Research Report 4/96

Designing Reliable Systems from Reliable Components Using the Context-Dependent Constraint Concept

by

Peter Molin

Department of
Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HK/R-RES—96/4—SE

Designing Reliable Systems from Reliable Components Using the Context-Dependent Constraint Concept

by Peter Molin

ISSN 1103-1581
ISRN HK/R-RES—96/4—SE

Designing Reliable Systems from Reliable Components Using the Context-Dependent Constraint Concept

Peter Molin
University College of Karlskrona/Ronneby
Department of Computer Science and Business Administration
Soft Center, S-372 25 Ronneby, Sweden
Peter.Molin@ide.hk-r.se

Abstract

The problem of composing a system using well-behaved components is discussed. Specifically, necessary conditions for preserving the behaviour in a system context are analysed in this paper. Such conditions are defined as Context-Dependent Constraints (CDC). A non-formal approach is taken based on common system integration errors. It is suggested that the identification and verification of CDCs should be part of any development method based on component verification. The CDCs can also serve as an aid for designing reliable and maintainable systems, where the goal of the design process is to reduce the number of CDCs.

1. Introduction

A major and basic problem in software reliability engineering is the question of how to design large complex software systems that are demonstrably reliable and maintainable. Demonstrability is a property that makes it possible to convince the developing organization, the customer, or in some cases, a third party software assessment organization that requirements can, and will be, met with sufficient confidence.

Component-based software engineering is a common approach for managing large complex systems where the systems are built using reusable well-tested or certified components. The basis of component-based software engineering is the well-known principle of decomposition, where a system is divided into simpler components. The components are designed and developed, and finally re-composed. Such a system development paradigm is founded on the *composability principle*:

A system consisting of components behaves correctly if all its components behave correctly, and if the interfaces between the components are correctly defined and used.

When a large component-based system is verified, a common complexity-reducing approach is to verify each component, integrate the components and continue with system testing. The assumption is that the system behaviour is related to the correct and verified behaviour of its components. It is therefore important to establish the *exact conditions* under which the composability principle is valid. The *composability problem* is the problem of identifying such conditions and finding ways to verify these.

These conditions can be viewed as a number of context-dependent constraints, which must be verified in order to imply composability. It is suggested in this paper that the CDCs must be defined during the design process, and verified as a means of establishing composability. Instead of trying to investigate formally and prove the conditions required for composability, a different approach is taken in this paper. Typical integration errors from different program categories serve as a basis for defining a number of CDCs. It is suggested in this paper that the process of defining and verifying the CDCs contribute to a higher demonstrability of the software system.

The paper further argues that there is a correlation between demonstrability and reliability, in the sense that a system that is easy to verify is probably more reliable than a system which is more difficult to verify.

Finally, it is proposed that demonstrability should be an important objective of the design process, thereby probably increasing the reliability of the system.

The paper starts with a discussion of how the composability problem is related to maintainability, reliability and reusability. The following section defines the CDC, and shows how the CDC notion is related to the composability problem. The subsequent two sections contain suggestions as to how CDCs can affect the verification process and the design process respectively. The paper concludes with three examples, a section on related work and conclusions.

2. The composability problem

The purpose of this section is to show how demonstrability, reliability, reusability and maintainability of large software systems are closely related to the composability problem.

The problem of reliable composition. It is very difficult to demonstrate the reliability of large complex systems by extensive system testing. An alternative approach is to rely on component testing or verification, and then apply the *composability principle* i.e. to demonstrate the reliability of the composed system based on the demonstrated reliability of the components [1]. Such an approach touches the problem of reliable composition, i.e. what are the address the composability problem in a systematic way? Unfortunately, it can and has been proven (see subsection 7.1), that for certain categories of programs it is impossible to predict system behaviour from the black-box behaviour of its components. The implication of this statement is that such programs must either be verified by system level testing and not by component testing (which may be unrealistic and against the very idea behind the decomposition principle) or, the actual source code implementation of the components must be present for static system verification. Such program categories are, for example, concurrent systems with shared memory, which are notorious for being difficult (compare Boehms productivity factors for concurrent systems [2]). The larger and more complex a system becomes, the more it must rely on the composability principle. It must therefore be demonstrated that the behaviour of the composed system is directly related to the behaviour of its components.

Maintenance and composability problem. Maintenance of a component-based software system can be seen as a repeated process of producing new or modified systems. Each new system is composed of some reliable, and some modified components. The reliable components are those that behaved correctly in the previous version of the system. After verification of the modified components it is important to make an impact analysis of the modifications. Will the modified components affect other parts of the system in an unpredictable manner? How much testing and verification are necessary at system level?

Reusability and composability problem. In the last couple of years, efforts have been made to establish component libraries, for example, the Software Components from Booch [3]. It seems as if the first really successful commercial use of components is the Microsoft VBX/OCX component market [4]. Certification of such components is very important. An exact description of the context in which the component is supposed to operate, and other possible restrictions on composability, are a major concern along with the question of component correctness.

3. Context-dependent constraints

This section defines the CDCs and gives a number of examples of CDCs. The CDCs in this section are derived from a list of integration test errors and system test errors found in Bezier's work on testing [5]. An attempt to classify the CDCs according to their origin is made.

The "design-by-contract" principle [6], where each component has well-defined preconditions and postconditions, is used as a basic model in this section. It is assumed that the contract specifications of the components are correctly defined, i.e. the behaviour of the composed system can be deduced from the component specifications. It is further assumed that each component has been implemented correctly in a programming language, for example, the component has been formally proven correct. Any problem that can cause the composed system to fail to behave as intended is defined as a CDC:

A context-dependent constraint of a component (CDC) is a constraint on different parts of the system necessary for correct component behaviour, and which cannot be validated by the input/output behaviour of the component.

The categories specified in table 1 is defined as follows: the *execution environment* is the combination of compiler, run-time system and hardware interpreting the component program; the *software environment* consists of all other components required for system composition. *Local verifiability* means that the constraint can be verified locally without any knowledge of execution or software environment.

Table 1: Constraint categories

CDC Type	Constraint target	Local verifi-ability
Execution environment constraints (EE)	Execution environment	No
Software resource con-straints (SR)	Software environment	No
Hardware resource con-straints (HR)	Execution environment	No
Preconditions (PC)	Software environment	No
Design constraints (DC)	Component	Yes
Integrity constraints (IC)	Complete system	No

There are six kinds of context-dependent constraint categories. The first category comprises constraints on the execution environment, the second and third category comprise hardware and software resource constraints, the fourth constraint category refers to precondition verification. A fifth category comprises design constraints on the component itself, for example, prohibiting global variable access or rules regarding resource allocation. The sixth category comprises integrity constraints, i.e. rules applicable to the complete software, e.g. deadlock or re-entrance avoidance. It is sometimes difficult to differentiate between design constraints and integrity constraints. One such example is deadlock avoidance, which could be considered an integrity constraint, i.e. the final software is analysed and it is determined whether the system is deadlock free or not. Another approach is to enforce specific design constraints on all components, e.g. that resources must be allocated in a specific order in order to avoid deadlocks. In this case the integrity constraints are still present but the verification is transformed to verification of design constraints.

The CDC examples in the following subsections are presented as problems. All the problems may however be translated into an equivalent constraint formulation. For example, the re-entrance problem could be formulated as a constraint: "Procedures in this component may not be called in a re-entrant way."

3.1. Execution environment constraints

The first CDC category is related to the execution environment, i.e. the combination of compiler, target hardware and operating system. Correct component behaviour is dependent on certain execution environment properties. Typical examples are hardware dependencies or indeterministic semantics of the chosen programming language. The source code is often correct but the mapping to a particular execution environment may affect the correctness of the component. The underlying reason for such problems is the discrepancy between a programming language model and the interpretation made by the actual hardware, compiler and operating system. Programs are often mathematically defined but executed in an environment with slightly different properties. CDCs are one way to express the exact requirements on the execution environment. Following are some examples of potential problems:

- Specific numeric precision required.
- Specific numeric accuracy required.
- Dependencies on bit or byte layout of the target machine.
- Dependence on hardware timing properties.
- Dependence on specific compiler decision when program semantics are undefined, such as evaluation order dependence.
- Dependence on specific scheduling decisions in the operating system, such as time sharing.
- Timing dependence on OS/compiler handling.

3.2. Resource constraints

Another category of CDCs is related to the fact that all components in a system share common resources such as memory and CPU execution time. The problem with resource sharing can be divided into two subproblems: the first problem is how resources are managed; and the second problem is related to whether or not there are enough resources available for the component. The following list contains typical resource constraints:

- Required stack memory.
- Required dynamic memory.
- Required shared operating system resources (buffers, messages).
- Required CPU time (fairness, starvation, priorities).
- Requirements regarding other application specific resources.

There are always implicit context-dependent constraints on components regarding memory and CPU utilisation.

Figure 1 shows the traditional view of system composition, concentrating on component behaviour and interfaces between components.

.

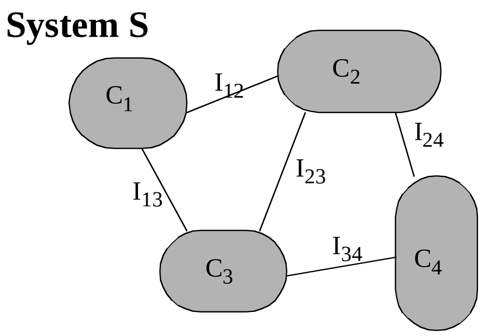


Figure 1 : Traditional system view.

Instead of the traditional view of software composition, a new resource dimension is proposed in figure 2 which shows resource sharing.

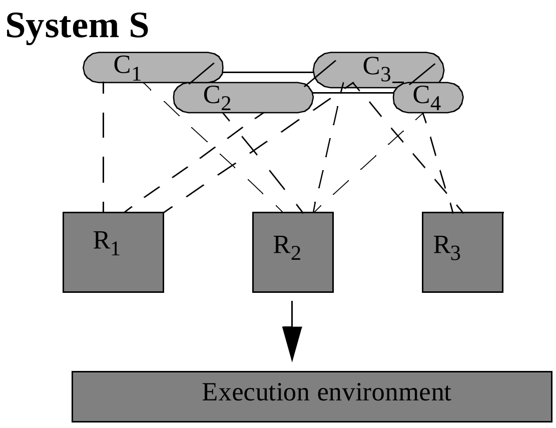


Figure 2 : A different view including shared resources.

3.3. Design constraints

When resources are shared, some kind of resource protocol must be defined, or restrictions on how shared resources should be accessed or allocated. Such restrictions are often formulated as design constraints:

- Shared variable access protocol violation, mutual exclusion, semaphores.
- Shared resource management, resource access protocol violation.
- Side effects, where components may access a global variable directly and at the same time refer to the same variable through a reference parameter.

3.4. Integrity constraints

Contracts defined by Helm et al. [7] introduce a notion of contract between abstract components, i.e. constraints covering several components which cannot be verified locally. Other integrity problems are:

- Deadlock avoidance.
- Re-entrance avoidance.

3.5. Consequences of CDC violation

Before discussing further how to verify systems based on the CDC concept, it is necessary to study the consequences of CDC violation. When a CDC is violated there are two possible consequences. The first case is when the component itself detects the problem and returns with an exception or indicates the problem by other means. Thereafter, the problem must be handled on a higher level. This could be considered as normal behaviour specified in the component specification, but it may be important to verify when the component actually fulfils its primary task. The second case is when a CDC violation results in an incorrect result, or some other undesirable effect. One example could be when the available stack is not sufficiently large; other data structures may be overwritten with unexpected consequences.

4. CDC impact on the verification process

For the discussions in this section a simplified design model has been adopted. Design is considered equivalent to a division of the system into well-specified components. A specification for a component consists of two parts: the behavioural specification (input/output specification or pre- and post-conditions) and the context-dependent constraints. The traditional point of view is to assume or require that the complete specification is defined when the system is decomposed, implying that full composability is assured at the design stage. Further development includes component implementation and verification against the complete component specification.

In this section, a more pragmatic approach is adopted in which it is recognized that the CDC-part of the specification may not be completely defined during the system design phase. Consequently, composability issues must be verified during component implementation. Table 1 shows an example of how some constraint categories are defined later in the development process and when corresponding composability verification is made. The table refers to constraint categories defined in table 1. Behavioural specification and verification have been omitted from the table. Note that the table reflects only one suggestion of when specification and verification shall take place. The only rule is that specification must take place before verification.

The CDC notion can be incorporated into most methods by adopting the following steps:

1. Include important CDCs in the specification of a component.
2. Identify all CDCs of an existing component implementation.
3. Verify the input/output behaviour of the component, i.e. decide whether the postconditions are true given that the specified preconditions or CDCs are valid.
4. Verify the design constraints for the component.
5. Verify systematically all other CDCs for all components once the complete system context is known.
6. Proceed with integration and system verification.

Testing can be used for steps 3 and 6. It is possible to include dynamic checking of most CDCs when a component starts its execution. If the CDC is not valid, an exception can be raised as in Eiffel [8].

Table 2: CDC specification and verification.

Development phase	Component specifications	Component verification	Composability verification
System design	IC, DC, PC		IC, DR
Component design	SR		SR
Component coding	EE, HR	DC	PC
System integration		EE	HR

An alternative verification approach is the more traditional one, in which the CDCs are implicitly verified by the system test. However, a more systematic CDC verification phase has several benefits. It increases the demonstrability of the system and directly locates faults without time-consuming debugging. In some cases, testing is not at all appropriate, e.g. finding shared variable problems or potential deadlock situations in concurrent systems.

5. CDC reduction as a design principle

The CDCs can serve as an instrument for improving the system design from a reliability point of view. In the first subsection, the hypothesis that a high degree of demonstrability implies higher reliability of the system is presented. In the second subsection it is explained how removing CDCs, or at least decreasing the verification effort of CDCs, can improve the demonstrability of the system. This approach is proposed as a design principle.

5.1. The demonstrability hypothesis

The hypothesis can be stated as “A highly demonstrable system implies high reliability”. or in other words, if it is easy to demonstrate the correctness of a system, it is probably also reliable.

Bezier [5] mentions that he normally makes an outline of what and how the software will be tested before the software engineers start design and implementation. The result is a considerable improvement in quality of the resulting software.

Mills et al. [9] express this as: “If a program looks hard to verify, it is the program that should be revised not the verification. The result is high productivity in producing software that requires little or no debugging.”

Apt and Olderog [10] state: “the smaller the grain of atomicity, the more realistic the program” and “the larger the grain of atomicity, the easier the correctness proof of the program” where atomicity is the “size” of the smallest atomic instruction. If updates are an atomic operation, there will be no update conflict errors. It is also plausible that large grain atomicity programs produce potentially fewer errors, simply because there are less error possibilities.

The process of designing highly demonstrable systems is an effort of reducing the number of possible execution paths in the program, reducing constructs which are potentially indeterministic, simplifying the design and thereby eliminating many error sources.

5.2. Removing CDCs

CDCs play an important role in the demonstrability of the system. If all CDCs are identified and verified, it is possible to relate the system behaviour to the behaviour of its components. Some of these constraints are difficult to verify, and since they often involve aspects of the whole system, they need to be verified every time a modification is made to the system. If the system and its components are designed in such a way that the CDCs disappear, or at least become easier to verify, the demonstrability of the system has increased. For example, if the system is designed in a concurrent way, one category of CDCs is constraints on shared variable access. If the system could be designed without shared variables, and without adding other constraints, the system has become more demonstrable.

It is important to spot the difficult CDCs early on, in the design phase. The systematic reduction of CDC verification effort is a concrete way of implementing the general principle of designing for verifiability or demonstrability. The following system design steps are thus proposed:

- During system design, identify potential CDCs of the components.
- Establish design rules or design constraints on all components, minimizing the verification effort of CDCs.
- During component design and implementation, identify all the CDCs of the component.
- Modify the design of the components in a way that reduces the corresponding CDC verification effort.

The strategy is to transfer effort systematically from verification, testing and debugging to design. If the degree of reusability is high (e.g. because the anticipated number of releases is high or the components are intended to be used in many different systems), it is clear that the pay-off is high. The ultimate, but impossible, goal is to have certified components that work in any context. Such components can be freely combined, thus producing systems that behave as intended in a demonstrable way.

6. Three examples

Three examples are presented in this section, two small and simple components, and one example relating to a complete system design outline.

6.1. A simple component

The following is an example in C++ of a component specification:

```
int Factorial (int n) ;
```

Precondition: n>=0
Postcondition: Factorial = n!

Using the proposed development method it is necessary to identify the CDCs of the actual component implementation. The following function is one possible implementation:

```
int Factorial (int n) {
  if (n == 0) return 1;
  return n * Factorial(n - 1);
}
```

The above function is a correct formulation in the programming language model, but the actual execution of Factorial (30) fails on most existing machines. One constraint is that stack memory, with a size proportional to *n*, must be available in order to guarantee correct execution. The exact requirement is, of course, dependent on the way the compiler generates code. Assume each stack frame requires 20 bytes including return address and the parameter *n* itself, and that the compiler does not perform tail-recursion removal, the constraint could be formulated as:

$$stack\_size \geq 20 \cdot \mathrm{n}$$

Furthermore, the result must be smaller than the largest *int* value, i.e. only parameters that are small enough could be handled correctly:

$$\mathrm{n}! \leq MAXINT$$

When the two context-dependent constraints are known, they must be verified, and both constraints require knowledge of the actual data used in the system. The first constraint, a resource constraint, is costly to verify since it involves knowledge of stack requirements for all components. The second constraint, a precondition constraint, is easier to verify, but still requires a verification of the actual parameters.

Another illustration of the use of CDCs is when the component is bought from a subcontractor. The example shows that it is not sufficient to require third party assessment, formal proofs or full test coverage of the component in order to guarantee correct behaviour in the final system, since the CDCs must be documented in order to assure correct component behaviour in the intended environment. When different component implementations are compared, not only must price and verified functionality be considered, but also resources required as well as the cost of CDC verification.

The developing organisation can use the CDCs as one means of evaluating some aspects of quality of a component. The previously mentioned component implementation can be judged to have a too costly CDC. An alternative solution is thus examined which exhibits simpler CDCs:

```
int Factorial (int n) {
  int Result = 1;
  for (int i = 0; i < n; i++)
    Result = Result * i;
  return Result;
}
```

Assuming that *Result* and the loop counter each requires 4 additional bytes, the corresponding CDCs become:

$$stack\_size \geq 28$$

$$\mathrm{n}! \leq MAXINT$$

The first constraint has become static, i.e. it can be verified without reference to actual execution data. This implementation can thus be judged as better than the first implementation.

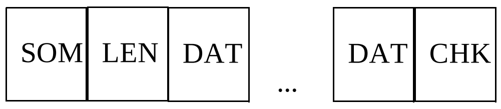
The next problem is to investigate the consequences of a precondition violation. As previously mentioned in subsection 3.5, there are two possible consequences: the first is that the component itself checks and detects the problem and notifies the client either by means of an exception or by returning some error code. The second was that the constraint violation is undetected and may corrupt data. By moving one of the constraints to run-time, thereby changing the definition of the component's interface, we have removed one constraint, but have put a burden on the calling side:

```
int Factorial (int n) {
  int Result = 1;
  for (int i = 0; i < n; i++) {
    if (MAXINT / i <= Result) return 0;
    Result = Result * i;
  }
  return Result;
}
```

There is obviously a performance trade-off between a run-time check of the constraints or a check at system build-time. This example shows how CDCs can be part of the verification process, and how they can be used to evaluate software components.

6.2. Protocol example

The next example is a function reading a simple message with the following specification:



where:

SOM	***Start of Message***
LEN	***Number of data bytes***
DAT	***Data byte***
CHK	***Checksum on LEN and DAT bytes***

Following is an implementation of the function in C++:

```
Status ReadMessage (unsigned char *
                   Buffer) {
  unsigned char c;
  int length, checksum ;
  Status ReturnStatus = ERROR;
  c = Read();
  if (c == SOM) {
    c = Read();
    if ( c <= MAXLENGTH ) {
      length = c ;
      for (int i = 0; i < length; i++) {
        Buffer[i] = Read();
      }
      Buffer[length] = 0;
      checksum = CheckSum(Buffer,length)
      if (checksum == Read()) {
        ReturnStatus = OK;
      } else CheckError++;
    }
  }
  return ReturnStatus;
}
```

A number of CDCs can be identified for the function:

- The component requires one thread with blocking wait ability.
- No other thread may call the function *Read*() between the *Read*() calls in this function.
- No other thread may have write access to the global variable *CheckError*.
- While this function is executing, no other thread may call it.
- The functions *Read*() and *CheckSum*() must not call this function.
- The *Buffer* must have space for MAXLENGTH characters.

It is clear that none of the above constraints can be checked by unit testing. At the same time it is obvious that they must all be satisfied in a reliable or trustworthy system. Possible constraint violation may be detected by system testing, but system testing does not really address demonstrability at all.

There are in principle two ways of achieving demonstrability in this case. Either introduce an explicit CDC verification phase when composing the system, probably relying on code inspection; or redesign the components or system in order to remove all these constraints, possibly by introducing design constraints.

Clearly, most of the above problems would not have occurred if the component had been designed by an experienced software engineer. The objective is, however, to discuss how to detect such problems in the development process.

6.3. Embedded system design example

Assume that the task is to design an embedded complex real-time, single processor system with high reliability and demonstrability requirements. Furthermore, assume that several similar versions of the target system have been planned, and that there will probably be several versions with improved functionality released for each target system.

Since this is a typical example where reliability, demonstrability, maintainability and reusability are of great importance, the CDC concept can be used to improve the system design. Composability is stated as the most important design principle, implying that CDCs must be eliminated or reduced as much as possible. The following design rules, which eliminate most of the CDCs mentioned in the previous section, are thus postulated:

- No fine-grain concurrency allowed.
- Components should, as much as possible, be self-contained, and not call other components.
- No use of dynamic memory.
- Necessary system resources are given to the components as input parameters, instead of letting the components themselves request these.
- Each component must be written in a non-blocking way with a well-defined maximum execution time.

These design rules may require significantly greater design effort than in the traditional way, for example, by using a real time kernel, but the benefits are important. Such a system can be considered fully composable, with only few CDC verifications left: after thorough component testing, system testing thus probably exposes few, if any, surprises. Reliability prediction can rely on component testing because of the composability property. Whenever components are changed, there are few other components affected, and system re-verification can be based on testing of the modified components. More effort is spent on a constrained but demonstrable design, instead of complex and time-consuming system testing, verification and debugging. This design effort is valuable for subsequent releases of new systems, but most of the testing and verification effort must be repeated for every new system. The increased design effort must be compared with the number of planned system releases times the gain of lower system verification effort for each release. Moreover, the resulting system probably contains fewer errors.

An advantage as regards management is that possible schedule overruns appear during the early design phase, instead of during the late system integration phase.

The example in this subsection is somewhat contrary to current trends in embedded systems development, but the point is that the example shows that there is a trade-off between the use of simple and elegant methods such as concurrency, and other attributes such as maintainability and reliability. An example of a project developed in the spirit of this example is described in an experience report [11].

7. Related work

7.1. Solving the composability problem with the help of formal methods

Some work has been done on the subject of verifying a composed program. For example, Apt and Olderog [10] have shown for a simple language that composed program behaviour can be deduced from the input/output behaviour of its components. Furthermore, they have shown that the behaviour of a disjoint parallel program is determined in a simple way by the input/output behaviour of its components. A disjoint parallel program is defined as a program where its process components have only read access to shared variables. Their proof also requires “fairness”, meaning that each component is activated infinitely often if it is not terminated. When the discussion includes true shared variables (read and write access) they report a remarkable (their words) result. The behaviour of a parallel program with shared variables **cannot** be determined by the input/output behaviour of the components. Instead, full information about the computation of all its components is needed. Furthermore, they show that a certain degree of atomicity is required to be able to draw any firm conclusions about the behaviour of the composed program.

CDCs are closely related to the rely-guarantee concept defined by C. B. Jones [12]. A component implementation must, in addition to the post-condition, fulfil the guarantee-condition and may depend on the rely-condition. The CDCs are a more general concept, and suitable for less rigorous development models.

The drawback of using formal methods for proving large program systems is the well-known fact that it is cumbersome work and practically very difficult due to the simple fact that there could be errors made in the complicated proving process [13].

Finally, CDCs referring to the execution environment is closely related to the machine-dependent axioms defined by Hoare [14] in one of the first attempts to prove formally computer programs.

7.2. Programming languages addressing CDC issues

Problems related to mapping from a mathematical programming language to interpreting hardware have been addressed in some programming languages. There are two main ways of solving this discrepancy: one is to define the language in pure mathematical terms and make sure the implementation fulfils the specified model. One example is Erlang [15] with mathematical integers without any upper bound. The other approach is to define an exact semantic of the operations available, and make the programmer responsible for mapping from a mathematic language to the programming language. One example is Ada [16] where fix point numbers are defined with specific precision information. Another example is Java [17], where floating point multiplication is defined as being non-associative!

Ada 95 [16] adopts a pragmatic approach to these problems, and displays a number of ad-hoc features intended to reduce composability problems and context dependencies. It is recognized that the use of tasking can potentially introduce deadlocks and racing conditions. Ada 95 thus provides the pragma “Restriction” which is used to impose a restricted set of the language in order to facilitate system verification. Pragma “Atomic” can also be used for shared variables to assure consistent access, and thereby increase the atomicity.

7.3. Local certifiability

The notion of CDC is very closely related to the local certifiability concept defined in the work of Weide and Hollingsworth [18]. Local certifiability is viewed as a fundamental property for any software development approach that is intended to be scalable. Correctness and composability are properties that must be locally certifiable for such an approach.

The CDCs represent a slightly different point of view, in which it is recognized that complete local certifiability is impossible: there are always some resource constraints on the intended system environment. Furthermore, CDCs are both used in the system verification process and in the component design process, where the goal is to achieve a high degree of local certifiability.

7.4. Component libraries

Booch component library [3] addresses many of the CDC related issues. Each component is implemented in different ways, with different resource requirements or different concurrency handling. Using CDC terminology, Booch provides a number of equivalent implementations with respect to input/output behaviour, but with different sets of CDCs. The most suitable component is chosen, with regard to the system and execution environment. CDCs are a generalisation of this approach.

8. Future work

The notion of CDCs needs more formal definition. Another problem is to define a systematic way to derive all CDCs from a given component, and to provide evidence that all CDCs have been found.

The problems of CDCs are also applicable to the case when an object is seen as a system and its methods are seen as its components. Modification of a class is primarily made by subclassing it, and modifying some of its methods. The composability problem can be defined as finding the conditions under which the modified and verified methods imply a correctly behaving object. As a consequence, a remaining problem is how to design an object suitable for “reliable subclassing”, i.e. it must be possible to predict the subclassed object’s behaviour after verification of the modified methods.

Object-oriented frameworks [19] can also be seen as components. An instantiated application can be seen as a system. The CDC concept can be used for establishing the required constraints on the instantiation. The problem, in this case, is how to design a framework in such a way that it supports “reliable instantiation”, i.e. to find a way to predict application behaviour based on verification of the instantiated parts with no knowledge of framework internals.

9. Conclusion

The fact that composability is absolutely necessary for large-scale software engineering has been stressed in this paper. The proposed CDC concept can serve as a valuable aid for verifying component-based software systems where CDC identification is required as well as definition of an additional verification phase between component or unit testing and system testing. In this additional phase the CDCs are verified. Furthermore, the reduction of the number of CDCs or their verification effort, is proposed as a systematic approach for designing composable, demonstrable and potentially reliable systems.

10. Acknowledgements

I would like to thank Lennart Ohlsson, Jan Bosch and Claes Wohlin for helpful advice in this work as well as some of the anonymous ISSRE reviewers for their suggestions.

References

[1] Wohlin, C., Runeson, P., "Certification of Software Components", *IEEE Transactions on Software Engineering*, Vol. 20, No 6, pp. 494-499, June 1994.

[2] Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, 1981.

[3] Booch, G., Vilot, M., "The Design of the C++ Booch Components", *ECOOP/OOPSLA'90*, special issue of SIGPLAN Notices, Vol. 25, No. 10, pp. 1 - 11, October 1990.

[4] Imagicom Component Resource, "http://www.imagicom.com"

[5] Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold, 1990.

[6] Meyer, B., "Applying 'Design by Contract'", *IEEE Computer*, Vol. 25, No. 10, pp. 40--51, October 1992.

[7] Helm, R., Holland, I. M., Gangopadhyay, D., "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems", *Proceedings of ECOOP/OOPSLA'90*, pp. 169-180, 1990.

[8] Meyer, B., *EIFFEL, The Language*, Prentice-Hall, 1992.

[9] Mills, H. D., Dyer, M., Linger, R., "Cleanroom Software Engineering", *IEEE Software*, Vol. 4, No 5, pp. 19-24, September 1987.

[10] Apt K. R., Olderog, E.-R., *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991.

[11] Molin, P., "Applying the Object-Oriented Framework Technique to a Family of Embedded Systems", *Research Paper 19/96*, University College of Karlskrona/Ronneby, 1996.

[12] Jones, C. B., "Accommodating Interference in the Formal Design of Concurrent Object-Based Programs", *Formal Methods in System Design*, Vol. 8, No. 2, pp. 105-122, March 1996.

[13] Rushby, J., Formal Methods and the Certification of Critical Systems, CSL-93-7 (technical report), *SRI International*, 1992.

[14] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", *Communication of the ACM*, Vol. 12, No 10, pp. 576-583, October 1969.

[15] Armstrong, J., Virding, R., Wikström, C., Williams, M., *Concurrent Programming in Erlang*, Prentice-Hall, 1996.

[16] *ADA 95 Language Reference Manua*l, AD A293760, National Technical Information Service, 1995.

[17] Gosling, J., Joy, B., Steele, G., *The Java Language Specification*, Addison-Wesley, 1996.

[18] Weide, B., Hollingsworth, J., "Scalability of Reuse Technology to Large Systems Requires Local Certifiability", *Proceedings of the 5th Annual Workshop on Software Reuse*, October 1992.

[19] Johnson, R., Foote, B., "Designing Reusable Classes", *Journal of Object-Oriented Programming*, Vol. 1, No 2, pp. 22-35, June 1988.