who created the system and will need to rely on documentation and their understanding of the developer’s intent.” [4]
- “As the system is more widely applied, its ability to be tailored to a given situation may result in it being used in circumstances that were not envisaged by the original design team.” [4]
- “Few static tools are available to investigate the correctness of such data.” [1]

Storey and Faulkner advocate the treatment of data as a separate system component subject to its own development process. This process, they argue, should include hazard or risk analysis and the development of *data integrity* requirements, which specify the level of data quality required to adequately address the identified risks and hazards. The lifecycle requires specification of the data component, verification and validation of the data, and management of data as it flows from its source to its eventual use.

The work presented in this thesis is motivated in part by Storey and Faulkner’s observations; it is an attempt to address the manner in which data components are speci-

fied. Describing the data in terms of the problem world, rather than in terms of the software that will consume it, is meant to address the problem of instance developers who were not part of the original software team. Likewise, our insistence on a formal description of the syntax of data is meant to help address the scarcity of static analysis tools they have observed.

7.2. The Role of Natural Language in a Software Product

In *The Role of Natural Language in a Software Product* [9], Strunk argues that it is inevitable that *some* informal, natural-language content will be used in *any* software specification, as it is needed to give meaning to otherwise semantically void formal models. To guide the development of this content so as to help address shortcomings in it and thus reduce the errors stemming from misunderstanding of the meaning of a specification or the resulting software artifacts, Strunk proposes:

- The *Hamilton process*, which guides the acquisition and recording of the application-specific knowledge needed to understand the meaning of the program.
- The *program+* (later called a *situated formalism*), an artifact for recording a formal representation of a program, the necessary application-specific knowledge, and the relationship between them.

The situated formalism contains both explanations of the application-specific terms used in the specification and a mathematical model of relevant properties of the problem world. The latter component is particularly important for control software, which is valid only if it gives rise to the desired changes in the problem world. To gain a complete understanding of the effect of such software it is necessary to understand how the real world will transform the effects of the actuators into the values monitored by sensors. Note that this portion of a situated formalism is not needed in a static configuration data

component specification. Unlike software programs, static configuration data are *static*: they describe values to be encoded rather than what the software must do in order to accomplish the changes to the environment demanded by the customer.

The technique we propose in Chapter 3 and the language we propose in Chapter 4 are intended to allow specification developers to create situated formal specifications of static configuration data components. We have considered the needs of the parties who have stakes in such components and created a particular kind of situated formalism that addresses the unique needs that they have.

7.3. Specification and Analysis of Data for Safety-Critical Systems

In [25], Knight et al. present a technique for the specification of data that divides the specification to *syntax*, *semantics*, and *context*. The syntax portion of a description of the structure of the data in Backus-Naur Form (BNF). The semantics portion uses the input language of the Prototype Verification System (PVS) theorem prover to formally state restrictions on data items and combinations of data items that disallow some syntactically-valid data documents. The context portion describes, in structured natural language, the problem world entities to which each datum relates.

By specifying the structure of the data formally, the proposed specification technique enables mechanical analysis of instance documents that purport to adhere to a given specification. The authors argue that such mechanical analysis can help to find some kinds of difficult-to-detect errors in the data and demonstrate the use of a prototype tool to find such errors.

However, mechanical analysis alone is insufficient to show that the data in a given instance document are valid. Users who write instance documents and application devel-

opers who write software that reads or writes instance documents may make mistaken assumptions about the meaning of each datum read or written. The context portion of a specification written in the form the authors propose is intended to engender in the mind of all readers as similar as possible an understanding of the meaning of each datum, in order to avoid catastrophic consequences.

Knight et al. describe an example specification format and a tool that can be used to verify that the data in a given file complies with the formally-expressed portions of a specification. A user uses the tool to generate lex and yacc specifications for a parser to parse instance documents. The tool's user must edit the generated parser to insert code to cause the parser to emit a description of a parsed instance document in PVS notation compatible with the specified semantics. Finally, the tool's user must supply a proof script—which may be very simple in some cases and quite complex in others—to direct PVS to prove that instance document, translated into PVS notation by the parser, satisfies the semantic constraints expressed in the specification.

The specification technique we present in Chapter 3 builds upon the ideas described by Knight et al. We have more carefully considered the stakeholders in static configuration data components and instances, and used the insight we gained to elaborate the goals for specifications and to propose a set of specification-related tools. After recognizing that no single specification language is ideally suited to describing all general forms, we have called for languages and tools customized to particular general forms. Moreover, after careful consideration of the benefits and drawbacks of the PVS system, we have decided to change our approach to specifying non-context-free syntactic proper-

ties in order to make verification accessible to persons lacking the specialized knowledge necessary to use PVS effectively.

The specification language we present in Chapter 4 also makes improvements over the original proposed by Knight:

1. We include explicit support for defining the lexical structure of data, including the legal character encodings, permissible white space, and tokens.
2. Our specification format includes a well-defined mapping from non-terminals in the context-free grammar to identifiers appearing in non-context-free properties.
3. We give an explicit structure to our description of the context in which the specification is situated.

Likewise, while the verification tool we describe in Chapter 5 was inspired by the one described by Knight, it extends on and enhances the original by being fully automatic. While the original required the user to modify the generated parser and guide PVS through the verification process, our tool requires no user intervention during verification.

7.4. ASN.1

ASN.1 is a standard notation for describing abstract data structures. It is widely used to describe data exchanged between systems, including X.509 digital certificates, messages exchanged under the Public Key Cryptography Standard, RSA keys and certificates, messages in ground-to-ground and ground-to-air protocols established by the FAA and the International Civil Aviation Organization, and messages sent as part of the Simple Network Management Protocol (SNMP) [26].

Data structures specified in ASN.1 notation are specified independent of any particular mapping to bits, and so can be used with a number of ASN.1 encodings [17]. The Basic Encoding Rules (BER) and Packed Encoding Rules (PER), for example, map ASN.1 data structures to binary messages. Users who want human readability or compatibility with the Extended Mark-up Language (XML) may use the XML Encoding Rules (XER) instead.

There are a number of commercial tools available to use with ASN.1, including tools that automatically parse ASN.1 messages and populate equivalent data structures in a variety of programming languages, including C, C++, C#, and Java. There are also syntax checkers for ASN.1 specifications and editors for building and verifying messages meeting a given specification [27].

While ASN.1 provides a powerful set of specification primitives, it does not provide a way to succinctly describe complex relationships among fields in a document. There is no way, for example, to specify that the value in a ‘total’ field of an invoice message is equal to the sum of the costs of each item listed in the message. In addition, ASN.1 relies upon identifiers and comments to specify message semantics. The language provides neither structure nor guidance for selecting identifier names and writing comments that adequately convey the meaning of the specified data.

The specification language for text-based data presented in this work addresses a different problem than ASN.1, as it is aimed at arbitrary text documents rather than messages encoded in one of the ASN.1 encodings. This difference aside, ASN.1 appears to be limited to the specification of the form of data. It has no mechanism permitting a *complete* description of syntax. Moreover, it has no features that explicitly support or require

situating the specifications in the problem world. It relies upon comments and identifier names to carry semantic meaning, just as the MSAW site data file specification does, and so developers using ASN.1 may well create specifications with the kinds of communication deficiencies we observed in our examination of MSAW.

These shortcomings aside, ASN.1 is a widely-used specification language that can be used to describe a very wide variety of data. Unlike plain text files and eXtensible Markup Language (XML) documents, it is just as easy to describe a message containing a graphic image or other binary data in ASN.1 as it is to describe a message containing only text fields. This variety, coupled with the widespread use of ASN.1, would cause us to seriously consider using an ASN.1 encoded form for a new static configuration data component. Doing this would require creating a specification language that uses ASN.1 to describe form (along with a supplemental features to complete the description of syntax) and a set of tools suited to that language. Such a language would enable the use of many of the ASN.1-enabled tools and would make the specification process more amenable to developers already familiar with ASN.1.

7.5. XML and XML Schemas

Developed by the World Wide Web Consortium (W3C), XML is a mark-up language for describing structured and semi-structured data. Similar in form to the HyperText Markup Language (HTML) and the Standard Generalized Markup Language (SGML), XML documents are text files augmented with annotations to describe the structure of the information they contain and guide its processing [28]. XML is widely used on the Internet and forms the basis of the XHTML and Really Simple Syndication (RSS) standards, amongst others.

As with ASN.1, there are a number of commercial tools available for use with XML documents. XML parsers are available for a wide variety of languages, including C, C++, Java, Perl, and C#. The Microsoft .NET Framework Class Library even contains a built-in `XmlSerializer` class that is capable of automatically serializing objects to XML [29]. There are XML editors for editing XML documents, tools for generating code to describe data structures equivalent to XML document formats and vice versa, and tools for transforming XML documents based on transform rules expressed in XSLT (itself based on XML).

Developers wishing to specify the form of XML documents have at least two choices: Document Type Definitions (DTDs) and XML Schemas [28]. While both allow the developer to constrain the general form of a document, including which elements and attributes are used and how they are composed, the XML Schema language permits finer control over the values that can appear in valid documents. Again, as with ASN.1, neither XML DTDs nor XML Schemas allow the developer to specify complex relationships between values in disparate parts of a document, and so neither allows a complete description of syntax. Moreover, both rely solely on comments and identifiers to carry semantic meaning, just as ASN.1 specifications do. The popularity of XML is, however, undeniable. As with ASN.1, it would be possible to create a specification language that uses XML Schemas to describe form, thus making possible the use of XML tools and XML-savvy developers.

7.6. Constraint languages

Constraint languages have been used to specify properties of data. Demsky and Rinard have, for example, proposed a constraint language for describing data as part of their work

on repairing damaged data structures [30]. Their technique generates algorithms that check data structures for syntax violations and use goal-directed reasoning to repair detected violations. A programmer using this technique must describe the form and required syntactic properties of the data structures to be protected in a language Demsky and Rinard provide for this purpose. Like the static configuration data specifications we propose, these descriptions are formal; unlike the specifications we propose, these descriptions only incidentally convey the meaning of the described data.

Pirri and Pizzuti have proposed defining data dictionaries in Prolog so that developers can take advantage of Prolog's reasoning capability to find inconsistencies arising from the integration of local dictionaries into a comprehensive data dictionary [31]. The data dictionary model they propose includes entities, relationships, and refinements, but does not describe the form data are to be encoded in when transferred between systems and makes no effort to describe the meaning of the data.

Any Prolog program that reads a file and checks the data in it for syntax errors before use could be considered a formal specification of that data. Such programs, however, use the powerful Prolog language to describe the form of the data and a combination of the programmer's choice of identifiers and natural language documentation (including comments) to describe its meaning. Even if the programmer deliberately and carefully creates the program with the idea of using all or part of its source and documentation as a description of the consumed data, it is unlikely that the program will be as accessible to human readers and as suitable for use as input to general-purpose data-related tools as a specification created to support these uses.

While constraint languages have been used to specify syntactic properties of data, we know of no prior work which uses constraint languages as part of a comprehensive effort to document the form, requirements, and semantics of static configuration data.

8. Conclusion

A growing number of systems make use of static configuration data to adapt a standard package of hardware and software to the site at which it is deployed. When errors in this data can cause harm to humans or the environment, developers must take appropriate measures to ensure the validity of the data. Storey has argued that static configuration data should be treated as a separate system component, complete with its own lifecycle. One of the artifacts that will be created and used during this lifecycle is a specification of the static data component.

A static configuration data component specification serves as a contract to the instance developers who create instance documents, describing what they must deliver; as a contract to application designers, detailing what the applications they build can expect; and as a basis for automated and manual verification activities. To support these uses, the specification must describe the form the data are to be encoded in, the properties the data are required to have, and the meaning of the data.

Both the form of data and a portion of the properties it is required to have concern the syntax of the data. That is, together they distinguish the set of bit strings that are not valid instance documents from those that may or may not be, depending upon factors other than the bit sequence alone. Specifying the form of data formally permits the development and use of a valuable set of tools that includes mechanical verification tools which can detect syntax violations in instance documents.

The remaining properties and the meaning of the data cannot be described formally; they must be described using natural language. Unfortunately, natural language

descriptions are prone to error. Application-specific or domain-specific terms can be incomprehensible to readers. Worse, readers may assume that terms are meant to indicate a different concept than the author intended to convey and consequently misunderstand what is written without being aware of the problem. Writers aware of these potential problems provide dictionaries, but the definitions in ad hoc dictionaries can suffer from a number of problems [8].

In this thesis we describe a general technique for specifying static configuration data components. The form of the data in such components should be specified formally and in a manner suited to the overall form of the data. The form of a text-based instance document, for example, should be specified in a manner that permits clear, concise formal descriptions of text-based data. The required properties of the data should be specified formally where practicable. Finally, the meaning of the data should be described by identifying the real-world concepts each field represents. Where these designations make use of application-specific or domain-specific terms, those terms should be explicated in order to reduce the risk of misunderstanding.

Our general evaluation technique was to re-write a specification for an existing safety-related static configuration data component using our technique, create a prototype mechanical verification tool, and use this tool to check a sample instance document for errors. Intentionally, our general specification technique calls for a specification of form in a manner suited to the overall form of the data. Because the specification we chose to re-write was for a text-based static configuration data component, we developed an instantiation of our technique for text-based data by creating a specification language for such data.

In order to evaluate the practicality of our specification technique and our claim that formally specifying static configuration data enables the construction of a valuable set of tools, we: (1) created a specification language that specializes our general specification technique to text-based static configuration data components, (2) re-wrote the specification for an existing safety-related static configuration data component in our language, (3) wrote a mechanical verification tool capable of checking an instance document for violations of the syntactic rules in a specification written in that language, and (4) used our mechanical verification tool to check a sample instance document for such violations.

In our text-based static configuration data component specification language the specification of the form of data is split into a specification of the lexical structure of the data and a context-free grammar. The required properties of the data are specified using a combination of a predicate logic language and unchecked prohibitions in natural language. The semantics of the data are specified using a combination of designations identifying the problem-world entities to which fields correspond and a context specification containing explications of the application-specific and domain-specific terminology used in the designations.

We chose to use the site data files that configure the FAA's MSAW system as an example safety-related static configuration data component with which to evaluate our system. While performing our evaluation, we found a number of discrepancies between the existing specification and the sample instance document we used our mechanical verification tool to check. Whether these discrepancies are errors in the specification or errors in the instance document we examined, it is alarming that so many errors exist in carefully-constructed, carefully-validated system in use in the real world. That we were able

to re-write the specification within the time and resource constraints of this project demonstrates that formal specifications are practical. That we discovered some of these errors while formalizing the existing specification and the remainder using our mechanical verification tool shows the power of formal specifications and of the tools that they make possible.

Unfortunately, with the limited resources available to us we were unable to evaluate some aspects of our proposed technique. We did not, for example, perform an experimental assessment of the accessibility benefits brought about by describing data in terms of the problem-world entities it represents and by using the CLEAR process to explicate the terms used in the descriptions. These issues should be explored. Our evaluation was also limited to one static configuration data component with a text-based form. Further work is needed to specialize the technique we describe to other general forms, to explore the practicality of formal specifications for data in these other forms, and to create and support a set of best practices for static configuration data component specification.

References

- [1] N. Storey and A. Faulkner, "Data Requirements for Data-Intensive Safety-Related Systems," in *Proceedings of the 21st Systems Safety Conference*, 2003, pp. 865-874.
- [2] W. Greenwell and J. Knight, "What Should Aviation Safety Incidents Teach Us?," University of Virginia, Department of Computer Science, Tech. Rep. CS-2003-12, 2003.
- [3] National Transportation Safety Board, *Controlled Flight Into Terrain, Korean Air Flight 801, Boeing 747-300, HL7468, Nimitz Hill, Guam, August 6, 1997*. Aircraft Accident Report NTSB/AAR-00/01, Washington, DC, 2000.
- [4] N. Storey and A. Faulkner, "Data Management in Data-Driven Safety-Related Systems," in *Proceedings of the 20th Systems Safety Conference*, 2002, pp. 466-475.
- [5] N. Storey and A. Faulkner, "The Characteristics of Data in Data-Intensive Safety-Related Systems," in *Proceedings of the 22nd International Conference on Computer Safety and Reliability*, 2003, pp. 396-409.
- [6] C. Heitmeyer, R. Jeffords, and B. Labaw, "Automated consistency checking of requirements specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 231-261, July 1996.
- [7] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," in *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 1-30, January 1997.
- [8] K. Wasson, "Cognitive Linguistic Elicitation and Representation (CLEAR): A Method for Creating Higher-Validity Software Requirements," Ph.D. dissertation, University of Virginia, Charlottesville, VA, 2006.
- [9] E. Strunk, "The Role of Natural Language in a Software Product," M.S. thesis, University of Virginia, Charlottesville, VA, 2006.
- [10] C. Gunter, E. Gunter, M. Jackson, and P. Zave, "A Reference Model for Requirements and Specifications," *IEEE Software*, vol. 17, pp. 37-43, May/June 2000.
- [11] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*. New York: Addison-Wesley Publishing Company, 2001.
- [12] K. Wasson, "A Case Study in Systematic Improvement of Language for Requirements," to appear in *Proceedings of the 14th IEEE International Requirements Engineering Conference*, 2006.

- [13] K. Wasson, J. Knight, E. Strunk, and S. Travis, "Tools Supporting the Communication of Critical Domain Knowledge in High-Consequence Systems Development," in *Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security*, 2003, pp. 317-330.
- [14] K. Hanks, and J. Knight, "Improving Communication of Critical Domain Knowledge in High-Consequence Software Development: an Empirical Study," in *Proceedings of the 21st International System Safety Conference*, 2003.
- [15] K. Hanks, J. Knight, and E. Strunk, "Erroneous Requirements: A Linguistic Basis for Their Occurrence and an Approach to Their Reduction," in *Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop*, 2001, pp. 115.
- [16] *Information Systems - Coded Character Sets - 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)*, New York: American National Standards Institute, 1986.
- [17] J. Larmouth, *ASN.1 Complete*. San Francisco: Morgan Kaufmann, 1999.
- [18] K. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions on Software Engineering*, vol. SE-6, January 1980.
- [19] D. Saff and M. Ernst, "Continuous Testing in Eclipse," in *2nd Eclipse Technology Exchange Workshop*, March 2004.
- [20] *RTCA: DO 200A Standards for Processing Aeronautical Data*. Washington: Radio Technical Commission for Aeronautics, 1998.
- [21] Federal Aviation Administration Technical Staff, *Common Arts Computer Program Functional Specification (CPFS), Site Adaptation*, Federal Aviation Administration, 2001.
- [22] A. Faulkner, "Data Integrity: An often-ignored aspect of safety systems," engineering doctorate executive summary, University of Warwick, 2004.
- [23] A. Faulkner and N. Storey, "Strategies for the Management of Data-Intensive Safety-Related Systems," in *Proceedings of the 21st International Systems Safety Conference*, 2003, pp. 855-864.
- [24] N. Storey and A. Faulkner, "Data - The Forgotten System Component?," *Journal of System Safety*, vol. 39, pp. 10-14, Fourth Quarter 2003.
- [25] J. Knight, E. Strunk, W. Greenwell, and K. Wasson, "Specification and Analysis of Data for Safety-Critical Systems," in *Proceedings of the 22nd International System Safety Conference*, 2004.
- [26] "ASN.1 Site - Uses," February 2006, <http://asn1.elibel.tm.fr/en/uses/index.htm>.

- [27] “OSS Nokalva - ASN.1 Products,” February 2006, <http://www.oss.com/products/products.html>.
- [28] D. Box, A. Skonnard, and J. Lam, *Essential XML: Beyond Markup*. Boston: Addison-Wesley Longman, 2000.
- [29] “XMLSerializer Class,” May 2006, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfssystemxmlserializationxmlserializer-classtopic.asp>.
- [30] B. Demsky and M. Rinard, “Data Structure Repair Using Goal-Directed Reasoning,” in *Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 176-185.
- [31] F. Pirri and C. Pizzuti, “Explaining Incompatibilities in Data Dictionary Design through Abduction,” *Data and Knowledge Engineering*, vol. 13, pp. 101-139, 1994.

Appendix A: Specification language syntax

A.1 Specification language lexical structure

Specifications are Unicode text documents. The space, newline, carriage return, and horizontal tab characters are considered white space and separate lexical elements. The # character introduces a comment, which continues to the end of the line. The remainder of any specification is comprised of following tokens:

Keywords. The following are keywords in the language: and, are, as, becomes, begin, by, character, context-free, context, derives, encoding, end, exists, explication, in, initial, is, lexer, lookahead, matching, non-context-free, nonterminal, not, of, or, predicate, prohibition, remains, represents, specification, start, state, states, such, symbol, syntax, that, there, token, unchanged, unchecked, upon, where, {, }, (,), ?, |, /, ., +, -, *, /, **, :, =, <>, >, >=, <, and <=.

Identifiers. Any string of characters beginning with an alphabetic character (a-z or A-Z) and containing only alphabetic characters, arabic digits, or underbars that is not a keyword is an identifier. Any string beginning with a back-tick character (‘) and containing only alphabetic characters, arabic digits, or underbars is also an identifier. Thus, while end is a keyword, ‘end is an identifier. The leading back-tick character is considered a hint to lexers and should be discarded before subsequent processing so that identifier names with and without the back-tick are considered identical. Thus ‘apple is considered identical to apple. All identifiers in our specification format share a common name space. Developers must therefore take care not to use the same identifier for two different entities, such as a lexer state and a non-terminal.

Character literals. Character literals are a pair of single quotes containing either any character other than a single quote, backslash, newline, carriage-return, or horizontal tab; the escape sequences `\\`, `\'`, `\"`, `\t`, `\r`, or `\n`, representing the backslash, single-quote, double-quote, horizontal tab, carriage-return, or newline characters, respectively; or the escape sequence `\uXXXX`, where `XXXX` is a Unicode character number expressed as a four-digit hexadecimal number.

String literals. String literals are double quotes containing a string comprised of characters other than a single quotes, backslashes, newlines, carriage-returns, or horizontal tabs and the same escape sequences that are legal in character literals.

Integer literals. Integer literals are either `0` or strings of arabic digits without a leading zero but optionally with a leading negative sign (`-`).

Floating-point literals. Floating point literals consist of a whole part followed by an optional fraction part followed by an optional exponent part. The whole part is identical in form to an integer literal. The fraction part is a dot (`.`) followed by one or more arabic digits. The exponent part is an `e` or `E` followed by a string identical in form to an integer literal. Strings that match the form of both integer and floating-point literals should be considered to be integer literals.

Character class literals. Character class literals are written in a notation similar to that used by Perl: brackets surround the set of characters and ranges in the class (unless it is negated as discussed below). Characters other than right bracket, backslash, newline, carriage-return, dash, or horizontal tab may appear unescaped. The set of escape sequences permitted in character classes is identical to the set permitted in string and character liter-

als except that character classes may not contain the `\'` and `\"` escape sequences but may contain the `\]` escape sequence, which represents the right bracket character. As in Perl character classes, ranges are represented by a dash between characters defining the lowest and highest Unicode character values in the range. Again as in Perl character classes, a leading `^` character negates the character class, changing the meaning of its contents into the set of characters that are *not* in the character class being declared.

A.2 Specification language grammar

A specification consists of the keyword `specification` followed in order by a string literal giving the specification's name, the keywords `is` and `begin`, a lexer specification, a context-free syntax specification, a non-context-free syntax specification, a context specification, and the keyword `end`. That is:

```
Spec ::= "specification" StringLiteral "is" "begin"
      LexicalSpecification
      ContextFreeSyntaxSpecification
      NonContextFreeSyntaxSpecification
      ContextSpecification
      "end" <EOF>
```

A.2.1 Lexical specification

The lexical specification consists of the keywords `lexer specification is begin` followed in order by a character encoding declaration, a lexer states declaration, a default lexer state declaration, one or more token definitions, and the keyword `end`. That is:

```
LexerSpecification ::= "lexical" "specification" "is" "begin"
                   CharacterEncodingDefinition
                   StatesDeclaration
                   DefaultStateDeclaration
                   ( TokenDefinition )+
                   "end"
```

Character encoding declaration. A character encoding declaration consists of the keywords `character encoding is in {` followed in sequence by one or more string literals separated by commas and the keywords `right brace and dot (}.)`. The string identifiers give the names of the character encoding schemes that instance documents may use. Duplicate strings are not permitted. Formally:

```
CharacterEncodingDefinition ::= "character" "encoding" "is" "in" "{"
    StringLiteral ( "," StringLiteral )* "}" "."
```

Lexer states declaration. A lexer states declaration consists of the keywords `states are {` followed by one or more identifiers separated by commas and then by a right brace and dot. Each identifier represents one lexer state. Duplicate declarations are not permitted.

Formally:

```
StatesDeclaration ::= "states" "are" "{" <IDENTIFIER>
    ( "," <IDENTIFIER> )* "}" "."
```

Default lexer state declaration. A default lexer state declaration consists of the keywords `initial state is` followed by an identifier followed by a dot. The identifier is the initial lexer state, and must be one of the identifiers declared in the lexer states declaration. Formally:

```
DefaultStateDeclaration ::= "initial" "state" "is" <IDENTIFIER> "."
```

Token definition. A token definition consists of the keyword `token` followed in order by an identifier for the token, the keywords `is begin,` one or more match declarations, and the keyword `end.` Each match declaration consists of an optional from state declaration followed in order by the keywords `upon matching,` a regular expression, a state transition declaration, and a dot. The from state declaration, if present, consists of the keywords `in {` followed in order by one or more identifiers separated by commas and the keyword `}.`

The effect of the `from state` declaration, if present, is that the match is only considered applicable whenever the lexer is in a state matching one of the given identifiers. (Identifiers may not be repeated in `from state` declarations.) If absent, the match applies under all lexer states. The form of regular expressions is described in section A.2.5. The state transition declaration is either the keyword sequence `state remains unchanged` or the keywords `state becomes` followed by the identifier of the state that the lexer should transition to upon matching the token. For a discussion of precedence rules in lexer match declarations, see section 4.1.7. Formally:

```
TokenDefinition ::= "token" <IDENTIFIER> "is" "begin" (
    ( FromStateDeclaration )?
    "upon" "matching" RE
    "state" ( ( "becomes" <IDENTIFIER> ) |
              ( "remains" "unchanged" ) )
    "."
)+
"end"
```

```
FromStateDeclaration ::= "in" "{" <IDENTIFIER>
    ( "," <IDENTIFIER> )* "}"
```

The regular expressions used in token match declarations may not include terminal or non-terminal references. The special identifier `comments` is used for tokens matching comments, and the special identifier `whitespace` is used for tokens matching white space tokens are limited to character class, string literal, and or expression components; they may not include concatenation or closure expressions.

A.2.2 Context-free syntax specification

The context-free syntax specification consists of the keywords `context-free syntax` is `begin` followed in order by a look-ahead declaration, a start symbol declaration, one or more production declarations, and the keyword `end`. Formally:

```
ContextFreeSyntaxSpecification ::= "context-free" "syntax" "is" "begin"
    LookaheadDeclaration
```

```

StartSymbolDeclaration
( ProductionDeclaration )+
"end"

```

Look-ahead declaration. The look-ahead declaration consists of the keywords `lookahead` `by` followed by the integer number of symbols that an LL parser would need to look ahead by followed by a dot. Formally:

```

LookaheadDeclaration ::= "lookahead" "by" <INTEGER_LITERAL> "."

```

Start symbol declaration. The start symbol declaration consists of the keywords `start` `symbol` `is` followed by the identifier representing the symbol that corresponds to the an entire instance document followed by a dot. The identifier must be one of the identifiers declared in the production declaration. Formally:

```

StartSymbolDeclaration ::= "start" "symbol" "is" <IDENTIFIER> "."

```

Production declaration. Each production declaration consists of the keyword `symbol` followed in order by the identifier of the symbol being declared, the keywords `is` `begin`, a derives declaration, an optional represents declaration, and the keyword `end`. The identifier being declared must not have been declared in a previous token or production declaration. The derives declaration consists of the keyword `derives` followed in order by a regular expression and a dot, and indicates that the symbol being declared matches any string that can be eventually derived by the regular expression. The represents declaration consists of the keyword `represents` followed by a mixed string expression followed by a dot. Formally:

```

ContextFreeSyntaxSpecification ::= "context-free" "syntax" "is" "begin"
    LookaheadDeclaration
    StartSymbolDeclaration
    ( ProductionDeclaration )+
    "end"

```

```

LookaheadDeclaration ::= "lookahead" "by" <INTEGER_LITERAL> "."

```

```
StartSymbolDeclaration ::= "start" "symbol" "is" <IDENTIFIER> "."
```

```
ProductionDeclaration ::= "symbol" <IDENTIFIER> "is" "begin" "derives"
    RE "." ( "represents" MixedStringExpression "." ) ? "end"
```

The regular expressions used in production declarations may not contain string literal or character class elements. Mixed string declarations are described in section A.2.7.

A.2.3 Non-context-free syntax specification

The non-context-free syntax specification consists of the keywords `non-context-free syntax is begin` followed by any number of predicate, prohibition, or unchecked prohibition declarations, followed by the keyword `end`. Formally:

Predicate declaration. A predicate declaration begins with the keyword `predicate` followed in order by the identifier of the predicate being declared, a left parenthesis, a comma-separated list of identifiers representing arguments, a right parenthesis, the keywords `is begin`, a predicate expression, and the keyword `end`. The identifier may not have previously been declared as a terminal, symbol, predicate, prohibition, or unchecked prohibition. Formally:

```
PredicateDeclaration ::= "predicate" <IDENTIFIER> "(" <IDENTIFIER>
    ( "," <IDENTIFIER> )* ")" "is" "begin" PE1 "end"
```

Predicate expressions are described in section A.2.6.

Prohibition declaration. A predicate declaration begins with the keyword `prohibition` followed in order by the identifier of the prohibition being declared, the keywords `is begin`, a predicate expression, and the keyword `end`. The identifier may not have previously been declared as a terminal, symbol, predicate, prohibition, or unchecked prohibition. Formally:

```
ProhibitionDeclaration ::= "prohibition" <IDENTIFIER> "is" "begin"
    PE1 "end"
```

Unchecked prohibition declaration. An unchecked prohibition represents a non-context-free restriction expressed in natural language. Unchecked prohibitions need not be syntactic in nature. They consist of the keywords `unchecked prohibition` followed in order by the identifier of the unchecked prohibition being declared, the keywords `is begin`, a mixed text expression, and the keyword `end`. The identifier may not have previously been declared as a terminal, symbol, predicate, prohibition, or unchecked prohibition. Formally:

```
UncheckedProhibitionDeclaration ::= "unchecked" "prohibition"
    <IDENTIFIER> "is" "begin" MixedTextExpression "end"
```

A.2.4 Context specification

The context portion of a specification consists of the keywords `context is begin` followed in order by one or more term explications and the keyword `end`. Each term explication consists of the keywords `explication of` followed in order by an identifier representing the explicated concept, the keywords `is begin`, a mixed text expression explicating the concept, and the keyword `end`. Formally:

```
ContextSpecification ::= "context" "is" "begin"
    ( TermExplication )+ "end"
```

```
TermExplication ::= "explication" "of" <IDENTIFIER> "is" "begin"
    MixedTextExpression "end"
```

A.2.5 Regular expression

Regular expressions are used in the definition of tokens and non-terminals. Each regular expression can contain (depending on what it is used to define) string literals, character classes, identifiers corresponding to terminals, and identifiers corresponding to non-terminals. These can be combined in the usual manner with the `+`, `*`, `?`, `{x}` (exactly x repeti-

tions of), and $\{x, y\}$ (at least x and no more than y repetitions of) operators and parenthesis. Formally:

```

RE    ::= RE1 ( "|" RE1 ) *
RE1   ::= RE2 ( RE2 ) *
RE2   ::= RE3 ( ( "+" ) | ( "*" ) | ( "?" ) |
               ( "{" <INTEGER_LITERAL> ( "," <INTEGER_LITERAL> )? "}" )
               ) ?
RE3   ::= ( ( "(" RE ")" ) | ( <CHARCLASS> ) | ( <STRING_LITERAL> ) |
           ( <IDENTIFIER> ) )

```

A.2.6 Predicate expression

Predicate expressions are used to define predicates and prohibitions. The basic predicate language consists of predicate clauses combined using the logical operators `and`, `or`, and `not`. Predicate clauses include existential quantification expressions, `is` expressions, comparisons, and predicate references.

Existential quantification expressions consist of the keywords `there exists` followed in order by a comma-separated set of identifiers followed by the keywords `such that`, and serve to introduce new variables into the expression. The new variables must not be token, symbol, predicate, prohibition, unchecked prohibition, explicated term, or argument identifiers and must not be introduced by another existential quantification expression further up the parse tree.

`Is` expressions serve to establish arithmetic relationships between variables. They consist of an identifier representing a quantity followed in order by the keyword `is` and a predicate arithmetic expression.

Comparisons serve to establish comparison relationships between variables. They consist of an identifier representing a variable followed in order by `=`, `<>`, `<`, `<=`, `>`, or `>=`, followed by a predicate parameter expression.

Predicate references consist of the predicate's identifier followed in order by a left parenthesis, a comma-separated list of predicate parameter expressions representing the arguments, and a close parenthesis.

Formally:

```

PE1 ::= ( "there" "exists" <IDENTIFIER> ( "," <IDENTIFIER> )*
        "such" "that" )? PE2
PE2 ::= PE3 ( "or" PE3 )*
PE3 ::= PE4 ( "and" PE4 )*
PE4 ::= ( "not" PE5 ) | ( PE5 )
PE5 ::= ( "(" PE1 ")" ) |
        ( <IDENTIFIER>
          ( ( "=" PEP1 ) | ( "<>" PEP1 ) | ( "<" PEP1 ) |
            ( "<=" PEP1 ) | ( ">" PEP1 ) | ( ">=" PEP1 ) |
            ( "(" ( PEP1 ( "," PEP1 ) * )? ")" ) |
            ( "is" PEA1 )
          )
        )

```

Predicate arithmetic expression. A predicate arithmetic expression represents the right-hand-side of an `is` expression. It consists of references to variables, integer literals, floating-point literals, or character literals combined using parenthesis and the usual algebraic operators: `+`, `-`, `*`, `/`, and `**` (exponentiation). Formally:

```

PEA1 ::= PEA2 ( ( "+" PEA2 ) | ( "-" PEA2 ) )*
PEA2 ::= PEA3 ( ( "*" PEA3 ) | ( "/" PEA3 ) )*
PEA3 ::= PEA4 ( "**" PEA4 )*
PEA4 ::= ( ( ( "-" ) ) PEA4 ) | ( PEA5 )
PEA5 ::= ( ( <IDENTIFIER> ) | ( <INTEGER_LITERAL> ) |
          ( <FLOAT_LITERAL> ) | ( <CHAR_LITERAL> ) | ( "(" PEA1 ")" )
        )

```

Predicate parameter expression. A predicate parameter expression represents a portion of a predicate expression that can appear as a parameter in a predicate reference or as the right hand side of a comparison. Predicate parameter expressions can be string literals, character literals, integer literals, floating-point literals, variables, or lists. A list consists of a left brace followed in order by one or more comma-separated instances of the other

kinds of entities permitted in predicate parameter expressions, an optional tail expression, and a right brace. A tail expression, if present, consists of the keyword `|` followed by an identifier representing a variable equal to the remainder of the list. Formally:

```
PEP1 ::= ( PEP2 ) |
        ( "{" ( PEP2 ( "," PEP2 )* ( "|" <IDENTIFIER> )? )? }" )
PEP2 ::= ( StringLiteral ) | ( <CHAR_LITERAL> ) |
        ( <INTEGER_LITERAL> ) | ( <FLOAT_LITERAL> ) | ( <IDENTIFIER> )
```

A.2.7 Mixed string expression

A mixed string expression is used to represent a designation or explication. It consists of a string of string literals and references. References are enclosed in parenthesis and take the form of an identifier optionally followed by the keyword `as` and a string literal. The identifier of a reference is the identifier of a token, symbol, or designated term. The string following `as`, if present, gives an alternate text form of the literal name and is used to change the first character case, introduce characters that can't be used in identifier names, make singular terms plural, and the like. Formally:

```
MixedStringExpression ::= (
    ( <STRING_LITERAL> ) |
    ( "(" <IDENTIFIER> ( "as" <STRING_LITERAL> )? ")" )
```

**Appendix B: Rewritten specification for
MSAW site data files**

Because the complete re-written specification is very large, we include only portions of it here. Elided material has been replaced with comments indicating its nature.

```

specification "MSAW configuration" is begin

  lexical specification is begin
    character encoding is in { "ASCII" }.

    states are { at_bol, not_at_bol, expecting_table_name }.
    initial state is at_bol.

    token whitespace is begin
      in { at_bol, not_at_bol } upon matching [ \t]
        state becomes not_at_bol.
      in { expecting_table_name } upon matching [ \t]
        state remains unchanged.
      upon matching "\n" | "\r\n" state becomes at_bol.
    end

    token comments is begin
      upon matching "#" [^\r\n]* "\r"? "\n" state becomes at_bol.
      upon matching "%" [^\r\n]* "\r"? "\n" state becomes at_bol.
    end

    token t_begin is begin
      in { at_bol } upon matching "begin"
        state becomes expecting_table_name.
    end

    token t_end is begin
      in { at_bol } upon matching "end;" state becomes not_at_bol.
    end

    token t_sim is begin
      upon matching ";" state becomes not_at_bol.
    end

    token t_apr_definitions is begin
      in { expecting_table_name } upon matching "APT_DEFINITIONS"
        state becomes not_at_bol.
    end

#
# Token declarations for the names of the remaining tables have been elided.
#

    token t_empty_quoted_string is begin
      upon matching "\"\"" state becomes not_at_bol.
    end

    token t_int_literal is begin
      upon matching "[+\\-]? [0-9]+" state becomes not_at_bol.
    end

# Note: [+\\-]? [0-9]+ is also a float, but tokens of that
# form will be matched as t_int_literal.
    token t_float_literal is begin

```

```

        upon matching [+\\-]? ( ( [0-9]+ "." [0-9]* ) |
          ( [0-9]* "." [0-9]+ ) ) state becomes not_at_bol.
end

token t_lat_literal is begin
  upon matching [0-9]{1,2} ":" [0-9]{1,2} ":" [0-9]{1,2}
    ( "." [0-9]{1,3} )? [NS]? state becomes not_at_bol.
end

token t_lon_literal is begin
  upon matching [0-9]{1,3} ":" [0-9]{1,2} ":" [0-9]{1,2}
    ( "." [0-9] )? [EW]? state becomes not_at_bol.
end

token t_hex_literal is begin
  upon matching "0" [xX] [0-9A-F]+ state becomes not_at_bol.
end

token t_ip_address_literal is begin
  upon matching [0-9]{1,3} ( "." [0-9]{1,3} ) {3}
    state becomes not_at_bol.
end

token t_poly_kw is begin
  upon matching "POLY" | "\"POLY\"" state becomes not_at_bol.
end

token t_rect_kw is begin
  upon matching "RECT" | "\"RECT\"" state becomes not_at_bol.
end

token t_circ_kw is begin
  upon matching "CIRC" | "\"CIRC\"" state becomes not_at_bol.
end

token t_rng3_kw is begin
  upon matching "RNG3" | "\"RNG3\"" state becomes not_at_bol.
end

token t_rng2_kw is begin
  upon matching "RNG2" | "\"RNG2\"" state becomes not_at_bol.
end

token t_rng6_kw is begin
  upon matching "RNG6" | "\"RNG6\"" state becomes not_at_bol.
end

token t_polar_kw is begin
  upon matching "POLAR" | "\"POLAR\"" state becomes not_at_bol.
end

token t_cart_kw is begin
  upon matching "CART" | "\"CART\"" state becomes not_at_bol.
end

token t_latlon_kw is begin
  upon matching "LATLON" | "\"LATLON\"" state becomes not_at_bol.
end

```

```

token t_unquoted_string_literal is begin
    upon matching ([^\r\n \t])+ state becomes not_at_bol.
end

token t_quoted_string_literal is begin
    upon matching "\"" [^"]* "\"" state becomes not_at_bol.
end
end

context-free syntax is begin
    lookahead by 1.
    start symbol is nt_spec.

    symbol nt_spec is begin
        derives
            t_begin nt_apt_definitions_table t_end
            t_begin nt_fix_definitions_table t_end
#
# References to the remaining tables have been elided.
#
        .
    end

    symbol nt_apt_definitions_table is begin
        derives t_apt_definitions nt_apt_definitions_row+.
        represents "The " (nt_apt_definitions_table)
            " represents the set of " (airport as "airports")
            " in the " (operating_area as "operating area")
            " that have " (auto_associate_dissociate_region as
            "auto associate/dissociate regions") ". ".
    end

    symbol nt_apt_definitions_row is begin
        derives nt_aptDefinitions_aptName nt_aptDefinitions_isPrimary
            t_sim.
        represents "An " (nt_apt_definitions_row) " represents an "
            (airport) " in the " (operating_area as "operating area")
            " that has an " (auto_associate_dissociate_region as
            "auto associate/dissociate region") ". ".
    end

    symbol nt_aptDefinitions_aptName is begin
        derives nt_alphanum_t.
        represents "An " (nt_aptDefinitions_aptName)
            " represents the " (three_letter_airport_name as
            "three-letter airport name") " of the "
            (airport) " represented by the " (nt_apt_definitions_row)
            " that derives it.".
    end

    symbol nt_aptDefinitions_isPrimary is begin
        derives nt_string_literal.
        represents "Whether this airport is a " (primary_airport as
            "primary airport") " (indicated by the value \"PRI\") or "
            "not (indicated by the value \"NOPRI\"). ".
    end
end

```

```

#
# Declarations of the context-free syntax of the remaining tables has been
# elided.
#

symbol nt_generic_row is begin
  derives nt_string_literal* t_sim.
end

symbol nt_string_literal is begin
  derives
    t_empty_quoted_string
  | t_unquoted_string_literal
  | t_quoted_string_literal
  | t_int_literal
  | t_float_literal
  | t_lat_literal
  | t_lon_literal
  | t_hex_literal
  | t_poly_kw
  | t_rect_kw
  | t_circ_kw
  | t_rng3_kw
  | t_rng2_kw
  | t_rng6_kw
  | t_polar_kw
  | t_cart_kw
  | t_latlon_kw
  | t_begin
  .
end

symbol nt_char_t is begin
  derives nt_string_literal.
end

symbol nt_disp_char_t is begin
  derives nt_string_literal.
end

symbol nt_alpha_t is begin
  derives nt_string_literal.
end

symbol nt_alphanum_t is begin
  derives nt_string_literal.
end

symbol nt_alpha_alphanum_t is begin
  derives nt_string_literal.
end

symbol nt_int_literal is begin
  derives
    t_empty_quoted_string
  | t_int_literal
  .
end

```

```
symbol nt_non_null_int_literal is begin
  derives t_int_literal.
end

symbol nt_float_literal is begin
  derives
    t_empty_quoted_string
    | t_int_literal
    | t_float_literal
    .
end

symbol nt_non_null_float_literal is begin
  derives
    t_int_literal
    | t_float_literal
    .
end

symbol nt_oct_literal is begin
  derives
    t_empty_quoted_string
    | t_int_literal
    .
end

symbol nt_non_null_oct_literal is begin
  derives t_int_literal.
end

symbol nt_hex_literal is begin
  derives
    t_empty_quoted_string
    | t_hex_literal
    .
end

symbol nt_non_null_hex_literal is begin
  derives t_hex_literal.
end

symbol nt_ip_address_t is begin
  derives
    t_empty_quoted_string
    | t_ip_address_literal
    .
end

symbol nt_lat_literal is begin
  derives
    t_empty_quoted_string
    | t_lat_literal
    .
end

symbol nt_non_null_lat_literal is begin
  derives t_lat_literal.
end
```

```

symbol nt_lon_literal is begin
  derives
    t_empty_quoted_string
    | t_lon_literal
  .
end

symbol nt_non_null_lon_literal is begin
  derives t_lon_literal.
end

symbol nt_region is begin
  derives
    ( t_poly_kw (
      (t_latlon_kw nt_altitude_range nt_poly_latlon_region )
      | (t_cart_kw  nt_altitude_range nt_poly_cart_region  )
      | (t_polar_kw nt_altitude_range nt_poly_polar_region )
    ) )
    | ( t_rect_kw (
      (t_latlon_kw nt_altitude_range nt_rect_latlon_region )
      | (t_cart_kw  nt_altitude_range nt_rect_cart_region  )
      | (t_polar_kw nt_altitude_range nt_rect_polar_region )
    ) )
    | ( t_circ_kw (
      (t_latlon_kw nt_altitude_range nt_circ_latlon_region )
      | (t_cart_kw  nt_altitude_range nt_circ_cart_region  )
      | (t_polar_kw nt_altitude_range nt_circ_polar_region )
    ) )
    | ( t_rng3_kw (
      (t_latlon_kw nt_altitude_range nt_rng3_latlon_region )
      | (t_cart_kw  nt_altitude_range nt_rng3_cart_region  )
      | (t_polar_kw nt_altitude_range nt_rng3_polar_region )
    ) )
    | ( t_rng6_kw (
      (t_latlon_kw nt_altitude_range nt_rng6_latlon_region )
      | (t_cart_kw  nt_altitude_range nt_rng6_cart_region  )
      | (t_polar_kw nt_altitude_range nt_rng6_polar_region )
    ) )
    | ( t_rng2_kw (
      (t_latlon_kw nt_altitude_range nt_rng2_latlon_region )
      | (t_cart_kw  nt_altitude_range nt_rng2_cart_region  )
      | (t_polar_kw nt_altitude_range nt_rng2_polar_region )
    ) )
  .
end

symbol nt_altitude_range is begin
  derives nt_float_literal{2}.
end

symbol nt_poly_latlon_region is begin
  derives nt_latlon_coord{3} nt_latlon_coord*.
end

symbol nt_poly_cart_region is begin
  derives nt_poly_sensor nt_cart_coord{3} nt_cart_coord*.
end

symbol nt_poly_polar_region is begin

```

```
    derives nt_poly_sensor nt_polar_coord{3} nt_polar_coord*.
end

symbol nt_rect_latlon_region is begin
    derives nt_latlon_coord nt_cart_coord nt_float_literal.
end

symbol nt_rect_cart_region is begin
    derives nt_cart_sensor nt_cart_coord nt_float_literal.
end

symbol nt_rect_polar_region is begin
    derives nt_polar_sensor nt_cart_coord nt_float_literal.
end

symbol nt_circ_latlon_region is begin
    derives nt_latlon_coord nt_float_literal.
end

symbol nt_circ_cart_region is begin
    derives nt_cart_sensor nt_float_literal.
end

symbol nt_circ_polar_region is begin
    derives nt_polar_sensor nt_float_literal.
end

symbol nt_rng3_latlon_region is begin
    derives nt_latlon_coord nt_rng3_azimuth3.
end

symbol nt_rng3_cart_region is begin
    derives nt_cart_sensor nt_rng3_azimuth3.
end

symbol nt_rng3_polar_region is begin
    derives nt_polar_sensor nt_rng3_azimuth3.
end

symbol nt_rng6_latlon_region is begin
    derives nt_latlon_coord nt_rng6_azimuth6.
end

symbol nt_rng6_cart_region is begin
    derives nt_cart_sensor nt_rng6_azimuth6.
end

symbol nt_rng6_polar_region is begin
    derives nt_polar_sensor nt_rng6_azimuth6.
end

symbol nt_rng2_latlon_region is begin
    derives nt_latlon_coord nt_rng2_azimuth2.
end

symbol nt_rng2_cart_region is begin
    derives nt_cart_sensor nt_rng2_azimuth2.
end
```

```

symbol nt_rng2_polar_region is begin
  derives nt_polar_sensor nt_rng2_azimuth2.
end

symbol nt_sensor_name is begin
  derives nt_string_literal.
end

symbol nt_poly_sensor is begin
  derives nt_sensor_name.
end

symbol nt_cart_sensor is begin
  derives nt_sensor_name nt_cart_coord.
end

symbol nt_polar_sensor is begin
  derives nt_sensor_name nt_polar_coord.
end

symbol nt_latlon_coord is begin
  derives nt_lat_literal nt_lon_literal.
end

symbol nt_cart_coord is begin
  derives nt_float_literal{2}.
end

symbol nt_polar_coord is begin
  derives nt_float_literal{2}.
end

symbol nt_rng3_azimuth3 is begin
  derives nt_float_literal{6}.
end

symbol nt_rng6_azimuth6 is begin
  derives nt_rng2_azimuth1{3}.
end

symbol nt_rng2_azimuth2 is begin
  derives nt_float_literal{4}.
end

symbol nt_rng2_azimuth1 is begin
  derives nt_float_literal{3}.
end

end

non-context-free syntax is begin
  predicate list_length(list, length) is begin
    list_length_acc(list, 0, length)
  end

  predicate list_length_acc(list, acc, length) is begin
    there exists a, t, i such that
      ( list = {} and
        length = acc

```

```

    ) or
    (   list = { a | t } and
      i is (acc + 1) and
      list_length_acc(t, i, length)
    )
  end

predicate char_is_in(char, charlist) is begin
  there exists c, rest such that
  charlist = { c | rest } and
  (   c = char or
    char_is_in(char, rest)
  )
end

#
# Other utility predicates elided.
#

predicate string_literal(id, string) is begin
  there exists tid, qstring such that
  nt_string_literal(id, { tid }) and
  (   (   t_empty_quoted_string(tid, qstring) and
        unquote_string(qstring, string)
      ) or
    t_unquoted_string_literal(tid, string) or
    (   t_quoted_string_literal(tid, qstring) and
      unquote_string(qstring, string)
    ) or
    t_int_literal(tid, string) or
    t_float_literal(tid, string) or
    t_lat_literal(tid, string) or
    t_lon_literal(tid, string) or
    t_hex_literal(tid, string) or
    t_poly_kw(tid, string) or
    t_rect_kw(tid, string) or
    t_circ_kw(tid, string) or
    t_rng3_kw(tid, string) or
    t_rng2_kw(tid, string) or
    t_rng6_kw(tid, string) or
    t_polar_kw(tid, string) or
    t_cart_kw(tid, string) or
    t_latlon_kw(tid, string) or
    t_begin(tid, string)
  )
end

predicate string_literal_with_default(id, string, default) is begin
  there exists s such that
  string_literal(id, s) and
  (   (   s = "" and
        string = default
      ) or
    (   s <> "" and
      string = s
    )
  )
end

```

```

predicate unquote_string(quoted, unquoted) is begin
  there exists tail such that
    quoted = { '"' | tail } and
    unquote_string_tail(tail, unquoted)
end

predicate unquote_string_tail(quoted, unquoted) is begin
  there exists c, t1, t2 such that
    (   quoted = "\"" and unquoted = ""
    ) or
    (   quoted = { c | t1 } and
      unquoted = { c | t2 } and
      unquote_string_tail(t1, t2)
    )
end

predicate char_t(id, string) is begin
  there exists sid such that
    nt_char_t(id, { sid }) and
    string_literal(sid, string)
end

prohibition no_invalid_char_t is begin
  there exists id, s such that
    char_t(id, s) and not
    (   string_chars_are_in(s,
      " !\"#$%&'()*+,-./0123456789:;<=>?@"
      "ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`"
      "abcdefghijklmnopqrstuvwxyz{|}~")
      # NOTE: we are assuming space is printable here --
      # that's not universal, but it is reasonable.
    )
end

#
# Predicates and prohibitions for the remaining string types elided.
#

predicate nn_int_literal(id, n) is begin
  there exists sid such that
    nt_non_null_int_literal(id, { sid }) and
    int_literal_value(sid, n)
end

predicate int_literal(id, n, default) is begin
  there exists sid such that
    nt_int_literal(id, { sid }) and
    (   (   there exists s such that
          t_empty_quoted_string(sid, s) and
          n = default
        ) or
      (   int_literal_value(sid, n)
        )
    )
end

predicate int_literal_value(ilid, n) is begin
  there exists s such that

```

```

t_int_literal(ilid, s) and
( ( there exists rest1, m such that
  s = { '-' | rest1 } and
  int_literal_value_acc(rest1, 0, m) and
  n is -m
) or
( there exists rest2 such that
  s = { '+' | rest2 } and
  int_literal_value_acc(rest2, 0, n)
) or
( int_literal_value_acc(s, 0, n)
)
)
end

```

```

predicate int_literal_value_acc(s, acc, n) is begin
( s = "" and
  n = acc
) or
( there exists c, rest, nextacc such that
  s = { c | rest } and
  c >= '0' and
  c <= '9' and
  nextacc is ((acc * 10) + (c - '0')) and
  int_literal_value_acc(rest, nextacc, n)
)
end

```

```

predicate oct_literal(id, n, default) is begin
there exists id1 such that
nt_oct_literal(id, { id1 }) and
( ( there exists s such that
  t_empty_quoted_string(id1, s) and
  n = default
) or
  oct_literal_value(id1, n)
)
end

```

```

predicate nn_oct_literal(id, n) is begin
there exists id1 such that
nt_non_null_oct_literal(id, { id1 }) and
oct_literal_value(id1, n)
end

```

```

predicate oct_literal_value(id, n) is begin
there exists s such that
t_int_literal(id, s) and
oct_literal_value_acc(s, 0, n)
end

```

```

predicate oct_literal_value_acc(s, acc, n) is begin
( s = "" and
  n = acc
) or
( there exists c, rest, nextacc such that
  s = { c | rest } and
  c >= '0' and
  c <= '7' and

```

```

        nextacc is ((acc * 8) + (c - '0')) and
        oct_literal_value_acc(rest, nextacc, n)
    )
end

prohibition no_invalid_oct_literal is begin
    there exists id1, id2, s such that
    (
        nt_oct_literal(id1, { id2 }) or
        nt_non_null_oct_literal(id1, { id2 })
    ) and
    t_int_literal(id2, s) and
    not string_chars_are_in(s, "01234567")
end

#
# Predicates and prohibitions for the other kinds of literals elided.
#

predicate apt_definitions(id, aptName, isPrimary) is begin
    there exists anid, ipid, simid, ansid, ipsid such that
    nt_apt_definitions_row(id, { anid, ipid, simid }) and
    nt_aptDefinitions_aptName(anid, { ansid }) and
    nt_aptDefinitions_isPrimary(ipid, { ipsid }) and
    alphanum_t(ansid, aptName) and
    string_literal(ipsid, isPrimary)
end

prohibition no_invalid_apt_definitions is begin
    there exists id1, an1, ip1 such that
    apt_definitions(id1, an1, ip1) and
    (
        (
            there exists id2, ip2 such that
            apt_definitions(id2, an1, ip2) and
            id1 <> id2
            # NOTE: it doesn't actually say in the original spec
            # that these must be unique. We made that up.
        ) or
        (
            not
            (
                string_length_is(an1, 3) and
                string_is_in(ip1, { "PRI", "NOPRI" })
            )
        )
    )
end

prohibition apt_definitions_row_count_legal is begin
    there exists id, rows, count, max such that
    nt_apt_definitions_table(id, rows) and
    list_length(rows, count) and
    const_max_naptq(max) and
    count > max
end

predicate fix_definitions(id, fixName) is begin
    there exists fnid, simid, fnsid such that
    nt_fix_definition_row(id, { fnid, simid }) and
    nt_fixDefinition_fixName(fnid, { fnsid }) and
    alphanum_t(fnsid, fixName)
end

```

```

prohibition no_invalid_fix_definitions is begin
  there exists id1, fn1 such that
  fix_definitions(id1, fn1) and
  (
    (
      there exists id2 such that
      fix_definitions(id2, fn1) and
      id1 <> id2
      # NOTE: it doesn't actually say in the original spec
      # that these must be unique. We made that up.
    ) or
    (
      there exists id3, ip2 such that
      apt_definitions(id3, fn1, ip2)
    ) or
    (
      not
      (
        string_length_is(fn1, 3)
      )
    )
  )
end

prohibition fix_definitions_row_count_is_legal is begin
  there exists id, rows, count, max_aptq, max_naptq, max such that
  nt_fix_definitions_table(id, rows) and
  list_length(rows, count) and
  const_max_naptq(max_naptq) and
  const_max_aptq(max_aptq) and
  max is (max_aptq - max_naptq) and
  count > max
end

#
# Predicates and prohibitions for the remainder of the tables elided.
#

predicate const_max_ahoaq(n) is begin
  n = 127
end

#
# Predicates for the remaining constants have been elided.
#

end

context is begin

  explication of airport is begin
    "What is an airport, anyway???"
  end

#
# Explications of the remaining terms elided.
#

end

end

```