Beherrschung von Schnittstelleninkonsistenzen in komponentenbasierten Softwaresystemen

Martin Jung, Francesca Saglietti, Verena Sauerborn

Lehrstuhl für Software Engineering
Universität Erlangen-Nürnberg
Martensstrasse 3
91058 Erlangen, Deutschland
jung@informatik.uni-erlangen.de

Abstract: Die Integration vorgefertigter Softwarekomponenten zu neuen Systemen unterstützt durch das Bausteinprinzip die Übersichtlichkeit der sich ergebenden Architekturen, durch den Einsatz betriebsbewährter Teile die Zuverlässigkeit des Gesamtsystems und durch die Wiederverwendung von Komponenten die Einsparung von Neuentwicklungskosten. Allerdings birgt der Integrationsprozess einige kritische Fehlerquellen an den Schnittstellen der Komponenten, die zu lokalen bzw. globalen Inkonsistenzen mit schwerwiegenden Konsequenzen führen können. Dieser Beitrag schildert einen Ansatz zu einer erweiterten Schnittstellenbeschreibung, die sowohl die automatische Erkennung einer Reihe von Inkonsistenzarten als auch ihre Behebung im Betrieb unterstützt.

1 Einleitung

Wirtschaftliche sowie sicherheitstechnische Aspekte sprechen für die Wiederverwendung bewährter Komponenten. Zum einen bietet der wiederholte Einsatz vorgefertigter Softwareelemente ein beträchtliches Einsparpotential an Neuentwicklungskosten; zum anderen erlaubt die mit bereits verwendeten Komponenten gewonnene Betriebserfahrung, Rückschlüsse auf deren Zuverlässigkeit zu ziehen. Der Einsatz von Komponenten in neuen Anwendungssystemen birgt allerdings das Risiko, die Bausteine so zusammenzusetzen, dass die sich ergebende Interaktion den neuen Anwendungskontext nicht mehr korrekt reflektiert. Infolge derartiger Inkonsistenzen an den Komponentenschnittstellen kann es zu kritischem Softwareversagen kommen, wie in Abschnitt 2 anhand einiger Beispiele belegt wird. Um dies zu vermeiden, wird in diesem Beitrag ein Ansatz vorgestellt, der es erlaubt, auf Basis vorliegender Informationen über die Komponenten sowie über ihre geplante Anwendung mögliche Inkonsistenzarten rechtzeitig (zur Konfigurationszeit) zu erkennen bzw. sie nachträglich (während des Betriebs) zu beheben.

Durch das Bausteinprinzip bietet die komponentenbasierte Entwicklung eine erhöhte Übersichtlichkeit der zugrunde liegenden Systemstruktur, wozu eine einfache Black-Box-Sicht auf die Komponenten ausreichen würde. Für die Beschreibung einer Black-

Box-Sicht existieren mehrere Ansätze, z.B. [WSDL], [IDL02] oder [MIDL]. Zur Erkennung bzw. Behebung besagter Inkonsistenzen hingegen sind spezielle interne Informationen über aktuelle Komponentenzustände und Randbedingungen der Anwendungen erforderlich, die in keinem der bestehenden Ansätze berücksichtigt werden. Um die oben genannte Übersichtlichkeit nicht unnötig durch zusätzliche überflüssige Informationen zu beeinträchtigen (White-Box-Sicht) werden in diesem Ansatz lediglich die zur Konsistenzanalyse erforderlichen Interna berücksichtigt (Gray-Box-Sicht).

Im Folgenden bezeichnet eine Komponente eine Softwareeinheit, die an ihrer Schnittstelle bestimmte Dienste zur Verfügung stellt; diese werden jeweils durch Angabe einer vorgegebenen Anzahl Eingabeparameter aufgerufen und können jeweils einen Rückgabeparameter ausgeben.

2 Inkonsistenzarten in komponentenbasierten Systemen

Es gibt verschiedene Kategorien von Inkonsistenzen, die bei der Konfiguration einer neuen Anwendung aus bereits bewährten Komponenten auftreten können (s. [SJ04]). Über die einfache syntaktische Inkonsistenz hinaus, die von heutigen Übersetzern problemlos behandelt wird, befasst sich der vorliegende Artikel mit einer Reihe wesentlich komplexerer Inkonsistenzarten, deren Analyse noch eine beträchtliche Herausforderung darstellt. Es handelt sich unter anderem dabei um folgende Kategorien:

2.1 Semantische Inkonsistenz

Semantische Inkonsistenz ergibt sich im Falle einer unterschiedlichen Interpretation identischer Daten seitens unterschiedlicher Komponenten. Die Interpretation kann beispielsweise die numerische bzw. logische Umsetzung eines Symbols oder dessen Bedeutung im physikalischen Umfeld betreffen.

- Numerische Inkonsistenzen entstehen unter anderem bei Verwendung unterschiedlicher Zahlendarstellungen, etwa englische bzw. deutsche Dezimaldarstellung, sowie bei Verwendung gleicher Präfixe mit unterschiedlicher Bedeutung, etwa „Mega“ im metrischen System oder zur Quantifizierung der Datenspeicher.

- Sprachliche Inkonsistenzen entstehen durch Mehrdeutigkeit der Sprachsyntax, d.h. syntaktisch korrekte Symbole können auf unterschiedlichen Weisen semantisch interpretiert werden, was unter anderem für das Versagen der Mariner 1 (s. [NASA62]) verantwortlich war. Erfreulicherweise sind derartige Inkonsistenzen dank der neueren Programmiersprachen nicht mehr denkbar.

- Referenzsystem-Inkonsistenzen entstehen durch Verwendung unterschiedlicher Bezüge zur realen Welt (Einheit zur Erfassung von Zeit, Ort, Währung usw.). Diese Inkonsistenzart verursachte zum Beispiel den Verlust der Mars-Mission Climate Orbiter (s. [St99]) infolge der Verwendung unterschiedlicher Referenzsysteme für die Krafterfassung.

2.2 Anwendungsbasierte Inkonsistenz

Über die Interpretation gemeinsamer Daten hinaus kann Inkonsistenz auch hinsichtlich des Anwendungskontexts auftreten. Diese äußern sich durch Verletzung folgender anwendungsspezifischer Relationen:

- Einschränkungen von Dienstparametern und evtl. von Komponentenzuständen zur Sicherung legalen Anwendungsverhaltens: Durch Einhaltung derartiger Einschränkungen ließen sich etwa Unfälle wie die in [Mo92] beschriebene explosive Mischung chemischer Substanzen vermeiden.

- Einschränkungen von Werten auf erlaubte Bereiche: Durch ihre Einhaltung ließen sich Unfälle wie die durch Wertebereichsüberschreitung verursachte Explosion der Ariane 5 (s. [Li96]) vermeiden.

2.3 Pragmatische Inkonsistenz

Eine weitere Inkonsistenzkategorie betrifft Unstimmigkeiten in den Rechnerumgebungen der einzelnen Komponenten, wie etwa durch Verletzung folgender Bedingungen:

- Absolute Zeitanforderungen schränken die zugelassene Ausführungszeit von Operationen bzw. die zugelassene Dauer zwischen zwei Aufrufen ein. Durch Einhaltung derartiger Einschränkungen ließen sich Unfälle wie das Versagen einer Patriot-Abwehrrakete (s. [GAO92]) vermeiden.

- Relative Zeitanforderungen schränken die zugelassene Reihenfolge nebenläufiger Komponentenausführungen ein. Durch ihre Einhaltung lassen sich typische Versagen in Folge von “race conditions” bzw. mangelndem Transaktionsschutz in Datenbanken vermeiden. Auch das Fehlverhalten der Familie von Bestrahlungsgeräten Therac-25 (s. [LT93]) ist auf ein Nebenläufigkeitsproblem zurückzuführen.

- Anforderungen an die Kompatibilität von Rechnerarchitekturen, zum Beispiel durch Ausschluss der gleichzeitigen Verwendung einer nachrichtenbasierten und einer objektorientierten Client/Server-Architektur.

3 Benötigte Information zur Erkennung von Inkonsistenzen

Für oben genannte Inkonsistenzarten wird im Folgenden festgehalten, welche Informationen zu ihrer Erkennung und eventueller Behebung (s. Abschnitt 5) im Kontext einer vorgegebenen Anwendung notwendig sind. Dabei wird vorausgesetzt, dass bekannt ist, welche Komponente Dienste nutzt, die von einer anderen Komponente angeboten werden.

3.1 Informationen zur Erkennung semantischer Inkonsistenz

Geht es um die Erkennung von semantischen Inkonsistenzen, muss geprüft werden, ob die Semantik eines angebotenen Parameters mit der des genutzten Parameters übereinstimmt. Zu diesem Zweck müssen die Parameter entsprechend der anzustrebenden semantischen Überprüfung feingranularer als auf rein syntaktischer Ebene gestaltet werden. Dazu werden die Typen der Parameter um zusätzliche Attribute mittels Paaren der Gestalt (Attribut,Wert) ergänzt. Stimmen die Werte des gleichen Attributs bei genutztem und angebotenem Parameter nicht überein, so wird die entsprechende Inkonsistenz statisch erkannt.

- Um numerische Inkonsistenz zu erkennen, werden Attribute eingesetzt, die die verwendete Zahlendarstellung etwa über ein Länderkürzel, z.B. nach ISO 639 (s. [ISO02]) oder durch Angabe der Zahlenbasis wiedergeben.

- Referenzsystem-Inkonsistenz lässt sich nach ähnlichem Prinzip erkennen, wenn an den Parametern das jeweilige Referenzsystem angegeben ist, etwa die verwendete physikalische Einheit oder die zugrunde liegende Zeitzone bzw. Währung.

3.2 Informationen zur Erkennung anwendungsbasierter Inkonsistenz

Zur Erkennung anwendungsbasierter Inkonsistenzen werden Informationen über legale Relationen der Dienstparameter $param_1,...,param_n$ und der Zustände $state_1,\ldots,state_m$ der Komponenten der Form

$$R_i\left(param_1,\ldots,param_n,state_1,\ldots,state_m\right),\ i \in \{1\ldots k\}$$

benötigt. Darüber hinaus werden anwendungsweite Informationen über die Wertebereiche von Parametern festgelegt: $W(param_j) = [min_j;max_j]$, $j \in \{1,...,n\}$. Diese Informationen werden als eine Liste von Einschränkungen der Anwendungsbeschreibung beigefügt, die in einer OCL-ähnlichen Notation formuliert werden. Die Komponentenbeschreibungen werden um Angabe der verarbeitbaren Wertebereiche $W_c(param_j) = [min_{c,j};max_{c,j}]$, $c \in \{1,...,m\}$, $j \in \{1,...,n\}$ ergänzt.

- Einschränkungsverletzungen von Dienstparametern lassen sich erkennen, wenn Dienste der Anwendung mit Parameterwerten $(param_1,...,param_n)$ im Anwendungszustand $(state_1,...,state_m)$ aufgerufen werden, die mindestens eine der Relationen R_i $(i \in \{1,...,k\})$ verletzen.

- Inkonsistenz zwischen von der Anwendung benötigten und von der Komponente c verarbeitbaren Wertebereichen lässt sich erkennen, wenn für mindestens einen Parameter $param_j$ der von der Komponente c verarbeitet wird, $W(param_j)$ nicht in $W_c(param_j)$ enthalten ist.

3.3 Informationen zur Erkennung pragmatischer Inkonsistenz

Um komponenten- und anwendungsspezifische Anforderungen an die Rechnerumgebung miteinander zu vergleichen, werden folgende Informationen benötigt:

- Informationen über minimale bzw. maximale Ausführungszeiten werden als Teil der Anwendungsbeschreibung notiert. Festgehalten werden mittels Attribut-Wert-Paaren die minimale und maximale Ausführungszeit „min $time_a$“ bzw. „max $time_a$“ des jeweiligen Dienstes $service_a$; überprüft wird die folgende Ungleichung:

$$\min \text{time}_a \leq \text{execution time}(\text{service}_a) \leq \max \text{time}_a$$

  Ähnliches gilt für die minimale bzw. maximale Zeit „min $time_{ab}$“ bzw. „max $time_{ab}$“ zwischen dem Aufruf zweier Dienste $service_a$ und $service_b$:

$$\min \text{time}_{ab} \leq \text{time}(call(service_b)) - \text{time}(call(service_a)) \leq \max \text{time}_{ab}$$

- Transaktionsschutz in der Anwendung wird erreicht, indem die Aktivierung bestimmter Dienste während der Bearbeitung vorgegebener Nachrichtensequenzen gezielt eingeschränkt wird. Dies wird durch eine Relation R($message_1$...$message_h$, $service_1$,...,$service_k$) festgehalten, die angibt, welche der Dienste $service_1$,...,$service_k$ während der Nachrichtensequenz $message_1$...$message_h$ aktiviert werden dürfen.

- Schließlich wird Information über die Anforderung an die Rechnerarchitektur bzw. vorhandene Middleware einzelner Komponenten in Form von Paaren (Attribut, Wert) an den Komponenten festgehalten.

4 Anreicherung des UML 2.0 Komponentenmodells

Die in Abschnitt 3 als erforderlich identifizierten Erweiterungen wurden als Profil der UML 2.0 (s. [OMG03]) mit der Bezeichnung CCI (Consistent Component Integration) umgesetzt. Die Erweiterungen lassen sich grob in drei Kategorien aufteilen: Erweiterungen an der Parameterdarstellung (s. Abb. 1), an der Anwendungsbeschreibung (s. Abb. 2) und an der Komponentenbeschreibung (s. Abb. 3).

Die beiden in Abschnitt 3.1 identifizierten Erweiterungen können im Profil über einen einzigen Mechanismus abgebildet werden. Dafür werden zunächst alle im Profil verwendeten Daten durch CCIDataType typisiert und durch CCITypeAttribute attributiert. Der entsprechende Ausschnitt aus dem Metamodell ist in Abbildung 1 zu sehen.

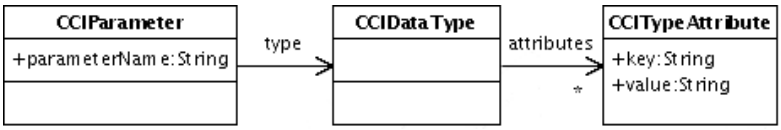

Abbildung 1: Erweiterung der Parameterdarstellung

Die identifizierten Informationen zur Beschreibung anwendungsspezifischer Einschränkungen werden im Profil durch eine Reihe spezieller Klassen abgebildet.

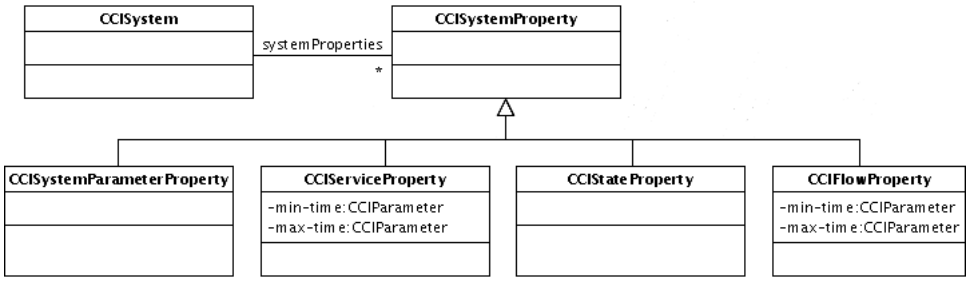

Abbildung 2: Anwendungseigenschaften

Der Anwendungsbeschreibung CCISystem können beliebig viele Eigenschaften in Form von Instanzen der Klasse CCISystemProperty zugeordnet werden, die OCL-ähnliche Konstrukte[1] bzw. Attribute enthalten. Für die unterschiedlichen Arten von Eigenschaften gibt es die im Folgenden beschriebenen spezialisierte Unterklassen.

Die in Abschnitt 3.2 angesprochenen Relationen zwischen Dienstparametern und Zuständen können als Instanzen von CCIServiceProperty in der Beschreibung untergebracht werden. Der Bezug auf die Dienste und Zustände der Komponentenbeschreibungen wird über Assoziationen hergestellt, die Relationen sind in OCL notiert. Als abkürzende Schreibweise für spezielle Relationen, die sich nur auf den Zustandsraum beziehen, werden CCIStateProperty-Instanzen verwendet.

Die Klasse CCISystemParameterProperty wird verwendet, um einen Parameter der Anwendung mit in OCL notierten Wertebereichseinschränkungen zu versehen. Diese können dann mit den von den Komponenten angebotenen Wertebereichen, beschrieben mittels Instanzen der Klasse CCIServiceGuard (vgl. Abbildung 3), verglichen werden.

Die Informationen min-time und max-time, die in CCIServiceProperty-Objekten gespeichert sind, werden verwendet, um Ausführungszeiten der Dienste anzugeben, die dann zur Laufzeit überprüft werden können, wie in Abschnitt 3.3 beschrieben.

Die Information, die in CCIFlowProperty-Objekten festgelegt wird, dient zum einen zur Festlegung von absoluten Zeitanforderungen zwischen den Aufrufen verschiedener Dienste über die Attribute min-time und max-time. Zum anderen werden hiermit relative Zeitanforderungen festgelegt: Mittels Objekten der Metaklasse CCIFlowProperty

[1] CCISystemProperty ist von der UML-Klasse Constraint abgeleitet, die als Attribut eine OCL-Expression besitzt.

werden Sequenzen von Nachrichten angegeben, während derer der Aufruf vorgegebener Dienste der Anwendung untersagt wird.

Die Informationen zum Festlegen bestimmter Anforderungen einer Komponente an ihr Rechenumfeld werden mit Hilfe von CCIComponentProperty festgelegt (s. Abbildung 3). Die Abbildung zeigt auch die Struktur der Information, die das Komponentenverhalten abbildet.

Der Einfachheit halber wurden hier nur die wichtigsten Elemente des Profils aufgeführt, insgesamt umfasst das Profil noch einige Klassen und Assoziationen mehr. Eine zusammenfassende Übersicht der Inkonsistenzarten, der für die Erkennung erforderliche Informationen und der entsprechenden Erweiterungen im Modell gibt Tabelle 1.

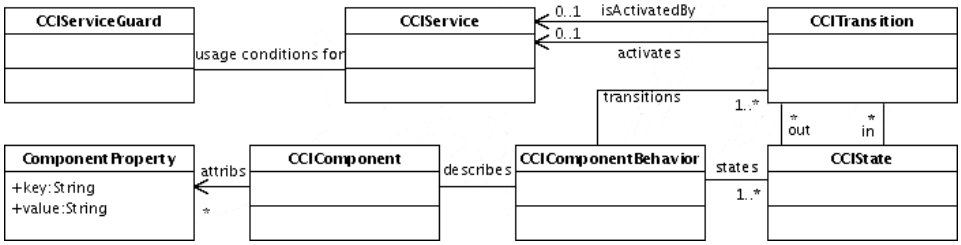

Abbildung 3: Komponentenverhalten

5 Erzeugung von Wrappers zur Vermeidung von Inkonsistenzen

Zum Zweck einer möglichst automatischen Überprüfung und Sicherstellung der Konsistenzeigenschaften mittels des vorgestellten Profils werden im Folgenden die Erzeugung und der Einsatz von Wrappers untersucht. Diese bezeichnen Konstrukte, die über die (bis auf eventuelle Typanpassungen) gleiche Schnittstelle wie die ursprüngliche Komponente verfügen, und an deren Stelle verwendet werden. Wird ein Dienst des Wrappers aufgerufen, kann zunächst festgestellt werden, ob der Dienst alle Vorbedingungen erfüllt und ob seine Ausführung die Konsistenzanforderungen der Anwendung nicht verletzt. Ist dies sichergestellt, so wird der Dienstaufruf vom Wrapper an die Komponente weitergegeben. Die Rückgabeparameter können ebenfalls vom Wrapper bearbeitet werden, bevor sie an die aufrufende Komponente zurückgegeben werden.

5.1 Maßnahmen

Zunächst lassen sich aus den oben identifizierten Inkonsistenzklassen, beruhend auf den Modellerweiterungen Maßnahmen ableiten, die zur Beherrschung von Inkonsistenzen getroffen werden können. Diese werden durch Wrappers realisiert, wie in den folgenden Abschnitten 5.2 und 5.3 gezeigt wird.

Inkonsistenzarten		Benötigte Information	Erweiterte Beschreibung der Komponentenschnittstellen bzw. der Anwendung
Semantische Inkonsistenz	Numerische Darstellung	Angaben zur Zahlendarstellung	Format-Attribute der Datentypen
	Referenz-system	Angaben zu den verwendeten Referenzsystemen	Einheits-Attribute der Datentypen
Anwendungs-basierte Inkonsistenz	Dienstparameter und Zustände	Zugelassene Relation zwischen den Parametern eines Dienstes in bestimmten Anwendungszuständen	OCL-ähnliche Notation von Einschränkungen über den Service-Parametern und dem Anwendungszustand
	Wertebereiche	Angabe der von der Anwendung geforderten sowie der von den Komponenten verarbeitbaren Wertebereiche der Dienstparameter	OCL-ähnliche Notation der entsprechenden Wertebereiche der Dienstparameter
Pragmatische Inkonsistenz	Absolute Zeit-anforderung	Minimal bzw. maximal zugelassene Ausführungszeiten von Diensten	Zeit-Attribute der Dienste auf Anwendungsebene sowie auf Komponentenebene
	Relative Zeit-anforderung	Dauer zwischen den Aufrufen vorgegebener Dienste	Zeit-Attribute von Dienstsequenzen auf Anwendungsebene
	Angabe der Komponenten-architektur	Anforderungen der Komponenten an die Rechnerumgebung	Architektur-Attribute an den Komponenten

Tabelle 1: Inkonsistenzklassen und zu ihrer Beherrschung notwendige Informationen

Die Maßnahmen lassen sich in vier Kategorien aufteilen:

- **Statische Erkennung und Abbruch der Integration:** Einige der obigen Inkonsistenzen lassen sich vor dem Betrieb erkennen, aber nicht beheben. Darunter fallen nicht erfüllte Anforderungen von Komponenten an die Rechnerumgebung, sowie Überschreitungen von Wertebereichen oder absoluten Zeitanforderungen. Wird bei der Integration festgestellt, dass der Wertebereich eines Parameters nicht dem von der Anwendung geforderten entspricht bzw. die geschätzte best-case-Ausführungszeit eines Dienstes die maximal zulässige überschreitet, muss die Integration abgebrochen werden.

- **Statische Erkennung und dynamische Beherrschung:** Die Erkennung der Verwendung unterschiedlicher Einheiten von Parametern oder unterschiedlicher Zahlendarstellungen in verschiedenen Komponenten ist statisch mit Hilfe der Typattribute möglich. Im Falle einer solchen Inkonsistenz kann ein entsprechend erzeugter Wrapper passende Konversionen auf der Basis vorgegebener Umrechnungstabellen dynamisch vornehmen.

- **Dynamische Erkennung und dynamische Beherrschung:** Die Erkennung von Inkonsistenzen bei relativen Zeitanforderungen ist statisch nur in Ausnahmefällen möglich. Dadurch bedingt muss ihre Einhaltung dynamisch überwacht werden und eventuell durch eine geeignete beherrschende Maßnahme erzwungen werden. Soll etwa die Bearbeitung einer Nachrichtensequenz die gleichzeitige Aktivierung vorgegebener Dienste ausschließen, kann dies bei Bedarf dynamisch durch Zwischenspeicherung und verzögerte Weiterleitung nach Abschluss der Sequenz erzielt werden. Auch im Falle von Dienstausführungszeiten, die kürzer sind als von der Anwendung vorgeschrieben, kann ein Wrapper zum Zwecke einer verzögerten Rückgabe herangezogen werden.

- **Dynamische Erkennung und Fehlermeldung:** Auch die Erkennung von Verletzungen absoluter Zeitanforderungen ist statisch nicht immer möglich; meist muss die tatsächliche Ausführungszeit dynamisch gemessen werden. Im Falle der Überschreitung einer Zeitschranke muss etwa eine Fehlermeldung ausgegeben werden. Ähnliches gilt für die Verletzung von Relationen zwischen Dienstparametern und Zuständen, sowie für die Überwachung der Zeitdauer zwischen den Aufrufen vorgegebener Dienste.

5.2 Struktur der Wrappers

Die Überwachung sowie Sicherstellung von Konsistenzeigenschaften erfordert, wie oben gezeigt, verschiedene Maßnahmen, die durch einen Wrapper realisiert werden. So können Konversionen zwischen verschiedenen Parametern notwendig werden, Einschränkungen hinsichtlich der Verwendung von Diensten überwacht und Zustandsänderungen der Anwendung verfolgt werden.

Darüber hinaus können Wrappers auch um weitere analytische oder die Anwendung erweiternde Funktionen, wie zum Beispiel Logging, ergänzt werden. Diese Vielzahl von möglichen Funktionen legt eine modulare Strukturierung der Wrappers nahe, in die nach Bedarf verschiedene Teile der benötigten bzw. gewünschten Funktionalität integriert werden können.

Aus diesem Grund wird hier ein modularer Aufbau verwendet, wie er schematisch in Abbildung 4 zu sehen ist. Ein Wrapper ist damit zunächst eine Struktur ohne eigene Funktionalität, die um die Komponente gelegt wird und diese komplett verbirgt, vergleichbar einem Stellvertreter nach dem gleichnamigen Entwurfsmuster[2] (s. [GOF95]). Dies ist wichtig, um in der Anwendung ungeschützten Zugriff auf die Komponente zu verhindern. In diese Wrapper-Struktur werden nun nach und nach die Funktionen integriert, die zur Sicherstellung der Anwendungskonsistenz benötigt werden.

---

[2] Wächter bzw. engl. Proxy-Pattern

5.3 Funktionsweise

Die beschriebene Struktur eines Wrappers enthält also Funktionen, die alle Inkonsistenzen, für die die Integration nicht abgebrochen werden muss, beherrschen können. Für die jeweils in Frage kommenden Erkennungs- bzw. Behebungsmaßnahmen sind lokale, komponentenspezifische Informationen erforderlich, die zum Teil durch globale, anwendungsspezifische zu ergänzen sind.

Während Erstere im entsprechenden Wrapper lokal vorliegen, werden Letztere mit Hilfe einer zusätzlichen anwendungsweiten Instanz, dem sogenannten „SystemObserver“, bereitgestellt.

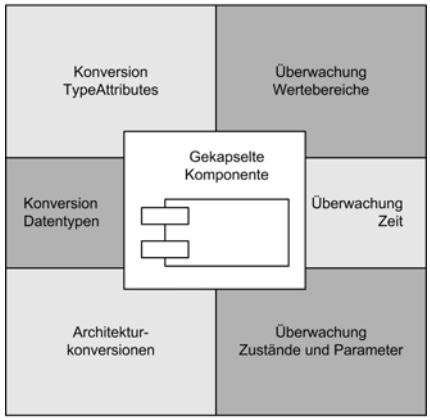

Abbildung 4: Aufbau eines Wrappers

Die Erstellung derjenigen Wrapperfunktionen, die sich lediglich auf lokale Informationen beziehen, wie die Überwachung bzw. Einschränkung der Verwendung bestimmter Dienste und die Konversionen von Parametern, ist relativ einfach. Es wird eine Programmeinheit, z.B. eine Methode, erzeugt, die die notwendige Funktionalität realisiert und im Wrapper abgelegt wird. Entsprechend des unten abgebildeten Schemas wird jeweils vor der Delegation (s. Abb. 5, „PreCall“) eines Dienstaufrufs an die eigentliche Komponente die lokale Wrapperfunktion ausgeführt. Sollte tatsächlich eine Inkonsistenz auftreten, wird diese durch eine angemessene beherrschende Maßnahme erkannt bzw. behoben; das auslösende Ereignis wird protokolliert.

Der SystemObserver zur Unterstützung derjenigen Wrapperfunktionen, die anwendungsweite Daten benötigen, wird durch zwei Bestandteile realisiert:

- zum einen durch einen sogenannten StateObserver, der vor der Ausführung eines Dienstes die Einhaltung vorgegebener Zustandseigenschaften vom oben eingeführten Typ CCIStateProperty überprüft (s. Abb. 5, „StateProperty“);

- zum anderen durch einen sogenannten FlowObserver, der die Einhaltung vorgegebener relativer Zeiteinschränkungen vom oben eingeführten Typ CCIFlowProperty überwacht (s. Abb. 5, „FlowProperty“).

Bei Feststellung verletzter Eigenschaften wird die unmittelbare Ausführung unterbunden. Im Falle einer Verletzung von Zustandseigenschaften wird die Ausführung abgebrochen, da in diesem Falle keine Inkonsistenzbehebung vorgesehen ist. Im Falle der Verletzung relativer Zeiteinschränkungen wird die Ausführung des Dienstes auf einen späteren zulässigen Zeitpunkt verschoben. Liegen keine Verletzungen vor, so werden nach Ausführung des Dienstes (s. Abb. 5, „Call“) die relevanten Informationen aktualisiert: Die lokalen Zustandsinformationen werden vom jeweiligen Wrapper (s. Abb. 5, „BodyPostCall“), die globalen Informationen zum Anwendungszustand vom StateObserver und diejenigen zum Abarbeitungszustand einer Nachrichtensequenz vom FlowObserver auf den letzten Stand gebracht (s. Abb. 5, „StateRegistration“ bzw. „FlowRegistration“). Zuletzt wird der von der Komponente gelieferte Rückgabewert bei Bedarf entsprechend der TypeAttributes des Aufrufers konvertiert und zurückgeliefert (s. Abb. 5, „Return“).

Sind für einzelne Dienste der Anwendung Vor- bzw. Nachbedingungen in Form von OCL-Ausdrücken angegeben, so wird der Wrapper entsprechende Funktionen enthalten, um diese Eigenschaften zu überwachen (s. Abb. 5, „Check Pre-Constraints“ bzw. „Check Post-Constraints“). Dazu werden die OCL-Ausdrücke vom Generator in Programmcode übersetzt, der zur Laufzeit die Einhaltung der Eigenschaften überprüft.

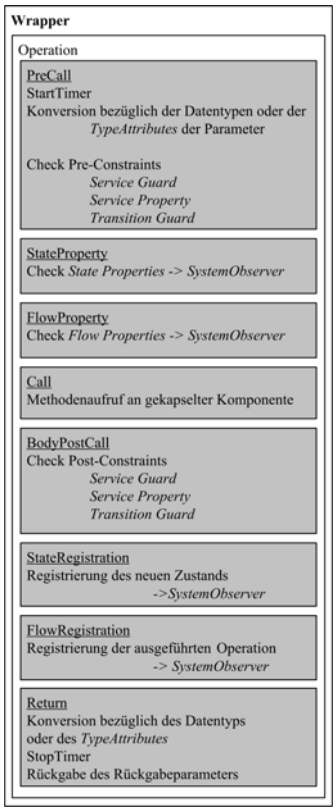

Abbildung 5: Funktionsweise eines Wrappers

6 Zusammenfassung

Ausgehend von den bestehenden Möglichkeiten, Komponenten und deren Schnittstellen zu beschreiben, sowie von den anhand realer Fälle aufgezeigten und klassifizierten Inkonsistenzen wurde eine Erweiterung des bestehenden UML 2.0 Komponentenmodells erstellt, um die notwendigen Informationen zur Beherrschung potentieller Inkonsistenzen bei der Spezifikation komponentenbasierter Anwendungen festzuhalten.

Basierend auf den identifizierten Inkonsistenzklassen und den zu ihrer Beherrschung erforderlichen Informationen wurden Erkennungs- und Behebungsmechanismen erarbeitet. Um diese Maßnahmen mittels Wrapper-Erzeugung automatisch umzusetzen, ist ein Werkzeug realisiert worden.

Unter Bearbeitung befinden sich derzeit Untersuchungen der Einsetzbarkeit des genannten Wrappers zwecks Aufzeichnung gewonnener Betriebserfahrung und deren Auswertung im Rahmen der Zuverlässigkeitsanalyse. Darüber hinaus wird die hier vorgestellte erweiterte Spezifikationssprache auch im Hinblick auf ihre Nutzbarkeit in der Integrationstestphase analysiert.

Literaturverzeichnis

[GAO92] Patriot Missile Defense – Software Problem Led to System Failure at Dhahran, Saudi Arabia. Report of the US General Accounting Office, 1992

[GOF95] Gamma, E. et al, Design Patterns, Addison Wesley, 1995

[IDL02] Object Management Group: CORBA 3.0 v. 3.0.3, Chapter 3 (2002) (formal/02-06-39)

[ISO02] International Standards Organization: Codes for the representation of names of languages, Standard 639, 2002

[Li96] Lions, JL. ARIANE 5 Flight 501 Failure: Report by the Enquiry Board, European Space Agency, Paris, 1996

[LT93] Leveson, N.; Turner, C. An Investigation of the Therac-25 Accidents, IEEE Computer vol. 26 nr. 7 p. 18-41, 1993

[MIDL] Microsoft Interface Definition Language, http://msdn.microsoft.com/library/en-us/dnanchor/html/midl.asp

[Mo92] Moonen, R. Dutch Chemical Plant explodes due to Typing Error, The Risks Digest, Forum on Risks to the Public in Computers and Related Systems, 1992

[NASA62] NASA, Mariner-Venus 1962: Final Project Report, NASA SP-59, Washington, 1965, pp. 278-279,294.

[OMG03] Object Management Group: UML 2.0 Superstructure Specification. (2003) OMG Adopted Specification (ptc/03-08-02)

[SJ04] Saglietti, F.; Jung, M: Classification, Analysis and Detection of Interface Inconsistencies in Safety-relevant Component-based Systems. Probabilistic Safety Assessment and Management 4, Berlin, 2004, S. 1864-1869.

[St99] Stephenson AG. Mars Climate Orbiter Mishap Investigation Board Phase 1 Report, 1999

[WSDL] World Wide Web Consortium: Web Service Description Language v.1.1 (2001) http://www.w3.org/TR/wsdl