SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2001; **11**:143–163 (DOI: 10.1002/stvr.218)

Automated structural analysis of SCR-style software requirements specifications using PVS

Taeho Kim*,† and Sungdeok Cha

Department of Electrical Engineering and Computer Science, Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology (KAIST), 373-1, Kusong-dong, Yusong-gu, Taejon, Korea

SUMMARY

The importance of effective requirements analysis techniques cannot be overemphasized when developing software requiring high levels of assurance. Requirements analysis can be largely classified as either structural or functional. The former investigates whether definitions and uses of variables and functions are consistent, while the latter addresses whether requirements accurately reflect users' needs. Verification of structural properties for large and complex software requirements is often repetitive, especially if requirements are subject to frequent changes. While inspection has been successfully applied to many industrial applications, the authors found inspection to be ineffective when reviewing requirements to find errors violating structural properties. Moreover, current tools used in requirements engineering provide only limited support in automatically enforcing structural correctness of the requirements. Such experience has motivated research to automate straightforward but tedious activities.

This paper demonstrates that a theorem prover, PVS (Prototype Verification System), is useful in automatically verifying structural correctness of software requirements specifications written in SCR (Software Cost Reduction)-style. Requirements are automatically translated into a semantically equivalent PVS specification. Users need not be experts in formal methods or power users of PVS. Structural properties to be proved are expressed in PVS theorems, and the PVS proof commands are used to carry out the proof automatically. Since these properties are application independent, the same verification procedure can be applied to requirements of various software systems.

KEY WORDS: formal specification; formal verification; requirements analysis; theorem proving

*Correspondence to: Taeho Kim, Department of Electrical Engineering and Computer Science, Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology (KAIST), 373-1, Kusong-dong, Yusong-gu, Taejon, Korea.
†E-mail: thkim@salmosa.kaist.ac.kr

Received 3 November 1998
Accepted 26 September 2000

Published online 26 April 2001

1. INTRODUCTION

Computer-controlled safety-critical systems are increasingly becoming a routine and integral part of modern society. Examples include fly-by-wire commercial jets or computerized emergency shutdown systems for nuclear power plants. Quality assurance requirements for such systems are quite demanding (e.g., requiring failure probabilities as low as 10^{-9}), and national or international regulation bodies routinely require safety demonstrations.

Among the many phases involved in software development, requirements engineering is generally considered to play a crucial role in determining the overall software quality. NASA data [1] indicate that significant portions (e.g., nearly 75%) of failures found in operational software are caused by errors in the software requirements specification (SRS).

Various approaches have been suggested for developing high-quality SRS and conducting cost-effective analysis. Inspection [2] and formal methods are generally shown to be effective [3,4]. Although inspection can, in principle, detect all types of errors in SRS, experience in conducting inspections on the SRS for the Wolsung[‡] SDS2 (shutdown system) [5] revealed that inspection is ineffective when verifying structural properties of large, complex, and evolving software requirements.

Structural properties are application independent and require that definitions and uses of variables, functions, and function groups are consistent. For example, circular dependencies in a specification are generally considered erroneous and are strongly discouraged [6,7]. If any of the external inputs is unused in detailed functional definitions, the SRS is clearly incorrect because either unnecessary inputs are being declared or appropriate functional requirements are missing.

Inspecting the SRS for compliance with structural properties is repetitive and tedious, although straightforward in concept, especially if requirements are subject to frequent changes. Unfortunately, most, if not all, industrial projects exhibit such characteristics. For example, the SRS for the Wolsung SDS2, 374 pages in length, was subject to four relatively minor releases in less than a year. Inspection of the initial release of the SRS, conducted by three staff members, to validate compliance with structural properties took about five hours (therefore, a total of 15 staff hours) of formal inspection meetings[§], during which only five trivial notational errors were discovered.

Data from the Wolsung SDS2 inspection sessions may create an impression that inspection was performed poorly. That is, industrial inspection teams usually consist of five to six people, not three, including an author. Furthermore, the inspection technique is known to detect, on average, one error per staff hour [4]. Inspection of the Wolsung SDS2 detected only about one third of the reported figure. Finally, the inspection rate of reviewing about 70 pages of requirements per hour clearly exceeds the recommended guidelines of a maximum 250 lines of source code per hour.

These numbers alone, however, fail to capture fully and accurately what really went on, and further explanations are in order. (1) Only about half of the 374 pages of the SRS contained information relevant to structural properties. 55 pages of the documentation provide an introduction,

[‡]The Wolsung nuclear power plant in Korea, used as a case study in this paper, is equipped with a software-implemented emergency shutdown system.

[§]Since quantitative measurement of inspection metrics was not one of the primary research objectives, the number of hours spent prior to the formal inspection meeting was not recorded.

notational conventions, cross-references, or appendices. Of the 172 pages of documentation for PDC1 (Programmable Digital Comparator), 66 pages dealt with detailed descriptions of various attributes of monitored and controlled variables. They were not relevant to structural correctness and so were not inspected although cross-references were used when inspecting functional requirements. (2) The Wolsung SDS2 consists of two subsystems, called PDC1 and PDC2, that are similar in style and content. The inspection team reviewed about 35 to 40 pages of software requirements per hour for structural property verification of PDC1; because of the similarities with PDC1, inspection of PDC2 took much less time. (3) Specification for the Wolsung SDS2 was largely derived from requirements for a similar system, in service at Ontario Hydro, whose requirements have been subject to formal and safety analyses [8]. Therefore, the SRS for the Wolsung SDS2 was essentially a reused version of well-documented and extensively analyzed requirements. (4) No attempt was made to detect functional errors in the specification.

Inspection is ineffective when verifying structural correctness. Whenever a new release of requirements is issued, unless the changes are minor in scope and isolated in location, there is no practical alternative but to conduct another round of inspection. It is difficult to keep the inspection teams focused and motivated if slightly modified requirements are subject to multiple reviews. The experience on SDS2 motivated research to explore ways to automate structural verification of the SRS so that inspection teams may focus on the more challenging task of detecting functional errors.

This paper demonstrates that the Prototype Verification System (PVS) is effective in automating verification of structural properties of SCR (Software Cost Reduction)-style requirements. Applicability of the proposed approach is not limited strictly to SCR-style requirements and to PVS. For example, a different translation algorithm could be developed to translate requirements written in a different formal specification language into PVS theorems. Likewise, structural properties, shown as PVS theorems in this paper, can be rewritten as logically equivalent statements accepted by other theorem provers.

The idea of developing a theorem prover just for the sole purpose of automating verification of structural properties can hardly be justified, technically and financially, as a meaningful activity. However, if an existing theorem prover can be used to automate labour-intensive and tedious activities, one can further improve productivity of the requirements engineering process and enhance software quality.

The rest of this paper is organized as follows. Using a portion of the Wolsung SDS2 requirements, Section 2 briefly reviews how an SCR-style SRS is organized. PVS is also introduced. Section 3 describes an algorithm to translate SCR-style requirements into PVS language. In addition, PVS theorems capturing the structural properties and commands used to carry out the proofs automatically are explained. Finally, Section 4 provides the conclusions reached during this activity. The SCR-to-PVS translation algorithm is described in Appendix A.

2. BACKGROUND

This section provides brief introductions both to SCR-style requirements and to PVS.

2.1. SCR-style requirements

The Wolsung SDS2 is designed to monitor reactor states continuously (e.g., temperature, pressure, and power), and to generate a trip signal (e.g., shutdown command) for the nuclear power plant to remain in safe states. Key attributes of SCR-style requirements[¶] are as follows.

Variable definitions. The interface between the computer system and its environment is described in terms of monitored and controlled variables. Monitored variables, whose names start with the `m_` prefix, refer to the inputs to the computer system, and controlled variables, whose names start with the `c_` prefix, refer to the outputs from the computer system. Attributes defined for each variable include type, units, range, and accuracy.

Functional overview diagrams (FODs). A FOD illustrates, in a notation similar to data flow diagrams, a hierarchical organization of functions. A group, denoted by the `g_` prefix, is further decomposed into groups or basic functions. Each atomic function name starts with the `f_` prefix. For example, group `g_Over`, highlighted in Figure 1(a), is refined into a basic function `f_PDLTrip` and a group `g_PZL` as shown in Figure 1(b). (Highlighting is used for illustration purposes only.) Similarly, a group `g_PZL` is decomposed into three basic functions as shown in Figure 1(c). In addition to the refinement relations, a FOD specifies inputs, outputs, and internal data dependencies among various components. Such data dependencies, in turn, implicitly dictate the proper order of carrying out a set of functions. For example, in Figure 1(c), the output of the `f_PZLSnr` function is used as an input to the `f_PZLTrip` function, and the latter function may be invoked only when the former is completed.

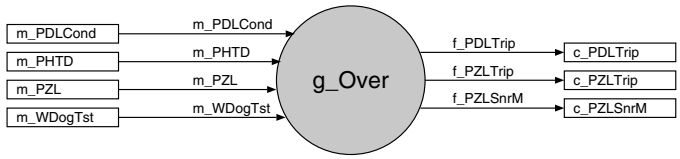
Structured decision table (SDT). The required behaviour of each basic function is expressed in a tabular notation, called the SDT, as shown in Figure 2. The function `f_PZLSnrM` produces a constant-valued output, denoted with the `k_` prefix, whose value is either `k_MsgOn` or `k_MsgOff`. As shown in the rightmost column, the function returns the value `k_MsgOff` when the value `m_WDogTst` is not equal to `k_Pressed` and the output of the function `f_PZLSnr` is not equal to `k_SnrTrip`. Otherwise, as shown in the first two columns, `k_MsgOn` is produced as the result. The symbol '-' denotes the 'don't care' condition.

In the case of the Wolsung SDS2, the SRS consisted of 30 and 59 monitored and controlled variables, respectively. 129 atomic functions were hierarchically organized into 15 function groups. The most complex SDT consisted of 17 rows and 12 columns.

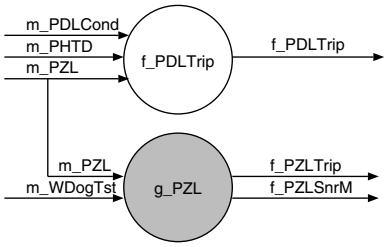
2.2. PVS

PVS, developed by a research group at SRI International, is an interactive tool for writing specifications and constructing proofs [9]. It has been successfully used in several industrial applications. Examples

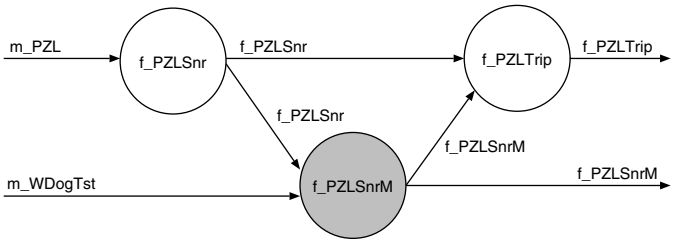
[¶]An SCR-style formal specification, used in the Wolsung SDS2, is slightly different from the SCR specification developed by researchers at the Naval Research Laboratory and supported by the SCR* toolset. The difference lies in how primitive functions are described. While the former is represented as a time-triggered AND–OR table, the latter uses an event–action table format. However, such semantic difference is irrelevant when it comes to verification of structural properties; thus specifications written in either notation can be verified.



(a)



(b)



(c)

Figure 1. Examples of the function overview diagram (FOD): (a) part of the FOD for SDS2; (b) a lower level FOD of `g_Over`; (c) a lower level FOD of `g_PZL`.

[10–13] include verification of several communication protocols and real-time protocols. One of the most complex systems verified by PVS to date is the microprocessor called AAMP5 which has nearly half a million transistors [14]. PVS was also used to prove nine of Parnas' theorems [15], and to prove the completeness and consistency of conditions in the RSML specification of the TCAS II specification [16].

The decision to use PVS was influenced by the following factors.

Condition Statements			
m_WDogTst = k_Pressed	T	F	F
f_PZLSnr = k_SnrTrip	–	T	F
Action Statements			
f_PZLSnrM = k_MsgOn	X	X	
f_PZLSnrM = k_MsgOff			X

Figure 2. Example of the structural decision table for f_PZLSnrM.

```
axf_PZLSnrM : AXIOM f_PZLSnrM(t) =
  LET
  X = (LAMBDA (x1 : pred[bool]),
              (x2 : pred[bool]):
       x1(m_WDogTst(t) = k_Pressed ) &
       x2(    f_PZLSnr(t) = k_SnrTrip )) IN TABLE
      %     |    |                  % '%' is the notation for starting comment line
      %     |    |
      %     v    v
      %------|----|-----------%   % If m_WDogTst = k_Pressed, or
      | X( T , ~ )| k_MsgOn   ||   % m_WDogTst /= k_Pressed and
      %------|----|-----------%   % f_PZLSnr = k_SnrTrip then
      | X( F , T )| k_MsgOn   ||   % k_MsgOn.
      %------|----|-----------%
      | X( F , F )| k_MsgOff  ||
      %------|----|-----------%
  ENDTABLE
```

Figure 3. Converted SDT f_PZLSnrM.

- PVS, a state-of-the-art verification system which is freely available, is a useful component when developing an integrated safety analysis environment in which proofs of structural, functional, and safety properties are performed.
- PVS supports high-order logic expressions, is equipped with highly automated and interactive proof procedures, and supports top-down proof exploration and construction. A proof strategy is intended to capture patterns of proof steps by combining primitive proof commands. For example, the grind command is perhaps the most frequently used command in highly automated proofs. Intuitively stated, the grind command instructs PVS to proceed with the proof in as much of an automated manner as possible without user intervention. That is, the command first installs the given theories and rewrite rules based on all the known and relevant definitions. It then carries out the first level of simplification and carries out all the equality

replacements [13]. In addition to the `grind` command, PVS provides a rich and powerful set of prover commands.
- PVS supports model checking [17] as well as theorem proving. Model checking, an automated technique for verifying behaviour of finite-state concurrent systems, is useful when verifying structural properties.
- The current release of PVS provides built-in support for SDT notation and its analysis, as shown in Figure 3. It should be noted that the `%` character is used to insert comments, that the axiom is visually similar to the SDT, and that rows and columns of the SDTs are interchanged. Figure 3 is semantically equivalent to the SDT shown in Figure 2. PVS automatically determines if the SDT includes missing (i.e., incomplete) or contradictory (i.e., non-deterministic) information during the type-checking procedure [18].

3. VERIFICATION OF STRUCTURAL PROPERTIES

The proposed approach to structural analysis of an SCR-style specification, illustrated in Figure 4, proceeds as follows.

1. Use a graphical editor to edit SCR-style requirements and translate to corresponding PVS specifications. An SRS editor was implemented in GNU C++ providing a Motif-based graphical user interface running on a Sparc station. In addition to providing icon-based editing of FODs and SDTs, the tool automatically generates PVS statements and inserts PVS theorems corresponding to the structural properties. The outputs are stored in a file. See Figure 5 for a screen dump of the tool.
2. Start PVS, load the translated file, and move the cursor to the area of the screen where the property to be proved is shown. Issuing the `M-x pr` command initiates a new proof session, and the commands shown below are used to carry out automated proofs.

3.1. Translation of SCR-style SRS into PVS statements

The translation algorithm is summarized in Appendix A, and it is illustrated below using the actual names appearing in the Wolsung SDS2 SRS. The example uses the FODs and the STD shown in Figures 1 and 2.

Step 1. Declaration of variables. The translation algorithm defines `variable_type` as an enumeration of all monitored variable names, basic function names, and controlled variable names. (See Figure 6.) It further partitions them into the following subsets: `monitor_set`, `function_set`, and `control_set`. User-defined subset types (e.g., `monitor_type`, `function_type`, and `control_type`) are also declared so that dependency relations may be specified.

Step 2. Declaration of function groups and hierarchy. The next phase translates information related to hierarchical definitions in the FOD as a set of tuples. As shown in Figure 7, `group_type` is declared as an enumeration of all function group names used in the FOD.

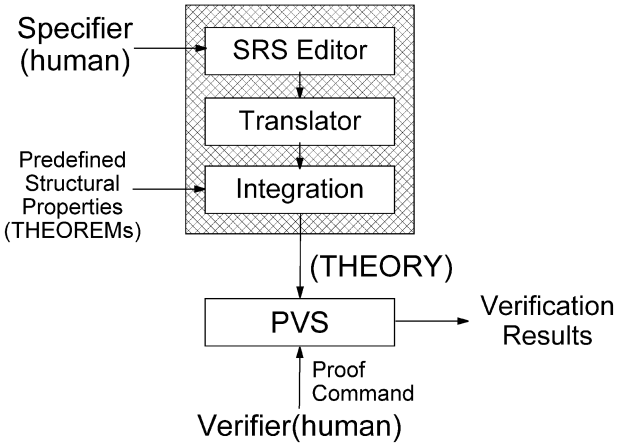


Figure 4. The outline of the overall approach.

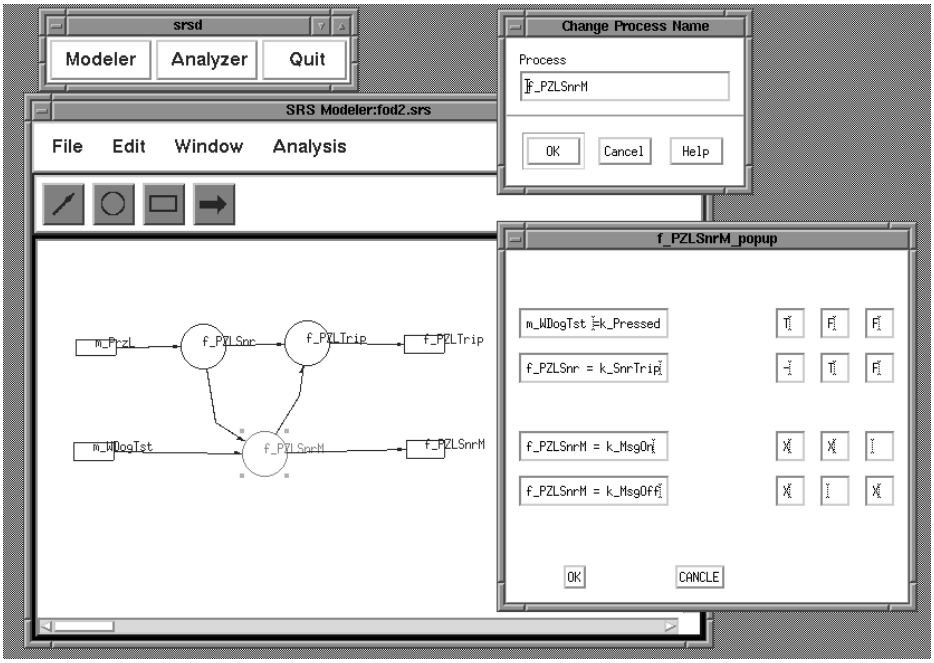


Figure 5. An SCR editor generating a PVS specification.


```
% variables in SRS
variable_type : TYPE = {
  m_PDLCond,  m_PHTD,     m_PZL,      m_WDogTst,
  f_PDLTrip,  f_PZLTrip,  f_PZLSnrM,  f_PZLSnr,
  c_PDLTrip,  c_PZLTrip,  c_PZLSnrM}

% monitored variables
monitor_set : setof[variable_type] = {x : variable_type |
  x = m_PDLCond     OR
  x = m_PHTD        OR
  x = m_PZL         OR
  x = m_WDogTst}
% monitored variable name type
monitor_type : TYPE = (monitor_set)

% functions
function_set : setof[variable_type] = {x : variable_type |
  x = f_PDLTrip     OR
  x = f_PZLSnrM     OR
  x = f_PZLTrip     OR
  x = f_PZLSnr}
% function name type
function_type : TYPE = (function_set)

% controlled variables
control_set : setof[variable_type] = {x : variable_type |
  x = c_PDLTrip     OR
  x = c_PZLTrip     OR
  x = c_PZLSnrM}
% controlled variable name type
control_type : TYPE = (control_set)
```

Figure 6. Declaration of variables (Step 1).

`group_in_group` definition captures hierarchical relationships among function groups. For example, group definition `g_Over` includes another group definition `g_PZL` as a subnode. Finally, `function_in_group` declaration, a tuple consisting of function name and group name, captures the group in which the definitions of primitive functions appear. For example, `f_PDLTrip` function is declared in the `g_Over` function group, whereas the other three functions are declared in the `g_PZL` group.

Step 3. Group inputs and outputs. In a manner similar to `function_in_group`, inputs to and outputs from each function group are declared as `input_group_dependency_set` and `output_group_dependency_set`, respectively. For example, Figure 8 captures that the group `g_Over` has four monitored variables as its inputs and produces three function outputs. It should be noted that, although not shown in the example, inputs to function groups are not

```
% function groups in SRS
group_type : TYPE = {
  g_Over,     g_PZL}

% groups within a group
group_in_group : TYPE = [group_type,group_type]
% (lower_group, higher_group)   ; hierarchy among groups
group_in_group_set : setof[group_in_group] = {x:group_in_group |
  x = (g_PZL, g_Over)}

% functions within a group
function_in_group:TYPE = [variable_type,group_type]
% (lower_function, higher_group); hierarchy among groups and functions
function_in_group_set : setof[function_in_group] = {x : function_in_group |
  x = (f_PDLTrip, g_Over)  OR
  x = (f_PZLSnr,  g_PZL)   OR
  x = (f_PZLSnrM, g_PZL)   OR
  x = (f_PZLTrip, g_PZL)}
```

Figure 7. Declaration of function groups and hierarchy (Step 2).

```
% inputs to groups
input_group : TYPE = [variable_type, group_type]
% (input, group)
input_group_dependency_set : setof[input_group] = {input_dep : input_group|
  % (input, g_Over)  ; input to g_Over
    input_dep = (m_PDLCond, g_Over)      OR
    input_dep = (m_PHTD, g_Over)         OR
    input_dep = (m_PZL, g_Over)          OR
    input_dep = (m_WDogTst, g_Over)      OR
  % (input, g_PZL)   ; input to g_PZL
    input_dep = (m_PZL, g_PZL)           OR
    input_dep = (m_WDogTst, g_PZL)}

% outputs of groups
output_group : TYPE = [group_type, variable_type]
% (group, output)
output_group_dependency_set : setof[output_group] = {output_dep:output_group|
  % (g_Over, output) ; output of g_Over
    output_dep = (g_Over, f_PDLTrip)     OR
    output_dep = (g_Over, f_PZLTrip)     OR
    output_dep = (g_Over,f_PZLSnrM)      OR
  % (g_PZL, output)  ; output of g_PZL
    output_dep = (g_PZL, f_PZLTrip)      OR
    output_dep = (g_PZL, f_PZLSnrM)}
```

Figure 8. Declaration of group inputs and outputs (Step 3).


```
dependency : TYPE = [variable_type, variable_type]

% (input, output);  dependency extracted from FOD
dependency_set : setof[dependency] = { element : dependency |
  % (input, f_PDLTrip) ; input to f_PDLTrip
    element = (m_PDLCond, f_PDLTrip)    OR
    element = (m_PHTD, f_PDLTrip)       OR
    element = (m_PZL, f_PDLTrip)        OR
  % (input, f_PZLSnr)  ; input to f_PZLSnr
    element = (m_PZL, f_PZLSnr)         OR
  % (input, f_PZLTrip) ; input to f_PZLTrip
    element = (f_PZLSnr, f_PZLTrip)     OR
    element = (f_PZLSnrM, f_PZLTrip)    OR
  % (input, f_PZLSnrM) ; input to f_PZLSnrM
    element = (m_WDogTst,f_PZLSnrM)     OR
    element = (f_PZLSnr, f_PZLSnrM)     OR
  % (input, controlled_variable) ; input to controlled variables
    element = (f_PDLTrip, c_PDLTrip)    OR
    element = (f_PZLTrip, c_PZLTrip)    OR
    element = (f_PZLSnrM, c_PZLSnrM) }

% (input, output);  dependency extracted from SDT
sdt_dependency_set : setof[dependency] = { element : dependency |
  % similar to dependency_set ... ...  }
```

Figure 9. Function inputs, outputs, and data flows (Step 4).

strictly limited to monitored variables. Data-flow dependencies between two functions declared in different groups need to be specified.

Step 4. Function inputs, outputs, and data flows. Data flows among primitive functions are captured in `dependency_set`. Similarly, dependency relations found in the SDT are captured in `sdt_dependency_set`. (See Figure 9.)

Step 5. Function inputs, outputs as a transition system. Data flows among functions are captured as a transition relation. A Boolean function, `transition (p,q)`, utilizes the `COND .. ENDCOND` constructor to avoid repeated use of nested IF clauses.

There are six trip conditions, three each for PDC1 and PDC2, specified in the Wolsung SDS2. The three trip conditions for PDC1 are referred to as PDL, PZL, and SLL. The PDL trip condition is the most complex of the three. There are eight and four monitored and controlled variables, respectively, as well as 14 primitive functions. Specification for the PDL trip is 22 pages long, and the most complex SDT has 12 rows and 17 columns. When applied to the specification for the PDL trip, it took about an hour to edit and convert the SCR-style specification into PVS specification language. Users of the proposed approach need to be knowledgable on how to write SCR-style requirements. On the other hand, technical expertise on PVS is not required because the translation process is entirely automated.

3.2. Structural properties and PVS theorems

P1: Each external input should be used in at least one function.

```
monitor_check: THEOREM
     FORALL (m_var : monitor_type): EXISTS (f_var : function_type):
     member((m_var, f_var), dependency_set)
```

This theorem is satisfied if and only if each of the monitored variables, referred to as the `m_var` in the theorem, appears as the first element in the definition of `dependency_set` relations. The second element in the tuple must be a primitive function name. The `member` function is one of numerous built-in PVS functions. In order to carry out the proof, a user enters the following PVS commands:

```
(apply (repeat (then (grind :if-match nil)(inst? :where +))))
```

This prover command specifies that the subcommands `(grind :if-match nil)` and `(inst? :where +)` are to be repeatedly applied as long as possible. Although the `grind` command is designed to complete the proof without requesting user intervention whenever possible, improper selection of proof heuristics during the instantiation or skolemization process may prevent the proof from being completed fully automatically. In such a case, the '`:if-match nil`' argument directs `grind` not to attempt heuristic instantiation and to leave the existential quantifier untouched in such cases. The '`inst?`' removes universal quantifiers by instantiation, and the '`:where +`' argument helps PVS to pick the correct instantiation in this case.

If the specification satisfies the property, the above proof should terminate successfully, while delivering the output message '`QED`', without requiring any user intervention. Otherwise, the proof has failed. For example, assume that `m_WDogTst` input is unused in the lower-level definition of the FOD shown in Figure 1(c). Such a situation may have been caused due to errors in editing or omission of a function definition (i.e., `f_PZLSnrM`) in the FOD. According to the translation algorithm, `dependency_set`, defined in Figure 9, will not include the tuple `(m_WDogTst, f_PZLSnrM)`. When the above PVS proof command is issued, PVS returns the following output to alert the user that it has detected an anomaly involving the variable `m_WDogTst`:

```
monitor_check :

{-1}  m_WDogTst? (m_var!1)
   |-------
```

The proof was carried out on an Ultra Sparc 2 station, equipped with one giga-byte of main memory, running version 2.3 of PVS. It took 28 s to complete the proof of the above property for the PDL trip specification automatically. While the exact duration needed to complete the proof varies, depending on such factors as workstation load, such data clearly suggest that an automated approach is clearly superior to the inspection method.

P2: Each external output should be generated by a function.

```
control_check1: THEOREM
     FORALL (c_var : control_type): EXISTS (f_var : function_type):
     member((f_var, c_var),dependency_set)
```


The theorem control_check1 is similar to the monitor_check used in **P1**. The only difference is that the membership relation is applied to the pairs of (function name, controlled variable) as opposed to (monitored variable, function name). It checks that all external outputs are actually generated by one or more functions. The same prover command for proving theorem monitor_check is applied to complete the proof.

If one wishes to enforce a further constraint so that each external output is generated by only one function, the theorem may be modified as follows so that a unique binding exists in the dependency_set for each control variable:

```
control_check2: THEOREM
     FORALL (c_var : control_type): FORALL (f_var1 : function_type):
     FORALL (f_var2 : function_type):
     member((f_var1, c_var), dependency_set) AND
     member((f_var2, c_var), dependency_set)
        => f_var1 = f_var2   % all bindings must be unique
```

The proof of the property control_check1 for the PDL trip took 21 s, whereas the one for control_check2 was completed in 37 s.

P3: All internal data flows must be generated from a source function and consumed by target functions.

```
function_check: THEOREM FORALL (f_var: function_type):
     EXISTS (var1 : variable_type): EXISTS (var2:variable_type):
     member((var1,f_var), dependency_set) AND
     member((f_var,var2), dependency_set)
```

The theorem function_check ensures that the requirements are free from input–output anomalies. The same prover command, used to prove properties **P1** and **P2**, is used. When applied to the PDL trip, PVS took 66 s to complete the proof automatically.

P4: All data flows declared in higher levels of a FOD are consistent with the ones defined in lower levels and *vice versa*.

A FOD is most likely to be organized in multiple levels of group hierarchy, and all data flows defined at different levels of the hierarchy must be consistent. For example, all inputs to a group must be declared as inputs to functions or nested groups when decomposed. A theorem specifying consistency for outputs is defined similarly.

```
hierarchy_check1 : THEOREM
  FORALL (hi_grp:group_type): FORALL (in_var:variable_type):
  member((in_var,hi_grp),input_group_dependency_set) =>    % an input to group
    ((EXISTS (lo_grp:group_type):
     member((lo_grp,hi_grp),group_in_group_set) AND        % lower level group's
       member((in_var,lo_grp),input_group_dependency_set)) % input
    OR
    (EXISTS (lo_fun:function_type):
     member((lo_fun,hi_grp),function_in_group_set) AND   % lower level function's
       member((in_var,lo_fun),dependency_set)))          % input
```


Another property to check is that all inputs to functions declared at a lower level must come from either outside the group boundary or as data flow originating internally from a group or function declared at the same level of abstraction.

```
hierarchy_check2 : THEOREM FORALL (in_var:variable_type):
  (FORALL (lo_grp:group_type):
   member((in_var,lo_grp),input_group_dependency_set) =>% an input to group
    (FORALL(hi_grp:group_type):
     member((lo_grp,hi_grp),group_in_group_set)         % higher level group
     =>
      ((member((in_var,hi_grp),input_group_dependency_set))
                                                 % input to higher level group
       OR
      (EXISTS (sa_grp:group_type):
       member((sa_grp,hi_grp),group_in_group_set) AND   % same level group's
         member((sa_grp,in_var),output_group_dependency_set))% output
       OR
      (member((in_var,hi_grp),function_in_group_set)))))% same level function's
                                                        % output
  AND
  (FORALL (lo_fun:function_type):
   member((in_var,lo_fun),dependency_set) =>            % an input to function
    (FORALL (hi_grp:group_type) :
     member((lo_fun,hi_grp),function_in_group_set)      % higher level group
     =>
      ((member((in_var,hi_grp),input_group_dependency_set))
                                                 % input to higher level group
       OR
      (EXISTS (sa_grp:group_type):
       member((sa_grp,hi_grp),group_in_group_set) AND   % same level group's
         member((sa_grp,in_var),output_group_dependency_set))% output
       OR
      (member((in_var,hi_grp),function_in_group_set)))))% same level function's
                                                        % output
```

Proofs of these properties for the PDL trip were completed in 12 and 15 s, respectively, for theorems `hierarchy_check1` and `hierarchy_check2`.

P5: The data flows of one function are consistent with the input–output relation of the function definition body written in a structured decision table.

Input and output relations declared in the FOD must be consistent with those in the structured decision table. That is, if function `f_A` accepts data flow input from function `f_B`, the SDT corresponding to function `f_A` must specify how the input is used in its computation. The `grind` command of PVS is sufficient to carry out the proof.

```
consistency_fod_sdt : THEOREM
     FORALL (a1:variable_type): FORALL (a2:variable_type):
     member((a1,a2),dependency_set)
     IFF
     member((a1,a2),sdt_dependency_set)
```

Proof of this theorem, the simplest of the six properties, took 7 s.


```
IMPORTING ctlops % import modules to perform model checking

p,q,r,s: VAR variable_type

transition(p,q) : bool =
  COND
  % input -> f_PDLTrip ; input to f_PDLTrip
   % m_PDLCond?(p) is equivalent to the expression "m_PDLCond = p"
    m_PDLCond?(p)  -> f_PDLTrip?(q),
    m_PHTD?(p)     -> f_PDLTrip?(q),
    m_PZL?(p)      -> f_PDLTrip?(q),
  % input -> f_PZLSnr  ; input to f_PZLSnr
    m_PZL?(p)      -> f_PZLSnr?(q),
  % input -> f_PZLTrip ; input to f_PZLTrip
    f_PZLSnr?(p)   -> f_PZLTrip?(q),
    f_PZLSnrM?(p)  -> f_PZLTrip?(q),
  % input -> f_PZLSnrM ; input to f_PZLSnrM
    m_WDogTst?(p)  -> f_PZLSnrM?(q),
    f_PZLSnr?(p)   -> f_PZLSnrM?(q),
  % input -> controlled_variable ; input to controlled variables
    f_PDLTrip?(p)  -> c_PDLTrip?(q),
    f_PZLTrip?(p)  -> c_PZLTrip?(q),
    f_PZLSnrM?(p)  -> c_PZLSnrM?(q)
  ENDCOND
```

Figure 10. Function inputs, outputs as a transition system (Step 5).

P6: No circular dependencies exist among data flows.

Circular dependencies occur when there exists a function `f_A` that accepts data flow from another function `f_B` whose computation depends on the data flow originating from `f_A`. Although lack of circular dependencies can be interactively verified using the theorem-proving capabilities of PVS, the current implementation does not adequately support mechanisms to automate the proof. As an alternative, the authors chose to apply the model checking command by rewriting the input and output dependencies as transition relations shown in Figure 10. The property to check is expressed in the following branching time temporal logic called CTL (Computational Tree Logic):

```
circular_dependency_check : THEOREM
  FORALL (var: variable_type) :
  AG(transition, (LAMBDA p:((var = p) => NOT EX(transition,
             (LAMBDA q: EF(transition, (LAMBDA r: (var = r)))(q)))(p))))(s)
```

In PVS, a CTL formula obeys the following format: `AG(`*transition_relation*, *predicate*`)(`*state*`)`. The `AG` operator asserts that the *predicate* is true on all paths induced by the given *transition_relation* starting from the specified *state*. The modal operator `EX(`*transition_relation*, *predicate*`)(`*state*`)` evaluates to true when *predicate* holds in any of the immediate successors to the current state. Similarly,

the `EF` operator determines if *predicate* is true in the current state or any state eventually reachable from the current state. In the above theorem, `p` denotes the current state, and `q` refers to the immediate successor states to `p`, whereas `r` refers to the states eventually reachable from `q`. Therefore, the theorem states that all variables used in the state `p` are never used, as the `AG` operator indicates, in states `r`. The PVS command `(model-check)` is used to carry out the verification automatically.

When applied to the PDL trip, PVS took about 25 min to verify the property among 33 dependency relations. The proof took a long time, relative to other properties, because the model checking algorithm requires exhaustive, perhaps repetitive[‖], generation and search of all possible states.

Although the Wolsung SDS2 SRS was revised four times in a year, each release was not subject to separate and repeated verification using PVS. Research reported in this paper was initiated after completing four rounds of inspection whenever a new release of SRS became available. There were no errors in the fourth and the final release of the Wolsung SRS violating the structural properties discussed in this paper. Accordingly, the proofs were completed successfully without errors. However, if one were to employ the approach described in this paper, only the modified requirements would need to be entered using a graphical editor, and then PVS theorems generated, and the same proof commands executed. Since the first task is easy to carry out and the remaining tasks are automated, repeated verification of evolving requirements would thus be much simplified.

Applicability of the technique is not only limited to the six properties described above. Additional properties, if necessary, can be captured in PVS theorems and verified by PVS. Such a task is not as simple as entering requirements using a graphical editor and verifying the properties discussed earlier. One needs to be an expert with the PVS specification language and proof commands.

3.3. Related work

There have been several approaches in which an automated verification of the structural properties of requirements has been attempted, and modern CASE (Computer Aided Software Engineering) tools used in requirements engineering, as shown in Table I, provide limited support in enforcing structural correctness of the SRS. For example, SCR* [7,19] can automatically check if requirements are complete and consistent. Completeness means that the disjunction of all conditions covers all possible input values (i.e., yielding a tautology), and consistency means that conditions do not overlap (i.e., the SRS is deterministic) [20]. Additionally, correctness of type definitions, circularity among inputs and outputs of functions, and proper initialization of variables can be checked. However, SCR* is unable to check the property **P3**. Similarly, SCR* does not support consistency checks between the FOD and the SDT. RSML, another requirements language tool, has been used to specify the required behaviour of avionics equipment [21] but the RSML simulator provides little support beyond examining conditions for logical completeness or consistency [22]. Tablewise [23] is similar to the RSML simulator in its capability to verify structural correctness of requirements specifications. STATEMATE [24] provides built-in checks to make sure that all inputs are used, and that data flows declared in the activity chart are consistent with information specified in the corresponding statecharts. However, as indicated in

[‖] When the experiment was repeated using 10 or 20 dependency relations, model checking took about 8 and 15 min, respectively.

Table I. Comparison of CASE tools supporting analysis of structural properties.

	SCR*	RSML simulator	Tablewise	STATEMATE	Proposed approach
Completeness	✓	✓	✓		✓
Consistency (aka determinism)	✓	✓	✓	✓[a]	✓
P1. `monitor_check`	✓			✓	✓
P2. `control_check`	✓				✓
P3. `function_check`				✓	✓
P4. `hierarchy_check`	N/A[b]	N/A		✓	✓
P5. `consistency_fod_sdt`		N/A		✓	✓
P6. `circular_dependency_check`	✓				✓

[a]The presence of non-determinism is reported only during simulation.
[b]Irrelevant features are indicated as N/A. For example, SCR* and RSML simulator do not support the declaration of data flows at different levels of abstractions.

Table I, it, too, provides only limited support when it comes to enforcing structural correctness among requirements.

4. CONCLUSION

Earlier in this paper, based on the experience of inspecting the Wolsung SDS2 SRS, the authors demonstrated that manual inspection of structural properties of an SCR-style requirements specification is inefficient and would benefit from some automated assistance. In order to automate structural verification to overcome this inefficiency, a software tool was developed that provides a graphical user interface which converts SCR-style requirements specifications into PVS language. To use this automation, a set of structural properties to be proved are captured in PVS theorems, and then PVS proof commands are used to carry out the proof automatically.

Users of the proposed approach need not be experts on formal methods or power users of PVS. A graphical editor provides a user-friendly interface to allow editing of an SCR-style specification and automates the translation process. If the specification is structurally correct, the proof process can be completed without requiring user intervention, other than typing the PVS proof commands as explained in the paper.

The proposed technique complements, rather than replaces, inspection methods. Inspection has been shown to be generally effective in numerous and large-scale industrial projects. When compared to the 15 staff hours needed to complete one round of manual inspection for six trip conditions and a couple of miscellaneous functions, the 30 min (for one trip condition) demonstrated with the approach proposed here appears noticably more productive than inspection. It must be emphasized that the primary benefit of this research is to automate tedious and repetitive inspection activities and that proof optimization is not the primary goal of the research described in this paper. While the proposed approach offers significant help to practitioners, it is also clear that there are some practical limitations. Perhaps most

prominently, the correctness of the PVS code itself has not been formally verified (although it is highly reliable and well supported) and hence may itself contain bugs, thus rendering its conclusions less reliable than might be desired**.

Another benefit of the proposed approach is that it is highly reusable. That is, in order to conduct verification of structural properties for another system, one needs only to provide an SCR-style specification using the graphic editor provided. The proof procedures and commands do not vary from one system to another.

Although the technique described in this paper has been applied to date only on the Wolsung SDS2, it is expected that the next generation nuclear power plant shutdown system, currently in development in Korea, could utilize technical advances reported here. Furthermore, this technique can be readily applied to other application domains as long as requirements can be written in SCR-style notation.

Lastly, several enhancements seem possible. They include, but are not limited to, the following:

- additional definition of structural properties, representation of them in terms of PVS theorems, and derivation of proof commands;
- development of translation rules so that a formal specification written in statecharts or modechart [25] can be verified for structural properties;
- verification of application specific but essential properties within an integrated verification environment. If software safety requirements can be captured in PVS theorems, the satisfaction of safety properties may be formally verified; the consistency between natural language program functional specifications and software requirements specifications could also be checked.

APPENDIX A. TRANSLATION ALGORITHMS

```
print 'stucturalproperties : THEORY'
print 'BEGIN'
% Section 3.1
% Step 1: Declaration of Variables
construct a list of all variable names
print 'variable_type : TYPE = {'
print the list of all variable names
print '}'

construct a list of monitored variable names
print 'monitor_set : setof[variable_type]={x : variable_type|'
print the list of monitored variable names
print '}'
print 'monitor_type : TYPE = (monitor_set)'

construct a list of function names
print 'function_set : setof[variable_type]={x : variable_type|'
```

**In fact, in the process of using PVS to verify SRS for the Wolsung SDS2, PVS generated outputs that seemed incorrect. An inquiry to the PVS developers indeed confirmed that bugs were present in the PVS code.

print the list of function names
print '}'
print 'function_type : TYPE = (function_set)'

construct a list of controlled variable names
print 'control_set : setof[variable_type] = {x : variable_type|'
print the list of controlled variable names
print '}'
print 'control_type : TYPE = (control_set)'

% Step 2: Declaration of Function Groups and Hierarchy
construct a list of all group names
print 'group_type : TYPE = {'
print the list of all group names
print '}'

print 'group_in_group : TYPE = [group_type, group_type]'
construct a list of tuple (lower_group, higher_group)
print 'group_in_group_set : setof[group_in_group] = {x : group_in_group |'
print the list of tuple (lower_group, higher_group)
print '}'

print 'function_in_group : TYPE = [function_type, group_type]'
construct a list of tuple (lower_function, higher_group)
print 'function_in_group_set: setof[function_in_group]={x:function_in_group |'
print the list of tuple (lower_function, higher_group)
print '}'

% Step 3: Group Inputs and Outputs
print 'input_group : TYPE = [variable_type, group_type]'
construct a list of tuple (input, group)
print 'input_group_dependency_set: setof[input_group]={input_dep:input_group |'
print the list of tuple (input, group)
print '}'

print 'output_group : TYPE = [group_type, variable_type]'
construct a list of tuple (group, output)
print 'output_group_dependency_set: setof[output_group]={output_dep:output_group|'
print the list of tuple (group, output)
print '}'

% Step 4: Function Inputs, Outputs, and Data-Flows
print 'dependency : TYPE = [variable_type, variable_type]'
construct a list of tuple (input, function) and (function, controlled_variable) from FOD
print 'dependency_set : setof[dependency] = { element : dependency|'
print the list of tuple (input, function) and (function, controlled_variable)
print '}'

```
construct a list of tuple (input, function) and (function, controlled_variable) from SDT
print 'dependency_set : setof[dependency] = { element : dependency|'
print the list of tuple (input, function) and (function, controlled_variable)
print '}'

% Step 5: Function Inputs, Outputs as a transition system
print 'IMPORTING ctlops'
print 'p,q,r,s: VAR variable_type'
construct a list of tuple (input, function) and (function, controlled_variable) from FOD
print 'transition(p,q) : bool ='
print ' COND
print the list of tuple (input, function) in input'?(p)->'function'?(q)'
print the list of tuple (function, controlled_variable) in function'?(p)->'controlled_variable'?(q)'
print ' ENDCOND


  % Section 3.2
% Structural Properties in PVS
% P1: monitor_check
print 'monitor_check : THEOREM'
print '      FORALL (m_var : monitor_type): EXISTS (f_var : function_type):'
print '      member((m_var, f_var), dependency_set)'

% P2: control_check
print 'control_check1 : THEOREM'
print '      FORALL (c_var : control_type): EXISTS (f_var : function_type):'
print '      member((f_var, c_var), dependency_set)'

% print P3, P4, P5, P6
print 'END structuralproperties'
```

ACKNOWLEDGEMENTS

The authors acknowledge the comments from the reviewers and, in particular, those from the Editor, Lee White. Also comments from Matt Jaffe helped the authors write clearly. Ongoing support of the PVS toolset by SRI has been beneficial. This work was supported in part by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc) and through the Center for Advanced Reactor Research (CARR).

REFERENCES

1. Lutz RR. Targeting safety-related errors during software requirements analysis. *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993; 99–106.
2. Fagan ME. Advances in software inspections. *IEEE Transactions on Software Engineering* 1986; **12**(7):133–144.
3. Hinchey MG, Bowen J. *Application of Formal Methods*. Prentice-Hall, 1995; 1–442.

4. Wheeler DA, Brykczynski B, Meeson RN Jr. *Software Inspection: An Industry Best Practice*. IEEE Computer Society Press, 1996; 1–312.
5. AECL CANDU. *Software Requirements Specification, SDS2 Programmable Digital Comparators*. Wolsung NPP, 86-68350-SRS-001 Revision 2, AECL, 1994; 1–374.
6. AECL CANDU. *Software Work Practice, Procedure for the Specification of Software Requirements for Safety Critical Systems*. Wolsung NPP, 00-68000-SWP-002, AECL, 1993; 1–82.
7. Heitmeyer CL, Jeffords RD, Labaw BG. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(3):231–261.
8. Craigen D, Gerhart S, Ralston T. Case studies. *An International Survey of Industrial Applications of Formal Methods*, vol. 2. U.S. Department of Commerce, 1993; 1–188.
9. Crow J, Owre S, Rushby J, Shankar N, Srivas M. A tutorial introduction to PVS. *Workshop on Industrial-Strength Formal Specification Techniques (WIFT '95)*, 1995; 1–112.
10. Owre S, Shankar N, Rushby J, Stringer-Calvert D. *PVS System Guide Version 2.3*. Computer Science Laboratory, SRI International, 1999; 1–88.
11. Owre S, Shankar N, Rushby J, Stringer-Calvert D. *PVS Language Reference Version 2.3*. Computer Science Laboratory, SRI International, 1999; 1–87.
12. Owre S, Rushby J, Shankar N, von Henke F. Formal verification for fault-tolerant architecture: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering* 1995; **21**(2):107–125.
13. Shankar N, Owre S, Rushby J, Stringer-Calvert D. *PVS Prover Guide Version 2.3*. Computer Science Laboratory, SRI International, 1999; 1–117.
14. Miller SP, Srivas M. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Workshop on Industrial-Strength Formal Specification Techniques (WIFT '95)*, 1995; 2–16.
15. Rushby J, Srivas M. Using PVS to prove some theorems of David Parnas. *Proceedings of 1993 International Conference on HOL Theorem Proving and Its Applications*, 1993; 163–173.
16. Heimdahl MPE, Czerny BJ. On the analysis needs when verifying state-based software requirements: An experience report. *Science of Computer Programming* 2000; **36**(1):65–96.
17. Clarke E, Grumberg O, Peled DA. *Model Checking*. MIT Press, 1999; 1–313.
18. Owre S, Rushby J, Shankar N. Integration in PVS: Tables, types, and model checking. *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)* (*Lecture Notes in Computer Science*, vol. 1217). 1997; 366–383.
19. Heitmeyer C, Kirby J, Lawbaw B. Tools for formal specification, verification, and validation of requirements. *Proceedings of 12th Annual Conference on Computer Assurance (COMPASS '97)*, 1997; 16–29.
20. Jaffe MS, Leveson NG, Heimdahl MPE, Melhart BE. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering* 1991; **17**(3):241–258.
21. Leveson NG, Heimdahl MPE, Hildreth H, Reese JD. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering* 1994; **20**(9):684–706.
22. Heimdahl MPE, Leveson NG. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering* 1996; **22**(6):363–377.
23. Hoover DN, Chen Z. Tablewise, a decision table tool. *Proceedings of 10th Annual Conference on Computer Assurance (COMPASS '95)*, 1995; 97–108.
24. Harel D, Lachover H, Naamad A, Pnueli A, Politi M, Sherman R, Shtull-Trauring A, Trakhtenbrot M. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering* 1990; **16**(4):403–414.
25. Jahanian F, Mok AK. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering* 1994; **20**(12):933–947.

