JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, no. 3
Special issue: TOOLS USA 2002 proceedings

"Design by Contract" + "Componentware" = "Design by Signed Contract"[1]

Andreas Rausch, Technische Universität München, Germany

Abstract

The main goal of "Design by Contract" is to improve correctness and robustness of software systems. For this purpose, the interfaces of classes or modules are augmented with precise specifications containing assertions. By means of these assertions, a supplier of a service imposes contractual obligations that his clients have to fulfill.

"Componentware" introduces a new software development paradigm. Systems are no longer implemented from scratch, but glued together from existing components. In this paper, we show why and how the concepts of pure design by contract fail in the context of component-based system development. In order to leverage the vision of design by contract to its full extent for component-based system development, we introduce the new concept of "Design by Signed Contract".

Signed contracts enable us to specify not only what a supplier provides to its environment, but also what a client needs from its environment. Signed contracts guarantee that client needs are satisfied by corresponding properties provided by suppliers. We show how signed contracts can be used for a more precise specification of the composition of component-based systems and a more formal verification of the correctness of these systems. Thereby, software system defects can already be detected and prevented at the specification level.

[1] This work originates from the research project ZEN – Center for Technology, Methodology and Management of Software & Systems Development – a part of Bayerischer Forschungsverbund Software-Engineering (FORSOFT), supported by the Bayerische Forschungsstiftung.

Cite this article as follows: Andreas Rausch: *"Design by Contract" + "Component Ware" = "Design by Signed Contract"*, in Journal of Object Technology, vol. 1, no. 3, Special issue: TOOLS USA 2002 proceedings, pages 19-36. http://www.jot.fm/issues/issue_2002_08/article2

1 INTRODUCTION

The idea of using assertions to check the correctness of programs has been born a long time ago. According to Tony Hoare, in the 1950's none other than Alan Turing already proposed to make a number of assertions from which the correctness of a program can be checked [Hoare81]. In this context the correctness of a program means that the program is consistent with its specification. The well-known notion of a Hoare triple provides a simple mathematical notation for reasoning about the correctness of programs [Hoare69]:

$$\{P\}C\{Q\}$$

In this notation, P and Q are predicates (or functions from the state space to the boolean domain) and C is a command or program. The meaning of the Hoare triple is that any terminating execution of C, starting in a state where P holds (pre-condition), will terminate in a state where Q holds (post-condition). This meaning of correctness is also known as partial correctness.

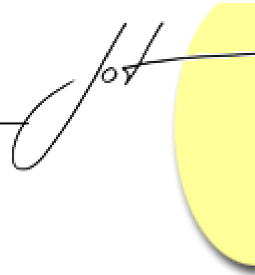
"Design by Contract" (DbC), introduced by Bertrand Meyer in 1987 [Meyer87], was one of the next milestones in the evolution of the idea of using assertions to improve the correctness and robustness of software systems. In DbC, interfaces of classes or modules are governed by precise specifications containing assertions. In the notion of DbC, these assertions define a contract between the client and the supplier of a service provided by an interface.

Three different kinds of assertions can be used: pre-conditions, post-conditions and invariants. A pre-condition states the properties that must hold before an operation is called. A post-condition describes the properties that are guaranteed after the operation is executed. And finally, an invariant is a condition that must be preserved by all operations of a certain instance.

According to DbC, these assertions are specified within the program code. Whenever the program is executed, the assertions can be validated. In case of violated assertions exceptions are thrown. Thus, executing test cases on the system as a whole helps you identifying, analyzing, and finally eliminating system defects.

Nowadays, as systems become more and more complex, component-based software development (CBSD) is to a greater extent applied in industry. CBSD changes the development paradigm – components are for composition. Systems are no longer implemented from scratch, but glued together from existing components.

In order to leverage CBSD to build correct programs we need sophisticated specification and high level programming techniques. On the one hand, we have to specify and realize software components as self-contained units of deployment. On the other hand, we have to specify the composition of those components to component-based systems.



As it turns out, the current concepts of DbC are not sufficiently powerful for these issues. The main reason for this is that the concept of a contract in DbC is actually not a contract, but only a service supplier's "offer" to potential clients. The supplier's "needs" are not completely specified, as they are seen as "implementation details". But in the context of CBSD a component is a self-contained unit of deployment. Therefore, you have to make the needs of a component visible. They must not be hidden as an implementation detail.

In this paper, we show how the vision of DbC can be leveraged to its full extent for component-based systems. For this purpose, in Section 2 we provide a small working example that illustrates the problem of the existing notion of contracts in DbC. Then in Section 3, we enhance the specification techniques of DbC towards "Design by Signed Contract" and show how the problems with pure DbC can be avoided. In Section 4, we finally provide the theoretical foundation of the proposed concepts. A short conclusion and a section about related work rounds up the paper.

2 "DESIGN BY CONTRACT" – APPLIED

A small toy example serves to clarify the general problem of applying DbC in the context of CBSD. Consider a simple production planning system (PPS). The PPS has to schedule and optimize the assignment of jobs to corresponding robots handling these jobs. Each robot can treat only a single job at any time. Each job has to be handled by a single robot. Overlapping jobs assigned to the same robot cause conflicts. The major goal of the PPS is to assign all jobs to robots without a conflict and to minimize the required production time.
As we apply a component-based approach, the PPS is built from existing components. The PPS contains two components: `Job` and `Robot`[2].

The important parts of the specification and implementation of our two components `Job` and `Robot` are shown in Figure 1 and Figure 2. The notation we use imitates the one known from DbC and Eiffel [Meyer97]. Keywords are written in capital letters.

`Job` contains the attribute `assigned` which refers to the robot handling this job. On the other hand, `Robot` has the attribute `scheduled` which refers to a set of jobs it has to handle. Both `Job` and `Robot` provide the method `hasConflict()` to calculate whether they cause a conflict or not. Corresponding to DbC, each method description consists of three parts: The first covers the pre-conditions of the method, which is not required in this example. The second includes the implementation of the method starting with the keyword `DO`. The third – with keyword `ENSURE` – contains the post-condition of the method.

[2] Although `Job` and `Robot` are more objects than components, it keeps the example small but expressive enough to illustrate the problem in general.

```
COMPONENT Job
  assigned : Robot
  start : Time
  end : Time
  INVARIANT interval_non_negative: start <= end
  .
  .
  hasConflict() : Boolean
    DO
      RESULT := False
      FORALL j IN assigned.scheduled LOOP
        RESULT := RESULT OR ((j NOT EQUALS CURRENT) AND
          (start <= j.end) AND (j.start <= end))
      END
    ENSURE
      RESULT = EXISTS j IN assigned.scheduled WITH
        (j NOT EQUALS CURRENT) AND (start <= j.end) AND
          (j.start <= end)
    END
END
```

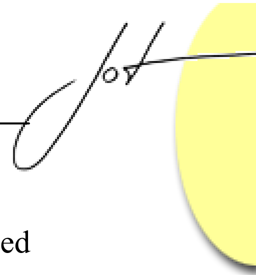
Fig. 1: First DbC version of component `Job`

The post-condition of the method `hasConflict()` of the component `Job` determines whether a job causes a conflict or not. A conflict appears if the assigned robot is scheduled for another job that overlaps with the current one. The implementation of the method is a simple translation of the post-condition into an operational form.

```
COMPONENT Robot
  scheduled : Set(Job)
  .
  .
  hasConflict() : Boolean
    DO
      RESULT := False
      FORALL j IN scheduled LOOP
        RESULT := RESULT OR j.hasConflict()
      END
    ENSURE
      RESULT = EXISTS j,k IN scheduled WITH
        (j NOT EQUALS k) AND (j.start <= k.end) AND
          (k.start <= j.end)
    END
END
```

Fig. 2: DbC version of component `Robot`

The post-condition of the method `hasConflict()` of the component `Robot` calculates whether the robot has a conflict or not. A conflict appears if at least two scheduled jobs of the robot overlap. For the corresponding implementation the already existing method of



the component Job is (re-)used. For reasons of reuse and encapsulation the presented solution seems absolutely reasonable.

Now, we can glue these two components together to implement and deliver the PPS to our customers. Once a system is shipped it usually takes only a couple of months until new requirements come up. In our particular case, we assume that our customers want the PPS to schedule jobs not only for a single robot, but also for a certain number of robots – the jobs they want to manage get more complex. Therefore a new version of the component Job has to be specified, implemented, and finally used within the PPS.

Figure 3 shows this new version of this component. The modified parts are highlighted in gray color. A job can now be assigned to a set of robots with respect to the number of required robots to handle the job. The method hasConflict() has also been modified. Now, a job causes a conflict if there is another job assigned to one of the robots the current job is assigned to, which overlaps with the current job.

The new version of the component Job fulfills the required new features. Moreover, it still fits together with the already existing component Robot. Probably the new version of the PPS will be again glued together, compiled, tested and eventually shipped to customers.

```
COMPONENT Job
  assigned : Set(Robot)
  numberOfRequiredRobots : Integer
  start : Time
  end : Time
  INVARIANT interval_non_negative: start <= end
  .
  .
  hasConflict() : Boolean
    DO
      RESULT := False
      FORALL r IN assigned LOOP
        FORALL j IN r.scheduled LOOP
          RESULT := RESULT OR ((j NOT EQUALS CURRENT) AND
            (start <= j.end) AND (j.start <= end))
        END
      END
    ENSURE
      RESULT = EXISTS r IN assigned WITH
        EXISTS j IN r.scheduled WITH
          (j NOT EQUALS CURRENT) AND (start <= j.end) AND
            (j.start <= end)
    END
END
```

Fig. 3: Second DbC version of component Job

Unfortunately the new version of the PPS has a defect: A robot is expected to signal a conflict if at least two of its scheduled jobs overlap, corresponding to the post-condition of hasConflict() in Figure 2. However, the implementation of the Robot‘s method

`hasConflict()` (re-)uses the `Job`'s method `hasConflict()` which has been modified (see Figure 2). Hence, the behavior of the `Robot`'s method `hasConflict()` has also been changed. A conflict for a robot R1 may now also be signaled if a job J1 scheduled for robot R1 and robot R2 overlaps with a job J2 assigned to robot R2. This behavior violates the corresponding post-condition of the Robot's method `hasConflict()`.

The component `Robot` is no longer correct in the context of the new version of the PPS, although it has not been modified. The implementation is not consistent with the specification (see Figure 2). The resulting defect may cause fatal faults, as for instance the optimizing algorithm of the PPS relies on a correct calculation of the conflicts of jobs and robots. Hence, the core functionality of the PPS is no longer correct.

Of course, this defect could have been detected during the integration test of the new version of the PPS. In order to detect it, a corresponding test case containing proper test data must be available and executed. Usually, new test cases including new test data are only specified and implemented for new functionality. Existing functionality is typically tested with existing test cases in so-called regression tests. As the discussed defect only appears if existing functionality is executed with new test data, it is quite likely that it will not be detected during integration test.

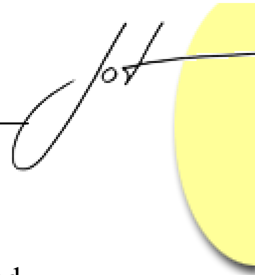
To sum up, applied CBSD means that systems are built from existing components. These components are self-contained units of deployment, but they have to work together to realize the functionality of the system as a whole. Correspondingly, the components of a component-based system rely on each other. The behavior of a single component depends on the "surrounding" components within the component-based system. It depends on the context in which the component is embedded. Hence, the correctness of a component-based system depends on an appropriate "component-mixture".

For instance, if a single component is correct but does not fulfill the needs of the others (like the modified `Job` component), the behavior of other components depending on it may be influenced unintentionally, resulting in software system defects.

Using the concepts of DbC in the way they are used in today's software engineering practice, namely for specification, programming, and testing issues, it is difficult to prevent those system defects. To detect these defects one has to either inspect the implementation or realize and execute a failure-producing system test scenario.

Both options are unacceptable in CBSD. Components are units of deployment and may be delivered by third parties. As you do not have access to the implementation of all components, you cannot inspect all of them. Therefore you still need to realize a complete set of system test scenarios for the system integration test, as the use of correct components does not enforce the correctness of the component-based system built from these components. But as we all know, one cannot identify all required test scenarios. Thus, one expected benefit of CBSD will not be achieved: improvement of system quality by (re-)using quality proven components.

The main reason for this is that DbC "only" guarantees local correctness at the level of objects, classes, or components, but it does not guarantee global correctness when



components are used and combined together. Therefore we have to provide a means for explicit specification of the dependencies between the components of a component-based system.

In the next section we will illustrate – based on our working example – how the concepts of DbC can be improved towards signed contracts which are needed for successful CBSD.

3 DESIGN BY SIGNED CONTRACT

Before we can introduce our new improved specification technique for CBSD, we need a clear understanding of the notion of a component and of CBSD. Instead of presenting our own, we use Clemens Szyperski's definition of a component, which is widely accepted:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (Quotation from [Szype97], page 34)*

These properties have several implications. For a component to be independently deployable, the component needs to be well separated from its environment and from other components. It needs to be sufficiently self-contained. A clear specification of what a component provides and needs is required. This specification has to be delivered together with the component by the component vendor.

The existence of such a specification is crucial for a component to be composable with other components by a third party, the component user. The component user needs an integrated but decoupled specification technique to explicitly describe the collaborations between the components under composition.

We need two kinds of specification techniques for CBSD:

- a self-contained component island specification provided by the component vendor and
- a component composition specification elaborated by the component user.

As shown in the previous section, the concepts of DbC are currently not sufficiently powerful to express the required component island specifications and component composition specifications. Based on our working example we show in the following how the concepts of DbC can be extended with respect to the requirements of CBSD.

For each component a component island specification has to exist. This island specification is structured in two parts. The first contains the provided properties. In our example this section starts with the keyword `PROVIDE`. The `PROVIDE` part is identical with the specifications well known from DbC that have been shown in the previous section. It specifies the properties the component provides to its environment, assuming the environment fulfills the second part, the `NEED` part.

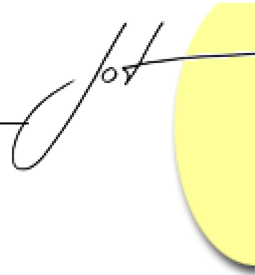
The second part of the specification captures the needed properties of the component, therefore we use the keyword NEED. The NEED part is syntactically identical to the PROVIDE part. It also contains a complete behavior specification based on pre-conditions, post-conditions, and invariants. In contrast to the PROVIDE part it specifies behavior the component expects from its environment. Hence, the NEED part will never be implemented, instead the needed behavior will be mapped to a provider-component during system compostion.

```
COMPONENT Job
  PROVIDE
    assigned : n_Robot
    start : Time
    end : Time
    INVARIANT interval_non_negative: start <= end
    .
    .
    hasConflict() : Boolean
      DO
        RESULT := False
        FORALL j IN assigned.n_scheduled LOOP
          RESULT := RESULT OR ((j NOT EQUALS CURRENT) AND
            (start <= j.end) AND (j.start <= end))
        END
      ENSURE
        RESULT = EXISTS j IN assigned.scheduled WITH
          (j NOT EQUALS CURRENT) AND (start <= j.end) AND
            (j.start <= end)
      END
  END
  NEED
    .
    .
    COMPONENT n_Robot
      n_scheduled : Set(Job)
  END
END
```

Fig. 4: First version of component island specification of Job

Figure 4 contains the component island specification of the component Job. The PROVIDE part of the component island specification is almost identical with the one shown in Figure 1. Only some of the identifiers have been exchanged. Instead, corresponding identifiers from the NEED part of the specification have been used.

For reasons of uniformity and clarity all needed properties of a component start with the prefix "n_". As shown in Figure 4 the component Job needs a component named n_Robot that has an attribute named n_scheduled which contains a set of jobs.



Note, a component island specification is a complete and self-contained specification, all used identifiers are defined. An implementation of such a specification can be independently tested and verified, an important feature for successful CBSD.

Figure 5 shows the corresponding component island specification of the component `Robot`. Again, this specification consists of the two parts `PROVIDE` and `NEED`. The `PROVIDE` part is similar to the one shown in Figure 2. The additional `NEED` part describes the required component `n_Job` including all needed properties.

```
COMPONENT Robot
  PROVIDE
    scheduled : Set(n_Job)
    .
    .
    hasConflict() : Boolean
      DO
        RESULT := False
        FORALL j IN scheduled LOOP
          RESULT := RESULT OR j.n_hasConflict()
        END
      ENSURE
        RESULT = EXISTS j,k IN scheduled WITH
          (j NOT EQUALS k) AND (j.n_start <= k.n_end) AND
            (k.n_start <= j.n_end)
      END
  END
  NEED
    .
    .
    COMPONENT n_Job
      n_assigned : Robot
      n_start : Time
      n_end : Time
      INVARIANT n_interval_non_negative: n_start <= n_end
      n_hasConflict() : Boolean
        ENSURE
          RESULT = EXISTS j IN n_assigned.scheduled WITH
            (j NOT EQUALS CURRENT) AND (n_start <=
              j.n_end) AND (j.n_start <= n_end)
        END
  END
END
```

Fig. 5: Component island specification of `Robot`

Once these component island specifications are finished, the components can be implemented, tested, and shipped to component users. Then, component users glue these components together to implement their envisioned system.

Therefore, component users need a specialized component composition specification technique. This specification technique has to enable component users to explicitly state the behavioral dependencies between the components under composition. This means needed properties of all components have to be mapped to provided properties of other components.

In our example the component user glues the components Job and Robot together to realize the PPS. Therefore, he has to map the needed properties of our two components to corresponding provided properties. For instance in Figure 6, which contains the component composition specification of the PPS, the needed method `n_hasConflict()` of the needed component `n_Job` is mapped to the provided method `hasConflict()` of the provided component `Job` (see gray colored line in Figure 6).

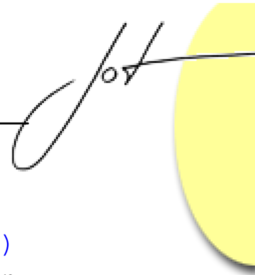
```
SIGNED CONTRACT BETWEEN Component Job, Robot
  MAPPING Job
    n_Robot -> Robot
    n_Robot.n_scheduled -> Robot.scheduled
  END
  MAPPING Robot
     n_Job -> Job
     n_Job.n_assigned -> Job.assigned
     n_Job.n_start -> Job.start
     n_Job.n_end -> Job.end
     n_Job.n_interval_non_negative ->
       Job.interval_non_negative
     n_Job.n_hasConflict() -> Job.hasConflict()
  END
```

Fig. 6: Component composition – signed contract between `Job` and `Robot`

Note, an important feature of the proposed specification technique is that the `NEED` part covers not only the syntax but also behavior – the `NEED` part is more than an "import" statement in common programming languages. For instance, the specification includes a post-condition for the needed method `n_hasConflict()` specifying the behavior of this required method (see Figure 5). Accordingly, the correctnes of the mapping does not require syntactical or logical equality of required and provided pre- and post-conditions, but "merely" suitable implications (see Section 4).

Hence, a component composition specification allows the component user to explicitly state the behavioral dependencies between the components under composition. Such a specification forms a so-called signed contract. Thereby the needed properties of all components of a system are mapped to provided properties of other components of this system. These signed contracts enable tools or at least developers to check and validate at the specification level whether all needed properties of the used components are fulfilled or not.

Consequently, a component-based system is correct if all components are correct and the signed contract of the system is fulfilled. If the signed contract is not fulfilled, at least one component may cause failures leading to system failures. Using signed contracts can help detecting and avoiding system defects at the specification level in advance.



For instance in our example from the previous section the method hasConflict() of the component Job has been modified. Figure 7 shows the corresponding new version of the component island specification of the component Job. (Re-)checking the signed contract from Figure 6 by a tool or a developer shows that this method is used within the component Robot with the synonym n_hasConflict(). The post-conditions of the needed method n_hasConflict() and the provided method hasConflict() are no longer logically equal. The signed contract is broken. The whole system is not correct any more. Applied “Design by Signed Contract” helps you identifying those defects at the specification level and thus preventing system failures.

```
COMPONENT Job
  PROVIDE
    assigned : Set(n_Robot)
    numberOfRequiredRobots : Integer
    start : Time
    end : Time
    INVARIANT interval_non_negative: start <= end
    .
    .
    hasConflict() : Boolean
      DO
        RESULT := False
        FORALL r IN assigned LOOP
          FORALL j IN r.n_scheduled LOOP
            RESULT := RESULT OR ((j NOT EQUALS CURRENT) AND
              (start <= j.end) AND (j.start <= end))
          END
        END
      ENSURE
        RESULT = EXISTS r IN assigned WITH
          EXISTS j IN r.n_scheduled WITH
            (j NOT EQUALS CURRENT) AND (start <= j.end) AND
              (j.start <= end)
      END
  END
  NEED
    .
    .
    COMPONENT n_Robot
      n_scheduled : Set(Job)
  END
END
```

Fig. 7: Second version component island specification of Job

4 FORMAL FOUNDATION

To apply or integrate the presented specification techniques into existing approaches a precise understanding of the basic concepts and notations is required. For these reasons, in this section we elaborate a formal foundation of the concepts introduced in the previous section.

Such a formal foundation usual incorporates two levels: The instance level represents the individual operational units of a component-based system that determine its overall behavior. The specification level contains a normalized abstract description of a subset of common instances with similar properties.

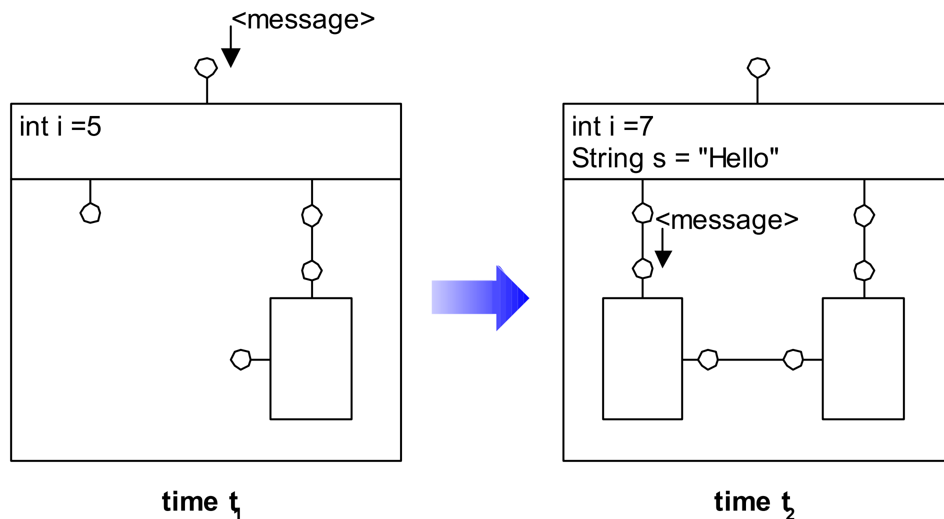


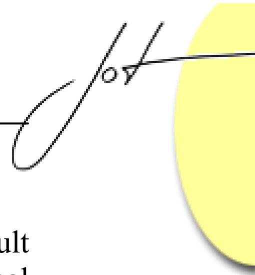
Fig. 8: Instance Level of the Formal Foundation

Although the instance level is the reliable semantic foundation of the specification level, we cannot discuss the complete mathematical definitions for the constituents of a component-based system at runtime – the instance level. This is beyond the scope of this article, as the resulting formulae are rather lengthy.

However, for the formal foundation of the specification level you still need at least a small number of basic concepts from the instance level. In [Rausc00] and [Rausc01] we have already presented a complete formal model for the instance level. In this model we distinguish between system, component, interface, connection, variable, message, and value instances, as shown in Figure 8.

In order to uniquely address these basic elements of the instance level we introduce the infinite set $\mathrm{INSTANCE}$ of all instances:

$$\mathrm{INSTANCE} =_{def} \{\mathrm{SYSTEM} \cup \mathrm{COMPONENT} \cup \mathrm{INTERFACE} \cup \mathrm{ATTRIBUTE} \cup \mathrm{CONNECTION} \cup \mathrm{MESSAGE} \cup \mathrm{VALUE}\}$$



The formal foundation in [Rausc01] is powerful enough to handle the most difficult aspects of component-based systems: dynamically changing structures, a shared global state, and at last mandatory call-backs. For this purpose the behavior of a component-based system is separated into three essential parts:

- Structural behavior captures the changes in the system structure, including the creation or deletion of instances and changes in the connection as well as aggregation structure:

$$\text{ALIVE} =_{def} \text{INSTANCE} \rightarrow \text{BOOLEAN}$$
$$\text{ASSIGNMENT} =_{def} \text{INTERFACE} \rightarrow \text{COMPONENT}$$
$$\text{ALLOCATION} =_{def} \text{ATTRIBUTE} \rightarrow \text{INTERFACE}$$
$$\text{CONNECTS} =_{def} \text{CONNECTION} \rightarrow \{\{i, j\} \mid i, j \in \text{INTERFACE}\}$$

- Variable valuations represent the local and global data space of the system. This enables us to model a shared global state:

$$\text{VALUATION} =_{def} \text{ATTRIBUTE} \rightarrow \text{VALUE}$$

- Component communication describes message-based asynchronous interaction between components. Thus, we can specify mandatory call-backs without problems.

$$\text{EVALUATION} =_{def} \text{INTERFACE} \rightarrow \text{MESSAGE}^{*}$$ [3]

As illustrated in Figure 8, the behavior of a component-based system is given by an infinte sequence of finite subsets of the set SNAPSHOT, which covers any possible system snapshot.

$$\text{SNAPSHOT} =_{def} \text{ALIVE} \times \text{ASSIGNMENT} \times \text{ALLOCATION} \times \text{CONNECTS} \times \text{VALUATION} \times \text{EVALUATION}$$

Finally, as shown in [Rausc01] this system behavior can be derived from the behavior functions of all components. The behavior function of a single component is a simple transition function that takes a snapshot and calculates the following snapshot:

$$\text{behavior} : \text{SNAPSHOT} \rightarrow \text{SNAPSHOT}$$

Based on this formal foundation of the instance level we can provide a precise definition of the sepcification level. A specifier at the specification level models all common properties of a set of instances in an abstract way. Let SPECIFIER be the infinite set of all specifiers, as for instance system specifications, component specifications, interface specifications, attribute specifications, and method specifications.

The function specified assigns to each instance its corresponding specifier. This function models the semantic bridge from the instance level to the specification level and vice versa:

$$\text{specified} : \text{INSTANCE} ® \text{SPECIFIER}$$

For the formal foundation of the specifiers we use the infinite set TERM^{v} of all logical expressions with a single free variable v. For instance, one specified property of a system could be: All instances of the attribute $\text{AttributeWithConstantValue}$ should always have

[3] Whereas MESSAGE^{*} denotes any finite sequence of messages.

the value 5. This specification would be formulated by the following logical expression $t \in TERM^{\nu}$:

$$\forall a \in Attribute_{\nu} \,.\, specified(a) = AttributeWithConstantValue \Rightarrow (a,5) \in valuation_{\nu}$$

where $Attribute_{\nu}$ is the set of attribute instances in an arbitrary component-based system ν and $valuation_{\nu}$ assigns values to attribute instances in the system ν.

An instance $s \in INSTANCE$, particularly a system during runtime, is a valid interpretation of such a $t \in TERM^{\nu}$ if the predicate $t\lfloor s \rfloor$ holds:

$$.\lfloor . \rfloor : TERM^{\nu} \times INSTANCE \rightarrow BOOLEAN$$

This function is the foundation of our semantics. It defines the set of predicates we use in our specifications, similar to the predicates used in Hoare triples. They allow us to determine whether an instance is a correct implementation of a specification or not.

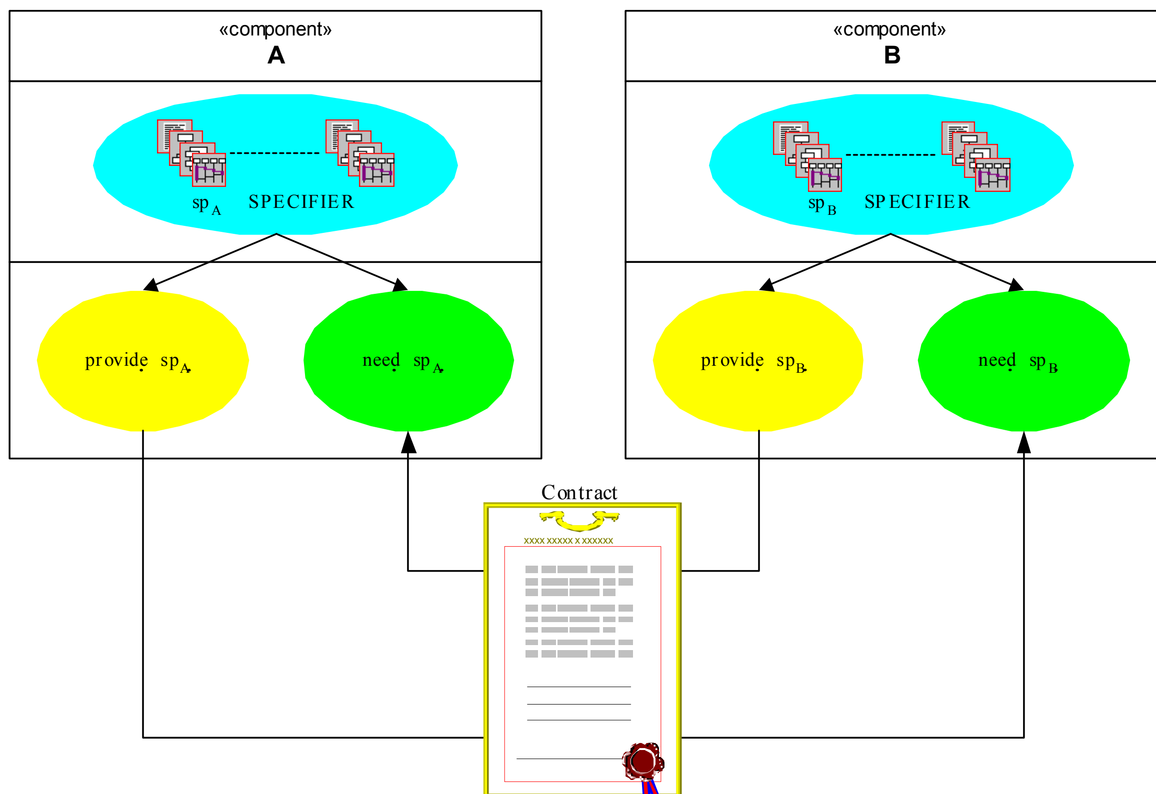
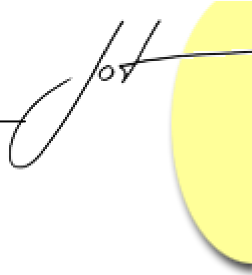


Fig. 9: Formal Foundation of Design by Signed Contract

To each specifier, especially to each component specification, we can now assign a set of provided properties and a set of needed properties[4]:

[4] $\wp(A)$ denotes the powerset of the set A.



$$\mathrm{provide}: \mathrm{SPECIFIER} \rightarrow R(\mathrm{TERM}^{\vee})$$

$$\mathrm{need}: \mathrm{SPECIFIER} \rightarrow R(\mathrm{TERM}^{\vee})$$

These sets correspond to the NEED and PROVIDE part of the specifications presented in the previous section (see Figure 9). The function $\mathrm{need}(\mathrm{spec})$ models all needed properties of a certain specifier $\mathrm{spec} \in \mathrm{SPECIFIER}$. Are all properties valid the specifier spec provides the properties described by $\mathrm{provide}(\mathrm{spec})$. Hence, if an instance $s \in \mathrm{INSTANCE}$ is a correct implementation of a given specification $\mathrm{spec} \in \mathrm{SPECIFIER}$, the following condition must hold:

$$\forall p \in \mathrm{provide}(\mathrm{spec}).\left(\bigwedge_{\forall n \in \mathrm{need}(\mathrm{spec})} n\right)[\![s]\!] \Rightarrow p[\![s]\!]$$

Based on these two functions $\mathrm{provide}$ and need we are able to explicitly model the dependencies between the various specifiers used within a specification. A signed contract $\mathrm{Contract} \subseteq \mathrm{CONTRACT}$ maps a set of specified needed properties of a certain specifier to a set of specified provided properties of another specifier:

$$\mathrm{CONTRACT} =_{def} \mathrm{SPECIFIER} \times \mathrm{TERM}^{\vee} \times \mathrm{SPECIFIER} \times \mathrm{TERM}^{\vee}$$

For a given signed contract the predicate $\mathrm{fulfilled}$ denotes whether the contract is valid for a specific specifier or not:

$$\mathrm{fulfilled}: \mathrm{SPECIFIER} \times R(\mathrm{CONTRACT}) \times R(\mathrm{SPECIFIER}) \rightarrow \mathrm{BOOLEAN}$$

Let $\mathrm{Contract} \subseteq \mathrm{CONTRACT}$ be a given signed contract and $\mathrm{Specifier} \subseteq \mathrm{SPECIFIER}$ a set of specifiers used within a specification, then the signed contract holds for the $\mathrm{specifier} \in \mathrm{Specifier}$ if all needed properties of $\mathrm{specifier}$ in the contract, are assigned to provided properties of other specifiers, and finally the needed and provided properties are logical equal.

$$\mathrm{fulfilled}(\mathrm{specifier}, \mathrm{Contract}, \mathrm{Specifier}) \Leftrightarrow_{def} \forall n \in \mathrm{need}(\mathrm{specifier}) \Rightarrow$$
$$\exists(\mathrm{specifier}, n, x, p) \in \mathrm{Contract}\,.\, x \in \mathrm{Specifier} \wedge p \in \mathrm{provide}\langle x \rangle \wedge \mathrm{holds}(p, n)$$

The predicate holds thereby denotes the logical equivalence of two properties. This predicate is valid, if the provided property implies the needed property with respect to all possible interpretations with an arbitrary instance $s \in \mathrm{INSTANCE}$:

$$\mathrm{holds}: \mathrm{TERM}^{\vee} \times \mathrm{TERM}^{\vee} \rightarrow \mathrm{BOOLEAN}$$

$$\mathrm{holds}(p, n) \Leftrightarrow_{def} (p[\![s]\!] \Rightarrow n[\![s]\!])$$

Whenever a component-based system is glued together from components the developer or a tool have to validate whether the signed contract of the system is fulfilled for all used components. Still not satisfied needed properties of components can be identified. Thus system defects may be detected and prevented in advance. These not satisfied needed properties have to be mapped to provided properties of other components.

Note, the correctness of this mapping is not calculable by a tool in general. To accomplish this the tool would have to calculate the predicate holds. But the number of instances for which the tool would have to prove the implication of properties is infinite.

However, `holds` can be proven with the use of specialized tools that require human interactions, e.g. theorem proving techniques, but this is beyond the scope of this article.

5 CONCLUSION

"Design by Contract" (DbC) is a well-known applied approach for the specification, the programming, and the testing of object-oriented systems. Component-based software development (CBSD) is a new paradigm. Systems are no longer implemented, they are glued together from existing components, which are self-contained units of deployment.

In our working example we have shown that applying the pure concepts of DbC fails in the context of CBSD. The main reason for this is that the components of a component-based system rely on each other, but one cannot explicitly specify the dependencies between these components with the concepts of DbC. Hence, to validate the correctness of the system as a whole, one has either to inspect all component implementations or to design and execute all failure-producing test scenarios. Both options are not possible in CBSD.

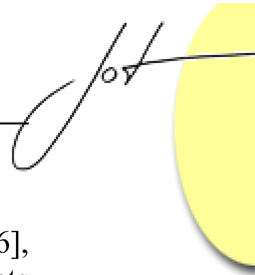
For these reasons we have elaborated a new sophisticated specification technique "Design by Signed Contract", based on the concepts of DbC. Thereby, we distinguish between component island specifications provided by component developers and component composition specifications developed by component users. With component island specifications we precisely describe what a component provides to and needs from its environment. In component composition specifications the mapping of needed properties to provided properties is specified within the context of a specific component-based system.

These composition specifications form a signed contract which can be checked and validated by developers or tools. Thereby situations can be detected where the needs of a single component are not fulfilled within a component-based system. Thus, software system defects can be identified and prevented in advance at the specification level. This will improve the correctness and robustness of component-based systems.

The presented formal foundation of the proposed concepts of "Design by Signed Contract" provide a reliable base to integrate these concepts into existing specification techniques and programming languages. This may be the next step towards a successful applied component-based software development in practice.

6 RELATED WORK ON CONTRACTS

As already mentioned, a lot of work on the integration of the concepts of DbC into programming languages has been done by Bertrand Meyer and the Eiffel-Community (see [Meyer87] and [Meyer97]). Surely, more sophisticated specification techniques have been developed based on the concepts of DbC. The most important ones that have



influenced this work are Interaction Contracts [Helm90], Reuse Contracts [Steya96], Evolving Interoperation Graphs [Rajli99], and Requirements/Assurances Contracts [Rausc00].

Interaction Contracts are used to specify the collaborations between objects. Although the basic idea of interaction contracts – to specify the behavioral dependencies between objects – seems to be quite a good suggestion, this approach takes neither CBSD nor DbC sufficiently into account. Interaction contracts strongly couple the behavior specification of the component seen as an island and the behavioral dependencies to other components. Hence, those components are still not self-contained units of deployment as required for successful CBSD.

Reuse Contracts address the problem of changing implementations of a stable abstract specification. There, defects in the scope of object-oriented software evolution are discussed. This might be helpful to predict the consequences of evolving a single component, but effects for a component-based system glued together from existing components are not clear at all.

Evolving Interoperation Graphs provide a framework for change propagation if a single class changes. These graphs only take the syntactical interface of classes and the static structure (class hierarchy) of the system into account, but not the behavioral dependencies. Moreover, neither CBSD nor DbC is taken into account.

Finally, Requirements/Assurances Contracts can be used to model and track the dependencies between the set of specification documents of a component-based system. Based on this approach, developers are able to track and manage the software evolution process and to recognize and avoid failures during software evolution. However, this is done at the level of specification documents and not at the level of specific specification specifiers. For these reasons DbC is not taken into account in this approach.

7 ACKNOWLEDGMENTS

I am grateful to Klaus Bergner, Manfred Broy and Siegfried Schäfler for interesting discussions and comments on earlier versions of this article.

REFERENCES

[Helm90] Helm R., Holland I. M., Gangopadhyay D. Contracts: Specifying Behavioral Compositions in Object-Oriented System. In ECOOP/OOPSLA '90 Proceedings, pages 169-180. 1990.

[Hoare69] Hoare, C.A.R. An axiomatic basis for computer. Commun. ACM 12, 10, October 1969, 576-585.

[Hoare81] Hoare, C.A.R. The Emperor's Old Clothes. Commun. ACM 24, 2, February 1981, 75-83.

[Meyer87] Meyer, B. Design by Contract, Technical Report TR-EI-12/CO, ISE Inc., 1987.

[Meyer97] Meyer, B. Object-Oriented Software Construction, Second Edition. Prentice Hall International. 1997.

[Rajli99] Rajlich V. Modeling Software Evolution by Evolving Interoperation Graphs. In Proceedings of the International Workshop on Software Change and Evolution 1999. 1999.

[Rausc00] Rausch A. Software Evolution in Componentware using Requirements/Assurances Contracts. In Proceedings of the ICSE'00. 2000.

[Rausc01] Rausch A. Componentware: Methodik des evolutionären Architekturentwurfs. PhD Thesis, Technische Universität München. 2001.

[Steya96] Steyaert P., Lucas C., Mens K., D'Hondt T. Reuse Contracts: Managing the Evolution of Reusable Assets. In OOPSLA 1996 Conference Proceedings, ACM Sigplan Notices, pages 268-285, AXM Press. 1996.

[Szype97] Szyperski C. Component Software, Beyond Object-Oriented Programming. Addison Wesley Longman Limited. 1997.

About the author

Andreas Rausch received his Ph.D. in 2001 from the Technische Universität München at the chair of Prof. Dr. Manfred Broy, with the dissertation titled "Componentware - Evolution-based Development of Software Architectures". He is part of the large interdisciplinary research project FORSOFT, leading the subproject ZEN that is concerned with the foundations of software engineering. He has been leading various industrial software projects, developing large distributed systems, and is one of the four founders of the software house 4Soft GmbH. He can be reached at rausch@computer.org.