Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5540--96-7872

The RS-232 Character Repeater Refinement and Assurance Argument

Andrew P. Moore

Center for High Assurance Computer Systems
Information Technology Division

Charles N. Payne

Secure Computing Corporation
Roseville, MN

July 25, 1996

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget. Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE July 25, 1996	3. REPORT TYPE AND DATES COVERED	
4. TITLE AND SUBTITLE The RS-232 Character Repeater Refinement and Assurance Argument			5. FUNDING NUMBERS 55-3284-0-6 PE: 334710G
6. AUTHOR(S) Andrew P. Moore and Charles N. Payne*			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5320			8. PERFORMING ORGANIZATION REPORT NUMBER NRL/MR/5540--96-7872
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency 9800 Savage Road Ft. Meade, MD 20755	COMSPAWARSYSCOM Washington, DC 20363		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES *Secure Computing Corporation 2675 Long Lake Road Roseville, MN 55113			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE
13. ABSTRACT (*Maximum 200 words*) Past experience in system security certification indicates the need for developers of high assurance systems to coherently integrate the evidence that their system satisfies its critical requirements. This document describes a method based on literate programming techniques to help developers present the evidence they gather in a manner that facilitates the certification effort. We demonstrate this method through the implementation and verification of a small but nontrivial, security-relevant example, an RS-232 character repeater. By addressing many of the important issues in system design, we expect that this example will provide a model for developing assurance arguments for full-scale composite systems with corresponding gains in the expediency of the system certification process.			
14. SUBJECT TERMS Assurance Certification Literate programming	System documentation Formal methods		15. NUMBER OF PAGES 124
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

Contents

The RS-232 Character Repeater Refinement and Assurance Argument

Chapter 1

Introduction

1.1 Goal

The goal of this document is to demonstrate a method for coherently integrating the evidence that a computing system satisfies its critical requirements. A *critical requirement* is any requirement that, if not satisfied, could result in catastrophic behavior such as loss of life or the unauthorized disclosure of classified information. The method we use is particularly appropriate for systems that require high assurance of such critical requirements.

1.2 Motivation

The evidence that a system satisfies its critical requirements is typically assessed by an independent certification team during the accreditation phase of the system's development cycle. Previous experience [25] developing a high assurance cryptographic controller called the ECA has taught us that, with respect to independent certification, the presentation of this evidence is at least as important as the kind of evidence gathered. The evidence should be presented as a coherent and integrated whole, which we call the assurance argument.

The certification of the ECA was largely unsuccessful. Although the developers had confidence that the ECA conformed to its security requirements, we were not able to convince the certifiers of this fact. This is partly due to a late start of the certification effort. The primary reason, however, was that the evidence, as presented, was not very convincing to those not intimately involved in the ECA development. The ECA documentation provided (1) inadequate guidance on how to piece the evidence together into a convincing assurance argument, and (2) inadequate assurance that the evidence gathered was relevant to the ECA implementation. This made it difficult for the certifiers to identify potential problems in the implementation. They had little reason to be convinced by the evidence provided.

Developers need methods to help them present the evidence they gather in a manner convincing to system certifiers. Documenting a convincing assurance argument is complicated by several factors. First, non-trivial systems usually require the use of many different methods, both formal and informal, during the development process, e.g., for requirements analysis, design simulation and implementation. Disparate notations and methodological paradigms threaten the coherence of the assurance argument and put its certification at risk. Second, many aspects of an assurance argument may not be easily formalized, e.g., design decisions, strategies and assumptions. The analysis of these aspects are necessarily more subjective than those aspects that are formalized. Finally, although formal methods are more precise, they can also be less intuitive than informal methods. The documentation must explain and motivate the formalisms used.

Manuscript approved April 26, 1996.

1.3 Approach

Literate programming methods and tools [11] provide a foundation for solving the problems associated with documenting and managing a convincing assurance argument. The concept of literate programming is simple:

> A traditional computer program consists of a text file containing program code. Scattered in amongst the program code are comments which describe the various parts of the code.
>
> In literate programming the emphasis is reversed. Instead of writing code containing documentation, the literate programmer writes documentation containing code. No longer does the English commentary injected into a program have to be hidden in comment delimiters at the top of the file, or under procedure headings, or at the end of lines. Instead, it is wrenched into the daylight and made the main focus. The "program" then becomes primarily a document directed at humans, with the code being herded between "code delimiters" from where it can be extracted and shuffled out sideways to the language system by literate programming tools.
>
> The effect of this simple shift of emphasis can be so profound as to change one's whole approach to programming. Under the literate programming paradigm, the central activity of programming becomes that of conveying meaning to other intelligent beings rather than merely convincing the computer to behave in a particular way. It is the difference between performing and exposing a magic trick. [30]

We extend the use of literate programming beyond traditional programming to encompass the specification and verification process as well. The application that we choose to demonstrate this approach, an RS-232 character repeater, was originally posed as a non-trivial, security-relevant example on which to determine the feasibility of formal methods [15]. Our use of literate programming techniques (1) demonstrates how a formal assurance argument can be presented in a clear and intuitive manner and (2) ensures that the documentation of the argument is consistent with the actual specification, implementation, and proof. This document was written using literate programming techniques and tools and is itself a literate program.

The refinement of the repeater addresses many important issues that commonly arise in the development of more complex hardware/software systems. We analyze very abstract requirements of the repeater at the top level. We specify and verify both a logical design and a physical design of the repeater. We deal directly with concurrency in both the logical and physical designs. This approach required significant effort, as is evidenced by the length and complexity of this document. Nevertheless, by addressing these issues we demonstrate the potential scalability of the method. We expect this document will provide a model for developing assurance arguments for full-scale composite systems.

1.4 Structure of this Document

This document contains four parts. Part I describes the repeater problem and solution strategy; we adopt CSP [9] as the computational framework and casts the repeater critical requirements in terms of this model. Part II describes the repeater logical design, specification, and verification using the EVES interactive proof system [13]. Part III describes the physical implementation of this design and the verification that the implementation conforms to the design constraints using the Failures Divergence Refinement (FDR) model checker [26]. Part IV defines concepts that are used in the first three parts. This part presents a brief description of EVES library units that comprise the EVES and CSP background theory and repeater application modules.

Part I contains three chapters. Chapter 2 informally describes the repeater problem and solution. Chapter 3 presents an overview of the notation used to specify, verify, and implement the repeater and the notation used to document the repeater assurance argument. This chapter provides enough detail of the notations used for a fairly thorough overview of the assurance argument. It is organized for ease of reference with much of the relevant notation summarized in tables in the appendices. Chapter 3 also provides pointers to relevant documentation (user manuals, tutorials, etc.) for readers interested in a deeper understanding of the repeater assurance argument. Chapter 4 refines the critical requirements for the repeater based on the assumptions of the CSP model. A subset of these requirements expressible as CSP trace specifications are formalized in the EVES syntax. Chapter 4 forms the foundation for the rest of the assurance argument. The argument evolves as the repeater is refined using the formal techniques where possible and informal techniques where necessary to ensure full coverage of the critical requirements.

Part II contains three chapters. Chapter 5 specifies the repeater logical architecture as the concurrent composition of two components. It decomposes the critical requirements described in Chapter 4 accordingly and justifies the integrity of the decomposition using EVES for the formal aspects of the argument. Chapters 6 and 7 present the design for each component and verify that the design satisfies the derived requirements, again using EVES as appropriate.

Part III contains two chapters. Chapter 8 recasts the repeater design specified in Part II in the FDR syntax. Chapter 9 implements this design and verifies, using FDR as appropriate, that the implementation conforms to the requirements of the design. This process generates a set of critical requirements that must be satisfied by any further refinement. Although this is the extent of the repeater refinement in this document, this specification is consistent with the repeater specified in [1], from which a VHDL design and gate-level hardware description were derived.

Part IV contains four chapters describing, respectively, a theory for reasoning about character sequences, a module for representing character sequences as buffers, an interface to an abstract base machine providing CSP-like primitives, and an outline of the library units that comprise the repeater specification.

Part I

The Repeater Problem

Chapter 2

Informal Problem Statement and Solution Strategy

The RS-232 character repeater, henceforth referred to simply as the repeater, relays all characters of correct parity until it overflows. More specifically, the repeater has an input data line, an output data line, and an error port, as shown in Figure 2.1, that can operate at a range of speeds. Characters (each consisting of a sequence of `K` bits) received from the input data line are transmitted at the output data line, possibly after some delay. All characters of correct parity (assume even parity coding) received by the repeater are stored, until transmission, in an internal buffer that can grow to a maximum size of `N>0` `K`-bit characters. Characters of odd parity are not retransmitted and cause an error to be signaled on the error port. If the buffer overflows (grows to a size greater than `N` characters), an error is signaled on the error port, nothing more is accepted, all characters in the buffer received prior to the character causing the error condition are transmitted, and the repeater halts.

The repeater must be rigorously shown to satisfy the critical requirements described in Section 2.1 We use the strategy outlined in Section 2.2. We justify this strategy in Section 2.3.

2.1 Critical Requirements

Information security is increasingly recognized as a property of an information system as a whole, rather than a property of its components [31]. This raises the question as to why a low-level device like a character repeater was viewed as security-relevant by the originators of the problem. Typically as one focuses on the requirements of smaller and smaller components of a system, those requirements become less and less identifiable as security-relevant. Nevertheless, certain commonly

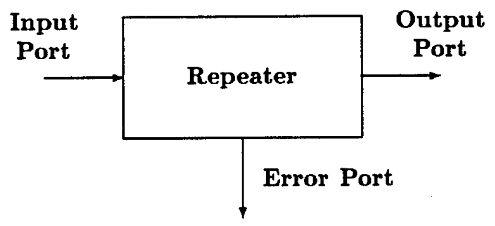


Figure 2.1: The RS-232 Character Repeater

reusable components, such as a memory management unit, have features that are generally useful in the construction of secure systems [14]. This is true as well for the repeater.

Confidentiality and integrity of information are important concerns for building secure systems. The primary concern for the repeater is the integrity of the data it processes, i.e., ensuring the data is not corrupted in some way. The first critical requirement demands that the characters received by the repeater are transmitted without change. Enforcing a strong notion of integrity might require using an error-correcting code; that is, however, beyond the scope of this repeater design.

> **Relay Characters Until Overflow** — Exactly those characters of even parity received by the repeater prior to the reception of the character causing an overflow are transmitted, K-bit character by K-bit character.

The repeater must also be concerned with ensuring that no new, possibly classified, information gets inserted into the character stream. The second and third critical requirements address which characters get transmitted and their ordering. Preserving the order of characters is necessary both to ensure that no sensitive information is encoded in a re-ordering (confidentiality) and to ensure that the data goes out as it came in (integrity).

> **No Spurious Characters** — Only received characters are transmitted.

> **Order Preserved** — Characters are transmitted in the order in which they were received.

2.2 Solution Strategy

The CSP language [9] forms the basis for the specification of the repeater and the verification that it satisfies its critical requirements. The EVES Verification System [5, 13] and the FDR model checker [6, 26] provide mechanical assistance for constructing and verifying the CSP specifications. EVES is a general purpose interactive verification system that can be used to prove mathematically that CSP descriptions conform to trace specifications. NRL extended an existing CSP theory [24, 12] to support such verifications [19]. FDR is a CSP model checker that automatically verifies (through an exhaustive state space analysis) that a CSP process implementation properly refines a CSP process specification.

Figure 2.2 illustrates our approach for constructing the assurance argument for the repeater CSP process implementation. Slanted arrows indicate a *refinement* of a specification to a more detailed specification or implementation; vertical arrows indicate a *translation* of a specification from one semantic domain to another semantic domain at a comparable specification level. Dashed arrows indicate a refinement/translation that is *informal*; solid arrows indicate a refinement that uses a combination of *informal* and *formal* techniques. The increase in width of the argument from top to bottom illustrates additional detail that is specified at the lower levels. Other work describes the synthesis of a gate-level hardware description from the repeater CSP physical design using VHDL synthesis tools [1].

The CSP computational paradigm makes a number of assumptions about the environment and implementation of CSP processes. Since we use this model to derive the repeater physical design, these assumptions result in critical requirements in addition to those described in the previous section. These additional critical requirements are not expressible in terms of the CSP model and, thus, must be verified informally. Tracing all the critical requirements through the levels of repeater refinement is a key aspect of the assurance argument that is not explicit in Figure 2.2.

Figure 2.3 illustrates the trace of requirements through the repeater logical and physical designs to the verification that those designs satisfy their derived requirements. An arrow from A to B means that A contributes to the derivation of B. Critical requirements for the repeater's physical architecture derive both from the top-level critical requirements and the fact that the detailed logical

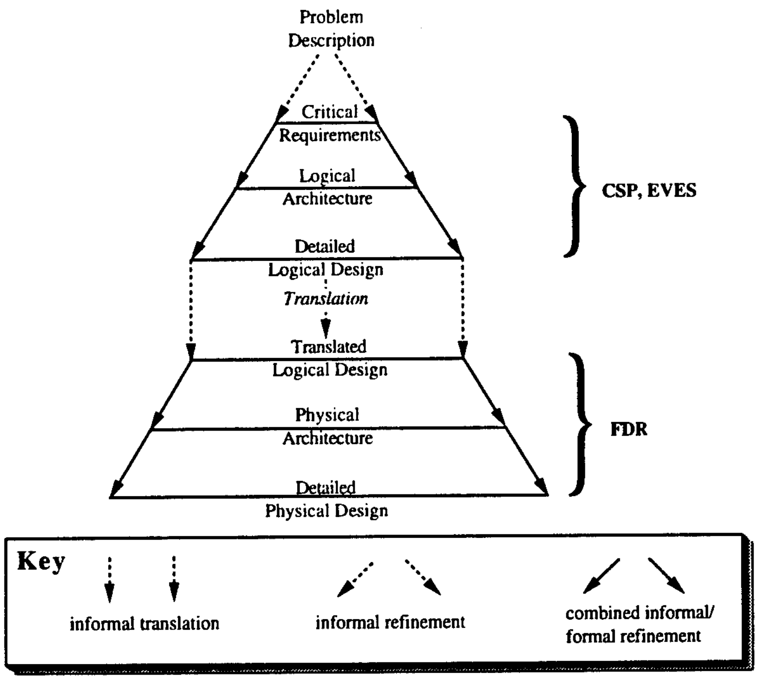


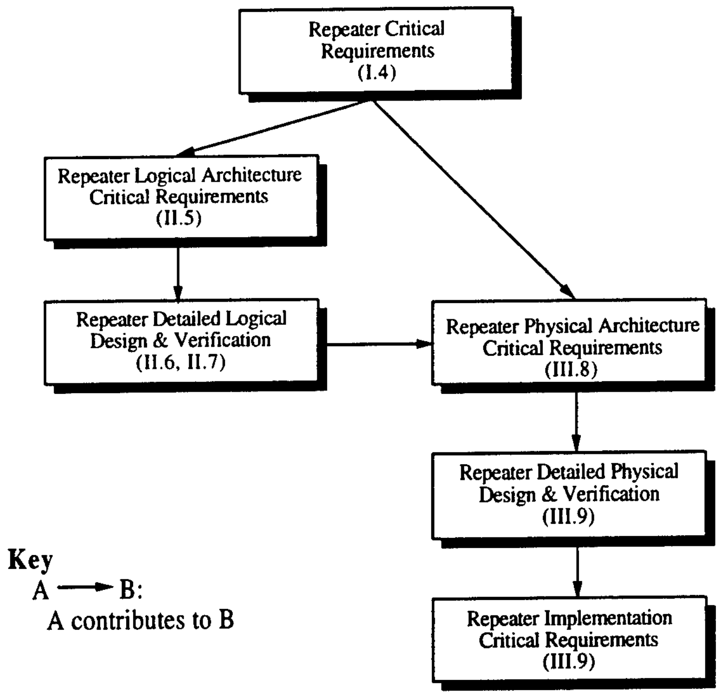
Figure 2.2: Repeater Assurance Argument

design satisfies the top-level critical requirements. This latter fact simplifies the physical architecture requirements since we need to show only that the physical design conforms to the logical design to guarantee that the requirements of Section 2.1 hold. A set of requirements for further refinement of the repeater are generated out of the process of refining the physical architecture.

Figure 2.3 provides part and chapter numbers indicating where in this document the primary elements of the requirements trace reside. Critical requirements are classified as either assumptions or assertions. *Assumptions* are those requirements that are necessary to satisfy the requirements of Section 2.1, but that can only be enforced by the repeater's environment. *Assertions* are those critical requirements that can be enforced by the repeater itself. *Formal* assumptions/assertions can be stated in the CSP model; *informal* assumptions/assertions are beyond the expressive power of the CSP model and are stated in English. We identify new or derived assumptions and assertions "in-line" during the presentation of a specification and summarize them near the end of the chapter in which they are identified. We use a numbering scheme similar to the chapter/section numbering scheme so that critical requirements can easily be traced back to their origin, e.g., "Assert 3.2" represents the second derived assertion that originated from assertion 3.

Refinement of the repeater design requires justification. Each chapter representing such a refinement concludes with a section that justifies the refinement. Each justification relies on a combination of formal and informal arguments. Formal arguments are used to justify that a repeater design conforms to its formal assertions; informal arguments are used to justify that a repeater design conforms to its informal assertions. We use narrative text to motivate and outline the formal arguments, but we rely on the mechanical tools to convince the reader that the details of the formal argument are correct.

Formal and informal arguments use identical notation to identify exactly what is to be proven and what facts are needed in the proof. The expression `c` using `a1,a2,...,an` describes a theorem in which `a1` through `an` are assumptions and `c` is the conclusion. This expression can be read "`c` follows from the assumptions `a1` through `an`." The justification of each theorem immediately follows



Repeater Critical Requirements (I.4)
Repeater Logical Architecture Critical Requirements (II.5)
Repeater Detailed Logical Design & Verification (II.6, II.7)
Repeater Physical Architecture Critical Requirements (III.8)
Repeater Detailed Physical Design & Verification (III.9)
Repeater Implementation Critical Requirements (III.9)
Key
A → B:
A contributes to B

Figure 2.3: Repeater Requirements Tracability

the sequent representing the theorem. Each argument ends with symbol □. Critical requirements that do not change through the decomposition are signified by the name of the requirement followed by the phrase "No change."

The rest of this section provides an overview of the tools that we use to specify and document the repeater assurance argument, with pointers to relevant documentation for a more thorough treatment.

2.2.1 Overview of Tools Used

The question arises why we use both interactive proof and model checking technology to construct the repeater argument. The use of interactive mathematical proof as a means of gaining assurance that an implementation conforms to its critical requirements is a (human) labor intensive process. Model checking can provide a comparable level of assurance for portions of a system verification at a substantial savings in time and human labor. Unfortunately, model checking currently applies only to the verification of relatively low-level requirements. FDR, for example, is used to verify that an implementation conforms to a CSP process specification. We need the EVES interactive prover to verify that the CSP process specification conforms to a more intuitive statement of the critical requirements.

CSP: The Computational Framework

CSP allows the description of systems composed of networks of communicating processes. A CSP process communicates with its environment through named communication channels. Olderog and Hoare [22] describe a family of increasingly sophisticated models for CSP; less sophisticated members of the family enable specification and proof of a subset of properties that the more sophisticated members enable. The Traces Refinement model is useful for ensuring that safety properties are preserved; the Failures Refinement is useful for ensuring that safety and liveness properties are preserved; and the Failures-Divergences Refinement model is useful for ensuring that safety and liveness properties are preserved and that the system does not diverge.[1] We chose the Traces Refinement model as the basis for the repeater assurance argument due to its comparative simplicity and its ability to prove safety properties of networks of processes.

The Traces Refinement model characterizes a process according to it's alphabet and set of traces. The alphabet of a process specifies all communication events, i.e., channel-value pairs, in which it is permitted to engage. A trace of a process is an observation of its execution. It consists of a finite sequence of all communication events in which the process has engaged at some moment in time. Properties specified about systems described in CSP take the form of restrictions on the traces in which a process representing the system may engage. If the set of traces associated with the process actually conform to these restrictions, the system is said to satisfy the properties.

The formal assertions are specified using the Traces Refinement model of CSP to allow formal specification and proof of the repeater as a network of communicating components. The repeater architecture is reflected in a CSP description. A decomposition of the Traces Refinement model requirements (i.e., trace specifications) onto the major components of the architecture is performed using the CSP proof theory in conjunction with a method developed at NRL [21]. This decomposition is performed down to the level of sequential CSP processes. These CSP processes are then implemented and proven to conform to their derived requirements using the CSP proof theory.

EVES: An Interactive Proof Assistant

EVES consists of a specification and programming language called Verdi, a proof obligation generator, and an interactive proof assistant called NEVER. An alternative syntax for Verdi, called Sugared Verdi (SVerdi), has been developed that is somewhat more conventional (Pascal-like) than the Lisp s-expression syntax of Verdi. We use SVerdi, rather than Verdi, in the refinement of the repeater assurance argument for increased readability. SVerdi consists of both executable (programming) constructs and non-executable (specification) constructs. SVerdi includes imperative statements (similar to Pascal), types for executable constructs, set theoretic concepts (including the axiom of choice) [7], first-order logic, and declarations such as mutually recursive procedures and functions, axioms, and types.

SVerdi development using EVES starts from an initial built-in theory, which is documented in Appendix C of [4]. The EVES database consists of the initial theory extended as appropriate by any declarations parsed. Each declaration parsed into EVES extends the database with new symbols and axioms. To maintain consistency of the database, the proof obligation generator constructs the formulas that need to be proven for each declaration parsed. The EVES database keeps track of these proof obligations and requires their proof before the consistency of the database may be declared. Additions to the database for each class of declaration are described in Appendix D of [4].

FDR: A CSP Model Checker

FDR offers the choice of verification using any of the three models of CSP: Traces Refinement, Failures Refinement, and Failures-Divergences Refinement. Although the CSP theory developed for EVES uses only Traces Refinement, the more sophisticated types of refinement are useful for the repeater argument to ensure that the repeater physical design makes progress processing characters and does not diverge. We, therefore, perform the more sophisticated types of refinement analysis during the FDR verification, since the analysis is performed automatically, but limit the EVES verification to the Traces Refinement, since the supporting theory for the more sophisticated models has yet to be encoded in SVerdi.

[1] A system is non-divergent if all recursion is guarded and there is no possibility of the network engaging in an infinite consecutive sequence of hidden events, i.e. livelock.

We claimed earlier that verification in FDR is automatic. Of course, this is true only if the CSP implementation described actually refines the CSP specification according to the type of refinement of interest. If the CSP implementation is in error, it is up to the developer to find and correct the error. FDR provides a graphical interface for determining the source of errors by analyzing the trace of events that led up to the error. The information provided depends on the type of refinement analysis performed.

FunnelWeb: A Literate Programming Tool

Literate programming (LP) tools allow users to produce typeset documentation and compiler-ready code from the same source document(s). This capability, along with automatic cross-referencing and the ability to interleave code and documentation in *any order*, gives the user great flexibility in the presentation of the program.

The original LP tool was Knuth's **WEB** toolset [10][2] for writing Pascal. LP tools for other programming languages quickly followed, and eventually programming language-independent tools became available. FunnelWeb [30], developed by Dr. Ross Williams of the University of Adelaide, Australia, is a language-independent LP tool. In fact, it supports many, arbitrary languages, which makes it a good candidate for this exercise. From a single document, we can produce not only the typeset assurance argument, but also the EVES and FDR specifications.

Except for some notational conventions described in Section 3.4, FunnelWeb's use should be invisible to the reader. Instead, the reader can be assured that the formal specifications herein are exactly those specifications processed by EVES and FDR.

2.3 Justification of the Strategy

The repeater problem statement, described in the introduction to this chapter, requires the repeater to receive and transmit bits at a range of speed. Support for asynchronous input and output suggests that we use a language suited to reasoning about asynchronously communicating components. Unfortunately, few mechanical tools directly support formal reasoning about such systems. The Gypsy Verification Environment [8] is useful for the verification of a variety of concurrent applications, but limits on the form of the specification and implementation of concurrent programs make it awkward to use for the repeater. At this writing, it is difficult, for example to specify in Gypsy program exit conditions of the form "at the time a message is received over channel A, the history of channel B satisfies property P."

Process algebras alleviate such problems by interleaving the individual channel histories in the order in which communications take place. Since the repeater is naturally described as a set of communicating processes and the repeater requirements are naturally stated as restrictions on traces of communication events, the process algebraic approach is ideal. While there are many process algebras documented in the literature, most are related in some way to one of the two early process algebras: CSP and CCS [17]. The tools that have been developed to aid reasoning about CSP and CCS are roughly comparable. However, we choose to use CSP because we had ready access to the CSP tools and we had experience using CSP to specify and verify security properties for a number of applications. Furthermore, the CSP Traces Refinement model is relatively easy to understand, allowing a clear exposition of the assurance argument.

Our solution strategy balances the assurance gained with the development cost incurred (including both human resources used and time to completion). A number of tools exist that permit users to graphically depict the simulation of communicating processes, e.g., LOTOS [29], but we decided to limit the scope of our effort to techniques complementary to testing in the verification process. This left interactive mathematical proof and model checking.

[2] A better description of **WEB** is provided by Sewell [28].

The choice of model checker is critical to achieving the goal of automating as much of the verification process as possible. FDR and the Concurrency Workbench [3] were the leading model checker candidates. While these tools are roughly equivalent in terms of functionality, FDR was chosen for the repeater refinement because of its basis in CSP, rather than CCS, and it's production-quality environment.

Verifying more intuitive properties of CSP specifications requires interactive proof. A number of interactive proof assistants in addition to EVES have been extended to support reasoning about CSP. Previous work at NRL encoded a subset of the CSP Traces Refinement model in the logic of EHDM [21]; work in the UK encoded the Failure-Divergences Model in the logic of HOL [2]. EVES was chosen for a number of reasons. Only EVES and HOL permit user-defined syntax of application theories, which is helpful for reasoning in a user-friendly CSP syntax. Only EVES and EHDM permit verification of actual software programs, which is important for future work involving literate assurance arguments about code. In the end, EVES was chosen because of its support for automating, rather than merely mechanizing, the reasoning process. The advent of other tools, e.g., PVS [23], may require a re-evaluation of this choice for future efforts.

Chapter 3

Notation Overview

The solution that we have proposed to the repeater problem stresses the importance of specifying the critical requirements in an intuitive manner while automating as much of the verification process as possible. Unfortunately, requirements stated to promote human understanding are usually not the easiest to verify automatically. This is exemplified by the low level requirements specification languages of existing model checkers. Our solution requires combining in a coherent manner the three tools we described in the last chapter:

- CSP which provides a concise notation and theory, but, by itself, provides no mechanical support;
- EVES which provides the ability to prove arbitrary properties, but, by itself, only provides a limited potential for automation; and
- FDR which provides complete automation, but, by itself, does not support specifying properties in an intuitively appealing manner.

Addressing the goals of our solution through the combined use of CSP, EVES and FDR comes at a cost. Each of these tools has its own set of notational conventions that often conflict. Our goal in the presentation of the repeater assurance argument is to use a notation that eases the understanding of the argument to the lowest levels of abstraction. Fortunately, EVES permits user-defined syntax for application theories, so that a CSP-like syntax for specifying and reasoning about CSP can be defined. However, EVES restricts user-defined syntax to certain sequences of ASCII character symbols; these restrictions do not permit full conformance with the syntax for CSP in [9] or the ASCII version of CSP on which FDR is based.

We choose a syntax for CSP that is as close as possible to FDR's CSP syntax and that we believe promotes readability of the repeater assurance argument. We present this syntax in Appendix A; the index at the end of the document identifies the page number on which specific operators are described. Appendix B describes variations on the syntax necessary for processing by the mechanical tools. We explicitly choose not to use a pretty (LaTeX) version of the syntax in the narrative descriptions, e.g., one which follows more closely the notation described in [9], because we feel this would tend to confuse rather than clarify the ASCII-restricted SVerdi specifications. We present enough detail about the relatively small subset of CSP used that additional reading on CSP is probably not necessary. For readers already familiar with CSP or readers that desire additional background, Appendix C provides a summary of the correspondence between the syntax of Hoare's CSP and the syntax we have adopted.

The rest of this section uses the notation of Appendix A to describe in more detail the CSP specification paradigm and the infrastructure provided by EVES and FDR to specify and mechanically verify properties about CSP descriptions. We also describe the notational conventions of the literate programming tool that we are using to document and manage the assurance argument. One need not memorize the notation presented at first reading. Only a small subset of the CSP and EVES notation is needed for the next chapter. Part II increasingly relies on a more in-depth understanding of the CSP and EVES notation. We do not use the FDR notation until Part III. An understanding of FunndelWeb notation is required uniformly through the document. We suggest a cursory review of the notation to start with followed by a more in-depth study as the need arises.

While progressing through the assurance argument, we hope the reader keeps in mind Hoare's views on learning a new notation:

> Notations are a frequent complaint. ... If it is any consolation, this should be the least of your worries. After learning the script, you must learn the grammar and the vocabulary, and after that you must master the idiom and style, and after that you must develop fluency in the use of the language to express your own ideas. All this requires study and exercise and time, and cannot be hurried. So it is with mathematics. The symbols may initially appear to be a serious hurdle; but the real problem is to understand the meaning and properties of the symbols and how they may and may not be manipulated, and to gain fluency in using them to express new problems, solutions, and proofs. Finally, you will cultivate an appreciation of mathematical elegance and style. By that time, the symbols will be invisible; you will see straight through them to what they mean.

3.1 The CSP Computational Framework

The CSP language permits describing systems as networks of communicating processes. A CSP process is an entity that communicates with its environment through named communication channels. The Traces Refinement model of CSP characterizes a process according to its alphabet and set of traces. The alphabet of a process *P*, denoted **alpha** *P*, specifies all communication events relevant to characterizing *P*. A communication event is described by a pair *c.m*; the alphabet of process *P* contains *c.m* if and only if *P* is permitted to communicate message *m* over channel *c*. The trace of a process is an observation of its execution. It consists of a finite sequence of events in which the process has engaged at some moment of time. The set of all traces of a process *P* is denoted **traces** *P*.

The CSP notation allows the description of processes using a variety of process constructors. The CSP subset used in this document contains nine primary process constructors, `STOP`, `SKIP`, the prefix constructor "`->`", the choice constructor "`[]`", the conditional constructor "**if then else endif**", the sequential constructor "`;`", the recursion constructor "`=`", the concurrent constructor "`||`", and the compose constructor "`|?|`". `STOP` is the process with alphabet *A* that never engages in any events of *A*; it describes the behavior of a broken process. The only trace of `STOP` is the empty trace. `SKIP` is the process with alphabet *A*, which does nothing but terminate successfully; it describes the behavior of a process that has successfully finished its job. To distinguish `STOP` and `SKIP` the event **tick** is used to signify successful termination. The only traces of `SKIP` is `.< >.` and `.<tick>.`.

The prefix process "*e* `->` *P*" describes a process that first behaves like *e* and then like process *P*. The trace of this process is *e* tacked onto the front of *t* where *t* is the trace of *P*. The choice constructor allows the behavior of a process to be influenced by outside events. If *P* and *Q* are processes and *e* and *f* are events, the process "*e* `->` *P* `[]` *f* `->` *Q*" behaves like process *P* if *e* is the first event to occur and behaves like process *Q* if *f* is the first event to occur. This generalizes to a process "`[]` $x : B$ `@` $P(x)$" where *B* is a non-empty set of events, *x* is an arbitrary event from *B*, and *P* is a function from events to processes. A trace of a choice process must be a trace of one of the alternatives. The conditional process "**if** *b* **then** *P* **else** *Q* **endif**" is defined as the process *P* if *b* is **true**, otherwise *Q*. This can be generalized to allow any number of conditional branches in the natural way. The trace of a conditional process is simply the trace of the process specified by the value of the conditional expressions.

The sequential composition of processes, "P ; Q", describes a process that acts like the successful termination of P followed by Q. If P does not terminate successfully then Q does not start. The trace of this process is a trace of P and, if this trace ends with the successful termination event, that event is replaced by a trace of Q. A process successfully terminates only if the process terminates on a SKIP process. The recursive process "$X = P(X)$" describes the process X that is the solution, i.e., fixed point, of the equation $X = P(X)$.

P [| X |] Q describes a process executing process P concurrently with process Q while synchronizing on events in X. Two processes synchronize on an event if and only if they engage in that event simultaneously. A commonly used abbreviation, P || Q describes a process executing process P concurrently with process Q with synchronization on those events that occur in both **alpha** P and **alpha** Q. Events occurring in **alpha** P but not **alpha** Q may be engaged in by P independently of Q. While the concurrency constructor is an operation on two processes, either of the two processes may itself be a concurrent process. This operator therefore allows the description of arbitrary networks of processes.

Processes executing concurrently communicate through channels. C ! m denotes the output of message m on channel C; C ? x denotes the input of value for x on channel C. These operations are communication events defined by

$$\begin{aligned}(C!m \texttt{->} P) &\equiv (C.m \texttt{->} P)\\ (C?x \texttt{->} P(x)) &\equiv (\square\ C.n{:}\textbf{alpha}\ P(n) \texttt{->} P(n)).\end{aligned}$$

Although the CSP notation distinguishes between the input and output of values over channels, the Traces Refinement model uses only the generic dot notation, $C.m$, to represent communications over channels. For example, the traces of the output process $C!m$ -> P are simply the traces of $C.m$ -> P.

A communication of message m over C can occur between two processes running in parallel if and only if both processes have the communication event $C.m$ in their alphabets and both processes simultaneously engage in that event. That is, whenever one process outputs a value onto the channel, the other process simultaneously inputs the same value from the channel. This implies that

$$(C!m \texttt{->} P) \,||\, (C?x \texttt{->} Q(x)) \equiv C.m \texttt{->} (P \,||\, Q(m))$$

where $C.m$ occurs in **alpha** P and **alpha** $Q(m)$. If only one process in a network of processes has a communication event in its alphabet, then that process may communicate over the channel associated with that event independently of the other processes. To simplify the theory involved, Hoare assumes that at most two processes in a network of processes can access the same communication channel and that communication over a channel occurs in only one direction [9]. If only one process in the network can access the channel, the channel is said to be external; if two processes can access the channel, the channel is said to be internal. The only way a process can communicate with another process executing concurrently is by engaging in a communication event; no shared memory is permitted.

The alphabet and set of traces of a concurrent process are defined in terms of its component processes. The alphabet of a concurrent process is simply the union of the alphabets of its component processes. The set of traces of a concurrent process includes any trace that, when restricted to one of its component alphabets, forms a trace for that component.

The above view of concurrency requires that any trace of a concurrent process P || Q include every event in which P or Q engage. The visibility of the communications over internal channels in the traces of P || Q reduces the amount of abstraction possible during the system design process. Hierarchical design, a proven method for managing the complexity of system design and verification, requires that the requirements of a component be based solely on the sequence of external communications in which it may engage. We would like to be able to hide the internal events and describe requirements of a process's external interface only. The CSP hiding operation "$P \setminus A$" describes a process P with the events in set A hidden. Therefore, P || Q \ (**alpha** P ** **alpha** Q) describes a concurrent process with all internal event hidden. For convenience, we define an abbreviation for this process using the compose operator, denoted |?|. P |?| Q ~ **tick** is equivalent to the P || Q except that the internal communications are hidden. The traces and alphabet of a compose process are the same as for a concurrent process with the internal events deleted.

The Traces Refinement model of CSP permits specifying safety, i.e., partial correctness, requirements of non-divergent processes [22]. A process P is non-divergent if P contains no unguarded recursion (i.e., if every recursive call to P is prefixed by some event) and P cannot engage in an infinite consecutive sequence of hidden events.[1] A requirement in CSP is viewed as a set of traces. Process P satisfies a requirement R, denoted P **sat** R, if and only if R contains every trace that may occur as an observation of P:

$$(P \text{ sat } R) \equiv (\text{traces } P \subseteq R).$$

The above discussion identifies a number of assumptions that the Traces Refinement model of CSP makes about the environment and implementation of a CSP process. In summary, these are

Shared Channel Communication — Communication between concurrent processes (or a process and its environment) can take place only over channels shared by the alphabets of the processes.

Two-Process Communication — Communication over a channel is unidirectional involving exactly two processes - one process acting as a sender and the other process acting as a receiver.

Atomic Communication — Communication over a channel is an atomic event.

Synchronous Communication — Communication over a channel requires synchronous participation of both sender and receiver.

Non-Divergent Processes — Processes are non-divergent, i.e., all recursion is guarded and there is no possibility of a process engaging in an infinite consecutive sequence of hidden events.

The validity of the assurance argument for the repeater developed in this document depends on the validity of these assumptions for the primitives of the repeater logical and physical designs. These assumptions will be continually refined and interpreted throughout the repeater refinement.

3.2 EVES Notation

We use SVerdi to write definitions, conjectures, and proofs. These entities can be organized using the built-in library mechanism. This mechanism provides a foundation from which to encode the CSP Traces Refinement model in EVES, to specify CSP applications and to prove that they satisfy their critical (trace) requirements.

3.2.1 Non-Executable Definitions

SVerdi can be partitioned into the constructs that permit execution in EVES and those that do not permit execution in EVES. SVerdi non-executable definitions are for specification purposes only; they can be stated as functions in the SVerdi Logic. The SVerdi Logic is based on the Predicate Calculus using an ASCII syntax for predicate logic operators (see Appendix A). Constants may be defined as functions of zero arguments. The EVES initial theory defines all the built-in functions including the set-theoretic extensions and operators for the pre-defined types.

User-defined functions in SVerdi have the format

[1] Note that while all terminating processes are non-divergent, not all non-divergent processes terminate.

```
function F (p1,p2,...,pn) =
  measure M(p1,p2,...,pn)
  begin
    FDef(p1,p2,...,pn)
  end F;
```

The `measure` expression in the above template is required only for recursive functions as a basis for proof of termination. The measure `M` must describe an expression associating natural numbers to recursive calls that is bounded below by zero and decreases each recurrence of the function. The body of a function may be "stubbed out" to postpone its definition until a later time. A simple example is integer exponentiation, the stub of which might be

```
function exp (base,exponent);
```

and the fully expanded body of which might be

```
function exp (base,exponent) =
  measure exponent
  begin
    if exponent >= 1
    then base * (exponent - 1)
    else 1
  end exp;
```

The value of `exponent`, the measure for this function, decreases each iteration until it reaches a value less than 1.

SVerdi provides a distinct notation called *zf functions* for constructing sets. To define, for example, the sets

$$foo(x) = \{y \in f(x) \mid P(x,y)\}$$
$$bar(x) = \{g(x,y,z) \mid y,z \in f(x)\}$$

we define in SVerdi

```
zf function foo (x) =
  begin
    { y in f(x) | P(x,y) }
  end foo;

zf function bar (x) =
  begin
    { g(x,y,z) | y,z in f(x) }
  end bar;
```

One other type of zf function allows you to choose an arbitrary value that satisfies some expression; for example, the function

```
zf function foobar (x) =
  begin
    y | P(x,y)
  end foobar;
```

chooses a `y` such that `P(x,y)` holds, provided that one exists. No measures are needed for zf functions since they may not be defined recursively. These and other set theoretic operators, which do not require embedding in zf function definitions, are summarized in Appendix A.4.

3.2.2 Executable Definitions

SVerdi specifications may use the full power of mathematics to describe the requirements to which a system must conform. SVerdi code, on the other hand, must be executable on a physical machine and, therefore, must conform to the constraints of that machine. For example, specifications may use unbounded integers, whereas any code will be constrained to a subset that is representable on the base machine chosen. The executable subset of SVerdi is strongly typed for representation purposes; the subset consists of type declarations, typed function declarations, and procedure declarations. SVerdi types include `bool` (Booleans), `char` (characters), `int` (integers), enumerated types, records and arrays.

Typed functions have the format

```
typed function TF (p11,p12,...,p1i : t1,
                   p21,p22,...,p2j : t2,
                   ...,
                   pm1,pm2,...,pmn : tm,) returns t =
  pre P(p11,...,pmn)
  begin
    TFDef(p11,...,pmn)
  end TF;
```

where `t, t1, t2, ..., tm` are pre-declared types and `p11, ..., pmn` are parameters of the associated types. The `pre` expression in the above template is needed only if the function `TF` is not total over the type space; in this case, `P(p11,...,pmn)` defines the restricted domain of the function.

SVerdi procedure declarations have the format

```
typed function PR (xvar p11,p12,...,p1i : t1,
                   xvar p21,p22,...,p2j : t2,
                   ...,
                   xvar pm1,pm2,...,pmn : tm) =
  initial I(p11,...,pmn)
  pre P1(p11,...,pmn)
  post P2(p11,...,pmn)
  measure M(p11,...,pmn)
  begin
    PRDef(p11,...,pmn)
  end PR;
```

where `t, t1, t2, ..., tm` are pre-declared types and `p11, ..., pmn` are parameters of the associated types. `xvar` is one of `lvar`, `pvar`, or `mvar`. Logical variable, or `lvar`, parameters are value parameters. Program variable, or `pvar`, parameters are variable parameters. Machine variable, or `mvar`, parameters provide restricted access to variables of the base machine, e.g., physical ports. Their exact value depends, ultimately, on linking the program to the specific observables.

The `initial` expression in procedure declarations gives names to the initial values of the variables for reference in later annotations. The `pre` expression describes the condition that is assumed to hold on entry to the procedure. The `post` expression describes the condition that is guaranteed to hold on exit of the procedure, if it can be shown that execution of statements `PRDef(p11,...,pmn)` of the procedure implies that the condition holds. The `measure` expression is required only for recursively defined procedures; `M` must describe an expression associating natural numbers to recursive calls that is bounded below by zero and decreases each recurrence of the procedure.

SVerdi statements include assignments, conditionals, procedure calls, loops and loop exits. Each loop must have the format

```
loop
  invariant Inv(p11,...,pmn)
  measure m(p11,...,pmn)
  LDef(p11,...,pmn)
end loop;
```

where the **invariant** expression describes a condition that holds at that point every iteration; the **measure** expression describes an integer expression that is bounded below by zero and decreases each iteration of the loop; and **LDef** describes the statements executed each iteration.

3.2.3 Conjectures

SVerdi provides several complementary ways to formulate conjectures, i.e., logical predicates, about SVerdi definitions. The simplest form of conjecture is an axiom:

```
axiom A (v1,v2,...,vn) =
  begin
    P(v1,v2,..,vn)
  end A;
```

where P is a predicate stated in terms of the variables v1 through vn. Although this type of conjecture is referred to as an axiom, EVES obligates the developer to prove the predicate submitted. The EVES prover, which is called NEVER, requires that the axiom definition be explicitly assumed (via a **use** command) in order for the predicate defined by the axiom to be used in subsequent proofs.

Three other types of conjecture definitions – rules, grules, and frules – allow the developer to direct NEVER to use the predicates defined automatically when certain conditions are met. Rules are rewrite rules of the form

```
rule R (v1,v2,...,vn) =
  begin
      C(v1,v2,..,vn)
    -> P(v1,v2,..,vn) = E(v1,v2,..,vn)
  end R;
```

where C is a predicate condition, P is a pattern to be matched, and E is an expression to be substituted. If R is in the EVES database and NEVER encounters the pattern P in the proof of a formula, then, if condition C can be proven automatically, expression E is substituted for P in the formula. If C is tautologically **true**, the definition of a rule can be simplified as expected.

A grule is used in the proof of a formula when a subexpression of the formula matches the trigger expression for the grule. Grules have the form

```
grule G (v1,v2,...,vn) =
  begin
      C(v1,v2,..,vn)
    -> P(v1,v2,..,vn)
  end G;
```

where C is a predicate condition, and P is a predicate containing the trigger expression. The trigger is the first full function applied to zero or more distinct free variables when P is scanned from left to right. If G is in the EVES database and NEVER encounters G's trigger in the proof of a formula then, if condition C can be proven, predicate P is assumed. If C is tautologically **true**, the definition of a grule can be simplified as expected.

Finally, a frule is used in the proof of a formula when an instance of its condition becomes **true**. Frules have the form

```
frule F (v1,v2,...,vn) =
  begin
      C(v1,v2,..,vn)
    -> P(v1,v2,..,vn)
  end F;
```

where `C` and `P` are predicates. If `F` is in the EVES database and NEVER detects that condition `C` is satisfied for some instantiation of `v1` through `vn` in the proof of a formula, then predicate `P` is assumed.

3.2.4 Proof Commands

SVerdi provides commands to interact with NEVER to prove the proof obligations generated while parsing definitions and conjectures into the EVES database. These commands can roughly be classified as to whether they provide coarse-grained control or fine-grained control over NEVER.

Commands providing coarse-grained control include **`simplify`**, **`rewrite`**, and **`reduce`**. Simplification uses frules and grules to transform an expression to one that the system considers to be simpler. Rewriting performs simplification and applies any rewrite rules that match the subexpression being traversed. Finally, reducing expands function definitions in addition to simplifying and rewriting the current formula. These coarse-grained commands can use a conjecture or function definition only if it is enabled. By default conjectures and definitions are enabled; they may be disabled by adding the keyword **`disabled`** in front of the conjecture or function definition. The coarse-grained commands can be incrementally strengthened/weakened using the **`with enabled`**/**`with disabled`** modifiers, specifying the definitions and conjectures to be enabled/disabled during the course of the command execution.

Commands that provide fine-grained control include commands that allow definitions and conjectures to be used manually (e.g., **`use`**, **`invoke`**, and **`apply`**) and commands that perform

- quantifier manipulation (e.g., **`instantiate`** and **`prenex`**),
- equality reasoning (e.g., **`equality substitute`**),
- formula rearranging (e.g., **`rearrange`** and **`split`**), and
- case splitting (e.g., **`cases`** and **`next`**).

SVerdi also provides an induction command, **`induct`**, which inducts on a recursive function specified by the user or chosen heuristically based on calls to recursive functions within the current formula.

3.2.5 Library Mechanism

SVerdi descriptions can be organized into library units for configuration and proof management. Library units are either **`spec`** (i.e., specification) units or **`model`** units. Each **`spec`** unit corresponds to a unique **`model`** unit; EVES requires the **`model`** unit to be a model (in the mathematical logic sense) of its corresponding **`spec`** unit. A library can be specified as default by calling the **`set library`** command with the path name of the directory in which the library resides. The current state of the EVES database can be saved as either a **`spec`** or **`model`** unit by calling the **`make`** command. **`model`** units may only be saved if all the proofs are complete and all the definitions are consistent with its corresponding **`spec`** unit. The definition of new units may be started by resetting the state of the EVES database using the **`reset`** command.

One library unit may **`load`** a previously defined **`spec`** unit to access the definitions and conjectures of that unit, if no circularities are introduced as a result. The definitions and conjectures loaded are referenced by prefixing their names with the name of the unit followed by an exclamation mark. For example, a definition **`def`** from the library unit **`lib`** is referred to as **`lib!def`**. The only exception to this naming convention is when the user defines specialized syntax for particular applications, in which case the user definition overrides the naming convention (see [16] for details on how this is accomplished). The special syntax used in the repeater refinement is described in Appendix A. The library distributed with EVES is described in [27]; a brief summary of the portion of the EVES Library that we use, including the names of the units defined, is presented in Section 1 of Chapter 13.

The SVerdi library mechanism supports abstraction, information hiding, modularization and reuse in the form of Ada-like package specifications or axiomatic descriptions of mathematical theories. The **spec** unit can be used to document the specification portion of a package as a CSP process stub and the critical requirements of that process. The **model** unit documents the body portion of the package as an detailed design of the CSP process and the verification that it satisfies the critical requirements defined. More important for the CSP theory development is the use of the library mechanism to formulate axiomatic descriptions of mathematical theories given in the **spec** unit and model theoretic proofs of their consistency given in the **model** unit. The CSP library developed for EVES and documented in [19] uses this approach to set up the background theory to specify and verify the repeater. A brief summary of its contents, including the names of the units defined, is presented in Section 2 of Chapter 13.

3.3 FDR Notation

CSP descriptions in FDR consist of three elements: a low-level process description language, a set of high-level process combination operators, and a supporting mathematical language. FDR's low-level process language allows the definition of relatively small finite state sequential components in terms of relatively complex mathematical objects. FDR's high-level process composition operators allow low-level process components to be combined into complex system models. FDR enforces two restrictions on the combination of these elements:

- no high-level operators may be nested inside low level processes, and
- process descriptions that involve high-level operators may not be parameterized.

These restrictions are made to ease the FDR verification process. Since our approach is to translate a logical CSP design into FDR syntax, we will have to ensure these restrictions are met throughout the design of the repeater. The FDR parser will reject any specification that does not meet these criteria.

Unlike EVES, FDR is tailored specifically to process CSP descriptions. CSP in FDR is strongly typed and has its own built-in language for defining channels, alphabets and processes. Channels and the values that may be communicated are declared as

```
pragma channel c1,c2,...,cn : v
```

where `ci` is the name of a distinct channel that may communicate the values in the set `v`. `v` may be either the name of a previously defined set or the set itself. Sets are user-defined finite enumerations that may include integers or truth values; conventional set notation is used, e.g., `MySet = {0,1,2}`.

Although the values transmitted over channels are limited to simple atomic values, process definitions may be parameterized with more complex structures such as sets and sequences. In particular, the repeater descriptions depend on sequences of bits to represent characters and buffer contents. These sequences are parameters to recursive processes that represent the current state of the repeater.

FDR supports the definition of functions on the CSP complex structures in the Standard ML (Meta-Language) programming language [18]. To use ML functions in a CSP process description, the developer must define the ML function, load the function into FDR, declare the name of the function to be used in the CSP description, and link the function definition with the appropriate name.

3.3.1 Defining and Loading ML Functions

ML functions are defined in a separate file and then loaded into FDR using the ML **use** command, e.g., **use "filename"**. The syntax for the subset of ML that we use is similar to that for EVES' CSP function definitions with extensions to support integration with FDR. ML function definitions have the form

```
fun F (v1,v2,...,vn) =
  Def(v1,v2,...,vn);;
```

where **Def** is an ML expression in terms of **F**'s parameters.

ML functions that pass values to FDR are required to have a rather awkward and unintuitive structure. Such functions must have type **expression list** $\mapsto$ **expression** or, if it is to be used as a predicate, **expression list** $\mapsto$ **bool**. To cast ML/FDR interface functions in this form, FDR provides a set of built-in functions to coerce values to type **expression** and to interrogate values contained in a variable of type **expression**. A sequence of values can be coerced to type **expression** by passing the sequence as the first parameter to the **EXPseqcomp** function:

EXPseqcomp: expression list $\times$ **expression list** $\mapsto$ **expression**

Since a parameter list can be viewed as an expression list, ML functions used to directly interface to FDR have the form

```
fun f [EXPseqcomp (list, [])]
      Def (list);;
```

ML functions called by **Def** use the more general syntax for ML function definitions described above. The first argument of **EXPseqcomp** is the list of parameters being passed; the second argument is always empty. The elaboration of **Def** requires the use of other coercion and interrogation functions, the signatures of which follow:

CheckAtom : expression $\mapsto$ **atomvalue**

Atom : atomvalue $\mapsto$ **expression**

InjectNum : int $\mapsto$ **atomvalue**

NumberOf : atomvalue $\mapsto$ **int**

3.3.2 Declaring and Linking ML Functions

FDR is instructed that a function call in a CSP description is supplied by an ML function definition using the command

```
pragma opaque "ML" fdr-name
```

where **fdr-name** is the name of the function in the CSP description. The function call is linked to the ML definition by the command

```
DefineMLFunction "fdr-name" ml-name
```

or, if the function is a predicate, by the command

```
DefineMLPredicate "fdr-name" ml-name.
```

3.4 FunnelWeb Notation

As we mentioned earlier, a literate programming (LP) tool allows the user to interleave documentation and code in whatever order is most appropriate for presentation. However, the tool must be able to extract and reorder the code "chunks" properly for the compiler. Like other LP tools, FunnelWeb accomplishes this feat with macros and automatic cross-referencing.

Each macro may contain some code as well as references to other macros. During the code extraction phase, FunnelWeb, starting at the "root" macro[2], extracts any code from the macro and expands all references to other macros. The process continues until there are no more references to expand and all of the code is extracted. The macros and their cross-references also appear in the typeset documentation so the reader can see how the code chunks fit together.

We use two types of FunnelWeb macros:

Simple Macro: The simple macro, by far the most common, has the following format. (The typographical conventions are consistent with FunnelWeb's output.)

⟨*macro name*⟩[definition number]≡
The code to be processed appears here
⟨*call to another macro*⟩ [called macro's definition number]
This macro is invoked in definition n.

The letters **M** and **Z** may follow the definition number, indicating that the macro may be called many times or zero times, respectively. The numbers of the definitions that use a macro are listed in the last line. Macros that are invoked in a macro that is not defined in this document are invoked in external files as described by Section 3 of Chapter 13.

Here is an example of a simple macro.

```
⟨procedure example⟩[5]≡
    procedure example is
        ⟨Constants⟩[4]
    begin
        for i in 1..n loop
            ⟨Write out first p powers of i⟩[8]
        end loop;
    end example;
This macro is invoked in definition 7.
```

Additive Macro: This macro is like the simple macro except that its definition may be distributed throughout the document. Each extension of an additive macro has the same name. The only syntactic difference between an additive macro and a simple macro is that ≡ is replaced with + ≡.

Parameterized Macro: As its name implies, the parameterized macro accepts one or more parameters for its definition. While not widely used in this report, this facility is useful in certain cases. Here is a simple example.

⟨*add*⟩[6] (◇2)**M**≡
◇1 + ◇2
This macro is invoked in definitions 3, 4 and 9.

[2]There may be more than one root macro as each root macro names an output file.

This macro takes two parameters and the second is added to the first. The small diamonds (◇) distinguish the quantity of parameters and the parameter numbers from the definition numbers and ordinary code. Since it makes little sense to define a parameterized macro to be invoked only once, the **M** will usually appear after the definition number. An invocation of *add* is illustrated below in the definition of another parameterized macro! The parameter list is enclosed in parentheses and the individual parameters are enclosed in single quotes and delimited by commas.

⟨*double*⟩[3](◇1)**M**≡
    ⟨*add*⟩[6](‘◇1’,‘◇1’)
This macro is invoked in definitions 10 and 11.

Chapter 4

Critical Requirements Specification

This chapter specifies the critical requirements for a CSP process called **Rptr** that represents the repeater. **Rptr**'s critical requirements derive from both the requirements introduced in Section 2.1 and the assumptions of the CSP Traces Refinement model introduced in Section 3.1. We start with the model assumptions since, in some sense, they lay the foundation for the specification of the other critical requirements. Section 4.1 discusses and motivates the definition of critical requirements that derive from the model assumptions. Section 4.2 presents the **Rptr** formal assertions in the literate style. Although the requirements derived from the model assumptions are critical they cannot be expressed in the CSP Traces Refinement model and, thus, are not formalized. Section 8.3 lists the collection of informal and formal **Rptr** critical requirements as a basis for future refinement.

4.1 Critical Requirements that Derive from Model Assumptions

As depicted in Figure 4.1, the channel **inbit** represents **Rptr**'s input port and the channel **outbit** represents **Rptr**'s output port. The **inbit** and **outbit** channels are limited to single-bit transmissions per communication event. In this chapter, we model only that portion of the repeater function necessary to describe its critical requirements. The omitted function, e.g., error processing and signaling over the error port, will be introduced during later refinement.

We derive critical requirements for **Rptr** by interpreting each Traces Refinement model assumption of Section 3.1 in terms of the primitives of the **Rptr**.

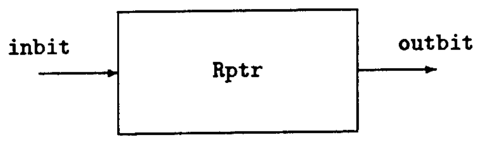


Figure 4.1: The Repeater: CSP External View

4.1.1 Shared Channel Communication

This assumption imposes a restriction on the way a CSP process description may be implemented in software or hardware. The Traces Refinement model assumes that the only way a process can communicate is through channels defined in it's alphabet. The proof theory associated with the model enforces this constraint on the CSP process description, but any refinement of the description to software or hardware is outside scope of the CSP theory and must be shown to satisfy the intent of the constraint independently.

Although this model assumption primarily imposes a requirement on the mapping of a CSP process description to an implementation, it is also helpful to identify assumptions of the environment during the CSP refinement process. The only assumption at this level is that power is continuously supplied to `Rptr`. This simplification is made so that it is not necessary to model a power channel explicitly. Although from a formal perspective the arguments made about `Rptr` are invalidated by a loss of power, practically speaking power loss only violates the guaranteed delivery aspect of the **Relay Characters Until Overflow** requirement.

Assump 1 Power is continuously supplied to `Rptr`.

Assert 1 If `Rptr` is continuously powered, `Rptr` and its environment can communicate only via external channels; communication between `Rptr` sub-processes can take place only over channels shared by the alphabets of the sub-processes.

4.1.2 Two-Process Communication

This assumption requires that the environment does not send data over `outbit` and that `Rptr` does not send data over `inbit`. Decomposing `Rptr` into sub-processes must ensure that communications are uni-directional involving exactly two processes.

Assump 2 The environment does not send data over `outbit`.

Assert 2 `Rptr` does not send data over `inbit`.

Assert 3 Communication between `Rptr` sub-processes must be uni-directional and involve exactly two sub-processes.

4.1.3 Atomic Communication

The assumption that communication events are atomic makes reasoning about the behavior of CSP processes simpler. This assumption appears to be inconsistent with the fact that transmission of data over a physical channel requires some finite, non-zero amount of time to complete. The intent of the assumption, however, is not to force instantaneous communication, but rather to ensure that the existence of arbitrary transmission delay does not affect the truth or falsity of the critical requirements of interest. Since the Traces Refinement model of CSP is limited to specifying and proving safety properties of CSP processes, the time it takes for communications to occur has no impact on the truth of a property specified in the Traces Refinement model. This assumption, therefore, implies no additional requirement for `Rptr`.

4.1.4 Synchronous Communication

This assumption imposes a restriction on the way a CSP process description may be implemented in software or hardware. The Traces Refinement model assumes synchronous communication in its theory for reasoning about CSP processes; any implementation of a CSP process description must ensure the synchrony of processes communicating via the CSP input and output operators. [1] Values sent over channels may not be lost due to mis-timed transmissions.

Assert 4 The implementation of communications over a channel in the `Rptr` process description must synchronize sender and receiver.

4.1.5 Non-Divergent Processes

This assumption imposes a restriction on the form of CSP process descriptions. The restriction ensures that CSP recursive process descriptions have a single solution, i.e., a unique meaning. In general, process descriptions must be fully refined to ensure non-divergence. Nevertheless, the assumption provides important guidance for the refinement process to ensure non-divergence can be proven of the final implementation.

Assert 5 `Rptr` must not engage in unguarded recursion nor engage in an infinite sequence of hidden events.

4.2 Specification of the Formal Assertions

The process representing `Rptr` has three parameters:

- `chsz` represents the length in bits of a character processed. All characters processed by `Rptr` are delimited by a `startbit/stopbit` combination. In terms of the variable `K` introduced in Chapter 2, `chsz` is just `K` reduced by two bits, one bit for each delimiter (i.e., `chsz` equals `K-2`).
- `buffsz` represents the capacity in characters of the internal buffer. From an external viewpoint, we have to interpret the size of the internal buffer referred to in Chapter 2 as the difference between the number of even parity characters received over `inbit` and the number of characters transmitted over `outbit`. `Rptr` may be processing a character in addition to those stored in the internal buffer; `buffsz` represents the actual maximum number of characters that can be stored by `Rptr` in addition to the one being processed, i.e., the value `N-1`. The maximum size of the internal buffer, `N`, referred to in Chapter 2 is therefore mapped to the maximum capacity of `Rptr`. This somewhat awkward interpretation is necessitated by the desire to start from an external specification of the critical requirements.
- `tick` represents the event representing successful termination. We use the variable `tick`, by convention, to represent a special event that occurs automatically when (and only when) a process terminates successfully. `Rptr` terminates successfully after it halts due to receiving a character that causes an overflow; otherwise, `Rptr` continues to process characters that it receives.

$\langle$*Definition Stub of Rptr*$\rangle$[1] $\equiv$
```
    function Rptr(chsz, buffsz, tick);
```
This macro is invoked in definition 219.

`Rptr` is a CSP process if it is constructed from legal CSP process operators. The fact that `Rptr` is actually a process is specified as an axiom at this level of specification since `Rptr` has yet to be defined.

$\langle$*Rptr is a CSP process*$\rangle$[2]M $\equiv$

[1] Note, however, that asynchronous communication can be modeled in CSP, for example, by inserting a buffer process or implementing a handshaking protocol between communicating processes.

```
grule is_process_rep (chsz, buffsz, tick) =
begin
         chsz >= 0
     and buffsz >= 0
     and not iscomm tick
  -> isprocess Rptr(chsz, buffsz, tick)
end is_process_rep;
```
This macro is invoked in definitions 219 and 221.

Assert 6 Rptr is a CSP process.

The alphabet of the repeater consists of communication of bits over the external channels `inbit` and `outbit`. A CSP channel is represented in EVES simply as a function with no parameters, i.e., a constant. Since these values are really constants we define `nilfix` aliases for each to make them easier to use, e.g., `inbit` for `inbit ()`.

⟨*Channels*⟩[3]M ≡
```
typed function in_bit () returns int;
nilfix inbit in_bit;

typed function out_bit () returns int;
nilfix outbit out_bit;
```
This macro is invoked in definition 203.

These are actually just names of channels representing the medium for transmission that will be implemented at some lower level. The values of these constants are irrelevant as long as they are distinct. We make this assumption explicit in the EVES specification by adding an axiom stating that the values returned by these functions are different. This axiom is trivially satisfied by any unique scheme for assigning values to the channels.

⟨*Unique channels*⟩[4]M ≡
```
grule in_not_eq_out () =
begin
  not inbit = outbit
end in_not_eq_out;
```
This macro is invoked in definitions 203 and 205.

The alphabet of `Rptr` can now be defined as `tick` and the set of all communications of bits over the external channels. We use abstract bit values of `0` and `1` to represent the voltages over lines.

⟨*Alphabet*⟩[5]M ≡
```
function Rep_alpha_ext(tick) =
begin
    reqs!seq_buff_alpha(-{inbit, outbit}-, -{0, 1}-, tick)
end Rep_alpha_ext;
```
This macro is invoked in definitions 203 and 205.

That this is the alphabet for `Rptr` is specified as an axiom at this level of specification since the details of `Rptr`'s design have yet to be defined. We assume that `chsz` is a natural number and `tick` is not a communication event. The latter assumption is made so that we do not need to worry about `tick` being indistinguishable from a communication event relevant to `Rptr`'s description.

⟨*Rptr's alphabet is defined by Rep_alpha_ext*⟩[6]M ≡

```
rule Rep_alphabet(chsz, buffsz, tick) =
begin
         chsz >=  0
     and buffsz >= 0
     and not iscomm tick
  -> alpha Rptr(chsz, buffsz, tick)
       = defs!Rep_alpha_ext(tick)
end Rep_alphabet;
```

This macro is invoked in definitions 219 and 221.

Assert 7 Rptr's alphabet is defined by Rep_alpha_ext

4.2.1 Top-Level Requirements Structure

Our goal is to specify formally as trace specifications the critical requirements for Rptr. Assuming valid_relay is the name for these requirements, our specification is

⟨*Rptr satisfies valid_relay*⟩[7]M ≡

```
axiom Rep_sat_spec(chsz, buffsz, tick) =
begin
         chsz >= 0
     and buffsz >= 0
     and not iscomm tick
  -> Rptr(chsz, buffsz, tick)
     sat valid_relay(⟨Rptr universe of events⟩[27], chsz, buffsz, tick)
end Rep_sat_spec;
```

This macro is invoked in definitions 219 and 221.

Assert 8 Rptr satisfies valid_relay.

A trace specification like valid_relay is simply a set of traces. Elements of the set are chosen from the universe of possible traces of events and must conform to valid_relay_spec:

⟨*valid_relay*⟩[8]M ≡

```
zf function valid_relay(a, chsz, buffsz, tick) =
begin
  { tr1 in a^* | valid_relay_spec(tr1, chsz, buffsz, tick) }
end valid_relay;
```

This macro is invoked in definitions 219 and 221.

We split valid_relay_spec into two pieces: one when Rptr has successfully terminated (i.e., tick is the last event of Rptr's trace) and one when it has not. Traces of Rptr that end with tick must satisfy Rptr's post condition; traces of Rptr that do not end with tick must satisfy Rptr's invariant. This forms a natural partition of valid_relay since we can say more about the requirements of Rptr when it has terminated. At termination we can say that all even parity characters received by Rptr before the buffer overflows have been successfully transmitted. Before termination, all we can say is that some subsequence of those characters have been transmitted. Henceforth, the phrase *valid characters* refers only to those characters of even parity received before an overflow.

⟨*Constraints on Rptr's trace*⟩[9]M ≡

```
function valid_relay_spec(tr1, chsz, buffsz, tick) =
begin
  if ⟨Rptr terminates⟩[10]
  then ⟨All valid characters were transmitted over outbit⟩[11]
  else ⟨A subsequence of valid characters were transmitted over outbit⟩[22]
  end if
end valid_relay_spec;
```
This macro is invoked in definitions 219 and 221.

⟨*Rptr terminates*⟩[10] ≡
```
      not null tr1
  and last(tr1) = tick
```
This macro is invoked in definition 9.

4.2.2 Repeater Post Condition

Under the assumption that **Rptr** has terminated, the three logical requirements taken together require that up to the point at which an overflow occurs, the output stream of characters must be identical to the input stream of characters with characters of odd parity removed. Also, nothing other than identifiable and complete characters may be transmitted over **outbit**.

⟨*All valid characters were transmitted over outbit*⟩[11] ≡
```
      ⟨Character sequence transmitted over outbit⟩[12]
  = ⟨Valid character sequence received over inbit⟩[13]
  and ⟨Only whole characters were transmitted over outbit⟩[21]
```
This macro is invoked in definition 9.

Character sequence transmitted over outbit

The sequence of bits traversing **outbit**, i.e., **tr1 |= outbit**, is a flat representation of the characters processed by **Rptr**. The following formats this bit sequence into the character sequence it represents:

⟨*Character sequence transmitted over outbit*⟩[12]M ≡
```
  ⟨Bits to chars⟩[122](‘tr1 |= outbit’)
```
This macro is invoked in definitions 11 and 22.

Valid character sequence received over inbit

valid_input_chars returns the sequence of even parity characters received over **inbit** in trace **tr1** before an overflow occurs. This sequence is calculated by restricting the bits received before an overflow occurs to those characters of even parity.

⟨*Valid character sequence received over inbit*⟩[13]M ≡
```
  valid_input_chars(tr1, chsz, buffsz, tick)
```
This macro is invoked in definitions 11 and 22.

⟨*Definition of valid_input_chars*⟩[14]M ≡
```
  function valid_input_chars(tr1, chsz, buffsz, tick) =
  begin
```

```
        ⟨Characters received over inbit before overflow⟩[15]
    |^ ⟨Set of even parity characters⟩[133]
    end valid_input_chars;
```
This macro is invoked in definitions 219 and 221.

The sequence of characters received over **inbit** before an overflow are derived by transforming to characters the sequence of bits received over **inbit** before an overflow.

⟨*Characters received over inbit before overflow*⟩[15] ≡
```
    ⟨Bits to chars⟩[122](‘⟨Trace before overflow⟩[16] |= inbit’)
```
This macro is invoked in definition 14.

The trace before an overflow occurs is derived by choosing the longest prefix of tr1 for which the predicate **no_error_condition** holds. This operation is performed by the function **filter** of the **tr** library unit. Intuitively, **no_error_condition** describes the condition under which no overflow of the internal buffer has occurred.

⟨*Trace before overflow*⟩[16] ≡
```
    tr!filter(tr1,
              defs!Rep_alpha(chsz,tick),
              no_error_condition(⟨Rptr universe of events⟩[27],
                                 chsz, buffsz))
```
This macro is invoked in definition 15.

Functions to be passed as parameters in SVerdi are represented as a set of ordered pairs where the first elements of the pairs form the domain and the second elements form the range. Since **no_error_condition** is a boolean function, we define it as a set of ordered pairs with domain equal to the set of traces of the alphabet passed in and the range equal to the set of Boolean values. Each trace is mapped to the value returned when passed to the predicate **no_over_flow**.

⟨*Definition of no_error_condition*⟩[17]M ≡
```
    zf function no_error_condition(a, chsz, buffsz) =
    begin
      { -<tr1, no_over_flow(tr1, chsz, buffsz)>- | tr1 in a^* }
    end no_error_condition;
```
This macro is invoked in definitions 219 and 221.

An overflow occurs when the difference between the number of even parity characters received over **inbit** and the number of characters transmitted over **outbit** exceeds **buffsz + 1**, at any point during execution. **buffsz** is incremented by 1 since **Rptr** may be processing a character in addition to the **buffsz** characters possibly held by the internal buffer.

⟨*Definition of no_over_flow*⟩[18]M ≡
```
    function no_over_flow(tr1, chsz, buffsz) =
    begin
      all tr2:
          tr2 .<=. tr1
       ->     (len ⟨Sequence of even parity inbit chars over tr2⟩[19])
            - (len ⟨Sequence of outbit chars over tr2⟩[20])
          <= buffsz + 1
    end no_over_flow;
```
This macro is invoked in definitions 219 and 221.

⟨*Sequence of even parity inbit chars over tr2*⟩[19]M ≡

⟨*Bits to chars*⟩[122](`tr2 |= inbit`)
|^ ⟨*Set of even parity characters*⟩[133]

This macro is invoked in definitions 18 and 33.

⟨*Sequence of outbit chars over tr2*⟩[20]M ≡
⟨*Bits to chars*⟩[122](`tr2 |= outbit`)

This macro is invoked in definition 18.

Only whole characters were transmitted over outbit

Specifying that no spurious bits were transmitted is a simple matter of stating that the sequence of bits that traversed **outbit** is a multiple of the character size, **chsz**, plus 2 for the start and stop bits delimiting each character transmitted.

⟨*Only whole characters were transmitted over outbit*⟩[21] ≡

```
(len tr1 |= outbit) mod (chsz + 2) = 0
```

This macro is invoked in definition 11.

4.2.3 Repeater Invariant

Before **Rptr** terminates, we cannot guarantee that every even parity character received has been transmitted. We can guarantee, however, that the sequence of characters transmitted over **outbit** must be a prefix of the sequence of even parity characters received over **inbit** before the overflow occurred. This is easily specified in terms of the primitives already defined.

⟨*A subsequence of valid characters were transmitted over outbit*⟩[22] ≡
⟨*Character sequence transmitted over outbit*⟩[12]
`.<=.` ⟨*Valid character sequence received over inbit*⟩[13]

This macro is invoked in definition 9.

4.3 Summary of the Critical Requirements

The assumptions and assertions of **Rptr** given the use of the CSP Traces Refinement model and the above trace specification are summarized below.

4.3.1 Assumptions

1 Power is continuously supplied to **Rptr**.

2 The environment does not send data over **outbit**.

4.3.2 Assertions

Informal Assertions

1 If **Rptr** is continuously powered, **Rptr** and its environment can communicate only via external channels; communication between **Rptr** sub-processes can take place only over channels shared by the alphabets of the sub-processes.

2 **Rptr** does not send data over **inbit**.

3 Communication between **Rptr** sub-processes must be uni-directional and involve exactly two sub-processes.

4 The implementation of communications over a channel in the **Rptr** process description must synchronize sender and receiver.

5 **Rptr** must not engage in unguarded recursion nor engage in an infinite sequence of hidden events.

Formal Assertions

6 **Rptr** is a CSP process.

7 **Rptr**'s alphabet is defined by **Rep_alpha_ext**.

8 **Rptr** satisfies **valid_relay**.

Part II

The Repeater Logical Design

Chapter 5

Repeater Logical Architecture

This chapter presents the logical CSP architecture for **Rptr** including the decomposition of **Rptr** critical requirements onto the components of this architecture. The goal of the logical architecture (and subsequent logical component refinement) is to describe the most abstract CSP process description that embodies the critical requirements of **Rptr** elaborated in the previous chapter. This process description will be used as the basis for the FDR refinement and verification described in Part III. By making the semantic gap between the **Rptr** critical requirements and its logical architecture as small as possible, we can minimize the extent to which the **Rptr** verification depends on interactive proof and maximize the extent to which it depends on automatic model checking. As mentioned previously, this approach minimizes the human resources needed to carry out a rigorous verification of intuitively appealing properties of **Rptr**.

Decomposition of the **Rptr** formal assertions onto the components of the logical architecture was carried out using the method described in [21]. We present only the final results of the application of this method; the process by which we reached these results are not particularly important for understanding the refinement and verification of **Rptr**. Section 5.1 presents an informal overview of the **Rptr** logical architecture and the primary responsibilities of its components. Sections 5.2 and 5.3 describe the alphabet and the critical formal assertions of each component. Section 5.4 summarizes the critical requirements of the logical architecture. Finally, Section 5.5 outlines the proof that the combination of the component formal assertions imply the **Rptr** formal assertions.

5.1 Overview of the Logical Architecture

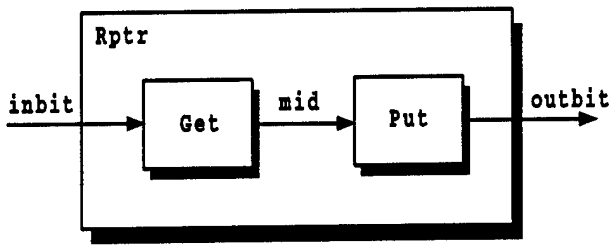


Figure 5.1: Repeater Logical Architecture

As depicted in Figure 5.1, **Rptr** is specified as the composition of two processes: **Get**, which receives and checks the parity of incoming characters, and **Put**, which maintains the internal buffer and transmits those characters that passed the parity check. The internal channel `mid` is used to pass valid characters for output.

Assump 3L The environment does not send or receive data over `mid`.

Assert 2.1L `Get` does not send data over `inbit`.

Assert 3.2L `Put` does not send data over `mid`.

Intuitively, `Rptr` operates as follows. The bit stream received over `inbit` and transmitted over `outbit` can be partitioned according to the delimited character sequence that the bit stream represents. Each segment of the bit stream representing a delimited character contains the `startbit` at the beginning of the segment and the `stopbit` at the end of the segment. For each such segment received over `inbit`, `Get` determines the parity of the character that the segment represents. If the character is of odd parity it is thrown out and the reception of a new character is initiated. Otherwise, the `startbit` and `stopbit` are stripped off and the character is sent in parallel form over `mid` to `Put`. `Put` serializes the data, adds the `startbit` and `stopbit` delimiters and stores the data until ready for transmission over `outbit`. `Rptr`'s components are tightly synchronized; `Get` can be processing a character and the internal buffer can store a maximum of `buffsize` characters. `Get` delays the reception of a new character until the transmission of the previous character to `Put`. At this level of specification, we have not indicated how or when errors are to be signaled over `err`.

Assert 3.1L Communication between `Get` sub-processes must be unidirectional and involve exactly two processes.

Assert 3.3L Communication between `Put` sub-processes must be unidirectional and involve exactly two processes.

A few characteristics of this partitioning of the problem are worth emphasizing. The black box view of the problem, described in Chapter 2, defines an overflow to occur when the number of bits received over `inbit` exceeds the number of bits transmitted over `outbit` by more than `K*N` (i.e., `(chsz + 2) * (buffsz + 1)` in terms of the formal `Rptr` specification). While errors can occur in this refinement due to the reception of characters of odd parity, the tight synchronization of `Rptr`'s components prohibits the occurrence of an overflow. `Rptr` can hold a maximum of `N` delimited characters, one from `Get` and `N-1` (i.e., `buffsz`) from `Put`. Once this maximum is reached, no more characters can be received until a character is transmitted.

5.2 Get Formal Assertions

This section presents the critical formal assertions for `Get`. The alphabet of `Get` consists of communication of bits over the previously defined channel `inbit` and the new channel `mid`. The fact that `Get` is actually a sequential process is specified as an axiom at this level of specification since `Get` has yet to be defined. We assume that `chsz` is a natural number and `tick` is not a communication event. The latter assumption is made so that we do not need to worry about `tick` being indistinguishable from a communication event relevant to `Get`'s description.

⟨*Definition Stub of Get*⟩[23] ≡
```
    function Get(chsz tick);
```

This macro is invoked in definition 207.

⟨*Get is a sequential CSP process*⟩[24]M ≡
```
grule Get_is_seqpr (chsz, tick) =
begin
         not iscomm tick
     and chsz >= 0
  -> pr!is_seqpr (Get (chsz, tick), tick)
end Get_is_seqpr;
```
This macro is invoked in definition 207.

Assert 6.1L `Get` is a sequential CSP process.

As before, a CSP channel is represented as a function with no parameters, i.e., a constant, so we define a `nilfix` alias for `mid` to make it easier to use, e.g., `mid` for `mid ()`. `mid` is actually just a name of a channel representing the medium for transmission that will be implemented at some lower level. Its value is irrelevant as long as it is different than previously defined channels. We make this assumption explicit in the EVES specification by adding a series of axioms stating that the value returned by `mid` is different than that returned by any other channel. These axioms are trivially satisfied by any unique scheme for assigning values to the channels.

⟨*mid channel*⟩[25]M ≡
```
typed function mid () returns int;
nilfix mid mid;
```
This macro is invoked in definition 203.

⟨*Unique mid channel*⟩[26]M ≡
```
grule in_not_eq_mid () =
begin
  not inbit = mid
end in_not_eq_mid;

grule mid_not_eq_out () =
begin
  not mid = outbit
end mid_not_eq_out;
```
This macro is invoked in definitions 203 and 205.

The universe of all events relevant to `Rptr` can now be defined as the externally visible events and the set of all communications of chsz-length characters over the `mid`.

⟨*Rptr universe of events*⟩[27]M ≡
```
defs!Rep_alpha(chsz,tick)
```
This macro is invoked in definitions 7 and 16.

⟨*Definition of Rptr universe of events*⟩[28]M ≡
```
function Rep_alpha (chsz,tick) =
  begin
    Rep_alpha_ext (tick)
    ++ reqs!seq_buff_alpha ({mid}, cseq!char_set(chsz), tick)
  end Rep_alpha;
```

This macro is invoked in definitions 203 and 205.

The alphabet of `Get` can be defined as `tick`, the set of all communications of bits over `inbit`, and the set of all communications of characters over `outbit`.

⟨*Get alphabet*⟩[29]M ≡
```
function Get_alpha (chsz,tick) =
  begin
    reqs!seq_buff_alpha ({inbit}, -{0, 1}-, tick)
    ++ reqs!seq_buff_alpha ({mid}, cseq!char_set(chsz), tick)
  end Get_alpha;
```
This macro is invoked in definition 207.

That this is the alphabet for `Get` is specified as an axiom at this level of specification since the details of `Get`'s design have yet to be defined.

⟨*Get's alphabet is defined by Get_alpha*⟩[30]M ≡
```
rule Get_alphabet(chsz, tick) =
begin
         chsz >=  0
     and not iscomm tick
  -> alpha Get(chsz, tick)
       = Get_alpha(chsz,tick)
end Get_alphabet;
```
This macro is invoked in definition 207.

Assert 7.1L `Get`'s alphabet is defined by `Get_alpha`.

5.2.1 Top-Level Requirements Structure

Our goal is to specify formally as trace specifications the critical requirements for `Get`. Assuming `valid_Get` is the name for these requirements, our specification is

⟨*Get satisfies valid_Get*⟩[31]M ≡
```
grule Get_sat_spec (chsz, tick) =
  begin
           chsz >= 0
       and not iscomm tick
    -> Get (chsz, tick)
       sat valid_Get (defs!Rep_alpha (chsz,tick), chsz, tick)
  end Get_sat_spec;
```
This macro is invoked in definition 207.

Assert 8.1L `Get` satisfies `valid_Get`.

`valid_Get` is simply a set of traces, the elements of which are chosen from the universe of possible traces of events from `Get_alpha (chsz,tick)` and must conform to `Get_not_over_capacity` and `valid_char_Get`:

⟨*valid_Get*⟩[32]M ≡
```
zf function valid_Get (a, chsz, tick) =
    begin
      { tr1 in a ^*
        |     Get_not_over_capacity (tr1, chsz, tick)
          and valid_char_Get (tr1, chsz, tick) }
    end valid_Get;
```

This macro is invoked in definition 207.

We must know the capacity of every component of **Rptr** to determine whether an overflow occurs. **Get** is constrained to process one character at a time. Therefore, at any point in **Get**'s execution, the number of even parity characters received over **inbit** can be at most one more than the number of characters transmitted over **mid**.

⟨*Definition of Get_not_over_capacity*⟩[33]M ≡

```
  function Get_not_over_capacity(tr1, chsz, tick) =
    begin
      all tr2:
              tr2 .<=. tr1
         and tr2 ^*? defs!Rep_alpha(chsz,tick)
        ->      (len ⟨Sequence of even parity inbit chars over tr2⟩[19])
              - (len tr2 |= mid)
           <= 1
    end Get_not_over_capacity;
```

This macro is invoked in definitions 207 and 209.

valid_char_Get can be specified much like the **Rptr**-level critical requirements, except that we need not worry about overflow. We split the specification of **valid_char_Get** into two pieces: one when **Get** has successfully terminated (i.e., **tick** is the last event of **Get**'s trace) and one when it has not. Traces of **Get** that end with **tick** must satisfy **Get**'s post condition; traces of **Get** that do not end with **tick** must satisfy **Get**'s invariant. This forms a natural partition of **valid_Get** since we can say much more about the requirements of **Get** when it has terminated. At termination we can say that all even parity characters received by **Get** have been successfully transmitted over **mid**. Before termination, all we can say is that some subsequence of those characters have been transmitted.

⟨*Definition of valid_char_Get*⟩[34]M ≡

```
  function valid_char_Get (tr1, chsz, tick) =
    begin
      if ⟨Get terminates⟩[35]
      then ⟨All even parity characters were transmitted over mid⟩[36]
      else ⟨A subsequence of even parity characters were transmitted over mid⟩[38]
      end if
    end valid_char_Get;
```

This macro is invoked in definitions 207 and 209.

⟨*Get terminates*⟩[35] ≡

```
          not null tr1
      and last(tr1) = tick
```

This macro is invoked in definition 34.

5.2.2 Get Post Condition

Under the assumption that **Get** has terminated, the critical requirements for **Rptr** taken together require that the output stream of characters must be identical to the input stream of characters with characters of odd parity removed.

⟨*All even parity characters were transmitted over mid*⟩[36] ≡
```
tr1 |= mid
  = ⟨Character sequence received over inbit⟩[37]
    |^ ⟨Set of even parity characters⟩[133]
```
This macro is invoked in definition 34.

The sequence of characters received over **inbit** before an overflow are derived by transforming to characters the sequence of bits received over **inbit** before an overflow.

⟨*Character sequence received over inbit*⟩[37]M ≡
```
⟨Bits to chars⟩[122](‘tr1 |= inbit’)
```
This macro is invoked in definitions 36 and 38.

5.2.3 Get Invariant

Before **Get** terminates, we cannot guarantee that every even parity character received has been transmitted. We can guarantee, however, that the sequence of characters transmitted over **outbit** must be a prefix of the sequence of even parity characters received over **inbit**. This is easily specified in terms of the primitives already defined.

⟨*A subsequence of even parity characters were transmitted over mid*⟩[38] ≡
```
tr1 |= mid
 .<=. (⟨Character sequence received over inbit⟩[37]
        |^ ⟨Set of even parity characters⟩[133])
```
This macro is invoked in definition 34.

5.3 Put Formal Assertions

This section presents the critical formal assertions for **Put** in much the same manner as those specified for **Get**. The alphabet of **Put** consists of communication of bits over the previously defined channels **outbit** and **mid**. The fact that **Put** is actually a sequential process is specified as an axiom at this level of specification since **Put** has yet to be defined. Again, we assume that **chsz** is a natural number and **tick** is not a communication event. We also assume that the buffer can hold at least one character.

⟨*Definition Stub of Put*⟩[39] ≡
```
function Put(chsz tick);
```
This macro is invoked in definition 211.

⟨*Put is a sequential CSP process*⟩[40]M ≡
```
rule Put_is_seqpr (chsz, buffsz, tick) =
  begin
          chsz >= 0
      and buffsz >= 1
      and not iscomm tick
    -> pr!is_seqpr (Put (chsz, buffsz, tick), tick)
        = true
  end Put_is_seqpr;
```
This macro is invoked in definition 211.

Assert 6.2L Put is a sequential CSP process.

The alphabet of Put can be defined as tick, the set of all communications of bits over inbit, and the set of all communications of chsz-length characters over outbit.

⟨*Put alphabet*⟩[41]M ≡
```
function Put_alpha (chsz,tick) =
  begin
    reqs!seq_buff_alpha ({outbit}, -{0, 1}-, tick)
    ++ reqs!seq_buff_alpha ({mid}, cseq!char_set(chsz), tick)
  end Put_alpha;
```
This macro is invoked in definition 211.

Assert 7.2L Put's alphabet is defined by Put_alpha.

⟨*Put's alphabet is defined by Put_alpha*⟩[42]M ≡
```
rule Put_alphabet(chsz, buffsz, tick) =
  begin
           chsz >= 0
       and buffsz >= 1
       and not iscomm tick
    -> alpha Put(chsz, buffsz, tick)
       = Put_alpha(chsz,tick)
  end Put_alphabet;
```
This macro is invoked in definition 211.

5.3.1 Top-Level Requirements Structure

Our goal is to specify formally as trace specifications the critical requirements for Put. Assuming valid_Put is the name for these requirements, our specification is

⟨*Put satisfies valid_Put*⟩[43]M ≡
```
grule Put_sat_spec (chsz, buffsz, tick) =
  begin
           chsz >= 0
       and buffsz >= 1
       and not iscomm tick
    -> Put (chsz, buffsz, tick)
       sat valid_Put (defs!Rep_alpha (chsz,tick), chsz, buffsz, tick)
  end Put_sat_spec;
```
This macro is invoked in definition 211.

Assert 8.2L Put satisfies valid_Put.

valid_Put is simply a set of traces, the elements of which are chosen from the universe of possible traces of events from Put_alpha (chsz,tick) and must conform to Put_not_over_capacity and valid_char_Put:

⟨*valid_Put*⟩[44]M ≡

```
zf function valid_Put (a, chsz, buffsz, tick) =
  begin
    { tr1 in a ^*
      |     Put_not_over_capacity (tr1, chsz, buffsz)
        and valid_char_Put (tr1, chsz, tick) }
  end valid_Put;
```
This macro is invoked in definition 211.

Put is responsible for maintaining the internal buffer which can hold a maximum of **buffsz** characters. Therefore, at any point in Put's execution, the number of characters received over **mid** can be at most **buffsz** more than the number of characters transmitted over **outbit**.

⟨*Definition of Put_not_over_capacity*⟩[45]M ≡
```
function Put_not_over_capacity (tr1, chsz, buffsz,tick) =
  begin
    all tr2:
            tr2 .<=. tr1
        and tr2 ^*? defs!Rep_alpha(chsz,tick)
      -> (len tr2 |= mid) - (len cseq!char_seq (tr2 |= outbit, chsz))
        <= buffsz
  end Put_not_over_capacity;
```
This macro is invoked in definitions 211 and 213.

As we did for **valid_char_Get**, we split the specification of **valid_char_Put** into two pieces: one when **Put** has successfully terminated and one when it has not. Traces of **Put** that end with **tick** must satisfy **Put**'s post condition; traces of **Put** that do not end with **tick** must satisfy **Put**'s invariant. At termination we can say that all characters received by **Put** have been successfully transmitted over **outbit**. Before termination, all we can say is that some subsequence of those characters have been transmitted.

⟨*Definition of valid_char_Put*⟩[46]M ≡
```
function valid_char_Put (tr1, chsz, tick) =
begin
  if ⟨Put terminates⟩[47]
  then ⟨All characters were transmitted over outbit⟩[48]
  else ⟨A subsequence of characters were transmitted over outbit⟩[49]
  end if
end valid_char_Put;
```
This macro is invoked in definitions 211 and 213.

⟨*Put terminates*⟩[47] ≡
```
    not null tr1
and last(tr1) = tick
```
This macro is invoked in definition 46.

5.3.2 Put Post Condition

Under the assumption that **Put** has terminated, the critical requirements for **Rptr** require that the output stream of characters must be identical to the input stream of characters. To ensure that no extraneous bits were transmitted we also specify that the length of the sequence of bits transmitted over **outbit** be divisible by the length of a delimited character.

⟨*All characters were transmitted over outbit*⟩[48] ≡
```
        tr1 |= mid = ⟨Bits to chars⟩[122]('tr1 |= outbit')
    and (len tr1 |= outbit) mod (chsz + 2) = 0
```
This macro is invoked in definition 46.

5.3.3 Put Invariant

Before Put terminates, we cannot guarantee that every character received has been transmitted. We can guarantee, however, that the sequence of characters transmitted over outbit must be a prefix of the sequence of characters received over mid.

⟨*A subsequence of characters were transmitted over outbit*⟩[49] ≡
```
    ⟨Bits to chars⟩[122]('tr1 |= outbit') .<=. tr1 |= mid
```
This macro is invoked in definition 46.

5.4 Summary of the Logical Architecture Critical Requirements

5.4.1 Assumptions

1 Power is continuously supplied to **Rptr**.

2 The environment does not send data over **outbit**.

3L The environment does not send or receive data over **mid**.

5.4.2 Assertions

Informal Assertions

2.1L **Get** does not send data over **inbit**.

3.1L Communication between **Get** sub-processes must be unidirectional and involve exactly two processes.

3.2L **Put** does not send data over **mid**.

3.3L Communication between **Put** sub-processes must be unidirectional and involve exactly two processes.

Formal Assertions

6.1L **Get** is a sequential CSP process.

7.1L **Get**'s alphabet is defined by **Get_alpha**.

8.1L **Get** satisfies **valid_Get**.

6.2L **Put** is a sequential CSP process.

7.2L **Put**'s alphabet is defined by **Put_alpha**.

8.2L **Put** satisfies **valid_Put**.

5.5 Justification of the Decomposition

This section presents an intuitive justification that the **Rptr** logical architecture and the critical requirements imposed on the components of that architecture are sufficient to imply the critical requirements of **Rptr** described in Chapter 4. More specifically we must show that the requirements listed in Section 4.3 follow from the requirements listed in Section 5.4 and the following elaboration of **Rptr**:

⟨*Definition of Rptr*⟩[50]M ≡
```
function Rptr (chsz, buffsz, tick) =
begin
  get!Get (chsz, tick) |?| put!Put (chsz, buffsz, tick) ^ tick
end rptr;
```
This macro is invoked in definition 221.

5.5.1 Assumptions Argument

Assump 1:

No change.

Assump 2:

No change.

5.5.2 Assertions Argument

Assert 1 using Assert 6.1L-6.2L, Assert 7.1L-7.2L:

Since this is a logical design only, no implementation of CSP communication is given. Each process may communicate only over channels identified in its alphabet. □

Assert 2 using Assert 2.1L, Assert 7.2L:

Assert 2.1L ensures that **Get** does not send data over **inbit**. The alphabet of **Put** defined by Assert 7.2L ensures that **Put** cannot access **inbit**. Since **Rptr** is composed solely of **Get** and **Put**, **Rptr** cannot send data over **inbit**. □

Assert 3 using Assert 7.1L-7.2L, Assert 3.1L-3.3L, Assump 3L:

Assert 7.1L and Assert 7.2L ensure that the only communication path between **Get** and **Put** is the channel **mid**. By Assert 3.2L **Put** does not send data over **mid**, so the communication path is uni-directional. The facts that **Rptr** is composed solely of **Get** and **Put** and that the environment cannot access **mid** (by Assump 3L) implies that all communication between the first level decomposition of **Rptr** into sub-processes is uni-directional and involves exactly two sub-processes. Assert 3.1L and 3.3L ensure that any further decomposition conforms to these restrictions as well. □

Assert 4:

Since this is a logical design only, no implementation of CSP communication is given. This assertion, therefore, implies no requirement at this level. □

Assert 5:

This assertion is proven during the FDR verification of the physical architecture. □

Assert 6 using Assert 6.1L-6.2L:

The pr library unit guarantees that a process is a CSP process if its components are processes. Since sequential processes are by definition CSP processes, Assert 6 trivially follows from Assert 6.1L-6.2L. □

Assert 7 using Assert 6.1L-6.2L, Assert 7.1L-7.2L:

The pr library unit guarantees that the alphabet of a compose process includes `tick` and any event that is in one of the component's alphabet and not in the other component's alphabet. Assert 7, thus, trivially follows from Assert 6.1L-6.2L, Assert 7.1L-7.2L. □

Assert 8 using Assert 6.1L-6.2L, Assert 7.1L-7.2L, Assert 8.1L-8.2L:

As mentioned in the introduction of this chapter we used the method described in [21] to decompose the requirements for `Rptr` into requirements on its components. The proof obligations that correspond to a requirements decomposition are also described in [21]. These proof obligations arise through the application of two inference rules that reside in the `reqs` library unit:

```
rule compose_sat_rule (p, q, tick, r) =
  begin
          pr!is_seqpr (p, tick)
      and pr!is_seqpr (q, tick)
      and (p || q) sat r)
      and compose_restriction_condition (p, q, tick, r)
    -> ((p |?| q ^ tick) sat r) = true
  end compose_sat_rule;

rule parallel_sat_rule (p, q, s, r, t, a) =
  begin
          isprocess p
      and isprocess q
      and a = (alpha p) ++ (alpha q)
      and p sat s
      and q sat r
      and concurrent_restriction_condition_conjunct (p, s, a)
      and concurrent_restriction_condition_conjunct (q, r, a)
      and conjunction_condition (s, r, t, a)
    -> ((p || q) sat t) = true
  end parallel_sat_rule;
```

`compose_sat_rule` describes a sufficient set of conditions for proving that a compose process satisfies some requirement given that the corresponding process with visible internal communications satisfies that requirement. The primary condition, the compose restriction condition, requires that the truth of the requirement be independent of the internal communications.

```
function compose_restriction_condition (p1, p2, tick, r) =
  begin
    all tr1:        not (tick -[ (nlast tr1))
                and tr1 in traces ((alpha p1) ++ (alpha p2))
                and tr1 in r
             -> (tr1 |^ (tick adj ((alpha p1) \\ (alpha p2)))) in r
  end compose_restriction_condition;
```

`parallel_sat_rule` describes a sufficient set of conditions for proving that a concurrent process satisfies some requirement given that its component processes satisfy some sub-requirements. The primary conditions are the concurrent restriction condition and the conjunction condition. The concurrent restriction condition requires the truth of each component's sub-requirement be independent of the events not in that component's alphabet.

```
function concurrent_restriction_condition_conjunct (p, s, a) =
  begin
    all tr1:        tr1 in a^*
                and (tr1 |^ (alpha p)) in s
             -> tr1 in s
  end concurrent_restriction_condition_conjunct;
```

The conjunction condition requires that every trace that satisfies both of the component sub-requirements also satisfies the requirement of the concurrent composition.

```
function conjunction_condition (s, r, t, a) =
  begin
    all tr1:        tr1 in a^*
                and tr1 in s
                and tr1 in r
             -> tr1 in t
  end conjunction_condition;
```

Each of the proof obligations of the requirements decomposition process is stated and proven as a distinct SVerdi rewrite rule.

⟨*Rptr Compose Restriction Condition*⟩[51]M ≡

```
    rule compose_restriction_condition_rep (chsz, buffsz, tick) =
      begin
              chsz >= 0
          and buffsz >= 0
          and not iscomm tick
       -> reqs!compose_restriction_condition
              (get!Get (chsz, tick),
               put!Put (chsz, buffsz, tick),
               tick,
            valid_relay (defs!Rep_alpha (chsz,tick), chsz, buffsz, tick))
         = true
   end compose_restriction_condition_rep;
```
This macro is invoked in definition 221.

⟨*Get Concurrent Restriction Condition*⟩[52] ≡
```
 rule concurrent_restriction_condition_conjunct_get (chsz, tick) =
   begin
            chsz >= 0
        and not iscomm tick
     -> reqs!concurrent_restriction_condition_conjunct
           (get!Get (chsz, tick),
            get!valid_get (defs!rep_alpha (chsz,tick), chsz, tick),
            defs!rep_alpha (chsz,tick))
         = true
   end concurrent_restriction_condition_conjunct_get;
```
This macro is invoked in definition 221.

⟨*Put Concurrent Restriction Condition*⟩[53] ≡
```
 rule concurrent_restriction_condition_conjunct_Put (chsz, buffsz, tick) =
   begin
            chsz >= 0
        and buffsz >= 0
        and not iscomm tick
     -> reqs!concurrent_restriction_condition_conjunct
           (put!Put (chsz, buffsz, tick),
            put!valid_Put (defs!rep_alpha (chsz,tick), chsz, buffsz, tick),
            defs!rep_alpha (chsz,tick))
         = true
   end concurrent_restriction_condition_conjunct_Put;
```
This macro is invoked in definition 221.

⟨*Rptr Conjunction Condition*⟩[54] ≡
```
 rule conjunction_condition_rep (chsz, buffsz, tick) =
   begin
            chsz >= 0
        and buffsz >= 0
        and not iscomm tick
     -> reqs!conjunction_condition
           (get!valid_get (defs!rep_alpha (chsz,tick), chsz, tick),
            put!valid_Put (defs!rep_alpha (chsz,tick), chsz, buffsz, tick),
            valid_relay (defs!rep_alpha (chsz,tick), chsz, buffsz, tick),
            defs!rep_alpha (chsz,tick)) = true
   end conjunction_condition_rep;
```
This macro is invoked in definition 221.

The proofs of these rules are given in the **rep** model library unit. □

Chapter 6

Get Component Refinement

We considered two approaches to define and verify sequential CSP processes in SVerdi: a functional approach and a procedural approach. The functional approach uses the theory specified in the CSP library, documented in [19], to construct sequential processes using EVES function declarations. Specification and verification proceeds in a manner similar to that documented for concurrent processes in the last chapter. The procedural approach models sequential CSP processes as SVerdi procedures. The critical (trace) requirements are specified much as they are in the functional approach, but they appear in the post condition of the SVerdi procedure that represents the CSP process. EVES's proof obligation generator constructs the conditions under which the process conforms to its requirements.

The procedural approach requires modeling an interface to a base machine on which the CSP described system executes. This interface, described in the `mach` library unit in Chapter 12, provides a set of communication routines very similar to the input and output primitives provided by CSP. An SVerdi procedure that communicates using these routines builds up a trace of its execution recorded in a machine variable parameter that represents the internal state. This trace forms the basis for the specification of the procedure's critical requirements.

The functional approach is more rigorous than the procedural approach since it does not rely on an informal interpretation of CSP processes as SVerdi procedures. Nevertheless, we adopt the procedural approach in the refinement of the `Rptr` sequential processes. The procedural approach reduces the complexity of specifying and verifying sequential processes by exploiting EVES's approach to decompose the requirements of SVerdi procedures and automatically generate the proof obligations required. We believe that this reduced complexity results in a more intelligible assurance argument than possible using the functional approach.

This chapter begins in Section 6.1 by describing an overview of the design of `Get` in CSP. Section 6.2 recasts this design in SVerdi by modeling CSP processes as SVerdi procedures and CSP operations as SVerdi statements. Critical requirements Assert 7.1L and Assert 8.1L are cast as a post condition on the procedure representing `Get`. Finally, Section 6.3 decomposes these requirements onto the primary procedures of the design and argues that the design satisfies the set of critical requirements of `Get`.

6.1 Overview of the Get Design

`Get` is a recursive non-terminating CSP process. Once `Get` receives the initiating `startbit` over channel `inbit`, processing of the next character may begin. One bit arrives over `inbit` with each iteration of **Get** until the whole character has been received, i.e., chsz + 2 bits including the delimiters. If the character received is of even parity and the final bit received is **stopbit**, the character is transmitted over **mid**; otherwise, **Get** ignores the character and initiates reception of the next character. The design of **Get** in CSP follows. **Inchar** is responsible for receiving the character once the **startbit** has arrived. A character has odd parity if the exclusive or of the bits that constitute the character holds true, where a **1** bit value represents true and a **0** bit value represents false.

```
Get(chsz, tick) =
   inbit ? b -> if b = startbit
                then Inchar(.<startbit>.,chsz,tick);Get(chsz,tick)
                else Get(chsz,tick) end if

Inchar(ch,chsz,tick) =
    if (len ch) <= chsz then
    then inbit ? b -> Inchar(ch ^ .<b>.,chsz,tick)
    else inbit ? b -> (if     not odd_parity (tail ch)
                          and b = stopbit
                         then mid ! tail ch -> SKIP
                         else SKIP end if)
    end if

function odd_parity (ch) =
    measure len ch
  begin
    if null ch
    then false
    else xor (head ch = 1, odd_parity (tail ch)) end if
  end odd_parity;

function xor (x, y) =
  begin
        (x or y)
    and not (x and y)
  end xor;
```

6.2 Formal Specification of the Get Design

The machine variable **st** representing internal state provides a means to transmit of values over channels and, for specification purposes, to store a record of the sequence of values transmitted. The current trace of **Get** during execution is evaluated from **st** and must be restricted to **Get**'s alphabet. The process trace is the basis for the specification of **Get**'s two primary requirements: **Get_not_over_capacity** and **valid_char_Get**. Showing that these requirements hold for every prefix of the final trace ensures that they hold invariantly during **Get**'s execution. Although the CSP process representing **Get** does not terminate, EVES requires termination. We, therefore, force the SVerdi representation to terminate at an arbitrary point, i.e., after the variable **cntdwn** reaches zero. **cntdwn** starts at the value of the maximum integer, **int'last()**. Later refinement will reveal how and when **cntdwn** decreases.

⟨*Get design*⟩[55] ≡

```
    procedure Get (mvar st : ⟨State type⟩[159],
                   lvar chsz : int,
                   lvar tick : ⟨Event type⟩[161]) =
      pre     chsz >= 0
          and not iscomm tick
          and ⟨Trace⟩[163](‘st’) = .<>.
      post     ⟨Trace⟩[163](‘st’) ^*? Get_alpha(chsz,tick)
           and all tr2:
                   (     tr2 .<=. ⟨Trace⟩[163](‘st’)
                     and tr2 ^*? Get_alpha(chsz,tick))
                -> (     Get_not_over_capacity (tr2, chsz, tick)
                     and valid_char_Get (tr2, chsz, tick))
      begin
        pvar cntdwn : int := int'last()
        Get_step(st,chsz,tick,cntdwn)
      end Get;
```

This macro is invoked in definition 209.

Put's buffer buff is represented as a first-in first-out queue of bits. An iteration of Put_step depends on the current state of buff. If buff is empty the only option is to receive the next character over mid. If the current length of buff will not accommodate storing another character the only option is to send the next bit stored in buff. Finally, if buff is not empty and there is room for storing another character, either of the above options may occur.

Get_step receives at most one character, stored in the variable chr. Character variables can be thought of as a trace of bits. Once Inchar has received the character, Outchar determines whether the character is appropriate for transmission over mid and, if so, transmits it. Note that an input or output operation is successful if and only if cntdwn remains greater than zero.

⟨*Get_step design*⟩[56]M ≡

```
    procedure Get_step (mvar st : ⟨State type⟩[159],
                        lvar chsz : int,
                        lvar tick : ⟨Event type⟩[161],
                        pvar cntdwn : int) =
      ⟨Get_step specification⟩[63]
      begin
        pvar b : ⟨Bit type⟩[139]
        pvar chr : ⟨Buffer type⟩[137] := ⟨Trace of single startbit⟩[147]
        ⟨inbit ? b⟩[170]
        if     cntdwn > 0
           and ⟨b is startbit⟩[148] then
             InOutChar(st,chr,chsz,tick,cntdwn)
             if cntdwn > 0 then Get_step(st,chsz,tick,cntdwn) end if
        end if
      end Get_step;
```

This macro is invoked in definition 209.

⟨*InOutchar design*⟩[57]M ≡

```
    procedure InOutchar(mvar st : ⟨State type⟩[159],
                        pvar chr : ⟨Buffer type⟩[137],
                        lvar chsz : int,
                        lvar tick : ⟨Event type⟩[161],
                    pvar cntdwn : int) =
    ⟨InOutchar specification⟩[68]
    begin
      Inchar(st,chr,chsz,tick,cntdwn)
      if cntdwn > 0 then
        Outchar(st,chr,chsz,tick,cntdwn)
      end if
    end InOutChar;
```
This macro is invoked in definition 209.

6.2.1 Inchar design

Inchar is a recursive process that iteratively accepts another bit over **inbit** and appends it to the end of **chr**. The recursion bottoms out when the length of **chr** becomes greater than **chsz**, i.e., the length of the character plus 1 for the **startbit**. **cntdwn** must decrease each iteration of **Inchar** since it will be used as the measure to prove the termination of **Get_step**.

⟨*Inchar design*⟩[58]M ≡
```
  procedure Inchar(mvar st : ⟨State type⟩[159],
                   pvar chr : ⟨Buffer type⟩[137],
                   lvar chsz : int,
                   lvar tick : ⟨Event type⟩[161],
                   pvar cntdwn : int) =
    ⟨Inchar specification⟩[72]
    begin
      pvar b : ⟨Bit type⟩[139]
      if ⟨len of chr⟩[157] <= chsz then
             ⟨inbit ? b⟩[170]
             if cntdwn > 0 then
                chr := ⟨Append b onto end of chr⟩[154]
                Inchar(st,chr,chsz,tick,cntdwn)
             end if
      else cntdwn := cntdwn - 1
      end if
    end Inchar;
```
This macro is invoked in definition 209.

EVES requires that executable recursive definitions be specified in terms of procedures. The function **odd_parity** specified previously in the overview of the **Get** design is defined as an EVES procedure **odd_parity_check**. The proof that this procedure terminates with **is_odd** equal to the value returned by **odd_parity** is trivial.

⟨*Definition of odd_parity_check*⟩[59]M ≡
```
  procedure odd_parity_check (lvar chr : bfr!buffer,
                              pvar is_odd : bool) =
    pre true
    post is_odd = odd_parity(bfr!contents(chr))
    measure lenp chr
    begin
      if nullp chr
      then is_odd := false
```

```
        else odd_parity_check(tailp chr,is_odd)
             is_odd:=xor(bfr!cnv_bit(headp chr)=1,is_odd)
        end if
      end odd_parity_check;
```
This macro is invoked in definition 209.

⟨*Get's definition of xor*⟩[60]M ≡
```
    typed function xor (a, b : bool) returns bool =
      begin
            (a or b)
        and not (a and b)
      end xor;
```
This macro is invoked in definition 209.

⟨*Definition of odd_parity*⟩[61]M ≡
```
    function odd_parity (chr) =
        measure len chr
      begin
        if null chr
        then false
        else xor (head chr = 1, odd_parity (tail chr)) end if
      end odd_parity;
```
This macro is invoked in definition 209.

6.2.2 Outchar design

Outchar receives the final delimiter and checks the parity of the character. If the final delimiter is **stopbit** and the character is of even parity, the non-delimited character is transmitted over **mid**.

⟨*Outchar design*⟩[62]M ≡
```
    procedure Outchar (mvar st : ⟨State type⟩[159],
                      lvar chr : ⟨Buffer type⟩[137],
                      lvar chsz : int,
                      lvar tick : ⟨Event type⟩[161],
                      pvar cntdwn : int) =
      ⟨Outchar specification⟩[76]
      begin
        pvar b : ⟨Bit type⟩[139]
        pvar is_odd : bool
        ⟨inbit ? b⟩[170]
         odd_parity_check(⟨tail of chr⟩[153],is_odd)
         if     cntdwn > 0
            and not is_odd
            and ⟨b is stopbit⟩[149] then
              ⟨mid ! tail of chr⟩[179]
          end if
      end Outchar;
```
This macro is invoked in definition 209.

6.3 Justification of the Get Design

This section presents an intuitive justification that the design of **Get** satisfies the assertions of **Get** described in Chapter 5. More specifically, we must show that the requirements listed in Section 5.4 follow from the design of **Get** described in the last section.

6.3.1 Assertions Argument

Assert 2.1L:

By inspection of the **Get** design, the only way **Get** can access external channels is through the **mach** library unit interface. The only calls **Get** makes are to send a character over **mid** using **snd_chr** and to receive a bit over **inbit** using **rcv_bit**. Of course, any refinement of these calls must be shown not to send any information over **inbit**. However, since we are using this model for design verification only, this level of analysis suffices. □

Assert 3.1L:

Get is not decomposed into concurrent sub-processes so no internal communication channels exist. □

Assert 6.1L using Assert 7.1L, Assert 8.1L:

We must argue that the procedural model of the Put CSP process has certain critical properties required of sequential processes. This argument is necessarily informal since the correspondence between our procedural model and CSP is informal. First we show that **Get** is a CSP process and then we show that it is sequential.

From the definition of **isprocess** in the **pr** model library unit, we must show that

1. the empty trace is a trace of **Put**,
2. the traces of **Get** are restricted to events in **Get_alpha(chsz,tick)**, and
3. any prefix of a trace of **Get** is also a trace of **Get**.

We define the traces of the procedural design of **Get** so that these properties hold. By inspection of this design, the trace of **Get** is constructed as it executes by appending to its end the communication event associated with every communication in which it engages. Viewing **Get**'s execution symbolically, we define the set of traces of **Get** as the set of all possible traces constructed up to any point in its execution. Clearly this ensures that (3) holds. (1) holds since the pre-condition for **Get** ensures that the trace is initially empty. (2) holds as a consequence of Assert 7.1L and the portion of the post-condition for **Get** that requires all traces generated to be confined to events in **Get_alpha(chsz,tick)**. We delay proof of the post-condition until the argument for Assert 8.1L.

From the definition of **is_seqpr** from the **pr** model library unit, a sequential process must have the successful termination event in its alphabet. The only place this event may appear in a trace of a sequential process is as the last event. The parameter **tick** represents the successful termination event for **Get**. **tick** is in the alphabet of **Get** as shown in Assert 7.1L. **tick** does not occur in any trace of **Get** since **Get** does not terminate successfully.[1] □

[1] The SVerdi procedural model of **Get** does complete execution, but only because EVES requires proof of termination. This may be considered unsuccessful termination in the CSP sense.

Assert 7.1L:

We define the alphabet of the procedural model for `Get` as the set of events in which it may engage. The post-condition for `Get` requires that this set be restricted to `Get_alpha(chsz,tick)`. We delay proof of the post-condition until the argument for Assert 8.1L. □

Assert 8.1L:

Assert 8.1L requires that all traces of `Get` be members of the set defined by `valid_Get`. This requires that every trace of `Get` satisfy `Get_not_over_capacity` and `valid_char_Get`. The post-condition for `Get` requires these properties of the `Get` design. The rest of this section presents an overview of the proof of the post-condition.

Proof Structure of Get The first task is to derive a set of valid requirements for `Get_step` that allows us to prove that `Get` satisfies its post-condition. The recursive nature of `Get_step` suggests that its specification is invariantly true before and after execution. We call this invariant `valid_Get_step`.

⟨*Get_step specification*⟩[63]M ≡

```
pre      cntdwn > 0
    and cntdwn <= int'last()
    and (len cseq!char_seq (⟨Trace⟩[163]('st') |= inbit, chsz)
                 |^ cseq!even_parity_chars (chsz))
          <= (len ⟨Trace⟩[163]('st') |= mid)
    and valid_Get_step(⟨Trace⟩[163]('st'),chsz,tick,cntdwn)
post valid_Get_step(⟨Trace⟩[163]('st'),chsz,tick,cntdwn)
measure cntdwn
```

This macro is invoked in definition 56.

`valid_Get_step` preserves the truth of both `Get_not_over_capacity` and `valid_char_Get`. In addition, it requires that only characters be permitted over `mid` and that no partial characters be accepted over `inbit`. Recall that this requirement must hold before and after each iteration of `Get_step`; clearly, during execution partial characters can be accepted.

⟨*Definition of valid_Get_step*⟩[64]M ≡

```
function valid_Get_step (tr1,chsz,tick,cntdwn) =
  begin
          chsz >= 0
      and chsz + 2 <= int'last()
      and not iscomm tick
      and tr1 ^*? Get_alpha(chsz,tick)
      and not (tick -[ tr1)
      and ⟨mid allows only chars⟩[121]
      and (cntdwn > 0 -> ⟨Partial char over inbit⟩[124]('tr1') = .<>.)
      and Get_not_over_capacity (tr1, chsz, tick)
      and all tr2:    (     tr2 .<=. tr1
                        and tr2 ^*? Get_alpha(chsz,tick))
                   -> valid_char_Get (tr2, chsz, tick)
  end valid_Get_step;
```

This macro is invoked in definition 209.

Showing that the specification for **Get** follows from the specification for **Get_step** requires showing that

1. the pre-condition and initial assignments for **Get** imply the pre-condition for **Get_step**, and
2. the post-condition for **Get_step** implies the post-condition for **Get**.

The first proof obligation follows trivially from the definitions involved:

⟨*Get lemma #1*⟩[65]M ≡
```
rule valid_Get_step_empty_empty (chsz,tick,cntdwn) =
  begin
           chsz >= 0
       and not iscomm tick
    -> valid_Get_step(.<>.,chsz,tick,cntdwn) = true
  end valid_Get_step_empty_empty;
```
This macro is invoked in definition 209.

The second proof obligation is split according to the structure of the post-condition for **Get**. Note that the requirement that the trace be restricted to **Get_alpha(chsz,tick)** follows trivially from the definition of **valid_Get_step**.

⟨*Get lemma #2*⟩[66]M ≡
```
rule Get_step_Get_not_over_capacity (tr1,tr2,chsz,tick,cntdwn) =
  begin
           tr2 .<=. tr1
       and tr2 ^*? Get_alpha(chsz,tick)
       and valid_Get_step (tr1,chsz,tick,cntdwn)
    -> Get_not_over_capacity (tr2,chsz,tick) = true
  end Get_step_Get_not_over_capacity;
```
This macro is invoked in definition 209.

⟨*Get lemma #3*⟩[67]M ≡
```
rule Get_step_valid_char_Get (tr1,tr2,chsz,tick,cntdwn) =
  begin
           tr2 .<=. tr1
       and tr2 ^*? Get_alpha(chsz,tick)
       and valid_Get_step (tr1,chsz,tick,cntdwn)
    -> valid_char_Get (tr2,chsz,tick) = true
  end Get_step_valid_char_Get;
```
This macro is invoked in definition 209.

Proof Structure of Get_step This level requires us to derive a set of requirements for **InOutchar** that allows us to prove that **Get_step** satisfies its post-condition. The post condition requires that **Get_step**'s post condition holds. The pre condition is similar to **Get_step**'s precondition except that the **startbit** has already been received. Furthermore, to prove that **Get_step** terminates, **InOutchar** must ensure that **cntdwn** decreases to a value no less than zero.

⟨*InOutchar specification*⟩[68]M ≡

```
initial cntdwn_0=cntdwn
pre     cntdwn > 0
    and cntdwn <= int'last()
    and InOutchar_pre(bfr!contents(chr),mach!hist(st),chsz,tick)
post    cntdwn < cntdwn_0
    and cntdwn >= 0
    and (cntdwn > 0
           -> ((len cseq!char_seq (mach!hist(st) |= inbit, chsz)
                |^ cseq!even_parity_chars (chsz))
               <= (len mach!hist(st) |= mid)))
    and valid_Get_step(mach!hist(st),chsz,tick,cntdwn)
measure cntdwn
```

This macro is invoked in definition 57.

⟨*Definition of InOutchar_pre*⟩[69]M ≡

```
function InOutchar_pre(cchr,tr1,chsz,tick) =
  begin
        chsz >= 0
    and not iscomm tick
    and chsz + 2 <= int'last()
    and tr1 ^*? Get_alpha(chsz,tick)
    and ⟨mid allows only chars⟩[121]
    and not (tick -[ tr1)
    and cchr = .<startbit>.
    and cseq!current_char(tr1 |= inbit,chsz) = cchr
    and (len cseq!char_seq (tr1 |= inbit, chsz)
              |^ cseq!even_parity_chars (chsz)) <= (len tr1 |= mid)
    and Get_not_over_capacity (tr1, chsz, tick)
    and all tr2:    (    tr2 .<=. tr1
                     and tr2 ^*? Get_alpha(chsz, tick))
                 -> valid_char_Get (tr2, chsz, tick)
  end InOutchar_pre;
```

This macro is invoked in definition 209.

Four cases are identified in the proof of **Get_step**'s post-condition:

1. while trying to accept the initial bit, **cntdwn** becomes zero, in which case **Put_step** terminates with no change to the buffer or trace;
2. the initial bit is received but is not **startbit**, in which case **Get_step** recurs with the bit appended to the end of the trace;
3. the initial **startbit** is received and **InOutchar** executes unsuccessfully with **cntdwn** equal to zero, in which case **Get_step** terminates;
4. the initial **startbit** is received and is **InOutchar** executes successfully, in which case **Get_step** recurs.

Each of these cases generates a proof obligation in addition to that generated for proof of termination. That these are all of the cases can be seen by inspection of the post-condition of the procedures involved. We deal with each case in turn. Termination follows directly from the fact that **cntdwn** decreases each iteration and is bounded below by zero.

The first case (1) is trivial since no change to the buffer or trace requires only that `valid_Get_step` hold initially, which is guaranteed by `Get_step`'s pre-condition.

The second case (2) requires us to prove the following theorem, which is trivial from the definitions involved since initial delimiters that are not `startbit` are ignored by `cseq!char_seq`.

⟨*Get_step lemma #1*⟩[70]M ≡
```
rule valid_Get_step_tack_inbit_skip (b,tr1,chsz,tick,cntdwn_0,cntdwn) =
  begin
          cntdwn_0 > 0
      and cntdwn > 0
      and not (b = startbit)
      and b in -{0, 1}-
      and valid_Get_step(tr1,chsz,tick,cntdwn_0)
    -> valid_Get_step(tr1 ^ .<inbit.b>.,chsz,tick,cntdwn) = true
  end valid_Get_step_tack_inbit_skip;
```
This macro is invoked in definition 209.

The third and fourth cases (3,4) requires us to prove that `InOutchar`'s pre condition is met and that its post condition implies `Get_step`'s post condition. The latter of these is trivial; the former is stated as the following theorem. This theorem's proof follows by noticing that `InOutchar_pre` is a simple restatement of `valid_Get_step` with the contents of `chr` as `.<startbit>.`.

⟨*Get_step lemma #2*⟩[71]M ≡
```
rule valid_Get_step_tack_startbit_pre (bs,tr1,chsz,tick,cntdwn) =
  begin
          cntdwn > 0
      and bs = .<startbit>.
      and (len cseq!char_seq (tr1 |= inbit, chsz)
              |^ cseq!even_parity_chars (chsz)) <= (len tr1 |= mid)
      and valid_Get_step(tr1,chsz,tick,cntdwn)
    -> InOutchar_pre(bs,tr1 ^ .<inbit.startbit>.,chsz,tick) = true
  end valid_Get_step_tack_startbit_pre;
```
This macro is invoked in definition 209.

Specification of InOutChar This level requires us to derive a set of requirements for `Inchar` and `Outchar` that allows us to prove that `InOutchar` satisfies its post-condition. Just as for `Get_step`, the recursive nature of `Inchar` suggests that its specification, called `valid_Inchar`, must be invariantly true before and after execution.

⟨*Inchar specification*⟩[72]M ≡
```
initial cntdwn_0=cntdwn,st_0=st,chr_0=chr
pre     cntdwn > 0
    and cntdwn <= int'last()
    and valid_Inchar (⟨Contents⟩[143]('chr'),⟨Contents⟩[143]('chr'),
                      ⟨Trace⟩[163]('st'),⟨Trace⟩[163]('st'), chsz, tick)
post     cntdwn < cntdwn_0
     and cntdwn >= 0
     and (cntdwn > 0 -> (len ⟨Contents⟩[143]('chr')) = chsz+1)
     and valid_Inchar (⟨Contents⟩[143]('chr_0'),⟨Contents⟩[143]('chr'),
                       ⟨Trace⟩[163]('st_0'),⟨Trace⟩[163]('st'), chsz, tick)
measure cntdwn
```

This macro is invoked in definition 58.

Discovering the invariant for **Inchar** requires understanding the crucial role that the partially received character **chr** plays. **cchr** represents the final contents of **chr**. **tr1** represents the final trace after execution of **Inchar**. **cchr_0** and **tr1_0** represent the initial contents of **chr** and the initial trace, respectively. **cchr** invariantly contains the partial character (including the **startbit**) that was received over **inbit**.

⟨*Definition of valid_Inchar*⟩[73]M ≡
```
    function valid_Inchar (cchr_0,cchr,tr1_0,tr1,chsz,tick) =
       begin
             chsz >= 0
         and not iscomm tick
         and chsz + 2 <= int'last()
         and tr1 ^*? Get_alpha(chsz,tick)
         and not (tick -[ tr1)
         and ⟨mid allows only chars⟩[121]
         and cchr_0 ^*? -{0,1}-
         and not null cchr_0
         and ⟨Partial char over inbit⟩[124]('tr1_0') = cchr_0
         and cchr_0 .<=. cchr
         and tr1_0 .<=. tr1
         and not null cchr
         and cchr ^*? -{0,1}-
         and ⟨Partial char over inbit⟩[124]('tr1') = cchr
         and ⟨Inchar prefix invariant⟩[74]
       end valid_Inchar;
```
This macro is invoked in definition 209.

The specification for Inchar must also describe the required properties of every prefix of the final trace. Let **pcchr** and **ptr1** represent some intermediate value of **cchr** and **tr1**, respectively. **Valid_Inchar** must ensure that communications that occur during execution of **Inchar**, i.e., those in the trace **tr!remove_first_n(ptr1,len tr1_0)**, contain only communications over inbit. The number of these communications must be bounded above by **chsz + 1 - (len cchr_0)**.

⟨*Inchar prefix invariant*⟩[74]M ≡
```
    all ptr1: some pcchr:
       invariant_over_char(cchr_0,pcchr,tr1_0,ptr1,tr1,chsz,tick)
```
This macro is invoked in definition 73.

⟨*Definition of invariant_over_char*⟩[75]M ≡
```
    function invariant_over_char (cchr_0,pcchr,tr1_0,ptr1,tr1,chsz,tick) =
       begin
              (    ptr1 ^*? defs!Rep_alpha(chsz,tick)
               and ptr1 .<=. tr1
               and tr1_0 .<=. ptr1)
           -> (    not null pcchr
               and pcchr ^*? -{0,1}-
               and ⟨Partial char over inbit⟩[124]('ptr1') = pcchr
               and (tr1_0 |= mid) = (ptr1 |= mid)
               and tr!remove_first_n(ptr1,len tr1_0)
                         ^*? reqs!buffer_alpha({inbit},-{0, 1}-)
```

```
                  and (len tr!remove_first_n(ptr1,len tr1_0))
                            <= chsz + 1 - (len cchr_0))
        end invariant_over_char;
```
This macro is invoked in definition 209.

The specification of **Outchar** is much simpler than that of **Inchar** since it is not recursively defined. In the case that the character to be transmitted has even parity and the correct final delimiter is received, the specification requires that the trace be appended only by the **stopbit** communication over **inbit** and the character transmission over **mid**. Otherwise, at most one bit is received over **inbit**.

⟨*Outchar specification*⟩[76]M ≡
```
    initial cntdwn_0=cntdwn,st_0=st
    pre     cntdwn > 0
        and Outchar_pre(⟨Contents⟩[143](‘chr’),⟨Trace⟩[163](‘st’),chsz,tick)
    post    cntdwn < cntdwn_0
         and cntdwn >= 0
         and valid_Outchar (⟨Contents⟩[143](‘chr’),⟨Trace⟩[163](‘st_0’),
                            ⟨Trace⟩[163](‘st’),chsz,tick,cntdwn)
    measure cntdwn
```
This macro is invoked in definition 62.

⟨*Definition of Outchar_pre*⟩[77]M ≡
```
    function Outchar_pre (cchr,tr1,chsz,tick) =
      begin
              chsz >= 0
          and not iscomm tick
          and not null cchr
          and cseq!is_char(tail cchr,chsz)
          and tr1 ^*? Get_alpha(chsz,tick)
      end Outchar_pre;
```
This macro is invoked in definition 209.

⟨*Definition of valid_Outchar*⟩[78]M ≡
```
    function valid_Outchar (cchr,tr1_0,tr1,chsz,tick,cntdwn) =
        begin
              tr1 ^*? Get_alpha(chsz,tick)
          and if     cntdwn > 0
                 and not odd_parity(tail cchr)
                 and last (tr1 |= inbit) = stopbit
              then tr1 = tr1_0 ^ .<inbit.stopbit,mid.tail cchr>.
              else    tr1 = tr1_0 ^ .<inbit.last (tr1 |= inbit)>.
                   or (cntdwn <= 0 and tr1 = tr1_0)
              end if
        end valid_Outchar;
```
This macro is invoked in definition 209.

Proof Structure of InOutchar Two cases are identified in the proof of **InOutchar**'s post-condition:

1. **Inchar** executes unsuccessfully with **cntdwn** = 0;

2. **Inchar** executes successfully;

The first case (1) follows directly from the following two theorems, the first of which guarantees satisfaction of Inchar's pre condition and the second of which guarantees satisfaction of its post condition:

⟨*InOutchar lemma #1*⟩[79]M ≡
```
rule valid_Inchar_tack_startbit_pre (cchr,tr1,chsz,tick) =
  begin
        InOutchar_pre(cchr,tr1,chsz,tick)
    -> valid_Inchar(cchr,cchr,tr1,tr1,chsz,tick) = true
  end valid_Inchar_tack_startbit_pre;
```
This macro is invoked in definition 209.

⟨*InOutchar lemma #2*⟩[80]M ≡
```
rule valid_Get_step_Inchar_timeout (cchr_0,cchr,tr1_0,tr1,chsz,tick,cntdwn) =
  begin
            cntdwn = 0
        and InOutchar_pre(cchr_0,tr1_0,chsz,tick)
        and valid_Inchar(cchr_0,cchr,tr1_0,tr1,chsz,tick)
    -> valid_Get_step(tr1,chsz,tick,cntdwn) = true
  end valid_Get_step_Inchar_timeout;
```
This macro is invoked in definition 209.

The second case (2) requires us to show that the pre condition for Outchar holds and that the post condition for Outchar guarantees that the post condition for Get_step holds. These facts follow directly from the following two theorems.

⟨*InOutchar lemma #3*⟩[81]M ≡
```
rule valid_Inchar_Outchar_pre (cchr_0,cchr,tr1_0,tr1,chsz,tick,cntdwn) =
  begin
            (len cchr) = chsz+1
        and InOutchar_pre(cchr_0,tr1_0,chsz,tick)
        and valid_Inchar(cchr_0,cchr,tr1_0,tr1,chsz,tick)
    -> Outchar_pre (cchr,tr1,chsz,tick) = true
  end valid_Inchar_Outchar_pre;
```
This macro is invoked in definition 209.

⟨*InOutchar lemma #4*⟩[82]M ≡
```
rule valid_Get_step_InOutchar (cchr_0,cchr,tr1_0,tr1,
                               tr2,chsz,tick,cntdwn) =
  begin
            InOutchar_pre(cchr_0,tr1_0,chsz,tick)
        and valid_Inchar(cchr_0,cchr,tr1_0,tr1,chsz,tick)
        and Outchar_pre(cchr,tr1,chsz,tick)
        and valid_Outchar(cchr,tr1,tr2,chsz,tick,cntdwn)
    -> valid_Get_step(tr2,chsz,tick,cntdwn) = true
  end valid_Get_step_InOutchar;
```
This macro is invoked in definition 209.

Proof Structure of Inchar Three cases are identified in the proof of **Inchar**'s post-condition:

1. the entire character (except the final delimiter) has been received, in which case **Inchar** terminates;
2. only part of the character has been received and reception of the next bit causes **cntdwn** decrease to zero, in which case **Inchar** terminates;
3. only part of the character has been received and the next bit is successfully received, in which case **Inchar** recurs.

The first case (1) is trivial since **Inchar**'s post condition follows trivially from the precondition and that fact that **cntdwn** is decremented.

The second case (2) follows from the following theorem:

⟨*Inchar lemma #1*⟩[83]M ≡
```
    rule valid_Inchar_tack_inbit_timeout (b,cchr,tr1,chsz,tick) =
      begin
              (len cchr) <= chsz
          and b in -{0, 1}-
          and valid_Inchar(cchr,cchr,tr1,tr1,chsz,tick)
        -> valid_Inchar(cchr,cchr ^ .<b>.,tr1,
                             tr1 ^ .<inbit.b>.,chsz,tick) = true
      end valid_Inchar_tack_inbit_timeout;
```
This macro is invoked in definition 209.

The third case (3) requires us to show that the pre condition for **Inchar** holds and that the post condition for the recurrence of **Inchar** guarantees that the post condition for **Inchar** holds. These facts follow directly from the following two theorems.

⟨*Inchar lemma #2*⟩[84]M ≡
```
    rule valid_Inchar_tack_inbit_pre (b,cchr,tr1,chsz,tick) =
      begin
              (len cchr) <= chsz
          and b in -{0, 1}-
          and valid_Inchar(cchr,cchr,tr1,tr1,chsz,tick)
        -> valid_Inchar(cchr ^ .<b>.,cchr ^ .<b>.,tr1 ^ .<inbit.b>.,
                                tr1 ^ .<inbit.b>.,chsz,tick) = true
      end valid_Inchar_tack_inbit_pre;
```
This macro is invoked in definition 209.

⟨*Inchar lemma #3*⟩[85]M ≡
```
    rule valid_Inchar_tack_inbit_step (b,cchr,pcchr,tr1,tr2,chsz,tick) =
      begin
              b in -{0, 1}-
          and valid_Inchar(cchr,cchr,tr1,tr1,chsz,tick)
          and valid_Inchar(cchr ^ .<b>.,pcchr,tr1 ^ .<inbit.b>.,
                                tr2,chsz,tick)
        -> valid_Inchar(cchr,pcchr,tr1,tr2,chsz,tick) = true
      end valid_Inchar_tack_inbit_step;
```
This macro is invoked in definition 209.

Chapter 7

Put Component Refinement

This chapter uses the same procedural approach to the design of **Put** as used in the last chapter in the design of **Get** (see the introduction to Chapter 6 for a description of the approach). We begin in Section 7.1 by describing an overview of the design of **Put** in CSP. Section 7.2 recasts this design in SVerdi by modeling CSP processes as SVerdi procedures and CSP operations as SVerdi statements. Critical requirements Assert 7.2L and Assert 8.2L are cast as a post condition on the procedure representing **Put**. Finally Section 7.3 decomposes these requirements onto the primary procedures of the design and argues that the design satisfies the set of critical requirements of **Put**.

7.1 Overview of the Put Design

Put is a recursive non-terminating CSP process. Each recurrence chooses either to send a bit from the internal buffer, i.e., **buff**, over **outbit** or receive a character over **mid**. A bit can be sent whenever there is one available to send; a character can be received only if the buffer has enough room to store it.

The buffer has sufficient capacity to store a character if the buffer is empty, since we assume **buffsz >= 1**, or if **buffsz*(chsz+2) >= (len buff)+chsz+2**, since the two delimiting bits are stored along with each **chsz**-length character. Characters received are stored at the right end of the buffer; bits transmitted are taken from the left end of the buffer. The design of **Put** in CSP follows:

```
Put (chsz, buffsz, tick,buff) =
    if null buff
    then mid ? ch -> Put(chsz,buffsz,tick,
                         buff ^ (startbit ]- (ch ^ .<stopbit>.)))
    elseif buffsz*(chsz+2) < (len(buff) + chsz + 2)
    then outbit ! head(buff) -> Put (chsz, buffsz, tick, tail(buff)
    else outbit ! head(buff) -> Put (chsz, buffsz, tick, tail(buff)
         [] mid ? ch -> Put(chsz,buffsz,tick,
                            buff ^ (startbit ]- (ch ^ .<stopbit>.)))
    end if
```

7.2 Formal Specification of the Put Design

As in the design of `Get`, the machine variable `st` representing internal state provides a means to transmit values over channels and, for specification purposes, to store a record of the sequence of values transmitted. The current trace of `Put` during execution is evaluated from `st` and must be restricted to `Put`'s alphabet. The process trace is the basis for the specification of `Put`'s two primary requirements: `Put_not_over_capacity` and `valid_char_Put`. Showing that these requirements hold for every prefix of the final trace ensures that they hold invariantly during `Put`'s execution. Although the CSP process representing `Put` does not terminate, EVES requires termination. We, therefore, force the SVerdi representation to terminate at an arbitrary point, i.e., after the variable `cntdwn` reaches zero. `cntdwn` starts at the value of the maximum integer, `int'last()`. Later refinement will reveal how and when `cntdwn` decreases.

⟨*Put design*⟩[86] ≡
```
    procedure Put (mvar st : ⟨State type⟩[159],
                   lvar chsz, buffsz : int,
                   lvar tick : ⟨Event type⟩[161]) =
      pre     chsz >= 0
          and buffsz >= 1
          and buffsz*(chsz+2) + chsz + 2 <= int'last()
          and not iscomm tick
          and ⟨Trace⟩[163]('st') = .<>.
      post ⟨Trace⟩[163]('st') ^*? Put_alpha(chsz,tick)
           and all tr2:
                   tr2 .<=. ⟨Trace⟩[163]('st')
                -> (    Put_not_over_capacity (tr2, chsz, buffsz, tick)
                    and valid_char_Put (tr2, chsz, tick))
      begin
        pvar buff : ⟨Buffer type⟩[137] := ⟨Empty buffer⟩[150]
        pvar cntdwn : int := int'last()
        Put_step(st,chsz,buffsz,tick,buff,cntdwn)
      end Put;
```
This macro is invoked in definition 213.

`Put`'s buffer `buff` is represented as a first-in first-out queue of bits. An iteration of `Put_step` depends on the current state of `buff`. If `buff` is empty the only option is to receive the next character over `mid`. If the current length of `buff` will not accommodate storing another character the only option is to send the next bit stored in `buff`. Finally, if `buff` is not empty and there is room for storing another character, either of the above options may occur. Note that an input or output operation is successful if and only if `cntdwn` remains greater than zero.

⟨*Put_step design*⟩[87]M ≡
```
    procedure Put_step (mvar st : ⟨State type⟩[159],
                        lvar chsz, buffsz : int,
                        lvar tick : ⟨Event type⟩[161],
                        pvar buff : ⟨Buffer type⟩[137],
                        pvar cntdwn: int) =
      ⟨Put_step specification⟩[90]
      begin
        pvar chr : ⟨Buffer type⟩[137]
        if ⟨buff is empty⟩[151] then
            ⟨mid ? chr⟩[174]
```

```
          if cntdwn > 0 then
            buff:= ⟨Append delimited chr onto end of buff⟩[155]
          end if
        elseif buffsz*(chsz+2) < (⟨len of buff⟩[156]+chsz+2) then
          ⟨outbit ! head of buff⟩[165]
          if cntdwn > 0 then
            buff:= ⟨tail of buff⟩[152]
          end if
        else poll_mid_and_outbit(st,chsz,buffsz,tick,buff,cntdwn)
        end if
        if cntdwn > 0 then
          Put_step(st,chsz,buffsz,tick,buff,cntdwn)
        end if
      end Put_step;
```

This macro is invoked in definition 213.

When both receiving a character over **mid** and sending a bit over **outbit** is possible, **mid** and **outbit** must be iteratively polled to determine whether a communication can take place. Once one of the communications occurs **buff** is updated as appropriate.

⟨*poll_mid_and_outbit design*⟩[88]M ≡

```
    procedure poll_mid_and_outbit(mvar st : ⟨State type⟩[159],
                                  lvar chsz, buffsz : int,
                                  lvar tick : ⟨Event type⟩[161],
                                  pvar buff : ⟨Buffer type⟩[137],
                                  pvar cntdwn : int) =
      ⟨poll_mid_and_outbit specification⟩[96]
      begin
        pvar sent,rcvd : bool := false
        pvar chr : ⟨Buffer type⟩[137]
        ⟨Poll mid to receive chr⟩[177]
        loop
          ⟨poll_mid_and_outbit loop specification⟩[99]
          exit when xor(sent,rcvd) or cntdwn=0
          ⟨Poll outbit to send first of buff⟩[168]
          if (not sent) and cntdwn > 0 then
              ⟨Poll mid to receive chr⟩[177]
          end if
        end loop
        if sent and cntdwn > 0 then
          buff:= ⟨tail of buff⟩[152]
        elseif rcvd and cntdwn > 0 then
          buff:= ⟨Append delimited chr onto end of buff⟩[155]
        end if
      end poll_mid_and_outbit;
```

This macro is invoked in definition 213.

⟨*Put's definition of xor*⟩[89]M ≡

```
    typed function xor (a, b : bool) returns bool =
      begin
             (a or b)
        and not (a and b)
    end xor;
```
This macro is invoked in definition 213.

7.3 Justification of the Put Design

This section presents an intuitive justification that the design of Put satisfies the assertions of Put described in Chapter 4. More specifically, we must show that the requirements listed in Section 5.4 follow from the design of Put described in the last section.

7.3.1 Assertions Argument

Assert 3.2L:

By inspection of the Put design, the only way Put can access external channels is through the **mach** library unit interface. The only calls Put makes are to send a bit over **outbit** using **snd_bit** and **poll_snd_bit** and to receive a character over **mid** using **rcv_char** and **poll_rcv_char**. Of course, any refinement of these calls must be shown not to send any information over **mid**. However, since we are using this model for design verification only, this level of analysis suffices. □

Assert 3.3L:

Put is not decomposed into concurrent sub-processes so no internal communication channels exist. □

Assert 6.2L using Assert 7.2L, Assert 8.2L:

We must argue that the procedural model of the Put CSP process has certain critical properties required of sequential processes. This argument is necessarily informal since the correspondence between our procedural model and CSP is informal. First we show that Put is a CSP process and then we show that it is sequential.

From the definition of **isprocess** in the **pr** model library unit, we must show that

1. the empty trace is a trace of **Get**,
2. the traces of **Put** are restricted to events in **put_alpha(chsz,tick)**, and
3. any prefix of a trace of **Put** is also a trace of **Put**.

We define the traces of the procedural design of **Put** so that these properties hold. By inspection of this design, the trace of **Put** is constructed as it executes by appending to its end the communication event associated with every communication in which it engages. Viewing **Put**'s execution symbolically, we define the set of traces of **Put** as the set of all possible traces constructed up to any point in its execution. Clearly this ensures that (3) holds. (1) holds since the pre-condition for **Put** ensures that the trace is initially empty. (2) holds as a consequence of Assert 7.2L and the portion of the post-condition for **Put** that requires all traces generated to be confined to events in **put_alpha(chsz,tick)**. We delay proof of the post-condition until the argument for Assert 8.2L.

From the definition of **is_seqpr** from the **pr** model library unit, a sequential process must have the successful termination event in its alphabet. The only place this event may appear in a trace of a sequential process is as the last event. The parameter **tick** represents the successful termination event for **Put**. **tick** is in the alphabet of **Put** as shown in Assert 7.2L. **tick** does not occur in any trace of **Put** since **Put** does not terminate successfully.[1] □

Assert 7.2L:

We define the alphabet of the procedural model for **Put** as the set of events in which it may engage. The post-condition for **Put** requires that this set be restricted to **put_alpha(chsz,tick)**. We delay proof of the post-condition until the argument for Assert 8.2L. □

Assert 8.2L:

Assert 8.2L requires that all traces of **Put** be members of the set defined by **valid_Put**. This requires that every trace of **Put** satisfy **Put_not_over_capacity** and **valid_char_Put**. The post-condition for **Put** requires these properties of the **Put** design. The rest of this section presents an overview of the proof of the post-condition.

Proof Structure of Put The first task is to derive a set of valid requirements for **Put_step** that allows us to prove that **Put** satisfies its post-condition. The recursive nature of **Put_step** suggests that its specification is invariantly true before and after execution. We call this invariant **valid_Put_step**.

⟨*Put_step specification*⟩[90]M ≡
```
pre      cntdwn > 0
    and buffsz*(chsz+2) + chsz + 2 <= int'last()
    and valid_Put_step (⟨Contents⟩[143]('buff'),⟨Trace⟩[163]('st'),
                        chsz,buffsz,tick)
post valid_Put_step (⟨Contents⟩[143]('buff'),⟨Trace⟩[163]('st'),
                        chsz,buffsz,tick)
measure cntdwn
```
This macro is invoked in definition 87.

Discovering the invariant for **Put_step** requires understanding the crucial role that the internal buffer **buff** plays. We refer to the contents of **buff** as **cbuff**. **cbuff** invariantly contains the delimited characters, or parts of characters, received over **mid** buf not yet transmitted over **outbit**. The length of **cbuff**, i.e., **len cbuff**, plus the number of bits of the last non-complete character transmitted over **outbit**, i.e., **(len tr1 |= outbit) mod (chsz+2)**, must be bounded above by the maximum capacity of the buffer, i.e., **buffsz*(chsz+2)**.

⟨*Definition of valid_Put_step*⟩[91]M ≡
```
function valid_Put_step (cbuff,tr1, chsz, buffsz,tick) =
  begin
        chsz >= 0
    and buffsz >= 1
    and not iscomm tick
```

[1]The SVerdi procedural model of **Put** does complete execution, but only because EVES requires proof of termination. This may be considered unsuccessful termination in the CSP sense.

```
        and tr1 ^*? put_alpha(chsz,tick)
        and not (tick -[ tr1)
        and ⟨mid allows only chars⟩[121]
        and invariant_over_buffer(cbuff,TR1, CHSZ, buffsz,tick)
        and ALL TR2: some cbuff2:
               TR2 .<=. TR1
            -> invariant_over_buffer(cbuff2,TR2, CHSZ, buffsz,tick)
      end valid_Put_step;
```
This macro is invoked in definition 213.

⟨*Definition of invariant_over_buffer*⟩[92]M ≡
```
    function invariant_over_buffer(cbuff,TR1, CHSZ,buffsz,tick) =
       begin
              cbuff ^*? -{0,1}-
          and ⟨mid chars to bits⟩[123] = ((TR1 |= OUTBIT) ^ cbuff)
          and buffsz*(chsz+2) >= (len cbuff)
                                   + ((len TR1 |= OUTBIT) mod (chsz+2))
       end invariant_over_buffer;
```
This macro is invoked in definition 213.

Showing that the specification for **Put** follows from the specification for **Put_step** requires showing that

1. the pre-condition and initial assignments for **Put** imply the pre-condition for **Put-step**, and
2. the post-condition for **Put-step** implies the post-condition for **Put**.

The first proof obligation follows trivially from the definitions involved:

⟨*Put lemma #1*⟩[93]M ≡
```
    rule valid_Put_step_empty_empty (chsz,buffsz,tick) =
      begin
               chsz >= 0
           and buffsz >= 1
           and not iscomm tick
        -> valid_Put_step(.<>.,.<>.,chsz, buffsz,tick) = true
      end valid_Put_step_empty_empty;
```
This macro is invoked in definition 213.

The second proof obligation is split according to the structure of the post-condition for **Put**. Note that the requirement that the trace be restricted to **put_alpha(chsz,tick)** follows trivially from the definition of **valid_Put_step**.

⟨*Put lemma #2*⟩[94]M ≡
```
    rule Put_step_Put_not_over_capacity (cbuff,tr1,tr2,chsz,buffsz,tick) =
      begin
               tr2 .<=. tr1
           and valid_Put_step (cbuff, tr1, chsz, buffsz,tick)
        -> Put_not_over_capacity (tr2, chsz, buffsz, tick) = true
      end Put_step_Put_not_over_capacity;
```
This macro is invoked in definition 213.

⟨*Put lemma #3*⟩[95]M ≡

```
rule put_step_valid_char_Put (cbuff,tr1,tr2,chsz,buffsz,tick) =
  begin
          tr2 .<=. tr1
      and valid_Put_step (cbuff,tr1, chsz,buffsz, tick)
    -> valid_char_Put (tr2, chsz, tick) = true
  end put_step_valid_char_Put;
```
This macro is invoked in definition 213.

Proof Structure of Put_step This level requires us to derive a set of requirements for poll_mid_and_outbit that allows us to prove that Put_step satisfies its post-condition. poll_mid_and_outbit can assume that either a character reception or a bit transmission is possible; thus, the buffer is not empty and there is room to store a character. We know that valid_Put_step must remain invariant in either case. Furthermore, to prove that Put_step terminates, cntdwn must decrease to a value no less than zero.

⟨*poll_mid_and_outbit specification*⟩[96]M ≡
```
initial buff_0=buff, st_0=st,cntdwn_0=cntdwn
pre      cntdwn > 0
     and not nullp buff
     and buffsz*(chsz+2) >= (len ⟨Contents⟩[143]('buff')) + chsz + 2
     and valid_Put_step (⟨Contents⟩[143]('buff'),⟨Trace⟩[163]('st'),
                         chsz,buffsz,tick)
post     cntdwn < cntdwn_0
     and cntdwn >= 0
     and valid_Put_step (⟨Contents⟩[143]('buff'),⟨Trace⟩[163]('st'),
                         chsz,buffsz,tick)
```
This macro is invoked in definition 88.

Put_step may receive a character over mid or transmit the head of the buffer over outbit depending, in part, on the status of the internal buffer. The cases are identified in the proof of Put_step's post-condition:

1. cntdwn becomes zero, in which case Put_step terminates with no change to the buffer or trace;
2. cntdwn remains greater than zero and a character is received over mid, in which case Put_step terminates with the delimited character appended to the end of the buffer and the associated communication event appended to the end of the trace; and
3. cntdwn remains greater than zero and a character is transmitted over outbit, in which case Put_step terminates with the head of the buffer deleted and the associated communication even appended to the end of the trace.

Each of these cases generates a proof obligation in addition to that generated for proof of termination. That these are all of the cases can be seen by inspection of the post-condition of the procedures involved. We deal with each case in turn. Termination follows directly from the fact that cntdwn decreases each iteration and is bounded below by zero.

The first case (1) is trivial since no change to the buffer or trace requires only that valid_Put_step hold initially, which is guaranteed by Put_step's pre-condition.

The second case (2) requires us to prove the following theorem. We can assume that there is room for storing a character in the internal buffer since Put_step only permits receiving characters over mid when this is the case.

⟨*valid_Put_step lemma #1*⟩[97]M ≡
```
rule valid_Put_step_tack_mid (cbuff,tr1,chsz,buffsz,tick,chr) =
  begin
           buffsz*(chsz+2) >= (len cbuff) + chsz + 2
       and cseq!is_char(chr,chsz)
       and valid_Put_step(cbuff,tr1,chsz,buffsz,tick)
    -> valid_Put_step(cbuff ^ (startbit ]- (chr ^ .<stopbit>.)),
                          tr1 ^ .<mid.chr>.,chsz,buffsz,tick) = true
  end valid_Put_step_tack_mid;
```
This macro is invoked in definition 213.

The third case (3) requires us to prove the following theorem. We can assume that the buffer is not empty since `Put_step` only permits transmitting bits over `outbit` when this is the case.

⟨*valid_Put_step lemma #2*⟩[98]M ≡
```
rule valid_Put_step_tack_outbit (b,cbuff,tr1,chsz, buffsz,tick) =
  begin
           not null cbuff
       and (head cbuff) = b
       and valid_Put_step(cbuff,tr1,chsz, buffsz,tick)
    -> valid_Put_step(tail cbuff,tr1 ^ .<outbit.b>.,
                     chsz, buffsz,tick) = true
  end valid_Put_step_tack_outbit;
```
This macro is invoked in definition 213.

Proof Structure of poll_mid_and_outbit The proof of `poll_mid_and_outbit` is similar to the proof of `Put_step` since the same two communications are possible during its execution: sending a bit over `outbit` and receiving a character over `mid`. The invariant for the loop hinges on the boolean variables `sent` and `rcvd`, which indicate which if either communication has occurred.

⟨*poll_mid_and_outbit loop specification*⟩[99]M ≡
```
invariant     cntdwn < cntdwn_0
          and cntdwn >= 0
          and chsz >= 0
          and buffsz >= 1
          and not iscomm tick
          and buff_0=buff
          and not (sent and rcvd)
          and if rcvd and cntdwn > 0 then
                       cseq!is_char(⟨Contents⟩[143]('chr'),chsz)
                   and ⟨Trace⟩[163]('st')
                           = ⟨Trace⟩[163]('st_0')
                             ^ .<mid.⟨Contents⟩[143]('chr')>.
                elseif sent and cntdwn > 0 then
                   ⟨Trace⟩[163]('st')
                       = ⟨Trace⟩[163]('st_0')
                         ^ .<outbit.head ⟨Contents⟩[143]('buff')>.
                else ⟨Trace⟩[163]('st') = ⟨Trace⟩[163]('st_0') end if
measure cntdwn
```
This macro is invoked in definition 88.

The loop in `poll_mid_and_outbit` terminates when the polling process results in a successful communication. Proof of the post-condition on termination relies on the same two lemmas as the proof of `Put_step`: ⟨*valid_Put_step lemma #1*⟩ and ⟨*valid_Put_step lemma #2*⟩.

The EVES proof of lemmas cited in this justification are documented in the `put` library model unit. □

Part III

The Repeater Physical Design

Chapter 8

Repeater Physical Architecture

This chapter presents the physical CSP architecture for **Rptr**. The target for the physical architecture and design is the CSP design specified in [1]. We present in Section 8.1 an overview of the physical architecture including the primary responsibilities of the components of that architecture. Section 8.2 translates the **Rptr** logical design described in SVerdi in Part II of this document to the syntax required by FDR. The FDR logical design is the CSP process specification to which the physical design must conform. Section 8.3 summarizes the critical requirements of the physical architecture. These requirements are derived from the top-level critical requirements described in Chapter 4. This derivation uses the fact that the **Rptr** logical design conforms to the requirements of Section 2.1 to simplify the physical architecture critical requirements. The critical requirements of the physical architecture were not derived directly from the logical architecture requirements since the logical architecture has a different structure than the physical architecture.

8.1 Overview of the Physical Architecture

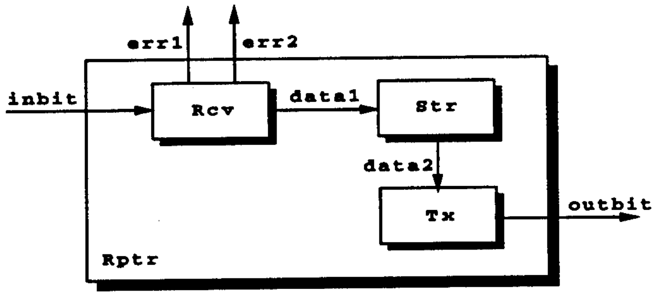


Figure 8.1: Repeater Physical Architecture

As depicted in Figure 8.1, the **Rptr** physical architecture composes three processes: **Rcv**, which receives and checks the parity of incoming characters; **Str**, which maintains the internal buffer; and **Tx**, which transmits those characters that passed the parity check. The internal channels **data1** and **data2** are used to pass characters of even parity for transmission over **outbit**. The channels **err1** and **err2** are used to relay status information, such as parity or framing error, to the operator.

Assert 2.1P The environment does not send data over **outbit**, **err1**, or **err2**.

At the external interface, the Rptr physical design behaves the same as the Rptr design if you ignore communications over the error channel. This fact is, of course, the subject of the justification that the physical design conforms to the design constraints. From an internal perspective, the behavior of the Rptr physical design is a refinement of the the behavior of the Rptr logical design. For each delimited character received over inbit, Rcv determines the parity of the character that the segment represents. A signal is sent over the error channels indicating whether the character received had a parity error (not even parity), a framing error (improper stopbit delimiter), or if no error occurred. If the character is of odd parity it is thrown out, and the reception of a new character is initiated. Otherwise, the startbit and stopbit are stripped off and the character is sent as an integer encoding over data1 to Str.

Assert 3.1P The environment does not send or receive data over data1 or data2.

Str has the capacity to store only a single character at a time. When Tx is ready, Str relays the character encoded as an integer over data2 to Tx. Tx decodes the character, adds the startbit and stopbit delimiters, and transmits the character in serial for over outbit. As in the logical design, Rptr's physical components are tightly synchronized. Rcv cannot receive a new character until it has transmitted the character it is processing to Str. Str cannot accept another character from Rcv until its buffer is empty. Finally, Tx cannot receive a character from Str until it has transmitted the character that it is processing over outbit. Note that this design implies that the size of the buffer, N, specified in the initial problem description equals three; each component has a capacity of one character.

8.2 Translation of the Logical Design to FDR

FDR places certain constraints on CSP descriptions to which the Rptr design currently does not conform:

- Process descriptions that involve high level operators may not be parameterized. This implies that the parameters of low level process descriptions must be concrete.
- The set of events that synchronize concurrent components must be explicit. Processes are composed with the [| X |] operator (where X is the set of events synchronized) rather than the || or the |?| operators.
- The set of events hidden for abstraction purposes must be explicit. The hiding operator \ is used to hide internal events rather than the |?| operator.
- Values that may be transmitted over channels are limited to integers or truth values. The characters transmitted over mid in the Rptr design must be encoded as integers before transmission by Get and decoded before transmission by Put.
- The successful termination event in FDR is implicit. The parameter tick in the Rptr design is suppressed in the FDR specification.
- Functions returning something other than a CSP process must be defined in ML.

This section recasts the logical design of Part II to conform to these constraints in the syntax required by FDR and ML. The CSP process called Rptr in Part II is renamed RptrSpec in the FDR specification. The CSP process Rptr now represents the repeater physical design. The goal of the FDR verification will be to show that Rptr refines RptrSpec.

Assert 8.1P traces Rptr .<=. traces RptrSpec

8.2.1 FDR Process Specification

The alphabet of a CSP process in FDR is implicitly specified as the union of the set of communication events associated with all channels declared via the channel pragma declarations.

⟨*Rptr specification channel declarations*⟩[100]M ≡

```
pragma channel inbit, outbit : bit
pragma channel mid : max_vals
```

This macro is invoked in definition 215.

The values that may be transmitted over a channel are strictly defined by the set of values specified in the channel pragma declarations. We limit the values that may flow over `mid` to integers between zero and fifteen, which restricts `Rptr` to process characters of at most four bits long, i.e., `K <= 4`.

Assert 9P `K <= 4`.

⟨*Values transmitted over external channels*⟩[101]M ≡

```
bit = {0,1}
```

This macro is invoked in definition 215.

⟨*Values transmitted over mid*⟩[102]M ≡

```
max_vals = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
```

This macro is invoked in definition 215.

The `Rptr` design was specified in Section 5.5 as the concurrent composition of `Get` and `Put` with internal event hiding. `mid` is the only channel on which `Get` and `Put` are synchronized and, thus, is the only internal channel whose communications are to be hidden. Note that we use the concrete values `K` and `N-1` as the character size and internal buffer size respectively. These values will be identified during later refinement.

⟨*FDR Rptr specification*⟩[103]M ≡

```
RptrSpec = Get(K) [|{mid}|] Put(K,N-1,<>) \ {| mid |}
```

This macro is invoked in definition 215.

The primary difference between the FDR process specification of `Rptr`'s components and the design specified in Sections 6.1 and 7.1 is the transmission of integers, rather than characters, over `mid`. Characters are encoded as integers by `Get` via a function `cnv_to_int` and decoded by `Put` via a function `cnv_to_int`.

⟨*FDR Get specification*⟩[104]M ≡

```
Get(chsz) =
    inbit ? b -> if b == startbit
                 then Inchar(<b>,chsz);Get(chsz)
                 else Get(chsz)
```

This macro is invoked in definition 215.

⟨*FDR definition of inchar*⟩[105]M ≡

```
Inchar(ch,chsz) =
  if #(ch) <= chsz then
    inbit ? b -> Inchar(ch^<b>,chsz)
  else inbit ? b -> if     not odd_parity(ch)
```

```
                                    and b == stopbit
                                  then mid ! cnv_to_int(tail(ch)) -> SKIP
                                  else SKIP
```

This macro is invoked in definition 215.

⟨*FDR Put specification*⟩[106]M ≡

```
    Put(chsz,buffsz,buff) =
      if null(buff) then
        mid ? i -> Put(chsz,buffsz,
                   buff^(<startbit>^(cnv_to_char(<i,chsz>)^<stopbit>)))
      else if buffsz*(chsz+2) < #(buff) + chsz + 2
         then outbit ! head(buff) -> Put(chsz,buffsz,tail(buff))
      else
        outbit ! head(buff) -> Put(chsz,buffsz,tail(buff))
         [] mid ? i -> Put(chsz,buffsz,
                        buff^(<startbit>
                              ^(cnv_to_char(<i,chsz>)^<stopbit>)))
```

This macro is invoked in definition 215.

8.2.2 ML Support Definitions

Recall that FDR requires that support functions, such as **odd_parity** and the character coercion functions, to be defined in ML. These functions are rather awkward to specify since FDR requires that the parameters to ML functions be of type ML **expression** and the return type be either ML **expression** or **bool**. FDR provides functions that allow coercion of **expressions** to/from **atoms** and **atoms** to/from **int** as described in Section 3.3.1 Using these coercion functions, **odd_parity** is specified as follows:

⟨*ML definition of odd_parity*⟩[107]M ≡

```
    fun xor(x,y) = (        (x orelse y)
                    andalso not(x andalso y));;

    fun is_odd ch =
            if null ch then false
            else (xor ((CheckAtom(hd ch)=InjectNum(1)),
                       is_odd (tl ch)));;

    fun odd_parity [EXPseqcomp(ch,[])] =
            is_odd(ch);;
```

This macro is invoked in definition 217.

Characters are converted to integers by simply calculating the decimal value of the binary representation of the bit sequence representing the character.

⟨*ML definition of cnv_to_int*⟩[108]M ≡

```
    fun char_to_int ch =
           if null ch then 0
              else NumberOf(CheckAtom(hd ch))
                    + 2*char_to_int(tl ch);;

    fun cnv_to_int [EXPseqcomp(ch,[])] =
        (Atom(InjectNum(char_to_int(ch))));;
```

This macro is invoked in definition 217.

Converting integers to characters is the inverse of the above operation.

⟨*ML definition of cnv_to_char*⟩[109]M ≡

```
    fun int_to_char (i,sz) =
      if i<1  andalso sz<1 then []
      else if i<1 then ([Atom(InjectNum(0))] @ int_to_char(i,sz-1))
      else ([Atom(InjectNum(i mod 2))] @ int_to_char(i div 2,sz-1));;

    fun cnv_to_char [EXPseqcomp([i,sz],[])] =
      EXPseqcomp(int_to_char(NumberOf(CheckAtom(i)),
                             NumberOf(CheckAtom(sz))), []);;
```

This macro is invoked in definition 217.

8.3 Summary of the Physical Architecture Critical Requirements

The following critical requirements were derived from the top-level Rptr critical requirements of Chapter 4 and the assumptions and assertions identified in throughout this chapter.

8.3.1 Assumptions

1 Power is continuously supplied to `Rptr`.

2.1P The environment does not send data over `outbit`, `err1`, or `err2`.

3.1P The environment does not send or receive data over `data1` or `data2`.

8.3.2 Assertions

Informal Assertions

1 If `Rptr` is continuously powered, `Rptr` and its environment can communicate only via external channels; communication between `Rptr` sub-processes can take place only over channels shared by the alphabets of the sub-processes.

2 `Rptr` does not send data over `inbit`.

3 Communication between `Rptr` sub-processes must be uni-directional and involve exactly two sub-processes.

4 The implementation of communications over a channel in the `Rptr` process description must synchronize sender and receiver.

5 `Rptr` must not engage in unguarded recursion nor engage in an infinite sequence of hidden events.

Formal Assertions

6 `Rptr` is a CSP process.

7 `Rptr`'s alphabet is defined by `Rep_alpha_ext`.

8.1P `traces Rptr .<=.  traces RptrSpec`

9P `K <= 4`

Chapter 9

Repeater Detailed Physical Design

This chapter describes the `Rptr` CSP process implementation in FDR and the justification that this physical design satisfies the critical requirements identified in the last chapter. This justification generates a set of critical requirements that must be satisfied of any further refinement of `Rptr`.

9.1 Formal Specification of the Physical Design

The `Rptr` physical design requires defining four new channels: two for passing characters between the components and two one-bit channels representing the two-bit error channel. We use two channels to represent the error channel for simplicity. We hide these four channels in the `Rptr` process implementation since they did not occur in the `Rptr` design. The models of refinement in CSP all require that the set of traces of the process implementation be a subset of the set permitted by the process specification.

⟨*Additional channels of the Rptr physical design*⟩[110]M ≡

```
    pragma channel data1, data2 : max_vals
    pragma channel err1, err2 : bit
```

This macro is invoked in definition 215.

⟨*Rptr physical architecture*⟩[111]M ≡

```
    Rptr = Rcv [|{data1}|] (Str [|{data2}|] Tx)
           \ {| data1,data2,err1,err2 |}
```

This macro is invoked in definition 215.

Once the `Rcv` component receives a startbit it begins processing the next `K`-bit character. The process `Data` receives all but the `startbit` and first bit of the character `ch`. Once `Data` has received `K-2` bits (making a total of `K` bits received including `startbit`), `Data` initiates reception of the last bit of the character.

⟨*Definition of Rcv*⟩[112]M ≡

```
    Rcv =
      inbit ? b -> if b != startbit then Rcv
                   else inbit ? b -> Data(<b>,0,b == 1)

    Data(ch,cnt,parity_err) =
    if cnt < K-2 then
      ⟨Accept the next bit and continue⟩[113]
    else ⟨Accept the last bit and continue⟩[115]
```
This macro is invoked in definition 215.

A running tally of the parity status of the incoming character is maintained by `Data` using a function of zero arguments called `prty_xor`. The Boolean variable `parity_err` is true if and only if the character thus far received has odd parity.

⟨*Accept the next bit and continue*⟩[113]M ≡
```
    inbit ? b -> Data(ch^<b>,cnt+1,prty_xor)
```
This macro is invoked in definition 112.

⟨*Definition of prty_xor*⟩[114]M ≡
```
    prty_xor = (parity_err or (b == 1))
                 and (not (parity_err and (b == 1)))
```
This macro is invoked in definition 215.

Once the last bit of the character has been received, the final delimiting bit is received and stored in the variable `frame_err`, is received. `Output` signals the error status of the character just received and transmits the character over `mid` if no error is indicated.

⟨*Accept the last bit and continue*⟩[115]M ≡
```
    inbit ? b -> Stop_bit(ch^<b>,prty_xor)
```
This macro is invoked in definition 112.

⟨*Definition of Stop_bit*⟩[116]M ≡
```
    Stop_bit(ch,parity_err) =
      inbit ? frame_err -> Output(ch,parity_err,frame_err)

    Output(ch,parity_err,frame_err) =
      if parity_err then ErrOut(0,1);Rcv
      else if frame_err != stopbit then ErrOut(1,0);Rcv
      else ErrOut(0,0); data1!cnv_to_int(ch) -> Rcv

    ErrOut(b1,b2) =
      err1 ! b1 -> err2 ! b2 -> SKIP
```
This macro is invoked in definition 215.

`Str` is a simple one-place buffer.

⟨*Definition of Str*⟩[117]M ≡
```
    Str = data1 ? i -> data2 ! i -> Str
```
This macro is invoked in definition 215.

`Tx` begins at `Input` by waiting to input an integer over `data2`, after which it decodes the integer returning the character. `Input` recursively transmits the delimited character bit by bit over `outbit`. Once complete `Input` starts over again.

⟨*Definition of Tx*⟩[118]M ≡
```
    Tx = Input(<>,K+2)

    Input(ch,outct) =
    if outct == K+2 then data2?i -> Input(cnv_to_char(<i,K>),0)
    else if outct == 0 then outbit ! startbit -> Input(ch,outct+1)
    else if outct == K+1 then outbit ! stopbit -> Input(ch,outct+1)
    else outbit ! head(ch) -> Input(tail(ch),outct+1)
```
This macro is invoked in definition 215.

9.2 Justification of the Physical Design

This section presents an intuitive justification that the FDR `Rptr` physical design satisfies the critical requirements of `Rptr` as described in Section 8.3. We argue informally that the physical design satisfies the informal assertions. We use a combination of informal and formal techniques to argue that the physical design satisfies the formal assertions. In particular we use the FDR model checker to demonstrate that the `Rptr` physical design satisfies the constraints of the `Rptr` specification.

9.2.1 Assumptions Argument

Assump 1:

No change.

Assump 2.1P:

No change.

Assump 3.1P:

No change.

9.2.2 Assertions Argument

Assert 1:

By inspection of the `Rptr` physical design, the only way that `Rptr` can communicate with its environment is through the `inbit`, `outbit`, `err1` and `err2` channels. Likewise, the components can communicate only through the channels defined. This requirement must be true of any refinement of the physical design and so remains in the list of critical requirements.

Assert 2:

By inspection, `Rcv` is the only component with access to `inbit` and `Rcv` does not transmit data over `inbit`. □

Assert 3:

Trivial by inspection. □

Assert 4:

This requirement must be true of any refinement of the physical design and so remains in the list of critical requirements.

Assert 5:

The non-divergence of `Rptr` is verifiable either through inspection of the physical design or, more formally, using FDR. FDR verification proceeds by showing that the process `Rptr` is a proper refinement of the process `CHAOS({|inbit,outbit|})`. This process is the most non-deterministic non-divergent process over the alphabet of bit communications over `inbit` and `outbit`. This verification proceeds fully automatically. For completeness, we also verified that `RptrSpec` is non-divergent. □

Assert 6:

We must argue that the FDR model of CSP processes has the properties that our EVES theory requires of those processes. This argument is necessarily informal since the correspondence between the EVES theory and the FDR model of CSP is informal.

From the definition of `isprocess` in the `pr` model library unit, we must show that

1. the empty trace is a trace of `Rptr`,
2. the traces of `Rptr` are restricted to events in `Rep_alpha_ext(chsz,tick)`, and
3. any prefix of a trace of `Rptr` is also a trace of `Rptr`.

The traces of a CSP process are the same in FDR as they are in the EVES theory. Thus, the empty trace is a trace of every process (1) and all prefixes of a trace of a process is also a trace of that process (3). (2) holds as a consequence of Assert 7.1P which is proved next. □

Assert 7:

`Rptr` hides all events that are not communications of bit values over `inbit` or `outbit`. Since the successful termination event `tick` is implicit in FDR, the alphabet of `Rptr` is exactly `Rep_alpha_ext(tick)`. □

Assert 8.1P:

This assertion is verified fully automatically by FDR but the result depends on the values of the character size `K` and the maximum character storage capacity `N`. The FDR Traces Refinement checks are successful for values in the range `2<=K<=4` and `N>=3`.

Assert 9.1P `2<=K<=4`

The upper bound on `K` is due to the need in the FDR specification to permit the communication of only a finite set of values over individual channels. Thus, values communicated over `mid` were constrained to lie between zero and fifteen. The algorithm used to convert between integers and characters implies that this constraint limits the character size to a maximum of four bits. Although this limit may seem rather arbitrary, the larger the character size the longer and the more memory it takes for the FDR verification to terminate. A character size of four was the longest possible before the state explosion exhausted memory on the machine on which the verification was performed.

The lower bound on `K` is due to the structure of the design of `Rcv`. `Rcv` requires that at least one bit of the character be received immediately after the `startbit` and that an additional character be received immediately before the `stopbit`. A Traces Refinement check fails for values `0<=K<2` since the design always receives at least two bits (in addition to the `startbit` and `stopbit`) whereas the specification receives exactly `K` bits. There exist traces of the process implementation that are not traces of the process specification.

`N` is a parameter only of the `Rptr` process specification. The process implementation assumes that the maximum capacity of `Rptr` is exactly three characters - each component can be processing/storing at most one character at a time. The specification, on the other hand, assumes only that the buffer be able to store at least one character. For values of `N` less than three, the process implementation exhibits behavior not permitted by the process specification — the Traces Refinement check in this range fails. The Traces Refinement check succeeded for `N = 3` and `N = 4` before the state explosion caused FDR to exhaust memory.

To test the limits of the `Rptr` physical design we applied the more sophisticated Failures Divergences Refinement check as well. The check succeeded for the same values of `K` as before but only for `N = 3`. The reason for this is clear. The Failures Divergences Refinement model requires that the implementation not refuse to engage in any event in which the specification may engage. For values of `N` greater than three, the implementation refuses to receive greater than three characters before transmitting a character. The specification does not refuse to receive characters in these circumstances. This is as one would expect for models of CSP that permit verification of liveness as well as safety properties. □

Assert 9P using Assert 9.1P:

As described above the FDR verification of `Rptr` succeeded for values `2<=K<=4`. This generates Assert 9.1P. □

9.3 Summary of the Implementation Critical Requirements

The following assumptions must be validated of any environment in which `Rptr` is embedded. The following assertions must be proven of any refinement of the `Rptr` physical design.

9.3.1 Assumptions

1 Power is continuously supplied to `Rptr`.

2.1P The environment does not send data over `outbit`, `err1`, or `err2`.

3.1P The environment does not send or receive data over `data1` or `data2`.

9.3.2 Assertions

Informal Assertions

1 If **Rptr** is continuously powered, **Rptr** and its environment can communicate only via external channels; communication between **Rptr** sub-processes can take place only over channels shared by the alphabets of the sub-processes.

4 The implementation of communications over a channel in the **Rptr** process description must synchronize sender and receiver.

9.1P **2<=K<=4**

Part IV

Supporting Definitions

Chapter 10

Character Sequence Theory

The formalization of the repeater critical requirements is based on a theory, called **cseq**, for partitioning a bit stream into a sequence of characters and for reasoning about the bit stream in an abstract manner. The theory also includes facilities for reasoning about the parity of characters. Characters include all **chsz**-length sequences of bits.

⟨*Definition of char_set*⟩[119]M ≡
```
function char_set(chsz) =
  measure chsz
begin
  if chsz > 0
  then pr!map_tack(0, char_set(chsz - 1))
            ++ pr!map_tack(1, char_set(chsz - 1))
  else unit .<>.
  end if
end char_set;
```
This macro is invoked in definition 201.

All characters processed by **Rptr** are delimited by a **startbit/stopbit** combination. These values are implemented as SVerdi functions, **start_bit** and **stop_bit**, with no parameters. Since these values are really constants we define **nilfix** aliases for each to make them easier to use. To avoid confusion, henceforth, we refer to a character that is delimited appropriately as a *delimited character*; the term *character* simply refers to the entity between the delimiters.

⟨*Definitions of startbit/stopbit*⟩[120] ≡
```
function start_bit () =
begin
  0
end start_bit;
nilfix startbit start_bit;

function stop_bit () =
begin
  1
end stop_bit;
nilfix stopbit stop_bit;
```
This macro is invoked in definition 201.

10.1 Primary Operations

The primary operations provided by **cseq** are to identify character sequences (**is_char_seq**), to convert between bit streams and character sequences (**char_seq** and **flatten**), and to determine the current (partial) character being processed (**current_char**). These functions are used many times in the definition and decomposition of the repeater critical requirements.

⟨*mid allows only chars*⟩[121]M ≡

```
    cseq!is_char_seq(tr1 |= mid,chsz)
```

This macro is invoked in definitions 64, 69, 73, and 91.

⟨*Bits to chars*⟩[122](◇1)M ≡

```
    cseq!char_seq(◇1,chsz)
```

This macro is invoked in definitions 12, 15, 19, 20, 37, 48, and 49.

⟨*mid chars to bits*⟩[123]M ≡

```
    cseq!flatten(tr1 |= mid,chsz)
```

This macro is invoked in definition 92.

⟨*Partial char over inbit*⟩[124](◇1)M ≡

```
    cseq!current_char(◇1 |= inbit,chsz)
```

This macro is invoked in definitions 64, 73, 73, and 75.

We begin by defining each of the above functions in terms of a set of secondary operations. **is_char_seq** describes a trivial predicate that holds if and only if each element of the sequence that it is passed is a character, via the **is_char** call.

⟨*Definition of is_char_seq*⟩[125]M ≡

```
    function is_char_seq (s, chsz) =
      measure len s
      begin
        if null s then true
        else     is_char(head s,chsz)
             and is_char_seq(tail s,chsz)
        end if
      end is_char_seq;
```

This macro is invoked in definition 199.

char_seq begins by searching through the bit stream for the first **startbit**. Once found, a decision must be made whether the bit sequence starting at that **startbit** begins with a valid delimited character, i.e., whether **has_char (s, chsz)** returns **true**. If so, the character is retrieved, via the **char_head** call, and the search continues by recurring on all but the first delimited character, via the first **char_tail** call. Otherwise, the malformed delimited character (which may be merely an incomplete delimited character) is discarded, via the second **char_tail** call, and the search continues. The recursion bottoms out when the bit sequence is null, returning the empty sequence. This process constructs the **chsz**-length character sequence formed from the bit sequence **s**.

⟨*Definition of char_seq*⟩[126]M ≡

```
    function char_seq(s, chsz) =
      measure len s
    begin
      if     not null s
      and not head(s) = startbit
    then char_seq(tail s, chsz)
    elseif has_char(s, chsz)
    then char_head(s, chsz) ]- char_seq(char_tail(s, chsz), chsz)
    elseif not null s
    then char_seq(char_tail(s, chsz), chsz)
    else .<>. end if
  end char_seq;
```
This macro is invoked in definitions 199 and 201.

If **flatten** is passed a valid character sequence, it should reverse the effect of **char_seq**. This is done by appropriately delimiting each character of the sequence it is passed.

⟨*Definition of flatten*⟩[127]M ≡
```
  function flatten (cs,chsz) =
    measure len cs
    begin
      if null cs
         or not is_char(head cs,chsz) then .<>.
      else (startbit ]- (head cs)) ^ (stopbit ]- flatten(tail cs, chsz))
      end if
    end flatten;
```
This macro is invoked in definition 199.

current_char traverses a bit sequence until it gets to a delimited character that is only partially specified. It ignores all false starts and fully specified characters. If no partial character remains at the end of the bit sequence after its traversal, the empty sequence is returned.

⟨*Definition of current_char*⟩[128]M ≡
```
  function current_char (s,chsz) =
    measure len s
  begin
    if     not null s
       and not head(s) = startbit
    then current_char(tail s, chsz)
    elseif     not null s
           and (len s) >= chsz+2
    then current_char(char_tail(s, chsz), chsz)
    elseif not null s
    then s
    else .<>. end if
  end current_char;
```
This macro is invoked in definitions 199 and 201.

10.2 Secondary Operations

10.2.1 Definition of has_char

has_char determines whether a bit sequence begins with a valid delimited character. It does this by asking whether there is some character that when delimited by a **startbit/stopbit** forms a prefix of the bit sequence.

⟨*Definition of has_char*⟩[129] ≡
```
function has_char (s, chsz) =
begin
  some c:     is_char (c, chsz)
          and startbit ]- (c ^ .<stopbit>.) .<=. s
end has_char;
```
This macro is invoked in definition 201.

10.2.2 Definition of is_char

is_char determines whether a given sequence **c** is a **chsz**-length character. This requires only that **c** be a sequence of only **0**'s and **1**'s and be exactly **chsz** in length.

⟨*Definition of is_char*⟩[130] ≡
```
function is_char (c, chsz) =
  measure len c
begin
  if     chsz > 0
     and not null c
  then     head c in -{0, 1}-
       and is_char (tail c, chsz - 1)
  else     c = .<>.
       and chsz = 0 end if
end is_char;
```
This macro is invoked in definition 201.

10.2.3 Definition of char_head

char_head returns the first character of a bit sequence whether the delimited character representing the character is valid or not. **char_head (s, chsz)** simply returns all but the delimiters of the first **chsz + 2** bits of the bit sequence **s**. By default, we return the empty sequence for all values of **chsz** less than 0.

⟨*Definition of char_head*⟩[131] ≡
```
function char_head (s, chsz) =
begin
  if chsz >= 0
  then tail (nlast tr!get_first_n (s, chsz + 2))
  else .<>. end if
end char_head;
```
This macro is invoked in definition 201.

10.2.4 Definition of char_tail

char_tail returns all but the first character of a bit sequence (including delimiters) independent of the first character's validity. **char_tail (s, chsz)** removes the first **chsz + 2** bits of the bit sequence **s**. By default, we delete only two bits (the bits representing the delimiters) for all values of **chsz** less than 0.

⟨*Definition of char_tail*⟩[132] ≡
```
function char_tail (s, chsz) =
begin
  if chsz >= 0
  then tr!remove_first_n (s, chsz + 2)
  else tr!remove_first_n (s, 2) end if
end char_tail;
```
This macro is invoked in definition 201.

10.3 Set of even parity characters

cseq also contains the definition of the subset of characters that have even parity. A character is of even parity if and only if the sum of its bits is even.

⟨*Set of even parity characters*⟩[133]M ≡
```
cseq!even_parity_chars (chsz)
```
This macro is invoked in definitions 14, 19, 36, and 38.

⟨*Definition of even_parity_chars*⟩[134]M ≡
```
zf function even_parity_chars(chsz) =
begin
  { c in char_set(chsz) | even(sum(c)) }
end even_parity_chars;
```
This macro is invoked in definitions 199 and 201.

⟨*Definition of even*⟩[135]M ≡
```
function even(i) =
begin
  i mod 2 = 0
end even;
```
This macro is invoked in definitions 199 and 201.

⟨*Definition of sum*⟩[136]M ≡
```
function sum(s) =
  measure len s
begin
  if null s
  then 0
  else head(s) + sum(tail s) end if
end sum;
```
This macro is invoked in definitions 199 and 201.

Chapter 11

Character Storage Module

The procedural approach to modeling CSP sequential processes using EVES requires using only executable SVerdi constructs. Although traces are useful for storing characters in the specifications about `Rptr`, they cannot be used in SVerdi procedure definitions since they are not executable. This section characterizes a module called **bfr** that includes a set of operations on a data type called **buffer** used to store sequences of characters.

⟨*Buffer type*⟩[137]M ≡

```
bfr!buffer
```

This macro is invoked in definitions 56, 57, 58, 62, 86, 87, 87, 88, 88, 176, 178, 181, 182, 193, and 193.

⟨*Definition stub for buffer type*⟩[138]M ≡

```
type buffer;
```

This macro is invoked in definition 197.

Characters and sequences of characters are represented simply as sequences of values of type `bit`. A function is provided to convert a bit to a one or a zero. Two bits are equal if and only if they are equal after conversion.

⟨*Bit type*⟩[139]M ≡

```
bfr!bit
```

This macro is invoked in definitions 56, 58, 62, 167, 169, 172, 173, 193, and 193.

⟨*Definition stub for bit type*⟩[140]M ≡

```
type bit;
```

This macro is invoked in definition 197.

⟨*Definition stub for cnv_bit*⟩[141]M ≡

```
typed function cnv_bit (b : bit) returns int;
```

This macro is invoked in definition 197.

⟨*Definition for equality of bits*⟩[142]M ≡

```
typed function bit_equal (b1,b2 : bit) returns bool =
  begin
    cnv_bit(b1) = cnv_bit(b2)
  end bit_equal;
```

This macro is invoked in definition 197.

Objects of type buffer are isomorphic to traces of bits in the sense that there is a mapping from buffers to traces of bits that is one-to-one and onto. We call this mapping contents; we call its inverse rebuff. We specify the properties of buffer only at its interface since we are not concerned at this point with an implementation. Of course, any implementation would require that buffers be bounded which would imply a more constrained interface.

⟨*Contents*⟩[143](◇1)M ≡
```
    bfr!contents(◇1)
```
This macro is invoked in definitions 72, 72, 72, 72, 72, 76, 76, 90, 90, 96, 96, 96, 99, 99, 99, 175, 175, 176, 176, 178, 178, 180, 181, and 182.

⟨*Buffer-Trace conversions*⟩[144]M ≡
```
    function contents (buff);

    function rebuff (tr);

    grule contents_is_trace (buff) =
      begin
           typeof(buff) = buffer()
        -> contents (buff) ^*? -{0,1}-
      end contents_is_trace;

    rule rebuff_is_buffer (tr) =
      begin
           tr ^*? -{0,1}-
        -> typeof rebuff(tr) = buffer ()
      end rebuff_is_buffer;

    rule contents_rebuff (tr) =
      begin
           tr ^*? -{0,1}-
        -> contents (rebuff (tr)) = tr
      end contents_rebuff;

    grule cnv_bit_is_0_or_1 (b) =
      begin
           typeof(b) = bit()
        -> cnv_bit(b) in -{0,1}-
      end cnv_bit_is_0_or_1;
```
This macro is invoked in definition 197.

The isomorphic nature of buffers and traces of bits implies that each operation on traces has a counterpart for buffers. As a notational convenience we name each operation on a buffer similar to its counterpart. In particular, operations named by a sequence of alphabetic characters are modified by appending the letter p onto the end, e.g., tailp for the tail of a buffer. Operations named by a special symbol are modified by delimiting the symbol with backward quote marks, e.g., '^' for ^.

⟨*Buffer operations*⟩[145]M ≡
```
    typed function start_bitp () returns bit;
    typed function stop_bitp () returns bit;
    typed function is_emptyp (buff : buffer) returns bool;
    typed function emptyp () returns buffer;
    typed function tackp (b : bit, buff : buffer) returns buffer;
    typed function first_eventp (buff : buffer) returns bit;
    typed function but_firstp (buff : buffer) returns buffer;
    typed function appendp (buff1,buff2 : buffer) returns buffer;
    typed function lengthp (buff : buffer) returns int;
```
This macro is invoked in definition 197.

⟨*Buffer operation abbreviations*⟩[146]M ≡
```
    nilfix startbitp start_bitp;
    nilfix stopbitp stop_bitp;
    delim ",";
    delim ">.'";
    plist "'.<" , >.' tackp emptyp;
    prefix nullp is_emptyp 18;
    infixr "']-'" tackp 24;
    prefix headp first_eventp 20;
    prefix tailp but_firstp 22;
    infix "'^'" appendp 22;
    prefix lenp lengthp 16;
    infix "'='" bit_equal 16;
```
This macro is invoked in definition 197.

The above definitions allow us to define the following special macros.

⟨*Trace of single startbit*⟩[147]M ≡
```
    '.<startbitp>.'
```
This macro is invoked in definition 56.

⟨*b is startbit*⟩[148]M ≡
```
    b '=' startbitp
```
This macro is invoked in definition 56.

⟨*b is stopbit*⟩[149]M ≡
```
    b '=' stopbitp
```
This macro is invoked in definition 62.

⟨*Empty buffer*⟩[150]M ≡
```
    '.<>.'
```
This macro is invoked in definition 86.

⟨*buff is empty*⟩[151]M ≡
```
    nullp buff
```
This macro is invoked in definition 87.

⟨*tail of buff*⟩[152]M ≡
```
    tailp buff
```
This macro is invoked in definitions 87 and 88.

⟨*tail of chr*⟩[153]M ≡
```
    tailp chr
```
This macro is invoked in definitions 62 and 179.

⟨*Append b onto end of chr*⟩[154]M ≡
```
    chr '^' '.<b>.'
```
This macro is invoked in definition 58.

⟨*Append delimited chr onto end of buff*⟩[155]M ≡
```
    buff '^' (startbitp ']-' (chr '^' '.<stopbitp>.'))
```
This macro is invoked in definitions 87 and 88.

⟨*len of buff*⟩[156]M ≡
```
    (lenp buff)
```
This macro is invoked in definition 87.

⟨*len of chr*⟩[157]M ≡
```
    (lenp chr)
```
This macro is invoked in definition 58.

As mentioned there is a one to one correspondence between the operations on buffers and the operations on traces of bits. This correspondence is specified as a sequence of rules that are assumed without proof. These rules are the natural ones expected.

⟨*Buffer operation properties*⟩[158]M ≡
```
    rule startbitp_is_startbit () =
      begin
        cnv_bit(startbitp) = startbit
      end startbitp_is_startbit;

    rule stopbitp_is_stopbit () =
      begin
        cnv_bit(stopbitp) = stopbit
      end stopbitp_is_stopbit;

    rule emptyp_is_empty () =
      begin
        '.<>.' = rebuff(.<>.)
      end emptyp_is_empty;

    rule nullp_is_null (buff) =
      begin
           typeof(buff) = buffer ()
        -> (nullp buff) = null contents(buff)
      end nullp_is_null;

    rule tackp_is_tack (b,buff) =
      begin
               typeof(b) = bit ()
           and typeof(buff) = buffer ()
        -> (b ']-' buff)
            = rebuff(cnv_bit(b) ]- contents(buff))
      end tackp_is_tack;

    rule headp_is_head (buff) =
      begin
```

```
           typeof(buff) = buffer ()
       and not null contents(buff)
    -> cnv_bit (headp buff) = head (contents (buff))
  end headp_is_head;

rule tailp_is_head (buff) =
  begin
           typeof(buff) = buffer ()
       and not null contents(buff)
    -> (tailp buff) = rebuff(tail contents(buff))
  end tailp_is_head;

rule appendp_is_append (buff1,buff2) =
  begin
           typeof(buff1) = buffer ()
       and typeof(buff2) = buffer ()
    -> (buff1 '^' buff2) = rebuff(contents(buff1) ^ contents(buff2))
  end appendp_is_append;

rule lenp_is_len (buff) =
  begin
       typeof(buff) = buffer ()
    -> (lenp buff) = len contents(buff)
  end lenp_is_len;
```
This macro is invoked in definition 197.

Chapter 12

Base Machine Interface

The underlying machine on which we model CSP sequential processes keeps track of the status of relevant communication channels and the values that have been transmitted over those channels. This section describes an SVerdi library unit called **mach** that characterizes the interface to this machine.

The state of a sequential process is characterized by a machine variable of type **state**. The only way this variable can be accessed or modified is by the facilities provided by **mach**. The structure of the **state** data type is hidden from any units using **mach**.

⟨*State type*⟩[159]M ≡

```
    mach!state
```

This macro is invoked in definitions 55, 56, 57, 58, 62, 86, 87, and 88.

⟨*Definition stub for state type*⟩[160]M ≡

```
    type state;
```

This macro is invoked in definitions 193 and 195.

⟨*Event type*⟩[161]M ≡

```
    mach!event
```

This macro is invoked in definitions 55, 56, 57, 58, 62, 86, 87, and 88.

⟨*Definition stub for event type*⟩[162]M ≡

```
    type event;
```

This macro is invoked in definitions 193 and 195.

For specification purposes, **mach** provides access to the current trace of each sequential process given that process's current state.

⟨*Trace*⟩[163](◇1)M ≡

```
    mach!hist(◇1)
```

This macro is invoked in definitions 55, 55, 55, 63, 63, 63, 63, 72, 72, 72, 72, 76, 76, 76, 86, 86, 86, 90, 90, 96, 96, 99, 99, 99, 99, 99, and 99.

⟨*Function returning the current process trace*⟩[164]M ≡

```
    function hist (st);
```

```
grule history_is_trace (st) =
begin
  istrace hist (st)
end history_is_trace;
```
This macro is invoked in definitions 193 and 195.

mach provides facilities to send **bits** over channels. As in the specification, channel identifiers are simply represented as integers. The variable **cntdwn**, which serves as the measure for the loop, represents an upper bound on the number of times the channel is to be polled before the communication is aborted. Note that the invariant for the loop of **snd_bit** is the same as the post-condition for the polling routine. **cntdwn** must decrease each iteration and be bounded below by zero. The **bit** is sent only if the variable **sent** is **true** and **cntdwn** is greater than zero; otherwise no change to the trace is incurred. The primary difference between **snd_bit** and **poll_snd_bit** below is that **snd_bit** guarantees that the **bit** is sent as long as cntdwn does not reach zero. **poll_snd_bit** polls the channel to determine whether a **bit** may be sent and, if so, sends it.

⟨*outbit ! head of buff*⟩[165]M ≡
```
mach!snd_bit(st,outbit,headp buff,cntdwn)
```
This macro is invoked in definition 87.

⟨*snd_bit specification*⟩[166]M ≡
```
initial st_0=st,cntdwn_0=cntdwn
pre cntdwn > 0
post     cntdwn < cntdwn_0
     and cntdwn >= 0
     and (if cntdwn>0 then
               hist(st) = hist(st_0) ^ .<chn.bfr!cnv_bit(b)>.
          else hist(st)=hist(st_0) end if)
```
This macro is invoked in definitions 167 and 193.

⟨*Definition of snd_bit*⟩[167]M ≡
```
procedure snd_bit (mvar st : state,
                   lvar chn : int,
                   lvar b : ⟨Bit type⟩[139],
                   pvar cntdwn : int) =
  ⟨snd_bit specification⟩[166]
  begin
    pvar sent : bool := false
    poll_snd_bit(st,chn,b,sent,cntdwn)
    loop
      invariant     cntdwn < cntdwn_0
                and cntdwn >= 0
                and if sent and cntdwn>0
                    then hist(st) = hist(st_0) ^ .<chn.bfr!cnv_bit(b)>.
                    else hist(st)=hist(st_0) end if
      measure cntdwn
      exit when sent or cntdwn=0
      poll_snd_bit(st,chn,b,sent,cntdwn)
    end loop
  end snd_bit;
```
This macro is invoked in definition 195.

⟨*Poll outbit to send first of buff*⟩[168]M ≡
```
    mach!poll_snd_bit(st,outbit,headp buff,sent,cntdwn)
```
This macro is invoked in definition 88.

⟨*Definition of poll_snd_bit*⟩[169]M ≡
```
    procedure poll_snd_bit (mvar st : state,
                            lvar chn : int,
                            lvar b : ⟨Bit type⟩[139],
                            pvar sent : bool,
                            pvar cntdwn : int) =
      initial st_0=st,cntdwn_0=cntdwn
      pre cntdwn > 0
      post      cntdwn < cntdwn_0
           and cntdwn >= 0
           and if sent and cntdwn > 0
                then hist(st) = hist(st_0) ^ .<chn.bfr!cnv_bit(b)>.
               else hist(st)=hist(st_0) end if;
```
This macro is invoked in definitions 193 and 195.

mach also provides facilities to receive bits. This specification is very similar in structure to that of snd_bit above. The primary difference is that rcv_bit and poll_rcv_bit guarantee that the object received is in fact a bit.

⟨*inbit ? b*⟩[170]M ≡
```
    mach!rcv_bit(st,inbit,b,cntdwn)
```
This macro is invoked in definitions 56, 58, and 62.

⟨*rcv_bit specification*⟩[171]M ≡
```
    initial st_0=st,cntdwn_0=cntdwn
    pre cntdwn > 0
    post     cntdwn < cntdwn_0
         and cntdwn >= 0
         and (if cntdwn > 0
              then hist(st) = hist(st_0) ^ .<chn.bfr!cnv_bit(b)>.
              else hist(st)=hist(st_0) end if)
```
This macro is invoked in definitions 172 and 193.

⟨*Definition of rcv_bit*⟩[172]M ≡
```
    procedure rcv_bit (mvar st : state,
                       lvar chn : int,
                       pvar b : ⟨Bit type⟩[139],
                       pvar cntdwn : int) =
      ⟨rcv_bit specification⟩[171]
      begin
        pvar rcvd : bool := false
        poll_rcv_bit(st,chn,b,rcvd,cntdwn)
        loop
          invariant     cntdwn < cntdwn_0
                    and cntdwn >= 0
                    and if rcvd and cntdwn > 0
                        then hist(st) =  hist(st_0)
                                      ^ .<chn.bfr!cnv_bit(b)>.
```

```
                     else hist(st)=hist(st_0) end if
        measure cntdwn
        exit when rcvd or cntdwn=0
        poll_rcv_bit(st,chn,b,rcvd,cntdwn)
      end loop
    end rcv_bit;
```

This macro is invoked in definition 195.

⟨*Definition of poll_rcv_bit*⟩[173]M ≡

```
  procedure poll_rcv_bit (mvar st : state,
                          lvar chn : int,
                          pvar b : ⟨Bit type⟩[139],
                          pvar rcvd : bool,
                          pvar cntdwn : int) =
    initial st_0=st,cntdwn_0=cntdwn
    pre cntdwn>0
    post     cntdwn < cntdwn_0
         and cntdwn >= 0
         and if rcvd and cntdwn > 0
             then hist(st) = hist(st_0) ^ .<chn.bfr!cnv_bit(b)>.
             else hist(st)=hist(st_0) end if;
```

This macro is invoked in definitions 193 and 195.

mach provides facilities to send and receive characters very similar to those provided to send and receive bits. The primary difference is that, on reception, the character size chsz is passed and the predicate cseq!is_char is guaranteed to hold for the character received.

⟨*mid ? chr*⟩[174]M ≡

```
  mach!rcv_char(st,mid,chr,chsz,cntdwn)
```

This macro is invoked in definition 87.

⟨*rcv_char specification*⟩[175]M ≡

```
  initial st_0=st,cntdwn_0=cntdwn
  pre cntdwn > 0
  post     cntdwn < cntdwn_0
       and cntdwn >= 0
       and (if cntdwn > 0
            then     cseq!is_char(⟨Contents⟩[143](‘chr’),chsz)
                 and hist(st) = hist(st_0) ^ .<chn.⟨Contents⟩[143](‘chr’)>.
            else hist(st)=hist(st_0) end if)
```

This macro is invoked in definitions 176 and 193.

⟨*Definition of rcv_char*⟩[176]M ≡

```
  procedure rcv_char (mvar st : state,
                      lvar chn : int,
                      pvar chr : ⟨Buffer type⟩[137],
                      lvar chsz : int,
                      pvar cntdwn : int) =
    ⟨rcv_char specification⟩[175]
    begin
      pvar rcvd : bool := false
      poll_rcv_char(st,chn,chr,chsz,rcvd,cntdwn)
```

```
   loop
     invariant     cntdwn < cntdwn_0
               and cntdwn >= 0
               and if rcvd and cntdwn > 0
                   then     cseq!is_char(⟨Contents⟩[143](‘chr’),chsz)
                        and hist(st) =  hist(st_0)
                                        ^ .<chn.⟨Contents⟩[143](‘chr’)>.
                   else hist(st)=hist(st_0) end if
     measure cntdwn
     exit when rcvd or cntdwn=0
     poll_rcv_char(st,chn,chr,chsz,rcvd,cntdwn)
   end loop
 end rcv_char;
```

This macro is invoked in definition 195.

⟨Poll mid to receive chr⟩[177]M ≡

```
    mach!poll_rcv_char(st,mid,chr,chsz,rcvd,cntdwn)
```

This macro is invoked in definitions 88 and 88.

⟨Definition of poll_rcv_char⟩[178]M ≡

```
    procedure poll_rcv_char (mvar st : state,
                             lvar chn : int,
                             pvar chr : ⟨Buffer type⟩[137],
                             lvar chsz : int,
                             pvar rcvd : bool,
                            pvar cntdwn : int) =
      initial st_0=st,cntdwn_0=cntdwn
      pre cntdwn>0
      post     cntdwn < cntdwn_0
           and cntdwn >= 0
           and if rcvd and cntdwn > 0
               then     cseq!is_char(⟨Contents⟩[143](‘chr’),chsz)
                    and hist(st) = hist(st_0) ^ .<chn.⟨Contents⟩[143](‘chr’)>.
               else hist(st)=hist(st_0) end if;
```

This macro is invoked in definitions 193 and 195.

⟨mid ! tail of chr⟩[179]M ≡

```
    mach!snd_char(st,mid,⟨tail of chr⟩[153],cntdwn)
```

This macro is invoked in definition 62.

⟨snd_char specification⟩[180]M ≡

```
    initial st_0=st,cntdwn_0=cntdwn
    pre cntdwn > 0
    post     cntdwn < cntdwn_0
         and cntdwn >= 0
         and (if cntdwn>0 then
                  hist(st) = hist(st_0) ^ .<chn.⟨Contents⟩[143](‘chr’)>.
             else hist(st)=hist(st_0) end if)
```

This macro is invoked in definitions 181 and 193.

⟨Definition of snd_char⟩[181]M ≡

```
procedure snd_char (mvar st : state,
                    lvar chn : int,
                    lvar chr : ⟨Buffer type⟩[137],
                    pvar cntdwn : int) =
  ⟨snd_char specification⟩[180]
  begin
    pvar sent : bool := false
    poll_snd_char(st,chn,chr,sent,cntdwn)
    loop
      invariant     cntdwn < cntdwn_0
                and cntdwn >= 0
                and if sent and cntdwn>0
                    then hist(st) = hist(st_0) ^ .<chn.⟨Contents⟩[143](‘chr’)>.
                    else hist(st)=hist(st_0) end if
      measure cntdwn
      exit when sent or cntdwn=0
      poll_snd_char(st,chn,chr,sent,cntdwn)
    end loop
  end snd_char;
```

This macro is invoked in definition 195.

⟨Definition of poll_snd_char⟩[182]M ≡

```
procedure poll_snd_char (mvar st : state,
                         lvar chn : int,
                         lvar chr : ⟨Buffer type⟩[137],
                         pvar sent : bool,
                         pvar cntdwn : int) =
  initial st_0=st,cntdwn_0=cntdwn
  pre cntdwn > 0
  post      cntdwn < cntdwn_0
       and cntdwn >= 0
       and if sent and cntdwn > 0
            then hist(st) = hist(st_0) ^ .<chn.⟨Contents⟩[143](‘chr’)>.
           else hist(st)=hist(st_0) end if;
```

This macro is invoked in definitions 193 and 195.

Chapter 13

Relevant Library Units

13.1 EVES Library

The library distributed with EVES includes the following units that we have used in the repeater refinement. The EVES library is documented fully in [27].

fn — a theory of first–class functions

nat — a theory of the Natural numbers

pair — a theory of ordered pairs and cross products

rel — a theory of binary relations

setrules — axioms about the primitive functions on sets

13.2 CSP Library

The library containing the extensions to EVES to allow specification and verification of CSP processes using the Traces Refinement model contains the following units. This library is documented fully in [19].

nset — a theory of finite sets containing up to five elements

tr — a theory of traces (or sequences) of some object

pr — a theory of a subset of the CSP processes

fpt — a theory of parameterized recursive processes

reqs — a theory of CSP alphabets and trace specifications

13.3 Repeater Application Modules

This section outlines the library units containing the repeater specification, implementation and proof. Because of their size, these units are not included *in their entirety* in this document. Instead, we selected certain functions and rules from each theory to elaborate. FunnelWeb helps us maintain consistency between the functions and rules as they are defined in this document and in the external unit. These units are specified fully in [20].

If the reader encounters a FunnelWeb macro definition whose definition invocation number is higher than any FunnelWeb macro presented in this document (e.g., the reader sees "This macro is invoked in definition 203.", and the highest macro definition number presented here is 191), then that macro definition is invoked in an external unit. The names of the files containing these macros are listed below in the context of aFunnelWeb additive macro. Files ending with "s" contain specification units; files ending with "m" contain model units.

bfr — a buffer data type

⟨*List of external files*⟩[183]**Z** + ≡
  ⟨*bfrs*⟩[197]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

mach — an interface to the base machine

⟨*List of external files*⟩[184]**Z** + ≡
  ⟨*machs*⟩[193]
  ⟨*machm*⟩[195]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

cseq — a theory for reasoning about sequences of bits as sequences of characters

⟨*List of external files*⟩[185]**Z** + ≡
  ⟨*cseqs*⟩[199]
  ⟨*cseqm*⟩[201]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

defs — the definition of the alphabet relevant to the repeater description and some rules for reasoning about the alphabet

⟨*List of external files*⟩[186]**Z** + ≡
  ⟨*defss*⟩[203]
  ⟨*defsm*⟩[205]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

rep — the specification and implementation of the repeater in terms of its two components, `Get` and `Put`

⟨*List of external files*⟩[187]**Z** + ≡
  ⟨*reps*⟩[219]
  ⟨*repm*⟩[221]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

get — the specification and implementation of the repeater's `Get` component

⟨*List of external files*⟩[188]**Z** + ≡
⟨*gets*⟩[207]
⟨*getm*⟩[209]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

put — the specification and implementation of the repeater's `Put` component

⟨*List of external files*⟩[189]**Z** + ≡
⟨*puts*⟩[211]
⟨*putm*⟩[213]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

rptr.fdr — The FDR physical design of `Rptr`.

⟨*List of external files*⟩[190]**Z** + ≡
⟨*fdr*⟩[215]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

rptr.ml — Supporting ML definitions for the FDR physical design of `Rptr`.

⟨*List of external files*⟩[191]**Z** + ≡
⟨*ml*⟩[217]

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is NEVER invoked.

Bibliography

[1] R. Auletta. Rapid-prototyping of high assurance systems. Technical report, George Mason University, Fairfax, Virginia, January 1993.

[2] A. Camilleri. A higher order logic mechanization of the CSP failure divergence semantics. In G. Birtwistle, editor, *Workshops in Computing*, pages 123–150. Springer-Verlag, September 1990.

[3] R. Cleaveland, J. Parrow, and B. Steffen. A semantics-based verification tool for finite-state systems. In *Protocol Specification, Testing, and Verification*. Elsevier Science Publishing Company B.V. (North-Holland), 1990.

[4] D. Craigen. Reference manual for the language verdi. Technical Report TR-91-5429-09a, ORA Canada, Ottawa, Ontario, September 1991.

[5] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M.Saaltink. Reference manual for the language verdi. Technical Report TR-91-5429-09a, ORA Canada, Ottawa, Ontario, September 1991.

[6] Formal Systems (Europe) Ltd. *Failures Divergence Refinement: User Manual and Tutorial*, January 1994.

[7] A. Fraenkel. *Abstract Set Theory*. North Holland, 1968.

[8] D. Good, R. Akers, and L. Smith. Report on gypsy 2.05. Technical Report 1-b, Computational Logic, Incorporated, Austin, Texas, January 1989.

[9] C. A. R. Hoare and J. C. Shepherdson, editors. *Mathematical Logic and Programming Languages*. Prentice-Hall, 1985.

[10] Donald E. Knuth. The **web** system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.

[11] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2), May 1984.

[12] S. Kromodimoeljo and B. Pase. Development of a skeletal CSP theory in EVES. Technical Report TR-92-5469-02, ORA Canada, Ottawa, Ontario, July 1992.

[13] S. Kromodimoeljo, B. Pase, M.Saaltink, D. Craigen, and I. Meisels. EVES: An overview. Technical report, ORA Canada, Ottawa, Ontario, February 1993.

[14] C. Landwehr and J. Carroll. Hardware requirements for secure computer systems: A framework. In *Symposium on Security and Privacy*. IEEE, 1984.

[15] Carl E. Landwehr. The RS-232 software repeater problem. Cipher Newsletter of the Technical Committee on Security and Privacy, Summer 1989.

[16] I. Meisels. An alternative syntax for verdi. Technical Report TR-94-5478-02, ORA Canada, Ottawa, Ontario, March 1994.

[17] R. Milner. *Communication and Concurrency.* Prentice Hall International (UK) Ltd, 1989.

[18] R. Milner. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[19] A. Moore. The EVES CSP library. Technical Report NRL Technical Memorandum 5540-153:apm, Naval Research Laboratory, Washington, D.C., August 1994.

[20] A. Moore. The RS-232 character repeater specification listing. Technical Report NRL Technical Memorandum 5540-035:apm, Naval Research Laboratory, Washington, D.C., August 1994.

[21] Andrew P. Moore. The specification and verified decomposition of system requirements using CSP. *IEEE Transactions on Software Engineering*, 16(9):932–948, September 1990.

[22] E.R. Olderog and C.A.R. Hoare. Specification oriented semantics for communicating sequential processes. *Acta Inform.*, 23:9–66, 1986.

[23] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Six Papers on Formal Verification.* SRI International, Menlo Park, California 94025-3493, May 1992.

[24] B. Pase and S. Kromodimoeljo. A user's guide to a skeletal CSP theory in EVES. Technical Report TR-92-5469-03, ORA Canada, Ottawa, Ontario, July 1992.

[25] Charles N. Payne, Jr., Andrew P. Moore, and David M. Mihelcic. An experience modeling critical requirements. In *Proc. COMPASS 94*, Gaithersburg, MD, June 1994. IEEE.

[26] A. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of CAR Hoare.* Prentice-Hall International, 1994.

[27] M. Saaltink. The EVES library. Technical Report TR-91-5449-03, ORA Canada, Ottawa, Ontario, August 1991.

[28] E. Wayne Sewell. *Weaving a Program: Literate Programming in* **WEB**. Van Nostrand Reinhold, New York, NY USA, 1989. ISBN 0-442-31946-0.

[29] P. van Eijk, C. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS.* Elsevier Science Publishing Company B.V. (North Holland), Amsterdam, The Netherlands, 1989.

[30] Ross Williams. FunnelWeb user's manual. Technical report, University of Adelaide, Adelaide, South Australia, Australia, May 1992. Available via anonymous ftp to `ftp.adelaide.edu.au` in `/pub/funnelweb`.

[31] William D. Young. Verifiable computer security and hardware: Issues. Technical Report 70, Computational Logic, Inc., Austin, TX, September 1991.

Appendix A

CSP Notation Overview

This appendix summarizes the notation that we use to write CSP specifications and process descriptions. This notation subsumes relevant notation provided by SVerdi in [16], by CSP in [9], and by FDR in [6]. The degree to which we could maintain compatibility was limited by the constraints on user-extensible syntax of SVerdi. Appendix B describes the variations on this notation required by the constraints of the mechanical tools.

In addition to briefly describing the meaning of the fundamental operators, information important for parsing CSP is presented in a tabular format. For each operator presented, its type, relative precedence, and full SVerdi function name, if any, is indicated. The type indicates whether the operator is

nilfix — a constant, e.g., `true`;

prefix — a unary operator occurring before its parameter, e.g., -1;

infix — a binary operator occurring between two parameters, e.g., 1+2;

postfix — a unary operator occurring after its parameter, e.g., 5!;

list — an n-ary operator representing a list, e.g., a pair represented in Verdi as pair!pair(a,b) is represented as -<a, b>-;

plist — an n-ary operator representing a paired list, e.g., a set represented in Verdi as (setadd a (setadd b (setadd c (nullset)))) is represented as $\{a, b, c\}$ and corresponds to the Verdi functions setadd and nullset; and

multifix — a more flexible n-ary operator that allows internal operator symbols to vary, e.g., the concurrent process denoted P [| X |] Q. [1]

The precedence of each operator is given on a scale of 0 to 24 where the higher numbers indicate higher precedence. Finally, the full function name, when written in prefix form, is provided so that the properties that the SVerdi operator inherits from Verdi, if any, are apparent from the Verdi Language Definition [4]. The function name is also needed when performing the `invoke` prover command, so that the parser can easily resolve which (potentially overloaded) operator should be expanded.

A.1 Logical Notation

[1] This type of operator is not supported in SVerdi.

Operator	Type	Verdi name	Precedence
`not`	prefix	`not`	14
`and`	infix	`and`	14
`or`	infix	`or`	12
`->`	infix	`implies`	8
`true`	nilfix	`true`	—
`false`	nilfix	`false`	—
`some`	prefix	`some`	—
`all`	prefix	`all`	—

Table A.1: Logical Notation Operators

Notation	**Meaning**
`not` p	$\neg p$ (see Appendix B for variations)
p `and` q	$p \wedge q$ (see Appendix B for variations)
p `or` q	$p \vee q$ (see Appendix B for variations)
p `->` q	p implies q
`true`	truth
`false`	falsity
`some` $x : p(x)$	there exists an x such that $p(x)$
`all` $x : p(x)$	for every x, $p(x)$

A.2 Integer Function Notation

Keyword	Type	Verdi name	Precedence
`-`	prefix	`-`	24
`*`	infix	`*`	20
`mod`	infix	`mod`	20
`div`	infix	`div`	20
`+`	infix	`+`	18
`-`	infix	`-`	18
`<`	infix	`<`	16
`<=`	infix	`<=`	16
`>`	infix	`>`	16
`>=`	infix	`>=`	16

Table A.2: Integer Function Notation Operators

Notation	**Meaning**
`-` i	the negation of i

$i * j$	the multiplication of i and j
i `mod` j	i integer modulus j; j not equal 0
i `div` j	the integer division of i by j with roundoff towards 0; j not equal 0
$i + j$	the addition of i and j
$i - j$	the subtraction of j from i
$i < j$	i is less than j
$i <= j$	i is less than or equal to j
$i > j$	i is greater than j
$i >= j$	i is greater than or equal to j

A.3 Ordered Pair Notation

Keyword	Type	Verdi name	Precedence
`lft`	prefix	`pair!fst`	20
`rgt`	prefix	`pair!snd`	20
`ispair`	prefix	`pair!is_pair`	18
`><`	infix	`pair!cross`	16
`-<, >-`	list	`pair!pair`	—

Table A.3: Ordered Pair Notation Operators

Notation	**Meaning**
`lft` p	the left element of the pair p
`rgt` p	the right element of the pair p
`ispair` p	p is an ordered pair
$S1$ >< $S2$	the set of pairs formed by pairing elements from set $S1$ with elements from $S2$
-<e, f>-	the pair with left element e and right element f

A.4 Set Notation

Notation	**Meaning**
`unit` a	the singleton set containing a
^^ $S1$	the power set of $S1$, i.e., the set of all subsets of $S1$
++ S	the union of all sets in S

Keyword	Type	Verdi name	Precedence
`unit`	prefix	`unit`	24
`^^`	prefix	`powerset`	24
`++`	prefix	`cup`	24
`adj`	infix	`setadd`	18
`<<`	infix	`subset`	18
`++`	infix	`union`	18
`**`	infix	`inter`	18
`--`	infix	`diff`	18
`//`	infix	`pr!set_div`	18
`in`	infix	`in`	16
−{,}−	list	`nset!two_set`	—
{,}	plist	`setadd, nullset`	—

Table A.4: Set Notation Operators

a `adj` S	a added to set S
$S1$ `<<` $S2$	$S1$ is a subset of $S2$
$S1$ `++` $S2$	the union of $S1$ and $S2$
$S1$ `**` $S2$	the intersection of sets $S1$ and $S2$
$S1$ `--` $S2$	the elements in $S1$ not in $S2$
$S1$ `//` $S2$	the set of elements in $S1$ or $S2$ but not in both
a `in` S	a in set S
$-\{e,f\}-$	the set containing e and f that is restricted to exactly two elements, i.e., no other set operators operate on these restricted sets – this construct is used to ease the proof process
$\{y$ `in` $f(x)$ | $P(x,y)\}$	the set of all values y in $f(x)$ such that $P(x,y)$
$\{g(x,y,z)$ | y,z `in` $f(x)\}$	the set of all values $g(x,y,x)$ such that y,z is in $f(x)$
e | $P(e)$	choose an element e such that $P(e)$, if one exists

A.5 Higher-Order Function Notation

Keyword	Type	Verdi name	Precedence
`<-`	infix	`fn!apply`	20
`dom`	prefix	`rel!dom`	20
`ran`	prefix	`rel!ran`	20

Table A.5: Higher-Order Function Notation Operators

Notation **Meaning**

f `<-` x	function f evaluated at x
`dom` f	the domain of function f
`ran` f	the range of function f

A.6 Trace Notation

Keyword	Type	Verdi name	Precedence
`]-`	infix	`tr!tack`	24
`iscomm`	prefix	`pr!is_comm`	24
`chan`	prefix	`pr!channel`	24
`msg`	prefix	`pr!message`	24
`tail`	prefix	`tr!tl`	22
`nlast`	prefix	`tr!but_last`	22
`^`	infix	`tr!append`	22
`|^`	infix	`tr!restrict`	22
`|^^`	infix	`pr!set_restrict`	22
`|=`	infix	`pr!vals`	22
`head`	prefix	`tr!hd`	20
`last`	prefix	`tr!last_event`	20
`^*`	postfix	`tr!trace_of`	20
`^*?`	infix	`tr!is_trace_of`	20
`istrace`	prefix	`tr!is_trace`	18
`null`	prefix	`tr!is_empty`	18
`-[`	infix	`tr!occurs`	18
`.<=.`	infix	`tr!subseq`	18
`len`	prefix	`tr!length`	16
`.<, >.`	plist	`tr!tack`, `tr!empty`	—

Table A.6: Trace Notation Operators

Notation	**Meaning**
e `]-` t	e tacked onto the front of t
`iscomm` e	e is a communication event
`chan` c	the channel associated with communication event c
`msg` c	the message associated with communication event c
`tail` t	all but the first element of trace t (see Appendix B for variations)
`nlast` t	all but the last element of trace t
$t1$ `^` $t2$	(between traces) $t1$ followed by $t2$ (see Appendix B for variations)
t `|^` A	trace t with elements not in set A removed
TS `|^^` A	the set of traces in TS, each restricted to events in A

`t |= C`	sequence of values sent over channel C in trace t
head t	the first element of trace t (see Appendix B for variations)
last t	the last element of trace t
`A^*`	the set of all traces of events in a (Kleene star)
`t ^*? A`	t **in** `A^*`
istrace t	t is a trace
null t	t is either not a trace or is empty
`e -[ t`	e in trace t
`t1 .<=. t2`	$t1$ is a prefix of $t2$
len t	the length of t
`.<e1, e2, ..., en>.`	the trace with event $e1$ through en in sequence (see Appendix B for variations)

A.7 Process Notation

Keyword	Type	Verdi name	Precedence
STOP	nilfix	None	24
SKIP	nilfix	None	24
`e -> P`	infix	None	24
`C?x -> P(x)`	multifix	None	24
`C!v -> P`	multifix	None	24
`[] x:B @ P(x)`	multifix	None	24
`e -> P [] f -> Q`	—	None	—
`P ; Q`	multifix	None	24
`X = P(X)`	multifix	None	24
`P [| X |] Q`	multifix	None	24
`P || Q`	infix	`pr!parallel(P,Q)`	24
`P \ A`	infix	None	24
`P |?| Q ~ c`	multifix	`pr!compose(P,Q,c)`	24
traces	prefix	`pr!process_traces`	20
`{|c1, c2, ..., cn|}`	list	—	
alpha	prefix	`pr!process_alphabet`	20
isprocess	prefix	`pr!is_process`	20
sat	infix	`reqs!satpr`	20

Table A.7: Process Notation Operators

Notation	Meaning
STOP	do nothing but terminate unsuccessfully with alphabet A

SKIP	do nothing but terminate successfully with alphabet A and termination event c
e -> P	event e then process P
$C?x$ -> $P(x)$	from channel C input value in variable x and then act like P evaluated at x
$C!v$ -> P	on channel C output value v and then act like P
[] $x:B$ @ $P(x)$	from B choose x engage in x then process F evaluated at x
e -> P [] f -> Q	for e not equal to f, an abbreviation for [] x :$\{e, f\}$ -> $F(x)$ where $F(e) = P$ and $F(f) = Q$
P ; Q	P and, if terminated by c, followed by Q
X = $P(X)$	process PROC such that PROC = F(PROC) and αPROC = A
P [| X |] Q	P composed in parallel with Q synchronizing on events in set X.
P || Q	P in parallel with Q synchronizing on events in common to both **alpha** P and **alpha** Q
$P \setminus A$	Process P while hiding the events in set A from external view. Internal transitions occur without synchronization.
P |?| Q ~ c	P parallel with Q with termination event c, hiding internal events
traces P	the traces of process P
$\{\|c1, c2, ..., cn\|\}$	the set of communication events possible with channels $c1$ through cn as defined by the channel declarations
alpha P	the alphabet of process P
isprocess P	P is a CSP process
P **sat** S	process P satisfies specification S

A.8 Miscellaneous Notation

Keyword	Type	Verdi name	Precedence
`mless`	infix	`m<`	16
`typeof`	prefix	`type_of`	16
`=`	infix	`=`	16
`nat`	nilfix	`nat!nat`	—
if *b* **then** *e*1 **else** *e*2 **endif**	infix	`(if b e1 e2)`	—

Table A.8: Miscellaneous Notation Operators

Notation	Meaning
i `mless` *j*	true if 0 <=*i* <=*j*
`typeof` *e*	the set of values corresponding to the type of *e*
*e*1 = *e*2	equality between expressions *e*1 and *e*2 (see Appendix B for variations)
`nat`	the set of Natural numbers
if *b* **then** *e*1 **else** *e*2 **endif**	expression *e*1 if *b* otherwise expression *e*2

Appendix B

Notational Variations

EVES Syntax	FDR Syntax	ML Syntax
p and q	p and q	p andalso q
p or q	p or q	p orelse q
$e1$ = $e2$	$e1$ == $e2$	$e1$ = $e2$
not ($e1$ = $e2$)	$e1$!= $e2$	—
$s1$ ^ $s2$	$s1$ ^ $s2$	$s1$ @ $s2$
head s	head s	hd s
tail s	tail s	tl s
len s	# s	—
.<$v1, v2, ..., vn$>.	$< v1, v2, ..., vn >$	$[v1, v2, ..., vn]$
if b then $e1$ else $e2$ endif	if b then $e1$ else $e2$	if b then $e1$ else $e2$

Appendix C

Notational Comparison with Hoare's CSP

Our CSP Syntax	Hoare's CSP Syntax	Primary Difference(s)
`STOP`	$STOP_A$	Syntax of STOP
`SKIP`	$SKIP_A$	Syntax of SKIP
e `->` P	$e \rightarrow P$	Syntax of prefix operator
$C?x$ `->` $P(x)$	$C?x \rightarrow P(x)$	Syntax of prefix operator
$C!v$ `->` P	$C!v \rightarrow P$	Syntax of prefix operator
`[]` x`:`B `@` $P(x)$	$x : B \rightarrow P(x)$	Syntax of choice operation
		Syntax of prefix operator
e `->` P `[]` f `->` Q	$e \rightarrow P \vert f \rightarrow Q$	Syntax of choice operator
		Syntax of prefix operator
$P\ ;\ Q$	$P\ ;\ Q$	None
$X = P(X)$	$X = P(X)$	None
P `[|` X `|]` Q	None	–
$P \setminus A$	$P \setminus A$	None
$P \vert\vert Q$	$P \vert\vert Q$	None
P `|?|` Q `~` c	$P \vert\vert Q \setminus P{*}{*}Q$	Syntax of internal event hiding

Index