Domain Consistency in Requirements Specification

Satyajit Acharya
Department of Comp. & Info. Sc.
University of Hyderabad
Hyderabad - 500046, India
satyajit2k3@yahoo.com

Hrushikesha Mohanty
Department of Comp. & Info. Sc.
University of Hyderabad
Hyderabad - 500046, India
hmcs@uohyd.ernet.in

Chris George
UNU-IIST, Macao SAR, P.R.China
cwg@iist.unu.edu

Abstract

Fixing requirements errors that are detected late in the software development life cycle can be extremely costly. So, finding problems in requirements specification early in the development cycle is critical and crucial. A formal specification can reduce errors by reducing ambiguity and imprecision and by making some instances of inconsistency and incompleteness obvious. In this paper, with an example of a moderately complex system of the mobile computing domain, we discuss how the consistency conditions found during initial abstract formal specification help in detecting logical errors during early stages of system development. We also discuss the importance of consistency conditions while modelling the domain of a complex system and show how the identified consistency conditions help in better understanding the specification and to gain confidence on the correctness of the specification. We use a combination of techniques, like specification inspection and testing the executable specification of a prototype using test cases, to validate the specification against the requirements as well as to ensure that the specified consistency conditions are respected and maintained by the operations defined in the specification.

Keywords: Consistency Conditions, Specification, Testing, RAISE, RSL

1. Introduction

While trying to develop a reliable software for a complex, dynamic and distributed system (e.g. a Mobile Computing system), it is required (and is often difficult) to make a clear distinction between correct and incorrect behaviors of components of the system. Also, for a large system, it becomes difficult to ensure that the implemented system will have consistent behavior. One possibility is to do constant runtime monitoring of several integrity or consistency conditions that must hold in a system to ensure the correct behaviors of its components. For instance, dangling pointers, corrupted data structures, lost resources, etc. indicate potential problems in the system, and by continuously checking the Consistency Conditions (henceforth CCs) defined for a system, these problems and other inconsistencies in the system can be identified before they lead to system failures.

During its lifetime, a robust system might run consistency checks after performing each action. But running such checks are sometimes too expensive in time, and often not feasible. This is especially true for distributed and dynamic systems like Mobile Computing (henceforth MC) systems. For example, when an action is performed by a task (being executed) in a mobile device of a MC system, it is not desirable (and is often not possible) to check the consistency of the whole system. Doing such a check would consume heavy resources and would take much time (e.g. to collect the relevant information about the system, since they may be distributed throughout the system). Note that, even if inconsistencies are discovered during the runtime of a system, it may not be possible to fix the problems, since the causes of the problem can not be identified or located. It may be required to redevelop the system from the beginning! So, ensuring the consistency of a system during the early stage of requirements specification is important.
The initial informal specification of requirements for a computer-based system often contain errors because of one or more of the following: (1) the domain of the specification may be large, (2) writers of the specification are fallible and are prone to errors, and (3) the requirements may be ill understood. System analysts need to examine this often incomplete and inconsistent specification and based on the available domain knowledge and expertise, transform this informal specification into more complete, consistent, and correct requirements specification [1].

In order to reduce the cost and risk of finding and fixing errors late in software development, it is desirable and important to detect faults in a specification before it is implemented. Finding errors at later stages of system development are costly, because correcting such errors may need changes in the whole specification and implementation. Detecting faults in specifications can help reduce the cost and risk of software development because incorrect implementation can be prevented early. This can be achieved by verifying the consistency and validity of specifications. Faults in specifications may arise if the consistency of the specification is violated or the user requirements are misrepresented by the specification. Consistency requires that no two or more requirements in a specification contradict each other [2]. Ensuring consistency among different requirements of a system is a challenging task.

In this paper, we discuss how the CCs found during initial abstract formal specification help in detecting logical errors during early stages of system development. We take an example of an MC environment that is highly dynamic in nature due to the mobility of various components. While writing the formal specification for such system using the RAISE specification language (RSL) [3], we found some CCs that helped us in better understanding the domain and in finding some logical errors. Some of these errors were conceptual, and occurred due to misunderstanding of the domain. Some other errors were related to misrepresentation of the information by the specification. If these consistency conditions had not been used while testing the specification, these conceptual/logical errors could have gone undetected. Due to lack of space, we do not present all the details of the specifications and the testing strategies we adopted to gain confidence in the specification. For such details, readers are referred to [4].

The rest of the paper is organized as follows. Section 2 gives an overview of the existing work on different types of consistency. In section 3, we describe the importance of CCs while writing formal specifications. We also discuss various techniques we used to gain confidence in the correctness of the specification. In section 4, we briefly describe the domain of MC environment. This model was used for writing the specification. After a short introduction to RSL, we also give a brief skeletal structure of the specification of this domain model. Section 5 describes some CCs found for the MC environment model and in section 6, we describe how these CCs helped us in finding some errors in the specification. In section 7, we describe the overall complexity of the specification and summarize the results found during the specification testing phase. Section 8 concludes the paper with a discussion.

2. Related Work

A number of works in the area of consistency in specification have been reported in the literature. Various aspects of consistency, like temporal consistency [5], data consistency [6], consistency between different UML diagrams [7, 8], viewpoints consistency in software processes [9], document consistency [10, 11] and so on, have been actively taken up by different researchers. In [12], the author discusses the utility of formal specification and argues that a formal specification must be adequate, internally consistent, unambiguous, and complete. In [13], the author gives a method to construct requirements specifications by using a combination of De-Marco Data Flow Diagram and VDM and calls the method as FRSM (Formal Requirement Specification Method). The author also mentions a set of rules to ensure the internal consistency of specification, especially concerning the pre- and post-condition structures in FRSM.

While developing a system using graphical description techniques, multiple description techniques are used to represent different views of the same system, and when these views are combined together, they form a complete specification. However, the consistency and completeness of the integration of these description techniques and views becomes a major issue. In [14], the authors provide an approach to ensure conceptual and semantic consistency when using different techniques that are used to represent various views of a system. In the paper [15], the authors discuss the issue of consistency of models made up of different sub-models. They introduce a consistency concept for software components modelled in the Unified Modelling Language (UML) and propose suitable consistency checks. In [8], the authors propose a method to test the consistency of different dynamic UML diagrams by using dynamic meta modelling rules as a notation for the CCs.

In [16], the authors describe a formal analysis technique called ‘consistency analysis’ that is applied to the tabular notation of SCR (Software Cost Reduction) method for detecting domain-independent errors in requirements specifications. The analysis method automatically identifies different kinds of errors, such as type errors, non-determinism, missing cases, and circular definitions, in requirements specifications. They used an automated consistency checker to check the SCR notation specification.
for syntax and type correctness, coverage, determinism, and other application-independent properties. In the paper [17], the authors describe an algorithm to automatically generate state invariants, properties that hold in every reachable state of a state machine model, from requirement specification using the SCR method. Such state invariants can be presented to system users for validation. The CCs presented in this paper are similar to invariants proposed in [17]. But, we believe that automatic generation of such invariants from a specification may not always capture all the invariants that are domain dependent. Some invariants may need additional domain knowledge that may not have been included in the specification. Hence, finding such invariants automatically from the specification may not always be possible. Also, there is a significant difference between generating state invariants from specifications as done in [16, 17] and writing them separately (as we do in this paper). If an invariant which should be maintained is not in fact maintained by the functions in the specification, then it cannot be automatically generated from that specification, and the error cannot be discovered. If one aims at discovering errors in the specification, then writing the invariants is much better.

The paper [18] illustrates some typical consistency management requirements and discusses the requirements in terms of both functionality and cross-cutting concerns that affect how this functionality is provided. The authors discuss the properties of a consistency management system, and the process of controlling the manipulation of objects in a system to ensure that their CCs are respected. The work in [19] uses a database to store the information about a system and the results of queries on the database are used for analyzing various properties and the consistency of the system. The author emphasizes the rapid application of formal method for system development and consistency analysis. In [2], the author puts forward specification testing as a practical technique for verification and validation of formal specifications using the approach to derive proof obligations from a specification and then test them, in order to detect faults leading to the violation of consistency or validity of the specification. Here, testing is done on the proof obligations derived from the specification. As we mentioned earlier, some domain information may not have been included in the specification and hence, this method may not find all types of errors in the system. Our work is more concerned with the use of CCs to aid in the correct evolution [1] of requirements specifications.

We next describe our views on specification consistency and different techniques to gain confidence in the correctness of requirements specifications.

### 3. Specification & Consistency

Consistency refers to situations where the specification of a system contains no internal (logical) contradictions. Some kinds of consistency apply to all specifications, such as function preconditions being satisfied by function calls, arguments of functions being in subtypes, and results of functions being in subtypes. A collection of these, termed **confidence conditions** in RAISE, is defined for RSL and can be generated automatically by the RAISE tool [20]. Checks for these conditions can also be included in implementations, so that they can be checked during testing, for example. Since confidence conditions are not domain-specific, and their generation and inclusion in testing is handled by tools, we henceforth use the term confidence condition (CC) to mean those particular to the domain, which must be identified by the designer.

To aid in identification, there is a useful slogan, “**No Loss, No Confusion**”, related to consistency. CCs try to ensure that there is neither any loss of information (**No Loss**) nor any confusion between various components in a system during its operation (**No Confusion**). The effects of loss of information may not be immediately apparent, but they may have devastating effects on the system in the long run. Also, when such an inconsistency is detected, it may be difficult to find its cause. **No Confusion** means that a system does not do any operation that leads to disparity in information in different parts of the system.

While designing a system, a designer can view it from two perspectives:

1. **Change Perspective**: Each operation on the system changes the state of the system. That is, an operation on the system may change some (or possibly all) attributes of that system, thereby changing its state. A designer tries to make sure that specification of such changes (dynamic behaviour of the system) comply to the requirements of the system (provided by the use cases).

2. **Invariants Perspective**: Almost every system has a set of properties (called **invariants**) that should not change during the system’s lifetime. So, every operation on the system should preserve the characteristics of these invariants. These invariants are, in fact, represented by the CCs.

Using these two perspectives together as guidelines helps a designer of a system to identify and deal with problems during early phases of design. An operation on a system may give the result expected from it, but, apart from the expected result, the operation may also have some side effects (e.g. loss of information about a resource) on the system that violates its invariants. The CCs help the designer to find and avoid such side effects.
3.1. Ensuring the Correctness of specifications

Once we have identified and defined the conditions that comprise consistency we can define predicates that state (a) that the initial state is consistent, and (b) that each generator preserves consistency when its precondition is satisfied. These take the form:

$$\text{consistent}(s) \land \text{pre}(\text{gen}) \Rightarrow \text{consistent}(\text{gen}(s))$$

Note that although these conditions are properties of the generators, consistency itself is a property of the state, and hence of the model of our domain.

We could take these conditions as proof obligations, but doing such formal proofs is expensive - they are time consuming and only experienced, expert people can do them. So, proofs are usually done for critical systems (or for some critical operations of a system) only.

It is also possible to use formal methods without formal proof, and such use of formal method is sometimes called “lightweight” [3]. We adopted this approach and used various techniques (like specification inspection, prototyping and testing) to increase the confidence in the specification correctness and to ensure that the consistency of a system is maintained, as explained below:

1. Generate and carefully inspect the confidence conditions. You can instead assume that errors in confidence conditions will be detected during testing, but experience suggests it is worth doing the inspection as the issues raised in checking confidence conditions are in practice common sources of error.

2. Test the consistency of the initial state of the system by using test cases on an executable specification.

3. Test the consistency of the result of every operation on the system by using test cases on the executable specification. Note that, the test cases developed for a specification can also be used in testing the final implementation of a system to check that the implementation maintains the consistency in the system.

Note that these operations are carried out while developing a specification. One has to be careful when refining the specification further towards the implementation. During such refinements all the properties specified by the CCs should be preserved.

Next, we briefly present the MC environment model for which the specification were written. The details of the model and its specification can be found in [4] (available at [21]).


An MC environment includes Mobile Support Stations (MSSs) and Mobile Hosts (MHs) as shown in Fig. 1. Each MSS is responsible for managing a number of MHs in a geographic location called a cell. The MSS provides different services to all MHs residing in its cell. Also, each MSS acts as the Base MSS for a number of MHs in the system and each MH in the system must have one MSS as its Base MSS.

![Figure 1. Mobile Computing Environment](image-url)

An MC application can be modelled by a set of interoperable objects located in different MHs and collaborating by passing messages among themselves. While these objects may move to different hosts, hosts can also change locations. We refer to such objects as MC tasks or simply as tasks. Difficulties raised by the resource constraints of the mobile devices [22] are compounded by mobility that induces variability in availability of both communication and computational resources. This makes the behavior of the environment more complex. In order to make a task run in MC environment the task should support the different services [22] like migration, cloning, hoarding, resource sharing, inheritance, location awareness and support for disconnected operations.

4.1 RAISE & RSL

“RAISE” is an acronym for ‘Rigorous Approach to Industrial Software Engineering’. The Raise specification Language (RSL) [3] is a formal, modular, and typed specification language suitable for formal specification and development of software systems. It provides support to a number of different styles of specification: abstract, property-oriented, sequential as well as concurrent specification of systems, and also allows the construction of model oriented designs. The language can be used to
formulate both initial, very abstract specifications and to express low level designs suitable for translation to programming languages. The software engineering method provides for formulating initial abstract specification, development, justification and translation [23].

In RSL, specifications are collections of related modules. Modules are the means by which specifications can be decomposed into comprehensible and reusable units. There are two kinds of modules in RSL: schemes and objects. Schemes are essentially named class expressions, and objects are instances of classes. A class expression represents a set of models. Each model associates an entity (value, type, variable, etc.) with each identifier defined within the class expression.

4.2 The Model of MC Environment

The class diagram that depicts the structure of an MC environment is given in Fig. 2. Here, the association between a set of MHs that may be present in an MSS’s cell is given by the link ‘Has residing MHs’. Also, each MSS acts as the Base MSS for a number of MHs and each MH has one MSS as its Base MSS. This is represented by a link ‘Has MHs as Base’. Next, we present the skeletal RSL type structure of the specification of different classes in the figure.

Figure 2. Class Diagram for MC Environment specifications

Class System consists of all the MHs and MSSs in the system. The system is modelled as a record type System given below. The subtype Sys is defined with the auxiliary function consistent to ensure the well-formedness of the system by specifying certain CCs. We will present some of these CCs in the next section.

The Class Mobile Host consists of a number of tasks and resources. The attributes of the class represents different components of an MH, as given below (details of the fields can be found in [4]).

Having given the type definitions for some entities of the domain, we next specify some CCs found while writing the specification.

5. Specification of Consistency Conditions

While specifying the mobile environment, we identified 45 different CCs at various levels of the specification. Out of these, 12 CCs were identified at the system level. Below, we give some examples of consistency conditions found at various levels of the specification. We used a bottom-up approach by first specifying the modules for basic entities in the system and then specifying the higher level modules. Table 1 gives the number of CCs found in different modules. For each module in the table, the entry ‘CCs’ indicates the number of consistency conditions, ‘Gen’ indicates the number of generator functions, ‘Obs’ indicates the number of observer functions, and ‘Total’ indicates the total number of functions in the module. The entries in column ‘Lines’ indicates the total number of lines in each module. Note that the total number of lines in the specification include the comments, which is approximately 20% of the total. The last column indicates the confidence conditions generated for each module.

5.1. CCs in Lower Level specifications

The collection of resources present in a device (MH or MSS) is represented by a map from the identifier (R.id) to its associated information (R.info). All resources in a device must satisfy all the CCs defined on it. So, Resource is a subtype of Resource_base constrained by the predicate consistent.

Here, msss, mhs and task represent all the MSSs, MHs and tasks present in the system. Some of these tasks may be running in the MSSs and some other in the MHs. res represents the collection of every resource located at different devices (MSSs and MHs) in the system.
Resource = \{ | m : Resource_base \cdot \text{consistent}(m) \} \}

consistent : Resource_base \rightarrow \text{Bool}
consistent(rm) \equiv \text{sharedCons(rm)} \land \text{hoardCons(rm)} \land
\text{hoardCons01(rm)} \land ...

Some of the CCs for the resources in a device are explained below.

CC1: sharedCons: If the kind of a resource is not sharable, then there can be at most one user of the resource.

sharedCons : Resource_base \rightarrow \text{Bool}
sharedCons(rm) \equiv
(\forall r : \text{R_id} \cdot r \in rm \Rightarrow
(\sim \text{is_Sharable(kind(rm(r)))}) \Rightarrow
\text{card}(\text{users(rm(r)))} \leq 1),

CC2: hoardCons: If a resource is hoarded, then it must be hoardable and its location must be an MSS (since hoarding is done from an MSS). There must be at least one task as the user of the resource that has hoarded the resource.

hoardCons : Resource_base \rightarrow \text{Bool}
hoardCons(rm) \equiv
(\forall r : \text{R_id} \cdot r \in rm \land \sim \text{hoarded}(rm(r)) \Rightarrow
\text{is_Hoardable(kind(rm(r)))} \land \text{is_mss_loc(loc(rm(r)))} \land
\text{users(rm(r)))} \neq \{ \}),

CC3: hoardCons01: If a resource is not hoardable, then its hoard status must always be false.

hoardCons01 : Resource_base \rightarrow \text{Bool}
hoardCons01(rm) \equiv
(\forall r : \text{R_id} \cdot r \in rm \land \sim \text{is_Hoardable(kind(rm(r)))} \Rightarrow
\sim \text{is_hoarded}(rm(r))),

Other CCs were found and specified for other modules and are described in [4].

5.2. System Level CCs

Some CCs can only be described at the system level because they involve relationships between system components. For example:

CC4: taskMHDisjointMSS: Every task in the system must be uniquely identified. To make every task in the system unique, we ensure that the set of tasks present in each MH in the system must be disjoint from the all tasks in each MSS in the system.

CC5: baseCons: When an MH resides in a cell, the location information of the MH should be consistent with the location information of the MSS responsible for that cell.

The full set of system CCs can be found in [4]. Having specified some CCs for the model, we next discuss how these CCs were used to find errors in the specification.

6. Confidence in Specification Correctness

In section 3.1, we discussed different techniques to gain confidence in the correctness of the specification. In this section we present how these techniques were applied.

We manually inspected the 859 automatically generated confidence conditions and satisfied ourselves that they were all true. Code for checking them is also included in the prototype implementation (generated by automatic translation). Table 1 gives the number of confidence conditions that were generated for each module in our specification.

<table>
<thead>
<tr>
<th>RSL Module Name</th>
<th>CCs</th>
<th>Gen</th>
<th>Obs</th>
<th>Total</th>
<th>Lines</th>
<th>Cap/Cnd</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESTING</td>
<td>4</td>
<td>5</td>
<td>22</td>
<td>101</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>I</td>
<td>54</td>
<td>1</td>
<td>12</td>
<td>68</td>
<td>268</td>
<td>101</td>
</tr>
<tr>
<td>SYS/MOBICHART</td>
<td>58</td>
<td>1</td>
<td>3</td>
<td>42</td>
<td>43</td>
<td>112</td>
</tr>
<tr>
<td>MH</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>MSSS</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>TOTAL</td>
<td>68</td>
<td>1</td>
<td>12</td>
<td>68</td>
<td>268</td>
<td>101</td>
</tr>
</tbody>
</table>

Table 1. Number of Functions and Lines in Each RSL Module.

To generate and run the test cases, we prototyped the system specification by doing a simplified refinement of all abstract types in it. We then used the RSL to SML (Standard ML of New Jersey, http://www.smlnj.org) translator provided by the RAISE tool [20] to run some test cases. We used a bottom-up testing strategy by first testing each lower level module (unit testing) in the specification and then testing the higher level modules (integration and system testing). The test cases were written by hand in RSL (which supports test case definition): the tool does not yet support automatic test case generation, but it can show expression coverage of the test cases. At least one test case was used to test each function in a module (function coverage). Some examples of test cases used at the system level are:

test_case
[test1] init(); consistent(s), /*Result: true*/
[test2] init(); consistent(taskleave(t1, s)), /*Result: true*/

init is a function we wrote to initialize the system with a number of MSSSs, MHs, tasks, etc. The test case [test1]

6
is used to ensure that this initial state is consistent. When we used the test case \texttt{[test1]} for the first time, the result was \texttt{false}: our initial state was not consistent.

Test case \texttt{[test2]} calls \texttt{taskleave} in the initial state and checks that the resulting state of the system is consistent. The precondition of \texttt{taskleave} is also being checked, of course, because of the inclusion of code to check confidence conditions. Other generators defined at the system level were tested for consistency in a similar fashion.

One error we found during testing was the omission of a CC. We noticed a typo error after testing had started, and then realised that the typo should have caused an inconsistency, but none had been reported. Clearly the defined CCs were not sufficient to capture all the consistency requirements. As a result, we found and added the required CC. This is an example of (accidental) mutation testing. Mutation testing is a technique for checking the adequacy of test cases. Errors are introduced in the specification (or program) to produce “mutants” and then test cases are run on all mutants and the original specification (or program) to find if the mutations are detected by these test cases [24].

In fact we found a number of inadequacies and other errors in our CCs during testing: see section 7.

Although we had inspected the confidence conditions and thought they were all true, we did get some messages like “Result of function... not in subtype” which indicates a confidence condition failure, and had to make the appropriate changes.

We also found that we needed more care in one place, when a top-level function is modelled in terms of a sequence of lower-level ones, to ensure consistency of the intermediate states.

## 7. Complexity of Specification

Table 1 gives an idea of the overall size and complexity of the specification of the MC environment and different Mobichart services [4].

We used a total of 319 test cases on different modules and found 28 errors in the specification. Out of these errors, 8 were typographic (typo) errors and 20 were conceptual/logical errors. A typo is a discrepancy between the author’s intention and the specification. Other errors (conceptual/logical) represent an inadequate or mistaken understanding of the problem. These are potentially more damaging than typos because they are less likely to be discovered during development. Table 2 gives the number of test cases and errors found while testing each RSL module.

Table 3 summarizes the categories of changes made at different levels of specification to fix the errors during testing. ‘New CCs added’ indicates the number of new CCs found necessary during testing. ‘CCs Changed’ gives the number of constraints changed in different CCs. Similarly, ‘Functions Changed’ indicates the number of functions (other than CCs) where we had to add new statements or remove old statements.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Test Cases</th>
<th>Errors Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources</td>
<td>63</td>
<td>0 4</td>
</tr>
<tr>
<td>RESREQBAG</td>
<td>10</td>
<td>0 0</td>
</tr>
<tr>
<td>RASSIGN</td>
<td>9</td>
<td>0 0</td>
</tr>
<tr>
<td>HOARD</td>
<td>11</td>
<td>0 0</td>
</tr>
<tr>
<td>TASK</td>
<td>43</td>
<td>0 1</td>
</tr>
<tr>
<td>MSSS</td>
<td>45</td>
<td>2 1</td>
</tr>
<tr>
<td>MHS</td>
<td>48</td>
<td>1 8</td>
</tr>
<tr>
<td>SYS/MOBICHART</td>
<td>90</td>
<td>5 6</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>319</strong></td>
<td><strong>8 20</strong></td>
</tr>
</tbody>
</table>

Table 2. Number of Test Cases and Errors found while Testing Each RSL Module. ‘Con.’ stands for Conceptual Errors.

<table>
<thead>
<tr>
<th>Type of Change Made</th>
<th>No. of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typographic</td>
<td>8</td>
</tr>
<tr>
<td>New CCs Added</td>
<td>4</td>
</tr>
<tr>
<td>CCs Changed</td>
<td>5</td>
</tr>
<tr>
<td>Sequence of Function Calls</td>
<td>1</td>
</tr>
<tr>
<td>Functions Changed</td>
<td>10</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>28</strong></td>
</tr>
</tbody>
</table>

Table 3. Faults found during Testing using 319 test cases.

## 8. Conclusion

Finding problems in the early stages of the software development cycle is critical. While developing a complex system, a system designer needs to consider two kinds of errors in specifications: (a) failure to specify what is wanted (validation problem) and (b) failure to specify what was intended by the specifier (verification error). A specification is valid if it represents the user requirements satisfactorily. Testing a specification can help with validation if the user looks at the test cases, but proof does not typically help with validation.

Our work here is concerned with finding and specifying the CCs for the domain, not checking the consistency of specifications that are derived using different methods as done in [13, 14, 15, 8]. The importance of finding and specifying CCs in a specification is discussed. Different methods were used to gain confidence in the correctness of the
specification, like confidence conditions, and specification-based testing using test cases. Some observations on the faults found during testing and their causes have been reported. It is interesting to observe that most of these errors were found because of the CCs. Results of some test cases did not give a consistent system. It implies that, if the CCs had not been taken into account, these conceptual errors would have gone unnoticed. The most interesting finding, we think, is that out of 20 conceptual errors, almost half of them are related to errors in the CCs. This is surprising since one would naturally expect consistency (a function of one state) to be easier to get right than change in the specification is more useful, since they generally go unnoticed and the effects of such faults can be catastrophic during later stages of system development.

References


