Chapter 5

Verification of the cryptlib Kernel

Wherein a new method for building a secure system is presented.

1. An Analytical Approach to Verification Methods

Having found the traditional methods used to build trusted systems somewhat lacking, we need to determine an alternative which is more suited to the task. The goal is to determine the most suitable means of creating a trustworthy system, one whose design is capable of earning the user's trust, rather than a trusted system, in which the user is required to trust that the designers and evaluation agency got it right, since the user has no real way to determine this for themselves. The previous chapter discussed the conventional approach to this problem which is to apply an analytical advocacy method (propose a formal theory or set of axioms, develop a theory, advocate its use), in place of this we take the highly unconventional approach of applying a mixture of scientific methods (observe the world, propose a model or theory of behaviour, analyse the results) and engineering methods (observe existing solutions, propose better ones, build or develop, analyse the results) to the problem.

To meet this goal we need to go to two very different fields, the field of cognitive psychology to determine how programmers understand programs and the field of software engineering to locate the tools and techniques used to verify the software. By combining knowledge from both of these fields, we can (hopefully) come up with a technique which can be employed by end users to evaluate the system for themselves, making it something which they can trust, rather than something which they are forced to trust. This mirrors real life, in which users base their trust on personal experience and the experiences of others whom they trust. For example at a time when it was very difficult to build a large bridge which wouldn't fall down within a few years, people trusted the Brooklyn Bridge not because someone had formally proven that it wouldn't fall down but because it was quite obviously constructed like an outdoor convenience of advanced structural integrity. More than a hundred years later people still trust it because it's stood for all that time without collapsing, in the same way that people will trust software which has been in active use and hasn't shown any sign of causing problems, regardless of whether it's been formally proven to be secure or not.

Our goal in building a trustworthy system is a twofold one:

1. The user must be able to examine the code and specifications to reassure themselves that they perform the functions expected of them. This requires very careful thought about how to present the work in a manner which users will find both palatable and comprehensible. The success of the assurance argument depends at least as much on presentation as production (possibly more so), so that rigorously produced evidence which is incomprehensible or present in such quantity that it can't be effectively assessed contributes little to assurance and user trust. As the previous chapter showed, current formal methods fail miserably in this regard.
2. The user must be able to use the formal specification to verify that that binary executable they have conforms to the specification. In other words it must be possible to pull the final, finished product out of the system it's running on and use an automatic verification process to check that what's running on the system is performing as the specification says it should, a goal which can be termed "Verification all the way down"[1]. As the previous chapter also showed, current formal methods don't do so well here either.

Similar sentiments have been expressed in a paper which lists a set of requirements for practical formal methods, which include a minimisation of the effort and expertise needed to apply the method, use of a language which developers find easy to use and understand, making formal analysis as automatic as possible, and providing a good suite of support tools [1].

This section will cover the approach used to try and meet these goals, with the rest of the chapter containing the actual details.

[1] This terminology was inspired by the following Stephen Hawking anecdote: An elderly lady confronted Bertrand Russell at the end of his lecture on orbiting planets saying "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise". Russell gave a superior smile before asking what the turtle was standing on. "You're very clever young man, very clever" replied the old woman "but it's turtles all the way down".

1.1. Peer Review as an Evaluation Mechanism

Encouraging examination of the code should provide the same benefits as peer review of journal articles, and has proven to be an effective way of fixing problems. In terms of the number of problems located, simply reading the code (that is, code inspection) is capable of locating many more defects than alternatives such as black box or white box testing. A variety of studies have found it to be several times more effective than other techniques for finding defects [2][3][4][5][6][7]. Although the previous chapter has pointed out the somewhat dubious basis of a number of software engineering practices so that claims made about the particular effectiveness of code review should, as with other practices, be taken with a grain of salt, there exists at least one analysis with broad enough scope and coverage that it avoids the criticisms levelled in the previous chapter [8]. In addition the fact that peer review is a standard practice for any scholarly journal and the earlier discussion of rather dissimilar techniques such as mathematical theorem proving being as much a social as mathematical process indicates that extensive review should be encouraged even for systems which have been otherwise “proven to be secure”. This claim is backed up by empirical evidence such as that provided from the evaluation of the first system which was certified at the Orange Book A1 level, in which the majority of the security problems (covert channels) were discovered not as a result of the very lengthy and laborious formal proving process, but through reviews and code walkthroughs [9]. Peer review also produced good results in the VAX VMM kernel implementation, resulting in the detection and fixing of many problems [10].

This type of review is formally defined as *N*-fold inspection and involves having a number of small teams or individuals examine code or specifications for defects, with results coordinated by a single moderator. *N*-fold inspection is based on the hypothesis that the *N* separate reviewers don’t significantly duplicate each other’s work so that there isn’t a large degree of fault-detection overlap. This is the same methodology which is used in most open source software development, although there it appears to have evolved naturally rather than as a result of any deliberate design process. In the open source world the phenomenon has been assigned the mantra “many eyes make bugs shallow”, although this only applies if the many eyes really are being applied to the code. With the exception of the OpenBSD effort, which has been making deliberate efforts to review the code they distribute, this type of examination seems to occur mostly for code which users have a direct interest in (for example a driver which is needed to make a new DVD player work) rather than for security-relevant code.

In terms of its effectiveness, one study of the *N*-fold inspection process found that, as further parallel inspections were performed (that is, as more individual users or small groups examined the code), the number of faults located increases cumulatively [11]. In one study it was found that while individuals would typically locate around 27% of all faults, with five inspections in parallel it went up to 65% [12]. Unfortunately these percentage figures are of somewhat dubious value since the 100% rate was arbitrarily set as being the number of faults found by 10 parallel inspections. A later experiment used a slightly different methodology which took as a baseline a document written by an experienced software project leader which was preprocessed by having it reviewed by approximately 40 people who found over 70 faults in the specification (this came as a surprise to the original author, who was amazed at their range and severity). The document was then revised and seeded with 99 known faults and subject to another round of *N*-fold inspection by nine teams, who produced a 78% detection rate of the known faults [13].

This study indicated a wide variation in individual team performance, with detection rates ranging from 22% to 50% and with no one fault being found by every team. These results underline the importance of extensive independent peer review, as well as showing how easy it is even for experienced designers to produce specifications with errors, a problem which was expanded on in the previous chapter.

This form of open peer review isn’t even feasible under a number of standard development methodologies for secure systems, which can require measures such as having all development performed in a sensitive compartmented information facility (SCIF), with optional TEMPEST shielding to deter particularly persistent peer reviewers [14]. An even more rigorous approach than this has been proposed which would be even more effective in deterring peer review, since it seems to be structured towards ensuring that no code is ever produced [15]. Although these measures were intended to prevent peer review by the opposition, they do little to inspire public trust in the resulting end product, since it can then require legal action or pressure from

government bodies to reveal what the resulting code really does (as opposed to what the vendor claims it does) [16][17].

1.2. Enabling Peer Review

In contrast to the systems which are designed to make peer review as difficult as possible, the goal of a trustworthy system design is to make it as easy as possible. In order to make peer review (and therefore the ability to detect various classes of faults) easy, we need to structure the code in a manner which makes it easily comprehensible to the typical programmer (although the connection between code comprehension and the ability to find faults has the potential to be yet another "intuitively obvious" but never verified facet of software engineering, there has in fact been a study carried out which found a strong correlation between code comprehension and fault detection [18]).

The standard response to the requirement to make code easily comprehensible is to rattle off a list of rules ("Use meaningful variable names", "Add plenty of comments", "Use structured code", and so on), seasoned to taste with personal preferences ("Use an OO methodology", "Write it in Java", "Document it using *insert name of favourite CASE tool*", and so on). However, instead of basing the code structure on these somewhat arbitrary choices, we can take advantage of the considerable amount of research which has been performed over the last 30 years on the subject of how programmers comprehend code in order to create code of optimum comprehensibility, and therefore code which is ideally suited for peer review. By tuning the code to match the human thought and comprehension process, we both ensure that the chances of any misunderstandings of the code's function and purpose are reduced, and encourage review by third parties by making it easy for them to examine the code. This is a process which needs to be examined from a psychological rather than the traditional software engineering perspective — if we can prove that a spaghetti mess of `goto`'s is logically equivalent to a structured program then why do we need to use structured code? The answer is that humans are better able to understand structured code than spaghetti code, an issue which is examined in more detail further on.

1.3. Selecting an Appropriate Specification Method

The final peer review problem which remains to be solved is the issue of the formal specification. As the previous chapter demonstrated, one almost universal property of formal specification languages is that they are incomprehensible to all but a few cogniscenti (the specification languages used by the two methodologies endorsed by the Orange Book have been described as "difficult to read, the machine language of specification languages" [19]). The end result of this is that the formal specification is never analysed by anyone other than the people who wrote it and possibly the people who were paid to evaluate it. This is exactly the opposite effect of the one desired.

We can address this problem by examining the precise roles of the DTLS and FTLS. The DTLS is meant to be a natural-language form of the specification, however this assumes that the "natural language" being used is English. For most programmers the natural language they use to describe the behaviour of a program is not English but a programming language, usually C. The US Ninth Circuit court has defined C source code as something that is "meant to be read and understood by humans and that can be used to express an idea or a method", something that is "meant for human eyes and understanding" [20], in other words the natural language of programmers. Going beyond the legal definition, psychological studies have shown that even complete non-programmers will spontaneously evolve programming-language-like constructs such as control statements when asked to create descriptions of algorithm-like tasks [21], indicating that this is indeed the natural language for use when communicating information about computer tasks. This means that the DTLS should be written in the programmers natural language (in this case C or a C-like language) rather than the average person's natural language (in this case English).

Studies into the understandability of software documentation have indicated that software developers and maintainers find it easier to understand closely related languages than distantly-related ones [22] so that the use of a C-like specification language will help their ability to comprehend the resulting specification. In addition since we can now choose a specification language which has a well-defined syntax and a well-defined semantics, all the details of the specification must be stated explicitly, so that missing or ambiguous information can be easily identified. In contrast the English specification which is typically used to guide

implementors makes it very difficult to write concisely and without ambiguity, making it necessary to produce a small essay at each step in order to ensure that all readers of the specification interpret it correctly. It is for this reason that formal specification languages are sometimes referred to as error avoidance systems, since they reduce the chances of ambiguity or errors in the specification.

Because an English DTLS can't be applied directly, it first needs to be manually translated into an executable form. This task is "error prone, expensive, time consuming, and contributes little to the standard development process" [23]. There has been a limited amount of experimental work in applying natural language processing (NLP) techniques to English specifications, but the results have been less than spectacular [24][25] and the case has been made that this approach represents, at best, a dangerous illusion since natural language is simply incapable of expressing precisely the exact semantics of a system even if the NLP problem is finally satisfactorily solved [26]. Making the specification directly executable through the use of a C-like specification language avoids this problem, and has the additional benefit that formal reasoning about and mechanical verification of the code to the specification is now possible. It has even been suggested that, since an implementation is the definitive specification of a program's behaviour, the source code itself should serve as the ultimate specification, providing a behavioural as well as conceptual specification of its operation [27]. This ensures that it will always be a correct (or at least current) specification (since only the code itself is guaranteed to be maintained and updated once the initial implementation has been completed, which is particularly critical when the implementation is subject to constant revision and change), but has the downside that implementation languages don't as a rule make terribly good specification languages.

Using this approach ties in to the concept of cognitive fit, matching the tools and techniques which are used to the task to be accomplished [28][29]. If we can perform this matching, we can assist in the creation of a consistent mental representation of the problem and its solution. In contrast if a mismatch occurs between the representation and the solution then the person examining the code has to first transform it into a fitting representation before they can apply it to the task at hand, or alternatively formulate a mental representation based on the task and then try and work backwards to the actual representation. By matching the formal representation to the representation of the implementation, we can avoid this unnecessary, error-prone, and typically very labour-intensive step. The next logical step below the formal specification then becomes the ultimate specification of the real system, the source code which describes every detail of the implementation and the one from which the executable system is generated.

Ensuring a close match between the specification and implementation raises the spectre of implementation bias, in which the specification unduly influences the final implementation. For example one source comments that "A specification should describe only *what* is required of the system and not *how* it is achieved […] There is no reason to include a *how* in a specification: specifications should describe what is desired and no more" [30]. Empirical studies of the effects of the choice of specification language on the final implementation have shown that the specification language's syntax, semantics, and representation style can heavily influence the resulting implementation [31]. When the specification and implementation languages are closely matched, this presents little problem. When the two bear little relation to each other (SDL's connected FSM's, Estelle's communicating FSM's, or LOTOS' communicating sequential processes, and C or Ada), this is a much bigger problem since the fact that the two have very different semantic domains makes their combined use rather difficult. An additional downside which was mentioned in the previous chapter is that the need to very closely follow a design presented in a language which is unsuited to specifying implementation details results in extremely inefficient implementations since the implementer needs to translate all the quirks and shortcomings of the specification language into the final implementation of the design.

However, it is necessary to distinguish implementation bias (which is bad) from designed requirements (which are good). Specifying the behaviour of a C implementation in a C-like language is fine since this provides strong implementation guidance, and doesn't introduce any arbitrary, specification-language based bias on the implementation since the two are very closely matched. On the other hand forcing an implementation to be based on communicating sequential processes or asynchronously communicating FSMs does constitute a case of specification bias since this is purely an artifact of the specification language and (in most cases) not at all what the implementation actually requires.

1.4. A Unified Specification

Using a programming language for the DTLS means that we can take the process a step further and merge the DTLS with the FTLS, since the two are now more or less identical (it was originally intended that languages like Gypsy also provide this form of functionality). The result of this process is a unified TLS or UTLS. All that remains is to find a C-like formal specification language (as close to the programmer's native language as possible) to write the UTLS in. If we can make the specification executable (or indirectly executable by having one which is usable for some form of mechanical code verification), we gain the additional benefit of having not only a conceptual but also a behavioural model of the system to be implemented, allowing immediate validation of the system by execution [32]. Even users who would otherwise be uncomfortable with formal methods can use the executable specification to verify that the behaviour of the code conforms to the requirements. This use of "stealth formal methods" has been suggested in the past in order to make them more palatable to users [33][34], for example by referring to them as "assertion-based testing" to de-emphasise their formal nature [35].

Both anecdotal evidence from developers who have worked with formal methods [36] and occasional admissions in papers which mention experience with formal methods indicate that the real value of the methods lie in the methodology, the structuring of the requirements and specification for development, rather than the proof steps which follow [37][38][39][40] (it was in recognition of this that early Orange Book drafts contained an entrée[2] class A0 which required an unverified FTLS, but this was later dropped alongside anything more than a discussion of the hypothesised "beyond A1" classes). As was pointed out several times in the previous chapter, the failing of many formal methods is that they can't reach down deep enough into the implementation phase(s) to provide any degree of assurance that what was implemented is what was actually required, however by taking the area where formal methods are strongest (the ability of the formal specification to locate potential errors during the specification phase) and combining it with the area where executable specifications are strongest (the ability to locate errors in the implementation phase), we get the best of both worlds while at the same time avoiding the areas where both are weak.

Another advantage to using specifications which can be verified automatically and mechanically is that it greatly simplifies the task of revalidation, an issue which presents a nasty problem for formal methods as was explained in the previous chapter but which becomes a fairly standard regression testing task when an executable specification is present [41][42]. Unlike standard formal methods which can require that large portions of the proof be redone every time a change is made, the mechanical verification of conformance to a specification is an automated procedure which, while potentially time-consuming for a computer, requires no real user effort. Attempts to implement a revalidation program using Orange Book techniques (the Rating Maintenance Program or RAMP) in contrast have been far less successful, leading to "a plethora of paperwork, checking, bureaucracy and mistrust" being imposed on vendors [43]. This situation arose in part because RAMP required that A1-level configuration control be applied to a revalidation of (for example) a B1 system, with the result that it was easier to redo the B1 evaluation from scratch than to apply A1-level controls to it.

1.5. Enabling Verification All the way Down

The standard way to verify a secure system has been to choose an abstract mathematical modelling method (usually on the basis of being able to find someone on staff who can understand it), repeatedly jiggle and juggle the DTLS until it can be expressed as an FTLS within the chosen mathematical model, prove that it conforms to the requirements, and then hope that functioning code can be magicked into existence based on the DTLS (in theory it should be built from the FTLS, but the implementers won't be able to make head or tail of that).

The approach taken here is entirely different. Instead of choosing a particular methodology and then forcing the system design to fit it, we take the system design and try and locate a methodology which matches it. Since the cryptlib kernel is a filter which acts on messages passing through it, its behaviour can best be expressed in terms of preconditions, postconditions, invariants, and various other properties of the filtering

[2] Given that the Orange Book comes to us from the US, it would probably have been designated an appetizer rather than an entrée.

mechanism. This type of system corresponds directly to the Design by Contract methodology [44][45][46][47].

Design by contract evolved from the concept of defensive programming, a technique created to protect program functions from the slings and arrows of buggy code, and involves the design of software routines which conform to the contract "If you promise to call this routine with precondition *x* satisfied then the routine promises to deliver a final state in which postcondition *x'* is satisfied" [48]. This mirrors real-life contracts which specify the obligations and benefits for both parties. As with real-life contracts, these benefits and obligations are set out in a contract document. The software analogue to a real-life contract is a formal specification which contains preconditions which specify under which conditions a call to a routine is legitimate, and postconditions which specify the conditions which are ensured by the routine on return.

From the discussion in the previous chapters it can be seen that the entire cryptlib kernel implements design-by-contract rules. For example the kernel enforces design-by-contract on key loads into an encryption context by ensuring that certain preconditions hold (the initial access check and pre-dispatch filter which ensures that the caller is allowed to access the context, the context is an encryption context, the key is of the appropriate type and size, the context is in a state in which a key load is possible, etc, etc) and that the corresponding postconditions are fulfilled (the post-dispatch filter which ensures that the context is transitioned into the high state ready for use for encryption or decryption). The same contract-based rules can be built for every other operation performed by the kernel, providing a specification against which the kernel can be validated.

By viewing the kernel as the enforcer of a contract, it moves from being just a chunk of code to the implementation of a certain specification against which it can be tested. The fact that the contract defines what is acceptable behaviour for the kernel introduces the concept of incorrect behaviour or failure, which in the cryptlib kernel's case means the failure to enforce a security condition. Determining whether the contract can be voided in some way by external forces is therefore equivalent to determining whether a security problem exists in the kernel, and this is what gives us the basis for verifying the security of the system. If we can find a way in which we can produce a contract for the kernel which can be tested against the finished executable, we can meet the requirement for verification all the way down.

2. Making the Specification and Implementation Comprehensible

A standard model of the human information processing system known as the Atkinson-Shiffrin model [49][50] which indicates how the system operates when information from the real world passes through it is shown in Figure 1. In the first stage of processing, incoming information about a real-world stimulus arrives in the sensory register and is held there for a brief amount of time (the longer it sits in the register, the more it decays). While the information is in the register, it is subject to a pattern recognition process in which it is matched against previously-acquired knowledge held in long-term memory. This complex interaction results (hopefully) in the new information being equated with a meaningful concept (for example the association of the shape **A** with the first letter of the alphabet), which is then moved into short-term memory (STM).

Data held in STM is held in its processed form rather than in the raw form found in the input register, and may be retained in STM by a process known as rehearsal, which recycles the material over and over through STM. If this rehearsal process isn't performed, the data decays just as it does in the input register. In addition to the time limit, there is also a limit on the number of items which can be held in STM, with the total number of items being around seven [51]. These items don't correspond to any particular unit such as a letter, word, or line of code, but instead correspond to chunks, data recoded into a single unit when it is recognised as representing a meaningful concept [52]. A chunk is therefore a rather variable entity containing more or less information depending on the circumstances[3]. People chunk information into higher-order units using knowledge of both meaning and syntax. Thus for example the C code corresponding to a while look might be chunked by someone familiar with the language into a single unit corresponding to "a `while` loop".

[3] This leads to an amusing circular definition of STM capacity as "STM can contain seven of whatever it is that STM contains seven of".

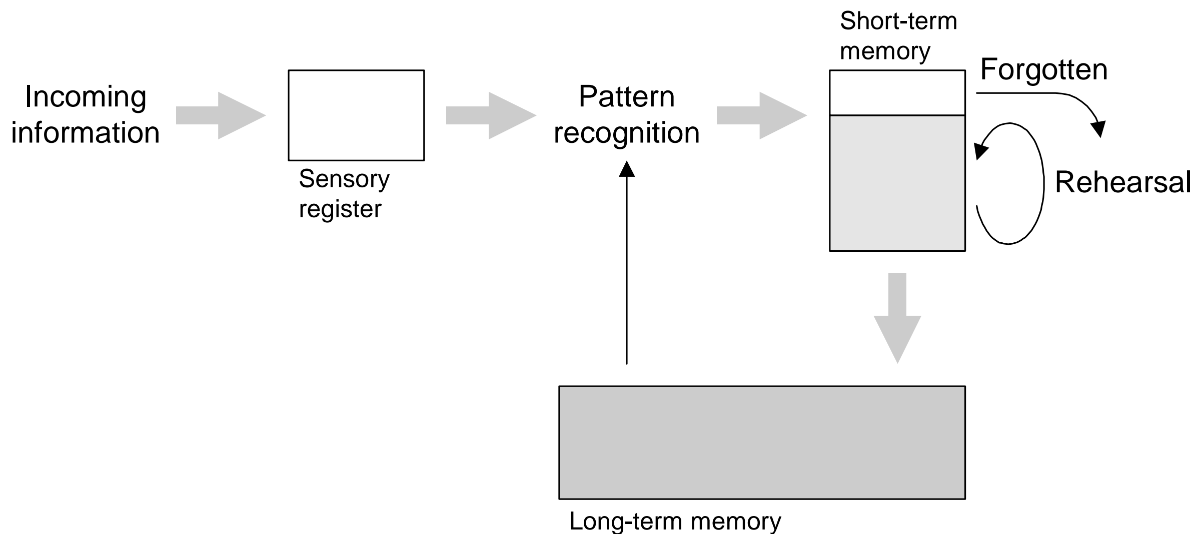

Figure 1: The human memory process

The final element in the process is long-term memory (LTM), into which data can be moved from STM after sufficient rehearsal. LTM is characterised by enormous storage capacity and relatively slow decay [53][54][55].

2.1. Program Cognition

Now that the machinery used in the information acquisition and learning process has been covered, we need to examine how the learning process actually works, and specifically how it works in relation to program cognition. One way of doing this is by treating the cognitive process as a virtual communication channel in which errors are caused not by the presence of external noise but by the inability to correctly decode received information. We can model this by looking at the mental information decoding process as the application of a decoder with limited memory. Moving a step further, we can regard the process of communicating information about the functioning of a program via its source code (or, alternatively, a formal specification) as a standard noisy communications channel, with the noise being caused by the limited amount of memory available to the decoding process. The more working storage (STM) that is consumed, the higher the chances of a decoding error or "decoding noise". The result is a discrepancy between the semantics of the information as received as input and the semantics present in the decoded information.

An additional factor which influences the level of decoding noise is the amount of existing semantic knowledge which is present in LTM. The more information which is present, the easier it is to recover from "decoding noise".

This model may be used to explain the differences in how novices and experts understand programs. Whereas experts can quickly recognise and understand (syntactically correct) code because they have more data present in LTM to mitigate decoding errors, novices have little or no data on LTM to help them in this regard and therefore have more trouble in recognising and understanding the same code. This theory has been supported by experiments in which experts were presented with plan-like code (code which conforms to generally-accepted programming rules, in other words code which contained recognisable elements and structures) and unplan-like code (code which doesn't follow the usual rules of discourse). When faced with unplan-like code, expert programmers performed no better than novices when it came to code comprehension because they weren't able to map the code to any schemas they had in LTM [56].

2.2. How Programmers Understand Code

Having examined the process of cognition in somewhat more detail, we now need to look at exactly how programs are understood by experts (and, with rather more difficulty, by non-experts). Research into program comprehension is based on earlier work in the field of text comprehension, although program comprehension

represents a somewhat specialised case since programs have a dual nature because they can be both executed for effect and read as communications entities. Code and program comprehension by humans involves successive recodings of groups of program statements into successively higher-level semantic structures which are in turn recognised as particular algorithms, and these are in turn organised into a general model of the program as a whole.

One significant way in which this process can be assisted is through the use of clearly structured code which makes use of the scoping rules provided by the programming language. The optimal organisation would appear to be one which contains at its lowest level short, simple code blocks which can be readily absorbed and chunked without overflowing STM and thus leading to an increase in the number of decoding errors [57]. An example of such a code block, taken from the cryptlib kernel, is shown in Figure 2. Note that this code has had the function name/description and comments removed for reasons explained later.

```
function ::=
   PRE( isValidObject( objectHandle ) );

   objectTable[ objectHandle ].referenceCount++;

   POST( objectTable[ objectHandle ].referenceCount == \
         ORIGINAL_VALUE( referenceCount ) + 1 );

   return( CRYPT_OK );
```

Figure 2: Low-level code segment comprehension

The amount of effort required to perform successful chunking is directly related to a program's semantic or cognitive complexity, the "characteristics which make it difficult for humans to comprehend software" [58][59]. The more semantically complex a section of code is, the harder it is to perform the necessary chunking. Examples of semantic complexity which go beyond obvious factors such as the choice of algorithm include the fact that recursive functions are harder to comprehend than non-recursive ones, the fact that linked lists are more difficult to comprehend than arrays, and the use of certain OO techniques which lead to non-linear code which is more difficult to follow than non-OO equivalents [60][61] (so much so that the presence of indicators such as a high use of method invocation and inheritance has been used as a means of identifying fault-prone C++ classes [62][63]).

At this point the reader has achieved understanding of the code segment, which has migrated into LTM in the form of a chunk containing the information "increment an object's reference count". If the same code is encountered in the future, the decoding mechanism can directly convert it into "increment an object's reference count" without the explicit cognition process which was required the first time. Once this internal semantic representation of a program's code has been developed, the knowledge is resistant to forgetting even though individual details may be lost over time [64]. This chunking process has been verified experimentally by evaluating test subjects reading code and retrogressing through code segments (for example to find the `while` at the start of a loop or the `if` at the head of a block of conditional code). Other rescan points included the start of the current function, and the use of common variables, with almost all rescans occurring within the same function [65].

At this point we can answer the rhetorical question which was asked earlier: If we can use the Böhm-Jacopini theorem [66] to prove that a spaghetti mess of `goto`'s is logically equivalent to a structured program then why do we need to use structured code? The reason given previously was that humans are better able to understand structured code than spaghetti code, and the reason that structured code is easier to understand is that large forward or backwards jumps inhibit chunking since they make it difficult to form separate chunks without switching attention across different parts of the program.

We can now step back one level and apply the same process again, this time using previously-understood code segments as our basic building blocks instead of individual lines of code, as shown in Figure 3, again taken from the cryptlib kernel. At this level the cognition process involves the assignment of additional meaning to the higher-level constructs than is present in the raw code, including control flow, transformational effects on data, and the general purpose of the code as a whole.

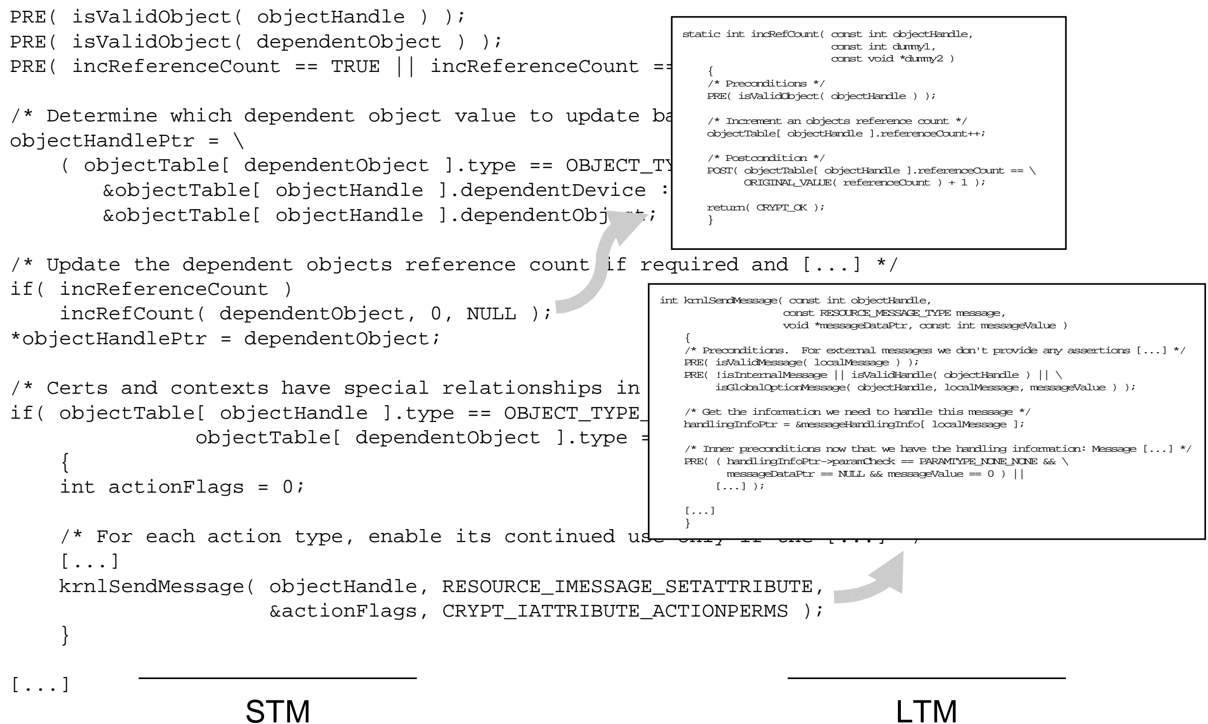

Figure 3: Higher-level program comprehension

Again, the importance of appropriate scoping at the macroscopic level is apparent: If the complexity grows to the point where STM overflows, comprehension problems occur.

A somewhat different view of the code comprehension process is that it is performed through a process of hypothesis testing and refinement in which the meaning of the program is built from the outset by means of features such as function names and code comments. These clues act as "advance organisers", short expository notes which provide the general concepts and ideas which can be used as an aid in assigning meaning to the code [67]. The code section in Figure 2 was deliberately presented earlier without its function name. It is presented again for comparison in Figure 4 with the name and a code comment acting as an advance organiser.

```
/* Increment/decrement the reference count for an object */

static int incRefCount( const int objectHandle )
    {
    PRE( isValidObject( objectHandle ) );

    objectTable[ objectHandle ].referenceCount++;

    POST( objectTable[ objectHandle ].referenceCount == \
          ORIGINAL_VALUE( referenceCount ) + 1 );

    return( CRYPT_OK );
    }
```

Figure 4: Low-level code segment comprehension with the aid of an advance organiser

Related to the concept of advance organisers is that of beacons, stereotyped code sequences which indicate the occurrence of certain operations [68][69]. For example the code sequence '`for i = 1 to 10 do { a[ i ] = 0 }`' is a beacon which the programmer automatically translates to 'initialise data (in this case an array)'.

2.3. Code Layout to Aid Comprehension

Studies of actual programmers have shown that the process of code comprehension is as much a top-down as a bottom-up one. Typically programmers start reading from the beginning of the code using a bottom-up strategy to establish overall structure, however once overall plans are recognised (through the use of chunking, beacons, and advance organisers), they progress to the use of a predictive, top-down mode in which lower levels of detail are skipped if they aren't required in order to obtain a general overview of how the program functions [70][71][72]. The process here is one of hypothesis formation and verification, in which the programmer forms a hypothesis about how a certain section of code functions and only searches down far enough to verify the hypothesis (there are various other models of code comprehension which have been proposed at various times, a survey of some of these can be found elsewhere [73]).

While this type of code examination may be sufficient for program comprehension, when in-depth understanding is required experienced programmers go down to the lower levels to fully understand every nuance of the code's behaviour rather than simply assuming the code works as indicated by documentation or code comments [74]. The reason for this behaviour is that full comprehension is required to support the mental simulation of the code which is used to satisfy the programmer that it does indeed work as required. This is presumably why most class libraries are shipped with source code even though OO theology would indicate that their successful application doesn't require this, since having programmers work with the source code defeats the concept of code reuse which assumes modules will be treated as black-box, reusable components (an alternative view is that since documentation is often inaccurate, ambiguous, or out of date, programmers prefer going directly to the source code which definitively describes its own behaviour).

In order to take advantage of both the top-down and bottom-up modes of program cognition we can use the fact that a program is a procedural text which expresses the actions of the machine on which it is running [75][76]. Although the code is expressed as a linear sequence of statements, what's being expressed is a hierarchy in which each action is linked to one or more underlying actions. By arranging the code so that the lower-level functions occur first in the listing, the bottom-up chunking mode of program cognition is accommodated for programmers who take the listing and read through it from start to finish. For those who prefer to switch to a top-down mode once they understand enough of the program to handle this, the placement of the topmost routines at the opposite end of the listing allows them to be easily located in order to perform a top-down traversal. In contrast, placing the highest-level routines at the start would force bottom-up programmers to traverse the listing backwards, significantly reducing the ease of comprehension for the code. The code layout which results from this design principle is shown in Figure 5. Similar presentation techniques have been used in software exploration and visualisation tools which are designed to aid users in understanding software [77].

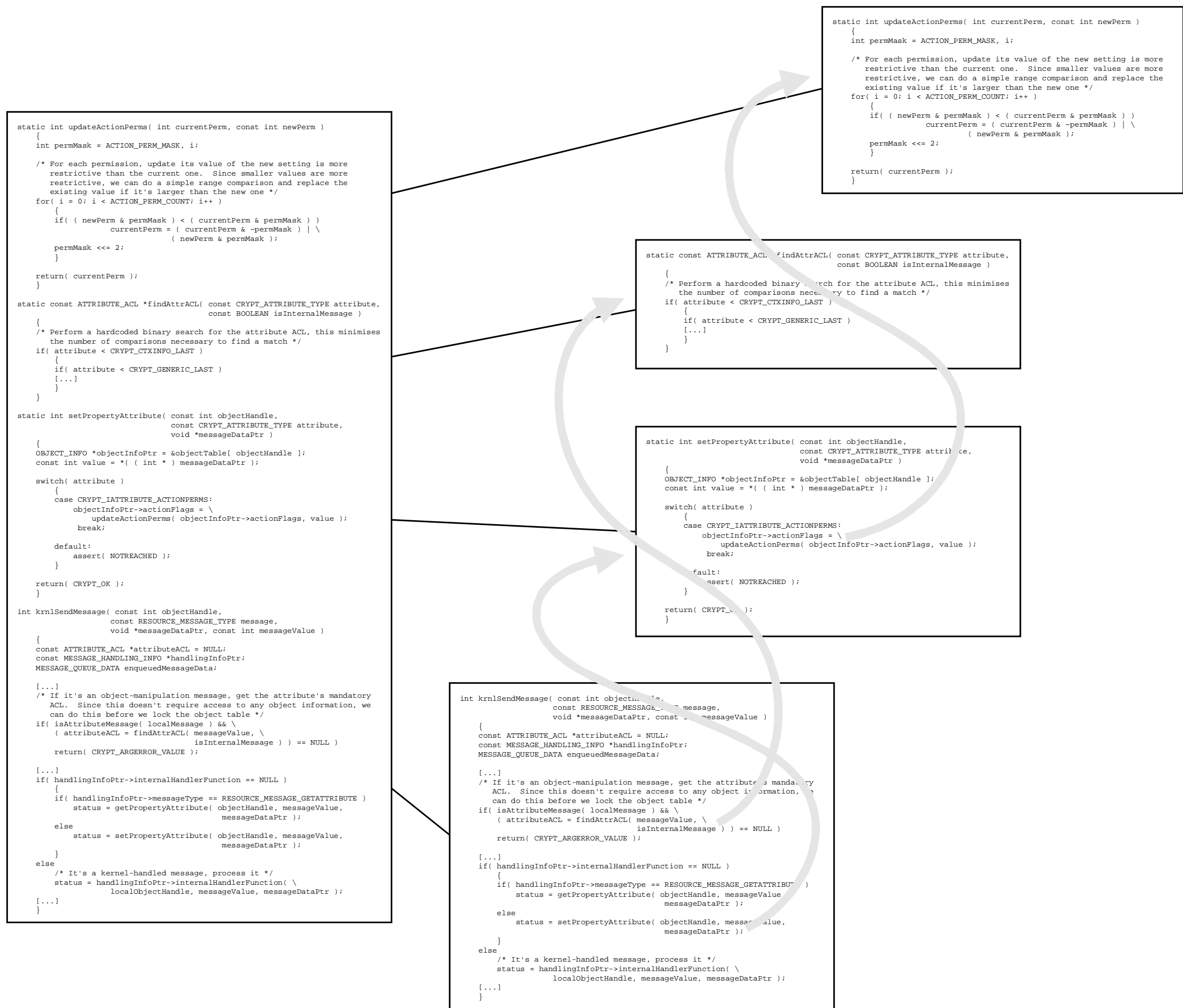

Figure 5: Physical (left) and logical (right) program flow

2.4. Code Creation and Bugs

The process of creating code has been described as one of symbolic execution in which a given plan element triggers the generation of a piece of code which the programmer then symbolically executes in their mind in order to assign an effect to it. The effect is compared to the intended effect and the code modified if necessary in order to achieve the desired result, with results becoming more and more concrete as the design progresses. The creation of sections of code alternates with frequent mental execution to generate the next code section. The coding process itself may be interrupted and changed as a result of these symbolic execution episodes, giving the coding process a sporadic and halting nature [78][79][80][81].

An inability to perform mental simulation of the code during the design process can lead to bugs in the design, since it's no longer possible to progressively refine and improve the design by mentally executing it and making improvements based on the results. The effect of an inability to perform this mental execution is that expert programmers are reduced to the level of novices [82]. This indicates that great care must be exercised in the choice of formal specification language, since most of them don't allow this mental simulation (or only allow it with great difficulty), effectively reducing the ability of its users to that of novice programmers.

The fact that the coding process can cause a trickle-back effect through various levels of refinement indicates that certain implementation aspects such as programming language features must be taken into account when

designing an implementation. For example specifying a program design in a functional language for implementation in a procedural language creates an impedance mismatch which is asking for trouble when it comes to implementing the design. Adhering to the principle of cognitive fit when matching the specification to the implementation is essential in order to avoid these mismatches, which have the potential to lead to a variety of specification/implementation bugs in the resulting code.

The types of problems which can occur due to a lack of cognitive fit can be grouped into two classes, conceptual bugs and teleological bugs, illustrated in Figure 6. Conceptual bugs arise due to differences between the actual program behaviour as implemented and the required behaviour of the program, for example as it is specified in a requirements document. Teleological bugs arise due to differences between the actual program behaviour as implemented and the behaviour intended by the implementer [83][84]. There is often some blurring between the two classes, for example if it is intended that private keys be protected from disclosure but the implementation doesn't do this then it could be due to either a conceptual bug (the program specification doesn't specify this properly) or a teleological bug (the programmer didn't implement it properly).

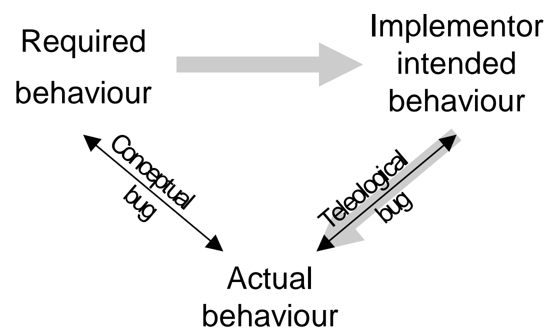

Figure 6: Types of implementation bugs

The purpose of providing a good cognitive fit between the specification and implementation is to minimise conceptual bugs, ones which arise because the implementer had trouble following the specification. Minimising teleological bugs, ones which arise where the programmer had the right intentions but got it wrong, is the task of code verification which is covered in the next section.

2.5. Avoiding Specification/Implementation Bugs

Now that we've looked at the ways in which errors can occur in the implementation, we can examine the ways in which the various design goals and rules presented above act to address them. Before we do this though, we need to extend Figure 6 to include the formal specification for the code, since this represents a second layer at which errors can occur. The complete process from specification to implementation is shown in Figure 7, along with the errors which can occur at each stage (there are also other error paths which exist, for example the actual behaviour not matching the specifier's intended behaviour, but this is just a generalisation of one of the more specific error types shown in Figure 7).

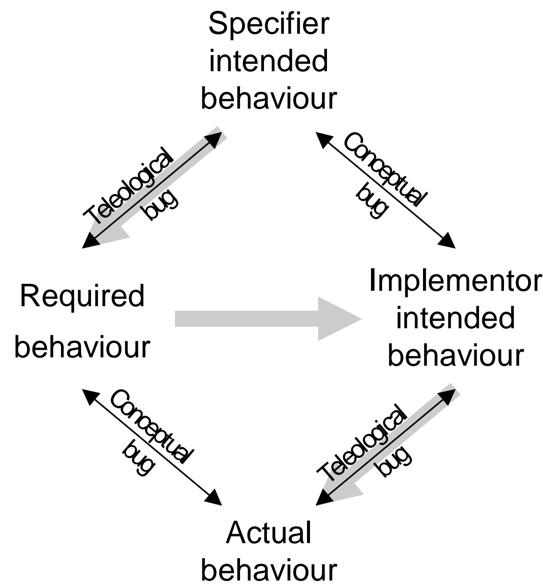

Figure 7: Specification and implementation bug types

Starting from the top, we have conceptual differences between the specifier and the implementor. We act to minimise these by closely matching the implementation language to the specification language, ensuring that the specifier and implementor are working towards the same goal. In addition to the conceptual bugs we have teleological bugs in the specification, which we act to minimise by making the specification language as close to the specifier's natural language (when communicating information about computer operations) as possible.

At the next level, we have teleological bugs between the implementor and the implementation they create, which we act to minimise through the use of automated verification of the specification against the code, ensuring that the behaviour of what's actually implemented matches the behaviour described in the specification. Finally, we have conceptual bugs between what's required and what's actually implemented, which we act to minimise by making the code as accessible and easily comprehensible for peer review as possible.

These error-minimisation goals also interact to work across multiple levels, for example since the specification language closely matches the implementation language the specifier can check that their intent is mirrored in the details of the implementation, allowing checking from the highest down to the lowest level in one single step.

This concludes the coverage of how the cryptlib kernel has been designed to make peer review and analysis as tractable as possible. The next section examines how automated verification is handled.

3. Verification All the way Down

The contract enforced by the cryptlib kernel is shown in Figure 8.

```
ensure that bad things don't happen;
```

Figure 8: The overall contract enforced by the cryptlib kernel

This is something of a tautology, but it provides a basis upon which we can build further refinements. The next level of refinement is to decide what constitutes "bad things" and then itemise them. For example one standard requirement is that encryption keys be protected in some manner (the details of which aren't important at this level of refinement). Our extended-form contract thus takes the form shown in Figure 9.

```
[…]
ensure that keys are protected;
[…]
```

Figure 9: Detail from the overall contract enforced by the kernel

This is still too vague to be useful, but it again provides us with the basis for further refinement. We can now specify how the keys are to be protected, which includes ensuring that they can't be extracted directly from within the architecture's security perimeter, that they can't be misused (for example using a private key intended only for authentication to sign a contract), that they can't be modified (for example truncating a 192-bit key to 40 bits), and various other restrictions. This further level of refinement is shown in Figure 10.

```
[…]
ensure that conventional encryption keys can't be extracted in plaintext form;
ensure that private keys can't be extracted;
ensure that keys can't be used for other than their intended purpose;
ensure that keys can't be modified or altered;
[…]
```

Figure 10: Detail from the key-protection contract enforced by the kernel

The specifications so far have been phrased in terms of expressing when things can't happen, however in practice the kernel works in terms of checking when things are allowed to happen and only allowing them in that instance, defaulting to deny-all rather than allow-all. In order to accommodate this we can rephrase the rules as in Figure 11.

```
[…]
ensure that conventional encryption keys can only be extracted in encrypted form;
ensure that keys can only be used for their intended purpose;
[…]
```

Figure 11: Modified key-protection contract

Note that two of the rules now vanish, since the actions which they were designed to prevent in the Figure 10 version are disallowed by default in the Figure 11 version. The technique of expressing an FTLS as a series of assertions which can be mapped to various levels of the design abstraction has been proposed before for use in verifying a B2 system by translating its FTLS into an assertion list which defines the behaviour of the system which implements the FTLS [85]. The mapping from FTLS was done manually, and seems to have been used more as an analysis technique than as a means of verifying the actual implementation.

We now have a series of rules which determine the behaviour of the kernel. What remains is to determine how to specify them in a manner which is both understandable to programmers and capable of being used to automatically verify the kernel. The most obvious solution to this problem is to use some form of executable specification or, more realistically, a meta-executable specification which can be mechanically mapped onto the kernel implementation and used to verify that it conforms to the specification. The distinction between executable and meta-executable is made because the term "executable specification" is often taken to mean the process of compiling a formal specification language directly into executable code, a rather impractical approach which was covered in the previous chapter.

Some possible approaches to meta-executable specifications are covered in the following sections.

3.1. Programming with Assertions

The simplest way of specifying the behaviour of the kernel is to annotate the existing source code with assertions which check its operation at every step. An assertion is an expression which defines necessary conditions for correct execution, acting as "a tireless auditor which constantly checks for compliance with necessary conditions and complains when the rules are broken" [86]. C's built-in `assert()` macro is a little too primitive to provide anything more than a relatively basic level of checking, however when applied to a design-by-contract implementation its use to verify that the preconditions and postconditions are adhered to can be quite effective. Since the cryptlib kernel was specifically designed to be verifiable using design-by-contract principles, it's possible to go much further with such a simple verification mechanism than would be possible in a more generalised design.

As the previously presented code fragments have indicated, the cryptlib kernel is comprehensively annotated with C assertions which function both to document the contract which applies for each function and to verify

that the contract is being correctly enforced. Even a mechanism as simple as this has helped to catch problems such as an optimiser bug in the gcc compiler which resulted in an object's reference count not being decremented under some circumstances (the author has resisted the temptation to publish a paper in a software engineering journal advocating the universal use of `assert()` based on this successful result).

Moving beyond the built-in assertion capability, there exist a number of extensions which provide the more powerful types of assertions needed for design by contract programming. The simplest of these just extend the basic `assert()` macro to support quantifiers such as ∀ and ∃, provided through the macros `forall()` and `exists()`, and access to the value of a variable at the time a function is called, provided through the macro `old()`. Combined with extensive preprocessor trickery and using special features of the C++ language, it's possible to provide this functionality without requiring any add-on programs or modifications to the C compiler [87].

Going beyond what's possible using the compiler itself were various efforts which looked at extending the concept of basic assertions to the creation of automatic runtime consistency checks. One of the earliest efforts in this area was the work on Anna (Annotated Ada), which uses annotations to Ada source code to perform runtime consistency checking of the executable code [88][89][90]. A derivative of Anna, GNU Nana [91], exists for C++, but has the disadvantage that it is tied heavily into the GNU software tools, being based on preprocessor macros and using language extensions in the gcc compiler and hooking into the gdb debugger. In terms of legibility, Nana-annotated programs have the unfortunate property of appearing to have been hit by a severe bout of line noise.

A slightly different approach is used with App, the Annotation PreProcessor for C, which is implemented as a preprocessor pass which recognises assertions embedded in source code comments and produces as its output (via the C compiler) an executable with built-in checks against the assertions [92]. Since App annotations exist outside the scope of the C code, they don't have to be implemented as preprocessor macros but can instead by handled through a C-like minilanguage which should be instantly understandable by most C programmers and which doesn't suffer from Nana's line-noise problem. App doesn't appear to be publicly available.

Another effort inspired by Anna was A++ (annotated C++), which allowed methods in C++ classes to be annotated with axioms specifying semantic constraints, with the annotations being of the form `[` *`quantifiers;`* `require` *`preconditions;`* `promise` *`postconditions`* `] statement;`. The annotations were to be processed by an A++ front-end which then fed the `statement` part on to the C++ compiler [93]. Work on A++ was abandoned at an early experimental stage so it's not known how verification would have been performed.

All of the mechanisms which rely on annotating program source code, from simple C assertions through to more sophisticated tools such as Anna/Nana, App, and A++ have two common disadvantages: they require modification of the original source code, reducing the comprehensibility of both the code and the annotations by creating a hybrid mix of the two, and they are all-or-nothing in that they can either be enabled and increase the program size and execution time, or be disabled with the result that the code runs without any checking. More seriously, the fact that they are implemented as inline code means that their presence can alter the behaviour of the code (for example by changing the way some compiler optimisations are performed) so that the behaviour of code compiled with the built-in checks differs from that compiled without the checks.

In order to solve these two problems we need to make two changes to the way the specification and verification is performed. Firstly, the specification needs to be written as a separate unit rather than being embedded in the code, and secondly the testing process needs to be non-intrusive so that the code under test doesn't need to be recompiled before or after testing.

3.2. Specification using Assertions

In order to achieve the two goals given above we need to have the ability to compile the specification into a separate piece of executable code which, in conjunction with the code under test forms an oracle which, for any given set of test data, is capable of judging whether the code conforms to the specification. The creation of tools to handle this was inspired by Guttag and Horning's work on the formal specification of the properties of abstract data types which combined a syntactic definition of the data type and a set of axioms

which specified the operations which were allowed on the data [94]. This work was contemporary with early efforts such as SELECT, which used symbolic execution of LISP code and tried to automatically determine appropriate test data (falling back to requesting user input if required) [95] and later lead to tools such as the Data Abstraction, Implementation, Specification, and Testing System (DAISTS) which allowed the specification of classes along with a set of axioms for the abstract data type implemented by each class and test data which checked the implementation against the axioms. The testing was performed by using the algebraic specification as an oracle for testing the implementation, utilising the left-hand side of each axiom as a test case which was compared using a user-supplied equality function to the right-hand side [96]. DAISTS was the first tool which allowed the semantics of an ADT to be specified and verified in the manner outlined earlier, but suffered from the problem that both sides of the equation (the formal specification and the implementation) had to be provided in the DAISTS implementation language SIMPL-D.

Although DAISTS itself appears to have faded from view, it did spawn some later (rather distant) derivatives and adaptations for C++ [97] and Eiffel [98]. The latter, A Set of Tools for Object-Oriented Testing (ASTOOT), is based on the concept of observational equivalence for objects. Two objects are said to be observationally equivalent if, after a sequence of operations on them, they end up in the same abstract state (even if their implementation details differ). A specification can be checked against its implementation by sending them a sequence of operations and then verifying that both end up in the same abstract state. Although this type of testing system is ideal for abstract data structures such as heaps, queues, lists, and trees, the functionality it provides doesn't provide a very good match for the operations performed by the cryptlib kernel.

When creating a specification which contains assertions about the behaviour of an implementation, we need to distinguish between definitional and operational specifications. Definitional specifications describe the properties which an implementation should exhibit, while operational specifications describe how those properties are to be achieved. For example a definitional specification for a sort function might be "upon termination the items are sorted in ascending order", while an operational specification might be a description of a bubble sort, heap sort, merge sort, or quicksort. In its most extreme form an operational specification is a direct implementation of an algorithm in a programming language. The pros and cons of definitional vs operational specifications have been considered earlier, for the cryptlib kernel an operational specification is used.

This introduction now leads us to the use of formal specification languages and assertion-based testing/stealth formal methods, of which the sections which follow provide a representative sample.

3.3. Specification Languages

The usual way to write specifications for a piece of software is in informal English, a DTLS in Orange Book terms. Unfortunately a DTLS has the disadvantage that it is written in a language unsuited for the creation of specifications, one in which it is both easy to create a vague and ambiguous specification, and one which is unusable with automated verifiers. This means that such an informal specification can't be checked for correctness using automated tools, nor can it be processed automatically for input to other tools such as ones which check the program code against the specification. Informal specifications condemn developers to manual verification and testing.

In order to express specifications precisely, an FTLS in Orange Book terms, we need to resort to the use of a formal specification language which is capable of capturing semantic rules and working with a precision not possible with plain English. This can then be passed through a language verifier to check that the content of the specification conforms to the rules, and the result passed on to other tools to conform that the code and/or final program conforms to the specification [99]. Although there has been some debate about the use of executable (or meta-executable) specifications among formal methods purists [100][32][101], we can take the standard criticism of this type of verification, that it can't be used to prove the absence of errors, and reverse it to show that it can at least demonstrate the presence of errors. This is no more or less useful than what model checkers do when they attempt to find counterexamples to security claims about a system, and indeed reported successful applications of model checkers to find faults often emphasise their use in showing the presence of errors in the same manner as more conventional types of testing would [102]. It should be noted here that the validation being performed goes beyond the standard functional-testing approach, which simply

checks that the system works correctly, to also verify that the system doesn't work incorrectly. The overall intent of the validation process then is to accumulate evidence that the implementation matches the specification, something which even a hypothetically perfect formal proof isn't capable of doing.

Another advantage of a formalised rather than descriptive specification is that it makes it rather difficult to fiddle a design decision, since any errors or ambiguities in the designer's thinking will be revealed when an attempt is made to capture it in the form of a formal specification. An example of an ambiguity is the fairly common practice of using the value –1 (or some similar out-of-band value) to indicate a "don't care" value in cases where a handle to an object is required. This practice was used in one location in the cryptlib kernel, but the semantics couldn't be captured in the specification which required that the entity which was present at this point be a cryptlib object and not a choice between an object and a special-case magic value with no significance other than to indicate that it had no significance. Redesigning the portion of the kernel which caused the problem in order to eliminate this ambiguity revealed a somewhat artificial constraint (which admittedly had made sense when the code was originally written) which came through from non-kernel code. Removing this constraint considerably simplified the semantics of the code once the kernel design change was made.

The following sections examine some sample specification languages which could potentially be used for specifying the behaviour of and verifying the cryptlib kernel. In each case a brief overview of a sample from a particular class of language is provided along with an example of how it might be used and an analysis of its applicability to the task at hand. Since many of these languages use an event-based or asynchronously-communicating process model of the world, the example is somewhat contrived in some cases (this also explains many specification language designer's apparent preoccupation with either elevator controllers or stacks when presenting their work, these being examples which fit the language's world view). More extensive surveys of specification languages, including coverage of BagL, Clear, CSP, Larch, PAISLey, Prolog, SEGRAS, SF, Spec, and Z, can be found elsewhere [103][104].

3.4. English-like Specification Languages

One standardised specification language is the Semantic Transfer Language (STL) [105], an English-like language for specifying the behaviour of programs. STL was designed to be a tool-manageable language capable of describing actions, information such as data and relationships among data, events, states, and connection paths. A portion of an STL specification for a left-shift function is shown in Figure 12.

```
[…]

Action leftshift
   is actiontype internal;
   uses dataitem value;
   uses dataitem amount;
   produces dataitem result;
   is tested exhaustively on dataitem value;
   is tested exhaustively on dataitem amount.

Dataitem value is an instance of datatype bitmask.
Dataitem amount is an instance of datatype integer.

Datatype bitmask
   is datatypeclass integer;
   has value range minimum 1;
   has value range maximum 32767;
   has value range resolution 1;
   has invalid subdomain out_of_bounds;
   has valid subdomain as_specified;

[…]
```

Figure 12: Excerpt from an STL specification

As a cursory examination of the sample shows, STL is an extremely expressive language, allowing every nuance of the code's behaviour to be expressed. An equally cursory examination will also indicate that it's a language which makes COBOL look concise by comparison. Note that the specification in Figure 12 still hasn't got to the point of specifying the operation which is being performed

(`result = value << amount` in C), and is also missing a number of supporting lines of specification which are required in order to make the whole thing work.

The corresponding advantage gained from all this verbosity is that it's possible to automatically generate many types of test cases from the specification. An example of a set of test cases generated automatically is given in Table 1, and includes high and low bounds, fencepost (off-by-one) errors, above- and below-bounds errors, and a reference value to make sure everything is working as required.

Subdomain	Equivalence class	Label	Value
invalid	below_bounds	below_bounds	0
valid	as_specified	low_bound	1
valid	as_specified	low_debug	2
valid	as_specified	reference	16384
valid	as_specified	high_debug	32766
valid	as_specified	high_bound	32767
invalid	above_bounds	above_bounds	32768

Table 1: Test data generated from STL specification

Although the automatic-test-case-generation ability is a powerful one, the incredible verbosity (and resulting unreadability due to their size) of STM specifications make it unsuited for use as a specification language for a security kernel, since the huge size of the resulting specification could easily conceal any number of errors or omissions which would never be discovered due to the sheer volume of material which would need to be examined in order to notice them. Other languages which have been designed to look English-like have also ended up with similar problems, for example the CATS specification language was specifically modified to allay the IEEE POSIX community's fears that the pool of potential developers, reviewers and users who could understand a formal specification language if it were used for POSIX specifications would be severely restricted, and ended up being very English-like at the expense of also being very COBOL-like [23].

3.5. Spec

Spec is a formal specification language which bears some resemblance to Pascal and which uses predicate logic to define a piece of code's required behaviour independently of its internal structure [106][107] (the Spec referred to here shouldn't be confused with another specification language of the same name and vaguely the same goals but which uses an incomprehensible mathematical notation [108]). Whereas other specification languages like Larch (see below) are intended for use with automated program-verification tools, Spec is intended more as a design tool for large-scale systems specification and development, specifically for use with event-driven real-time systems. An example Spec specification for the left-shift operation is given in Figure 13. For clarity this doesn't include constraints on the shift amount, which are specified elsewhere, or the ability to shift by more than a single bit position.

```
FUNCTION left_shift { amount: integer } WHERE amount > 0 & amount < 16

MESSAGE ( value : bitmask )
   WHEN value >= 0                      -- Shifting signed values is tricky
      REPLY ( shifted_value : bitmask )
      WHERE shifted_value >= 0 & shifted_value = value * 2
   OTHERWISE
      REPLY EXCEPTION negative_value
END
```

Figure 13: Excerpt from a Spec specification

Spec functional descriptions describe the response of a function to an external stimulus. The intent is that functions described in Spec provide a single service, with the function description containing the stimulus-response characteristics for the function. An incoming message which fits into a particular `when` clause triggers the given response, with the `otherwise` clause giving the response when none of the conditions in

a `when` clause are matched. The `reply` statement provides the actual response sent to the function which provided the original stimulus.

In addition to these basic properties, Spec also has an extensive range of properties and capabilities which are targeted at use with real-time, event-driven systems, as well as support for defining new types, and a facility for defining "machines" which work a bit like classes in object-oriented methodologies.

Although Spec meets the requirements for a programmer's natural language, it has some drawbacks which make it unsuited for use in specifying the cryptlib kernel. As the description above has indicated, Spec is more suited for working with event-driven stimulus-response models than the procedural model used in the cryptlib kernel, which provides something of an impedance mismatch with what's required for the kernel verification since the functions-as-event-handlers model, while it can be adapted to work with cryptlib, isn't really capable of adequately representing the true functionality present in the kernel, while the more sophisticated capabilities such as machines don't match anything in the kernel and aren't required. Another problem with Spec is the lack of any tool support for the language.

3.6. Larch

Larch is a two-tiered specification language with the upper tier consisting of a general-purpose shared language, Larch Shared Language or LSL which provides an implementation language independent specification for the properties of the abstract data types being used, and the lower tier consisting of an interface language which describes the mapping to the actual implementation language. For C, the lower-level language is LCL [109][110].

LSL works with sorts, which are roughly equivalent to data types, and operators, which map one or more input values to an output value. Specifications are presented in terms of traits which define an abstract data type or occasionally just a set of operators which aren't tied to any particular data type. The LSL specification doesn't specify things like the ADT representation, algorithms used to manipulate the ADT, or various exception conditions such as the behaviour when an illegal or out-of-bounds value is encountered. These lower-level details are left to the LCL specification. A portion of the Larch specifications for the shift operation are shown in Figure 14, although in this case the two-tier nature of the language and the fact that the shift operation is far more simplistic than what would usually be specified as a Larch trait make it somewhat artificial. Sitting at a third layer below LCL is the implementation itself, which in this case will be in C and is even more simplistic.

```
Left_shift: trait
   includes Integer
   introduces
      shift: Val, Amt → Val
   asserts ∀ a: Amt, v: Val
      v < INT_MAX ∀ ( a < 16 ∧ a >= 0 );
```

```
int left_shift( int Val, int Amt )
   {
   modifies Val;
   ensures result = ( Val < INT_MAX ∀
         ( Amt < 16 ∧ Amt >= 0 ) ) ∧
         ( Val' = Val << Amt );
   }
```

Figure 14: Excerpt from a Larch specification indicating LSL (left) and LCL (right)

Since Larch specifications can't (with occasional exceptions) be executed, users of LSL are expected to annotate the specification with assertions which can then be verified against the implementation, although some of the tools for this portion of the process are still at a somewhat experimental stage. LCL provides the operators `^` and `'` which can be used to obtain the value of an object (locs in Larch-speak) before and after a procedure. In the example above the `'` operator is being used to indicate the state of the loc after the shift operation has been performed.

As the example indicates, the Larch notation, which at the LSL level uses multi-sorted first-order logic, is far more powerful than the verbose and English-like specification languages which have been discussed so far. Unfortunately, despite it's C-friendliness Larch goes too far towards the nature of the formal specification and proof systems discussed in the previous chapter, requiring a considerable amount of mathematical skill from users with an accompanying steep learning curve as they come to terms with traits, locs, sorts, subgoals and proofs, and all the other paraphernalia which accompanies formal proof tools. As with other provers covered earlier, Larch also requires the use of an interactive proof assistant, the Larch prover (LP) in order to help users reason about conjectures. These problems mean that Larch doesn't meet the requirements given earlier

for understandability and automation. In addition the powerful range of facilities provided by Larch are overkill for our purpose, since a much simpler specification and verification system will also suffice for the task at hand.

3.7. ADL

The assertion definition language ADL is a predicate logic-based specification language which is used to describe the relationship between the inputs and outputs of a program function or module. An ADL specification consists of a set of first-order predicate logic assertions which hold immediately after the completion of a call to a function and which act to constrain the values of the input and output parameters of the function [111][112]. The use of imperative software functions rather than applicative mathematical functions solves one of the major headaches present in many formal methods languages in that software functions can change the state of the computation while mathematical ones can't, avoiding the need to sprinkle the formal specification with hidden functions in the manner described in the previous chapter.

An ADL specification for a function constitutes a formal description of the function's semantics, and usually begins by partitioning the behaviour of the function into normal and abnormal states, identified by the keywords `normal` and `exception` which identify what happens when the function behaves normally and what happens when it encounters an exception condition. For example the behaviour for many Unix system calls, which return –1 on encountering an error, would be characterised with `exception := ( return = –1), normal := !exception`, where `return` is a keyword indicating the return value from the function.

The remainder of the function specification contains a series of assertions which must evaluate to true once the function completes execution. Operators and expressions which are typically used in assertions are the call-state operator `@` which provides the state of a variable at the time the function was called and which is equivalent to the `old` keyword in Eiffel [113], an exception expression `<:>` (implicitly defined in terms of `exception`) which characterises error situations by defining the conditions which cause the function to fail and relating them to the error condition which arises, and the keyword `normally` (implicitly defined in terms of `normal`) which lists the behaviour of the function under non-exception conditions. For example a statement indicating that the function returns –1 (which ADL recognises as an exception condition using the previous definition of `exception`) if a value is nonzero would be given as `value != 0 <:> return = –1`.

There are two types of test conditions which can be derived from ADL specifications, call-state conditions (equivalent to the Eiffel `require` keyword for preconditions), and return-state conditions (equivalent to the Eiffel `ensure` keyword for postconditions). An ADL specification for the shift operation which contains these tests is shown in Figure 15, although this is slightly overspecified (having been chosen to illustrate the features described above) since in real life something as simple as a shift operation would probably be expected to throw an exception on encountering a programmer error rather than returning detailed error codes.

```
int left_shift( int value, int amount )

semantics {
   exception := ( return = -1 ),
   normal := !exception,

   amount < 0 || amount > 16
      <:> return == -1,

   normally {
      value == @value << amount
      }
   }
```

Figure 15: Excerpt from an ADL specification

The code fragment that was used earlier which increments an object's reference count is shown in Figure 16 alongside the corresponding ADL specification (because this is a sample chosen to illustrate an ADL specification and because the concrete C specification only contains a single line of actual code, the size of the abstract specification is about the same as the size of the concrete specification. In practice the former is

much smaller, but this can't be easily illustrated without using an impractically large code example). Both of these specifications say that, when given a valid object, the function will increment its reference count. The ADL version illustrates the use of the call-state operator to obtain the value of a variable when the function is called. In the C version the same effect is achieved through the use of a C preprocessor macro, which also throws an exception if the assertion condition is not met. As the example shows, ADL is close enough in appearance to C that it should be understandable by the typical C programmer after a brief explanation of what ADL is and how it works. Contrast this to more rigorous formal approaches such as Z, where after a week-long intensive course programmers rated a sample Z specification which they were presented with as either hard or impossible to understand [114].

```
int incRefCount( const int objectHandle )
   {
   PRE( isValidObject( objectHandle ) );

   objectTable[ objectHandle ].\
      referenceCount++;

   POST( objectTable[ objectHandle ].\
      referenceCount == \
      ORIGINAL_VALUE( referenceCount ) + 1
      );

   return( CRYPT_OK );
   }
```

```
int incRefCount( const int objectHandle )

semantics {
   exception := cryptStatusError( return ),
   normal := !exception,

   isValidObject( objectHandle )
      <:> return == CRYPT_ARGERROR_OBJECT,

   normally {
      objectTable[ objectHandle ].\
         referenceCount == \
         @objectTable[ objectHandle ].\
         referenceCount + 1,
      return == CRYPT_OK
   }
```

Figure 16: C and ADL specification for object reference count increment

A final ADL operator, which hasn't been required so far, is the implication operator `-->`. In the specification above we could have added a superfluous statement using the predefined function `unchanged` to indicate that `exception --> unchanged( objectTable[ objectHandle ].referenceCount )` but this isn't required since it's already indicated through the call-state test for a valid object.

ADL specifications are written as separate units which are fed into the ADL translator (ADLT) and compiled into an executable form which can be used to verify the implementation against the specification. Because this approach is completely non-intrusive so that there is no need to modify the implementation itself, it allows the code as it is currently running on a system to be verified against the specification, fulfilling the "verification all the way down" requirements. Figure 17 illustrates the process involved in building a test program to verify the C specification for a program (in other words its actual implementation) against its ADL specification. The output of ADLT is C code which is compiled alongside the C implementation code and linked into a single test program which can be run to verify that one matches the other.

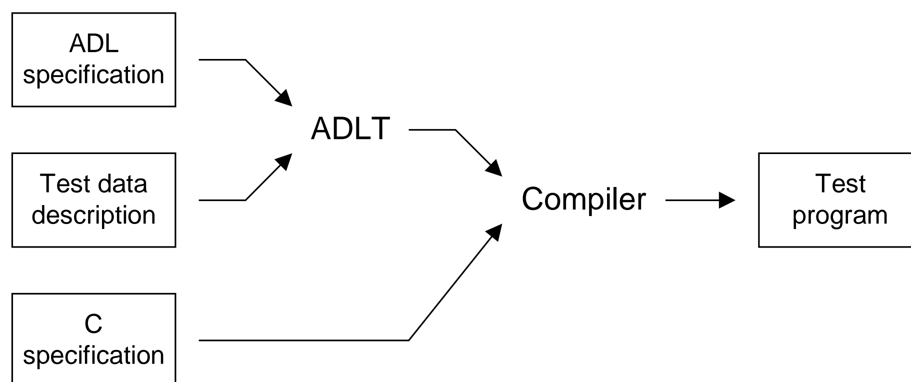

Figure 17: Building a test program from ADL and C specifications

The test program built from this process verifies the functions in the C specification against the semantics specified in the ADL specification by first evaluating all expressions qualified by the call-state operator, calling the function under test with the given test values, evaluating all assertions in the ADL version (using the values saved earlier where appropriate), and reporting an error if any of the assertions evaluate to false.

The process of using an ADL specification to verify an existing binary is shown in Figure 18. In this case the compiled form of the C specification already exists in the form of the executable code which is being run on the system, so the ADL specification is compiled and linked with the existing binary to produce the final test program.

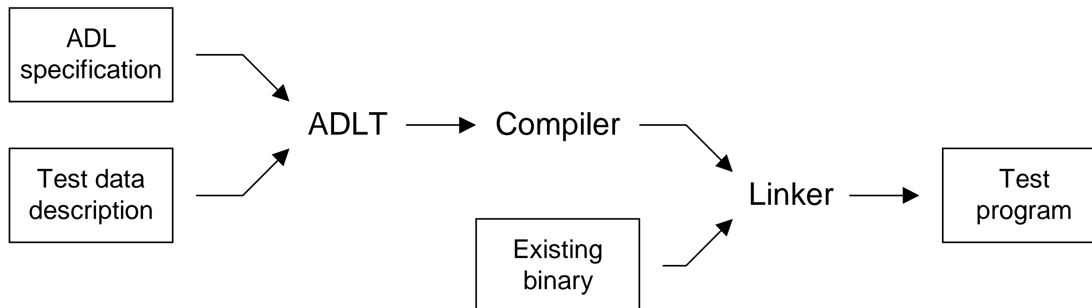

Figure 18: Building a test program for an existing binary

The two cases illustrated above indicate the use of a test data description file, which can either be generated manually or automatically based on the ADL specification. The issue of test data selection is covered in the next section.

An additional facility provided by ADL, which isn't useful for our purposes, is the ability to generate a natural-language document based on annotations in the formal specification. In Orange Book terms this means that it's possible to generate a DTLS based on extra information added to the FTLS. A similar approach has been used for the specification of a software-based RS232 repeater which used Knuth's literate programming techniques to generate EVES and FDR specifications as well as LATEX documentation from a single source file [115]. In our case since the UTLS subsumes the FTLS and DTLS, this extra step isn't necessary, although it could be added if required by third-party evaluators. As with the literate programming approach, ADL provides the capability to mix plain-English annotations with the formal specification. These annotations are then combined by ADLT with information extracted from the specification to produce a plain English version of the specification in troff or HTML format.

3.8. Other Approaches

Various other approaches have also been suggested for specification-based testing which build on the idea that the abstract and concrete implementations can be viewed as different versions of the same software with the hope that their differing form and content will keep common-mode errors to a minimum. Similar ideas exist in the form of *N*-version programming, where a particular error will (it is hoped) be caught by at least one of the *N* independently-developed program versions [116][117][118][4][119]. Note that this approach doesn't attempt to verify the entire implementation as do some formal methods but merely seeks to check it for particular cases, in return for a huge improvement in the success rate of the process and a lowering of the time and skill investment which is needed to obtain results. This type of self-checking implementation can be viewed as a special kind of 2-version programming which has a high degree of design diversity.

One approach to creating a complementary implementation of this kind builds an abstract specification of various ADTs in a Larch-like language and then uses a parallel concrete implementation in C++ with classes containing an additional `abstract` member which contains the abstract form and a concrete-to-abstract mapping function `concr2abstr()` to map the concrete implementation to its abstract form. Member functions of the class are modified to invoke the abstract form of the implementation and then verify that the result conforms to that of the concrete one [120]. In formal-methods terms the abstraction represents a V-function which is modified by an operation, the O-function, to the transformed version of the abstraction. This parallels the modification of the contents of a class instance via a method invocation. The resulting self-checking ADT system is shown in Figure 19.

[4] Readers using these and related references should be aware that there are some ideological differences among researchers involved in *N*-version programming work, which is sometimes reflected in the publications.

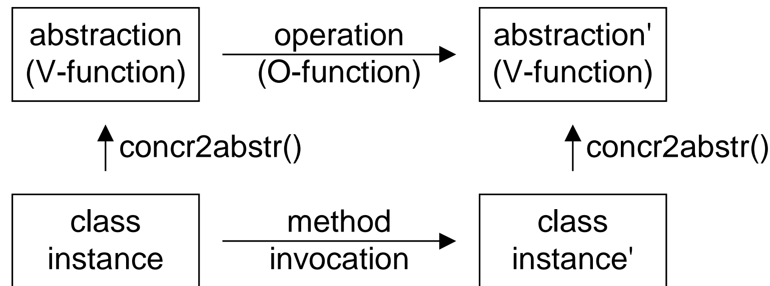

Figure 19: Self-checking ADT implementation

This approach differs from the ADL one in that it requires modification of the source code (although some suggested improvement include the use of a C++-to-C++ preprocessor which inserts the necessary statements into the class implementation and the use of a term rewriting system to help automate the creation of portions of the implementation from the specification). In addition the approach appears to be limited to C++ (rather than straight C) and is somewhat tricky to extend beyond checking of ADTs. A final disadvantage relative to ADL is that, as with the simpler types of assertion-based testing, the checks become embedded in the code, bringing with it the disadvantages already covered in Section 3.1 above.

4. The Verification Process

In order to verify the kernel implementation, we need to perform two types of testing, an inherently top-down form which verifies that the implementation follows the intent of the designer, and an inherently bottom-up form which verifies that the implementation follows the specification. As a previous section indicated, the purpose of this two-tier verification approach is to catch both teleological and conceptual bugs at every level. The inherently top-down testing is intended to ensure that all the design requirements are met, for example that setting certain attributes for an object under appropriate conditions functions as the designer intended. The inherently bottom-up testing is intended to ensure that the implementation corresponds exactly to the specification. This form of testing is generally referred to as specification-based testing. The two forms of testing can be viewed as enforcing the letter of the law (bottom-up or specification-based testing) and the intent of the law (top-down testing).

The top-down verification which checks that the implementation conforms to the designer's intent is relatively straightforward (in fact the kernel performs a core subset of these checks as part of the self-test which is performed to exercise the kernel mechanisms every time it starts up), the bottom-up verification which checks that the implementation complies with the letter of the specification is somewhat more complicated and is covered in the following sections.

4.1. Verification of the Kernel Filter Rules

A previous chapter described the kernel filter mechanism through which the kernel filter rules were implemented, we can now examine how the implementation is verified. Each message type is subject to three types of processing, the general access check which is applied to each message, and a message-type-specific pre- and post-dispatch filter which varies based on message type. Instead of treating the kernel as a monolithic collection of filters and mechanisms, we can decompose it into a number of independent { `general, pre-dispatch, post-dispatch` } triples and then verify each one individually. This decomposition of the complete set of filter rules into a plurality of discrete paths representing different equivalence classes is shown in Figure 20.

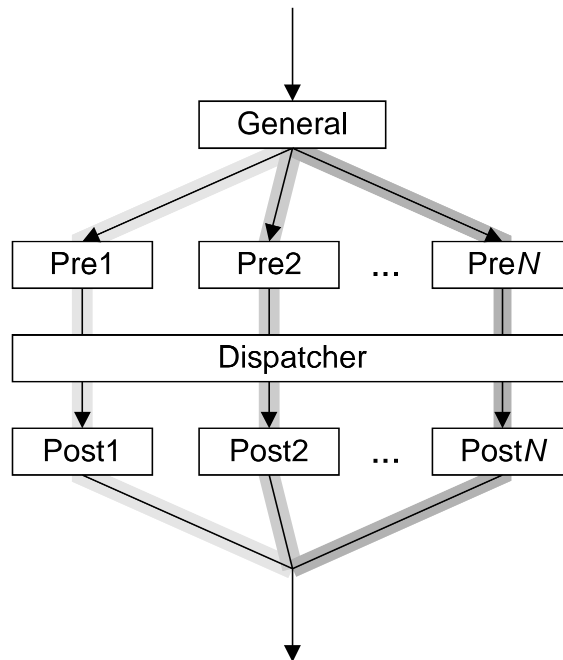

Figure 20: Verification of per-message filter rules

In order to verify that the kernel handles each message correctly, we can verify each path as an independent unit rather than having to verify the kernel as a whole. In many cases there is no post-dispatch filter (the message simply results in a status value being returned) so only the pre-dispatch step needs to be verified.

4.2. Specification-based Testing

An asserted program *p* can be viewed as a sequence of assertions *a*1, *a*2, …, *an* which, when executed on input *i*, transforms it to output *o* while satisfying a single global assertion *A* which is the sum of all the satisfied assertions. We can then say that the program is self-checking with respect to *A* [121]. Since *A* is typically too complex to test as a single postcondition, we break it down into a number of separate assertions *a*1, *a*2, …, *an* which are spread throughout the program as explained earlier. In order to verify the program with respect to the single global meta-assertion *A* we need to determine input data *i* which causes no assertion to be false and which results in the overall meta-assertion holding during the transformation from *i* to *o*.

The traditional functional testing approach is to partition the input domain into equivalence classes and take test data from each class. Each test case consists of an input criterion which describes data which satisfies the test case and an acceptance criterion which describes whether this test case is acceptable or whether it should generate an error. There are a variety of possible selection techniques for test data, including specification-based testing to detect specification-to-implementation mapping errors and oracle-based testing in which the specification acts as an oracle to be violated. Specification-based testing is typically used by selecting test cases which verify that for a given input criterion or assertion the output criterion or assertion is met, and oracle-based testing verifies the opposite. This represents a general overview of formal specification-based testing strategies, in practice there are many variants which can be used [122][123]. The literature on test case generation is at least as extensive as it is for formal methods, and most of the tools appear to be at a similar level of development as their formal methods counterparts.

The testing task is considerably simplified by the strong separation of policy and mechanism which is maintained by the cryptlib kernel. For example instead of specifically verifying that, once a key is loaded into an encryption context, the kernel moves it into the high state, we only need to verify that the mechanism to manage the transitioning from low to high state is functioning as required in order to determine that it will function correctly not only for key loads but also for key generation, certificate signing, and any other

operations which result in an object being transitioned from the low to the high state. This means that the operations performed by the kernel are already pre-partitioned into a set of equivalence classes which correspond to the different filter rules, and issue which was covered in the previous section.

To verify the correctness of a loop, we must verify that it iterates the correct number of times and then stops. The necessary conditions for loop termination are given by the loop variant, a boolean expression which relates variables which are increased or decreased on each iteration. Its predicate is true while the loop is within bounds and false if the loop goes out of control. In terms of specifying a concrete assertion, the loop variant is a restatement of the loop control predicate which contains an integer expression which can be evaluated after each iteration of the loop body, which after each iteration of the body produces a number smaller than at the previous iteration, and which can never go negative. The loop variant for the kernel routing function, along with the function itself, is shown in Figure 21. The magic value 3 is the maximum depth of a hierarchy of connected objects as explained in a previous chapter.

```
/* Route the request through any dependent objects as required until we reach the
   required target object type */
while( object != ε && object.type != target.type )
   {
   /* Try sending the message to the target */
   […]

   /* Loop variant */
   assert( 3 – loop_index > 0 );
   }
```

Figure 21: Loop variant for the kernel routing function

Since the cryptlib kernel is almost entirely loop-free, and the few loops which do exist are guaranteed to terminate after a small, fixed number of iterations (so that they could if necessary be unrolled and expressed as a small number of conditional expressions), verification of this aspect of the code should present no real difficulties.

4.3. Verification with ADL

The formal specification of the behaviour of the cryptlib kernel consists of a set of assertions which constrain the state of the computation being performed. When an assertion evaluates to false during program execution, there exists an incorrect state in the program. This type of full security testing ensures that the implementation both works correctly (corresponding to standard functional testing) and doesn't work incorrectly, a property which doesn't necessarily follow from having it work correctly [124]. In order to test a design-by-contract based program using assertion-based testing, it's necessary to generate test data which violates assertions, preferably automatically, and then check that the behaviour of the implementation corresponds to that specified by the assertions in the formal specification. The problem of finding program input on which an assertion is violated may be reduced to the problem of finding program input on which a selected statement is executed, so that a number of existing methods of test data selection can be applied [125][126].

In the testing processes shown in Figure 17 and Figure 18, the input test data was supplied by the user in the form of a test data description (TDD) specification which was fed to ADLT alongside the ADL specification for the program, from which ADLT generated code to verify the implementation against the specification. The manual creation of TDD specifications is a labour-intensive and error-prone process, and it would be of considerable benefit if this could be done automatically. The earlier discussion of STL indicated that it was possible to specify, in a rather long-winded manner, valid values for various data types defined using STL which allowed the automatic selection of test values to check the handling of conditions such as low and high range checking and off-by-one errors.

It turns out that it's possible to do exactly the same thing in ADL without requiring STL's incredibly verbose and long-winded description of what represents permitted values for variables. For an ADL specification this can be done by examining the call-state and return-state test conditions and creating test data based on the values used in the assertions. For example if an assertion indicated that `val >= 0 && val < 10` then a test data generation tool could use this to choose test values of –1, 0, 5, 9, and 10, corresponding to the earlier STL range checks for `below_bounds`, `low_bound`, `reference`, `high_bound`, and

`above_bounds`. Although this technique has the potential to run into problems with arbitrarily complex expressions in assertions, it is quite practical if a small amount of restraint is exercised by the specifier, so that test conditions are specified as a number of discrete assertions rather than as a single enormous obfuscated test statement.

The choice of values is obtained by walking the ADL parse tree and generating call-state test conditions from call-state evaluatable expressions and return-state test conditions from all evaluatable expressions. In order to test the normal behaviour of a function `exception` must have the value false, which means that all exception assertions must evaluate to false and all `normally` assertions must evaluate to true. In the case of the `incRefCount` function this means that the exception condition for `isValidObject( objectHandle )` must not be invoked on entry (in other words that the function must be passed a valid object) and that the normal execution condition for the reference count increment must occur. In order to test the exception behaviour of a function `exception` must have the value true, which means that, for each exception assertion being tested, all previous exception assertions must evaluate to false. Since `incRefCount` is simple enough that it doesn't contain any exception conditions (that is, provided the precondition holds it will always increment an object's reference count), there is nothing to test in this particular case.

The exact details of how test values can be automatically derived from the ADL specification are covered elsewhere [127]. Once the test data has been derived, it can be used to generate a set of coverage checking functions using the ADLscope tool which augments the test code introduced by ADLT as shown in Figure 17 to produce coverage statistics for the code under test. The resulting coverage information can be used to identify portions of the C specification which require more testing [128].

5. Conclusion

This chapter has presented a new approach to building a trusted system, introducing the concept of an (obviously) trustworthy system rather than a trusted (because we say so) system. The verification methodology which is used to construct this system has been specially designed to instil confidence in the users by allowing them to verify the design specification and implementation themselves through the use of "all the way down" verification. Although this type of verification has long been classed as "beyond A1" (also known as "impossible at the current state of the art"), by carefully matching the verification methodology to the system design it is possible to perform this type of verification in this particular instance. Michael Jackson (the other one) has observed that "It's a good rule of thumb that the value of a method is inversely proportional to its generality. A method for solving all problems can give you very little help with any particular problem" [129]. The method presented here has exactly the opposite properties. Far from trying to be a silver bullet, it constitutes a kryptonite bullet, one which is spectacularly effective against werewolves from Krypton, and not much good against any other kind. However, this doesn't matter to us since all that's important is that it's the right tool for the job. Attacking a werewolf with a Swiss army chainsaw is no more useful, it just make a bigger mess.

6. References

[1] "On the Need for *Practical* Formal Methods", Constance Heitmeyer, *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Real-Time Fault-Tolerant Systems (FTRTFT'98)*, Springer-Verlag Lecture Notes in Computer Science No.1486, September 1998, p.18.

[2] "Practical Software Metrics for Project Management and Process Improvement", Robert Grady, Prentice-Hall, 1992.

[3] "Software Inspection and the Industrial Production of Software", A.Frank Ackerman, Priscilla Fowler, and Robert Ebenau, *Proceedings of the Symposium on Software Validation*, Elsevier Science Publishers, 1984, p.13.

[4] "Software Inspections: An Effective Verification Process", A.Frank Ackerman, Lynne Buchwald, and Frank Lewski, *IEEE Software*, **Vol.6**, **No.3** (May 1989), p.31.

[5] "Handbook of Walkthroughs, Inspections, and Technical Reviews", Daniel Freedman and Gerald Weinberg, Dorset House, 1990.

[6] "Software Inspections: An Industry Best Practice", David Wheeler, Bill Brykczynski, and Reginald Meeson Jr., IEEE Computer Society Press, 1996.

[7] "Software Inspections and the Cost-Effective Production of Reliable Software", A.Frank Ackerman, *Software Engineering*, IEEE Computer Society Press, 1997, p.235.

[8] "National Software Quality Experiment: Results 1992-1996", Don O'Neill, *Proceedings of the 8th Annual Software Technology Conference*, April 1996.

[9] "Analysis of a Kernel Verification", Terry Vickers Benzel, *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1984, p.125.

[10] "A Retrospective on the VAX VMM Security Kernel", Paul Karger, Mary Ellen Zurko, Douglas Bonin, Andrew Mason, and Clifford Kahn, *IEEE Transactions on Software Engineering*, **Vol.17**, **No.11** (November 1991), p.1147.

[11] "The Performance of the *N*-Fold Requirement Inspection Method", Eliezer Kantorowitz, Arie Guttman, and Lior Arzi, *Requirements Engineering*, **Vol.2**, **No.3** (1997), p.152.

[12] "*N*-fold inspection: A requirements analysis technique", Johnny Martin and Wei-Tek Tsai, *Communications of the ACM*, **Vol.33**, **No.2** (February 1990), p.225.

[13] "An Experimental Study of Fault Detection in User Requirements Documents", G.Michael Schneider, Johnny Martin, and W.Tsai, *ACM Transactions on Software Engineering and Methodology*, **Vol.1**, **No.2** (April 1992), p.188.

[14] "Ensuring Software Integrity", Jonathan Weiss and Edward Amoroso, *Proceedings of the 4th Aerospace Computer Security Applications*, December 1988, p.323.

[15] "Toward an Approach to Measuring Software Trust", Ed Amoroso, Thu Nguyen, Jon Weiss, John Watson, Pete Lapiska, and Terry Starr, *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1991, p.198.

[16] "Mondex Blows Users Anonymity", Gavin Clarke and Madeleine Acey, *Network Week*., 25 October 1995.

[17] "Mondex's double life: E-Cash both 'private' and 'fully auditable'", Niall McKay, *Infoworld Canada*, 7 May 1997.

[18] "The role of comprehension in software inspection", A.Dunsmore, M.Roper, and M.Wood, *The Journal of Systems and Software*, **Vol.52**, **No.2/3** (1 June 2000), p.121.

[19] "The Evaluation of Three Specification and Verification Methodologies", Richard Platek, *Proceedings of the 4th Seminar on the DoD Computer Security Initiative* (later the National Computer Security Conference), August 1981, p.X-1.

[20] Bernstein vs USDOJ, U.S. Court of Appeals for the Ninth Circuit, Case Number 97-16686, 6 May 1999.

[21] "What non-programmers know about programming: Natural language procedure specification", Kathleen Galotti and William Ganong III, *International Journal of Man-Machine Studies*, **Vol.22**, **No.1** (January 1985), p.1.

[22] "Estimating Understandability of Software Documents", Kari Laitinen, *ACM SIGSOFT Software Engineering Notes*, **Vol.21**, **No.4** (July 1996), p.81.

[23] "Issues in the Full Scale use of Formal Methods for Automated Testing", J.Crowley, J.Leathrum, and K.Liburdy, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA'96)*, ACM, January 1996, p.71.

[24] "A Translation Method from Natural Language Specifications into Formal Specifications Using Contextual Dependencies", Yasunori Ishihara, Hiroyuki Seki, and Tadao Kasami, *Proceedings of the IEEE International Symposium on Requirements Engineering*, IEEE Computer Society Press, January 1993, p.232.

[25] "Processing Natural Language Software Requirement Specifications", Miles Osborne and Craig MacNish, *Proceedings of the 2nd International Conference on Requirements Engineering (ICRE'96)*, IEEE Computer Society Press, April 1996, p.229.

[26] "The Role of Natural Language in Requirements Engineering", Kevin Ryan, *Proceedings of the IEEE International Symposium on Requirements Engineering*, IEEE Computer Society Press, January 1993, p.240.

[27] "Lessons Learned in an Industrial Software Lab", Albert Endres, *IEEE Software*, **Vol.10**, **No.5** (September 1993), p.58.

[28] "Cognitive Fit: An Empirical Study of Information Acquisition", Iris Vessey and Dennis Galletta, *Information Systems Research*, **Vol.2**, **No.1** (March 1991), p.63.

[29] "Cognitive Fit: An Empirical Study of Recursion and Iteration", Atish Sinha and Iris Vessey, *IEEE Transactions on Software Engineering*, **Vol.18**, **No.5** (May 1992), p.368.

[30] "On the Nature of Bias and Defects in the Software Specification Process", Pablo Straub and Marvin Zelkowitz, *Proceedings of the 16th International Computer Software and Applications Conference (COMPSAC'92)*, IEEE Computer Society Press, September 1992, p.17.

[31] "An Empirical Investigation of the Effects of Formal Specifications on Program Diversity", Thomas McVittie, John Kelly, and Wayne Yamamoto, *Dependable Computing and Fault-Tolerant Systems*, **Vol.6**, Springer-Verlag, 1992, p.219.

[32] "Specifications are (preferably) executable", Norbert Fuchs, *Software Engineering Journal*, **Vol.7**, **No.5** (September 1992), p.323.

[33] "From Formal Methods to Formally Based Methods: An Industrial Experience", *ACM Transactions on Software Engineering and Methodology*, **Vol.8**, **No.1** (January 1999), p.79.

[34] "An Avenue for High Confidence Applications in the 21st Century", Timothy Kremann, William Martin, and Frank Taylor, *Proceedings of the 21st National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1999, CDROM distribution.

[35] "Formal Methods and Testing: Why the State-of-the Art is Not the State-of-the Practice", ISSTA'96/FMSP'96 panel summary, David Rosenblum, *ACM SIGSOFT Software Engineering Notes*, **Vol.21**, **No.4** (July 1996), p.64.

[36] Personal communications with various developers who have worked on A1 and A1-equivalent systems.

[37] "A Case Study of SREM", Paul Scheffer, Albert Stone, and William Rzepka, *IEEE Computer*, **Vol.18**, **No.3** (April 1985), p.47.

[38] "Seven Myths of Formal Methods", Anthony Hall, *IEEE Software*, **Vol.7**, **No.5** (September 1990), p.11.

[39] "Striving for Correctness", Marshall Abrams and Marvin Zelkowitz, *Computers and Security*, **Vol.14**, **No.8** (1995), p.719.

[40] "Symbol Security Condition Considered Harmful", Marvin Schaefer, *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1989, p.20.

[41] "A Method for Revalidating Modified Programs in the Maintenance Phase", S.Yau and Z. Kishimoto, *Proceedings of the 11th International Computer Software and Applications Conference (COMPSAC'87)*, October 1987, p.272.

[42] “Techniques for Selective Revalidation”, Jean Hartman and David Robson, *IEEE Software*, **Vol.7**, **No.1** (January 1990), p.31.

[43] “A New Paradigm for Trusted Systems”, Dorothy Denning, *Proceedings of the New Security Paradigms Workshop ’92*, 1992, p.36.

[44] “Applying ‘Design by Contract’”, Bertrand Meyer, *IEEE Computer*, **Vol.25**, **No.10** (October 1992), p.40.

[45] “iContract — The Java Design by Contract Tool”, Reto Kramer, *Technology of Object-Oriented Languages and Systems*, IEEE Computer Society Press, 1998, p.295.

[46] “The Object Constraint Language”, Jos Warmer and Anneke Kleppe, Addison Wesley, 1999.

[47] “Making Components Contract-aware”, Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins, *IEEE Computer*, **Vol.32**, **No.7** (July 1999), p.38.

[48] “Object-oriented Software Construction”, Bertrand Meyer, Prentice Hall, 1988.

[49] “Human memory: A proposed system and its control processes”, R.Atkinson and R.Shiffrin, *The psychology of learning and motivation: Advances in research and theory, Vol.2*, Academic Press, 1968, p.89.

[50] “The control of short-term memory”, R.Atkinson and R.Shiffrin, *Scientific American*, **No.225** (August 1971), p.82

[51] “Über das Gedächtnis”, Hermann Ebbinghaus, Duncker and Humblot, 1885.

[52] “The magical number seven, plus or minus two: Some limits on our capacity for processing information”, George Miller, *Psychological Review*, **Vol.63**, **No.2** (March 1956), p.81.

[53] “Human Memory: Structures and Processes”, Roberta Klatzky, W.H.Freeman and Company, 1980.

[54] “Learning and Memory”, William Gordon, Brooks/Cole Publishing Company, 1989.

[55] “Human Memory: Theory and Practice”, Alan Baddeley, Allyn and Bacon, 1998.

[56] “Empirical Studies of Programming Knowledge”, Elliot Soloway and Kate Ehrlich, *IEEE Transactions on Software Engineering*, **Vol.10**, **No.5** (September 1984), p.68.

[57] “An integrating common framework for measuring cognitive software complexity”, Zsolt Öry, *Software Engineering Journal*, **Vol.8**, **No.5** (September 1993), p.263.

[58] “Software Psychology: Human Factors in Computers and Information Systems”, Ben Shneiderman, Winthrop Publishers Inc, 1980.

[59] “A study in dimensions of psychological complexity of programs”, B.Chaudhury and H.Sahasrabuddhe, *International Journal of Man-Machine Studies*, **Vol.23**, **No.2** (August 1985), p.113.

[60] “Does OO Sync with How We Think?”, Les Hatton, *IEEE Software*, **Vol.15**, **No.3** (May/June 1998), p.46.

[61] “Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems”, R.Harrison, S.Counsell, and R.Nithi, *The Journal of Systems and Software*, **Vol.52**, **No.2/3** (1 June 2000), p.173.

[62] “Exploring the relationships between design measures and software quality in object-oriented systems”, Lionel Brand, Jürgen Wüst, John Daly, and D.Victor Porter, *The Journal of Systems and Software*, **Vol.51**, **No.3** (1 May 2000), p.245.

[63] “An Empirical Investigation of an Object-Oriented Software System”, Michelle Cartwright and Martin Shepperd, *IEEE Transactions on Software Engineering*, **Vol.26**, **No.8** (August 2000), p.786.

[64] "A effect of semantic complexity on the comprehension of program modules", Barbee Mynatt, *International Journal of Man-Machine Studies*, **Vol.21**, **No.2** (August 1984), p.91.

[65] "Program Comprehension Beyond the Line", Scott Robertson, Erle Davis, Kyoko Okabe, and Douglas Fitz-Randolf, *Proceedings of Human-Computer Interaction — INTERACT'90*, Elseview Science Publishers, 1990, p.959.

[66] "Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules", Corrado Böhm and Guiseppe Jacopini, *Communications of the ACM*, **Vol.9**, **No.5** (May 1966), p.336.

[67] "The Psychology of How Novices Learn Computer Programming", Richard Mayer, *Computing Surveys*, **Vol.13**, **No.1** (March 1981), p.121.

[68] "Towards a theory of the comprehension of computer programs", Ruven Brooks, *International Journal of Man-Machine Studies*, **Vol.8**, **No.6** (June 1983), p.543.

[69] "Beacons in computer program comprehension", Susan Weidenbeck, *International Journal of Man-Machine Studies*, **Vol.25**, **No.6** (December 1986), p.697.

[70] "Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental Results", Ben Shneiderman and Richard Mayer, *International Journal of Computer and Information Sciences*, **Vol.8**, **No.3** (June 1979), p.219.

[71] "Expertise in debugging computer programs: A process analysis", Iris Vessey, *International Journal of Man-Machine Studies*, **Vol.23**, **No.5** (November 1985), p.459.

[72] "Knowledge and Process in the Comprehension of Computer Programs", Elliott Soloway, Beth Adelson, and Kate Erhlich, *The Nature of Expertise*, Lawrence Erlbaum and Associates, 1988, p.129.

[73] "Program Comprehension During Software Maintenance and Evolution", Anneliese von Mayrhauser and A.Marie Vans, *IEEE Computer*, **Vol.28**, **No.8** (August 1995), p.44.

[74] "The Programmer's Burden: Developing Expertise in Programming", Robert Campbell, Norman Brown, and Lia DiBello, *The Psychology of Expertise: Cognitive Research and Empirical AI*, Springer-Verlag, 1992, p.269.

[75] "Advanced Organisers in Computer Instruction Manuals: Are they Effective?", Barbee Mynatt and Katherine Macfarlane, *Proceedings of Human-Computer Interaction (INTERACT'87)*, Elsevier Science Publishers, 1987, p.917.

[76] "Characteristics of the mental representations of novice and expert programmers: an empirical study", Susan Weidenbeck and Vikki Fix, *International Journal of Man-Machine Studies*, **Vol.39**, **No.5** (November 1993), p.793.

[77] "Cognitive design elements to support the construction of a mental model during software exploration", M.-A. Storey, F.Fracchia, and H.Müller, *The Journal of Systems and Software*, **Vol.44**, **No.3** (January 1999), p.171.

[78] "Towards a theory of the cognitive processes in computer programming", Ruven Brooks, *International Journal of Man-Machine Studies*, **Vol.9**, **No.6** (November 1977), p.737.

[79] "Change-Episodes in Coding: When and How Do Programmers Change Their Code?", Wayne Gray and John Anderson, *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing Corporation, 1987, p,185.

[80] "Cognitive Processes in Software Design: Activities in the Early, Upstream Design", Raymonde Guindon, Herb Krasner, and Bill Curtis, *Proceedings of Human-Computer Interaction (INTERACT'87)*, Elsevier Science Publishers, 1987, p.383.

[81] "A Model of Software Design", Beth Adelson and Elliot Soloway, *The Nature of Expertise*, Lawrence Erlbaum and Associates, 1988, p.185.

[82] "Novice-Export Differences in Software Design", B.Adelson, D.Littman, K.Ehrlich, J.Black, and E.Soloway, *Proceedings of Human-Computer Interaction (INTERACT'84)*, Elsevier Science Publishers, 1984, p.473.

[83] "Stereotyped Program Debugging: An Aid for Novice Programmers", Harald Wertz, *International Journal of Man-Machine Studies*, **Vol.16**, **No.4** (May 1982), p.379.

[84] "Expert Programmers Re-establish Intentions When Debugging Another Programmer's Program", Ray Waddington, *Proceedings of Human-Computer Interaction (INTERACT'90)*, Elsevier Science Publishers, 1990, p.965.

[85] "An Assertion Mapping Approach to Software Testing", Greg Bullough, Jim Loomis, and Peter Weiss, *Proceedings of the 13th National Computer Security Conference*, October 1990, p.266.

[86] "Testing Object-Oriented Systems: Models, Patterns, and Tools", Robert Binder, Addison-Wesley, 1999.

[87] "Powerful Assertions in C++", Harald Mueller, *C/C++ User's Journal*, **Vol.12**, **No.10** (October 1994), p.21.

[88] "An Overview of Anna, a Specification Language for Ada", David Luckham and Friedrich von Henke, *IEEE Software*, **Vol.2**, **No.2** (March 1985), p.9.

[89] "A methodology for formal specification and implementation of Ada packages using Anna", Neel Madhav and Walter Mann, *Proceedings of the 1990 Computer Software and Applications Conference*, IEEE Computer Society Press, 1990, p.491.

[90] "Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs", David Luckham, *Texts and Monographs in Computer Science*, Springer-Verlag, January 1991.

[91] "Nana — GNU Project — Free Software Foundation (FSF)", `http://www.gnu.org/software/nana/nana.html`.

[92] "A Practical Approach to Programming With Assertions", David Rosenblum, *IEEE Transactions on Software Engineering*, **Vol.21**, **No.1** (January 1995), p.19.

[93] "The Behavior of C++ Classes", Marshall Cline and Doug Lea, *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*, ACM, September 1990.

[94] "The Algebraic Specification of Abstract Data Types", John Guttag and James Horning, *Acta Informatica*, **Vol.10** (1978), p.27.

[95] "SELECT — A Formal System for Testing and Debugging Programs by Symbolic Execution", Robert Boyer, Bernard Elspas, and Karl Levitt, *ACM SIGPLAN Notices*, **Vol.10**, **No.6** (June 1975), p.234.

[96] "Data-Abstraction Implementation, Specification, and Testing", John Gannon, Paul McMullin, and Richard Hamlet, *ACM Transactions on Programming Languages and Systems*, **Vol.3**, **No.3** (July 1981), p.211.

[97] "Daistish: Systematic Algebraic Testing for 00 Programs in the Presence of Side-effects", Merlin Hughes and David Stotts, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA'96)*, ACM, January 1996, p.53.

[98] "The ASTOOT Approach to Testing Object-Oriented Programs", Roong-Ko Doong and Phyllis Frankl, *ACM Transactions on Software Engineering and Methodology*, Vol.3, No.2 (April 1994), p.101.

[99] "Automating specification-based software testing", Robert Poston, IEEE Computer Society Press, 1996.

[100] "Specifications are not (necessarily) executable", Ian Hayes and Cliff Jones, *Software Engineering Journal*, **Vol.4**, **No.6** (November 1989), p.330.

[101] "Executing formal specifications need not be harmful", Andrew Gravell and Peter Henderson, *Software Engineering Journal*, **Vol.11**, **No.2** (March 1996), p.104.

[102] "Feasibility of Model Checking Software Requirements: A Case Study", Tirumale Sreemani and Joanne Atlee, *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96)*, IEEE Computer Society Press, June 1996, p.77.

[103] "A Practical Assessment of Formal Specification Approaches for Data Abstraction", K.Ventouris and P.Pintelas, *The Journal of Systems and Software*, **Vol.17**, **No.1** (January 1992), p.169.

[104] "Languages for the Specification of Software", Daniel Cooke, Ann Gates, Elif Demirörs, Onur Demirörs, Murat Tanik, and Bernd Krämer, *The Journal of Systems and Software*, **Vol.32**, **No.3** (March 1996), p.269.

[105] "Standard Reference Model for Computing System Engineering Tool Interconnections", IEEE Standard 1175:1992, IEEE, 1992.

[106] "Languages for Specification, Design, and Prototyping", Valdis Berzins and Luqi, *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1990, p.83.

[107] "An Introduction to the Specification Language Spec", Valdis Berzins and Luqi, *IEEE Software*, **Vol.7**, **No.2** (March 1990), p.74.

[108] "Specifying Distributed Systems", Butler Lampson, Working Material for the International Summer School on Constructive Methods in Computing Science, August 1988.

[109] "A Tutorial on Larch and LCL, a Larch/C Interface Language", John Guttag and James Horning, *Proceedings of the 4th International Symposium of VDM Europe (VDM'91), Formal Software Development Methods*, Springer-Verlag Lecture Notes in Computer Science No.552, 1991, p1.

[110] "Larch: Languages and Tools for Formal Specification", John Guttag and James Horning, Springer-Verlag Texts and Monographs in Computer Science, 1993.

[111] "Preliminary Design of ADL/C++ — A Specification Language for C++", Sreenivasa Viswanadha and Sriram Sankar, *Proceedings of the 2nd Conference on Object-Oriented Technology and Systems (COOTS'96)*, Usenix Association, June 1996, p.97.

[112] "Specifying and Testing Software Components using ADL", Sriram Sankar and Roger Hayes, Sun Microsystems Technical Report TR-94-23, Sun Microsystems Laboratories Inc, April 1994.

[113] "Eiffel: The Language", Bertrand Meyer, Prentice-Hall, 1991.

[114] "Literate specifications", C.Johnson, *Software Engineering Journal*, **Vol.11**, **No.4** (July 1996), p.225.

[115] "Increasing Assurance with Literate Programming Techniques", Andrew Moore and Charles Payne Jr., *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96)*, IEEE Computer Society Press, June 1996, p.187.

[116] "Fault Tolerance by Design Diversity: Concepts and Experiments", Algirdas Avizienis and John Kelly, *IEEE Computer*, **Vol.17**, **No.8** (August 1984), p.67.

[117] "The *N*-version Approach to Fault Tolerant Systems", Algirdas Avizienis, *IEEE Transactions on Software Engineering*, **Vol.11**, **No.12** (December 1985), p.1491.

[118] "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study", Nancy Leveson, Stephen Cha, John Knight, and Timothy Shimeall, *IEEE Transactions on Software Engineering*, **Vol.16**, **No.4** (April 1990), p.432.

[119] "N-version design versus one good version", Les Hatton, *IEEE Software*, **Vol.14**, **No.6** (November/December 1997), p.71.

[120] "Automatically Checking an Implementation against Its Formal Specification", Sergio Antoy and Dick Hamlet, *IEEE Transactions on Software Engineering*, **Vol.26**, **No.1** (January 2000), p.55.

[121] "On the Use of Executable Assertions in Structured Programs", Ali Mili, Sihem Guemara, Ali Jaoua, and Paul Torrés, *The Journal of Systems and Software*, **Vol.7**, **No.1** (March 1987), p.15.

[122] "A Method for Test Data Selection", F.Velasco, *The Journal of Systems and Software*, **Vol.7**, **No.2** (June 1987), p.89.

[123] "Approaches to Specification-Based Testing", Debra Richardson, Owen O'Malley, and Cindy Tittle, *Proceedings of the Third ACM SIGSOFT Symposium on Software Testing, Analysis and Verification*, December 1989, p.86.

[124] "Security Testing as an Assurance Mechanism", Susan Walter, *Proceedings of the 3rd Annual Canadian Computer Security Symposium*, May 1991, p.337.

[125] "Predicate-Based Test Generation for Computer Programs", Kuo-Chung Tai, *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*, IEEE Computer Society/ACM Press, May 1993, p.267.

[126] "Assertion-Oriented Automated Test Data Generation", Bogdan Korel and Ali Al-Yami, *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*, IEEE Computer Society Press, 1996, p.71

[127] "Structural Specification-based Testing with ADL", Juei Chang and Debra Richardson, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA'96)*, January 1996, p.62.

[128] "Structural Specification-Based Testing: Automated Support and Experimental Evaluation", Juei Chang and Debra Richardson, *Proceedings of the 7th European Software Engineering Conference (ESE/FSE'99)*, Springer-Verlag Lecture Notes in Computer Science No.1687, November 1999.

[129] "Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices", Addison-Wesley, 1995.