(19) Bundesrepublik Deutschland
Deutsches Patent- und Markenamt

(10) **DE 60 2004 012 714 T2** 2009.04.16

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 1 592 975 B1**
(21) Deutsches Aktenzeichen: **60 2004 012 714.0**
(86) PCT-Aktenzeichen: **PCT/JP2004/001648**
(96) Europäisches Aktenzeichen: **04 711 445.9**
(87) PCT-Veröffentlichungs-Nr.: **WO 2004/072669**
(86) PCT-Anmeldetag: **16.02.2004**
(87) Veröffentlichungstag der PCT-Anmeldung: **26.08.2004**
(97) Erstveröffentlichung durch das EPA: **09.11.2005**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **26.03.2008**
(47) Veröffentlichungstag im Patentblatt: **16.04.2009**

(51) Int Cl.[8]: ***G01R 31/319*** (2006.01)

(30) Unionspriorität:

447839 P	**14.02.2003**	**US**
449622 P	**24.02.2003**	**US**

(84) Benannte Vertragsstaaten:
AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LI, LU, MC, NL, PT, RO, SE, SI, SK, TR

(73) Patentinhaber:
Advantest Corp., Tokio/Tokyo, JP

(74) Vertreter:
PFENNING MEINIG & PARTNER GbR, 10719 Berlin

(72) Erfinder:
PRAMANICK, Ankan, Tokyo 1790071, JP; ELSTON, Mark, Tokyo 1790071, JP; CHEN, Leon, Tokyo 1790071, JP; SAUER, Robert, Tokyo 1790071, JP

(54) Bezeichnung: **Verfahren und Struktur zum Entwickeln eines Prüfprogramms für Halbleiterschaltkreise**

Beschreibung

QUERVERWEIS AUF ZUGEORDNETE ANMELDUNGEN

[0001] Diese Anmeldung bezieht sich auch auf die gleichzeitig hiermit eingereichte US-Patentanmeldung, Nr. 10/972 404 „Verfahren und Struktur zum Entwickeln eines Prüfprogramms für Halbleiterschaltkreise", die Anspruch auf den Nutzen der am 14. Februar 2003 eingereichten Patentanmeldung Nr. 60/447 839 „Verfahren und Struktur zum Entwickeln eines Prüfprogramms für Halbleiterschaltkreise" erhebt.

HINTERGRUND DER ERFINDUNG

Gebiet der Erfindung

[0002] Die vorliegende Erfindung betrifft das Prüfen von Schaltkreisen (IC) und spezieller automatisches Prüfgerät (ATE) zum Prüfen von einem oder mehreren IC.

Beschreibung des Standes der Technik

[0003] Die zunehmende Komplexität von System-on-a-Chip-Bauelementen (SOC) und die gleichzeitige Forderung nach einer Reduzierung der Kosten der Chipprüfung hat sowohl Hersteller von Schaltkreisen als auch Prüfgerätehersteller gezwungen, nochmals zu überdenken, wie die Prüfung von Schaltkreisen durchgeführt werden sollte. Nach Studien der Industrie werden die projektierten Kosten von Prüfgeräten ohne eine Neugestaltung in naher Zukunft weiter dramatisch ansteigen.

[0004] Ein bedeutender Grund für die hohen Kosten der Prüfgeräte ist die spezialisierte Beschaffenheit der Architektur normaler Prüfgeräte. Jeder Prüfgerätehersteller hat eine Anzahl von Prüfgeräteplattformen, die nicht nur quer durch Firmen wie beispielsweise Advantest, Teradyne und Agilent inkompatibel sind, sondern auch quer über Plattformen wie die von Advantest hergestellten Prüfgeräte der Serien T3300, T5500 und T6600 inkompatibel sind. Wegen dieser Inkompatibilitäten erfordert jedes Prüfgerät seine eigenen spezialisierten Hardware- und Softwarekomponenten, die auf anderen Prüfgeräten nicht verwendet werden können. Außerdem ist ein erheblicher Übersetzungsaufwand erforderlich, um ein Prüfprogramm von einem Prüfgerät auf ein anderes zu übertragen, und es ist schwierig, Drittlösungen zu entwickeln. Selbst wenn eine Drittlösung für eine Plattform entwickelt ist, kann sie nicht auf eine andere Plattform übertragen oder wieder verwendet werden. Der Übertragungsprozess von einer Plattform zu einer anderen ist im Allgemeinen komplex und fehleranfällig, was zu zusätzlichem Aufwand, Zeit und erhöhten Prüfkosten führt.

[0005] Prüfgerätesoftware wie beispielsweise die Tools/Anwendungen für Rechnerbetriebssystem und Prüfanalyse laufen auf einem Host-Computer. Wegen der zugeordneten Beschaffenheit der Architektur bleibt die gesamte Hardware und Software in einer von vorn herein bestimmten Konfiguration für ein vorgegebenes Prüfgerät. Zum Prüfen eines IC wird ein zugeordnetes Prüfprogramm entwickelt, das einige oder alle der Fähigkeiten von Prüfgeräten nutzt, um sowohl die Prüfdaten, Signale, Wellenformen und Strom- und Spannungspegel zu definieren als auch die Reaktion des Prüfobjekts (DUT) zu sammeln und Abnahme/Nichtabnahme des DUT zu bestimmen.

[0006] Weil ein Prüfgerät eine breite Vielfalt von Schaltkreisen prüfen können sollte, werden sowohl Hardware- als auch Softwarekomponenten ausgeführt, die über einen breiten Bereich von Rechenoperationen arbeiten. Folglich enthält ein Prüfgerät viele Ressourcen, die in vielen Prüfsituationen genutzt werden können. Gleichzeitig kann das Prüfgerät für einen vorgegebenen Schaltkreis die meisten erwünschten Ressourcen, die für diesen Schaltkreis geeignet sind, nicht zur Verfügung stellen. Zum Beispiel kann ein logisches Prüfgerät, das geeignet ist zum Prüfen eines komplexen SoC A, das einen internen Mikrorechner, großen internen DRAM und Flash sowie verschiedene andere Speicher wie PCI und USB, usw. enthält, als unangemessen für eine anwendungsspezifische integrierte Schaltung (ASIC) B, die keinen internen Mikrorechner und großen internen DRAM/Flash aufweist, jedoch DAC und Sigma-Delta-Wandler umfasst, erkannt werden. Zum Prüfen einer anwendungsspezifischen integrierten Schaltung B würde ein geeignetes Prüfgerät anstelle von teurem Support für eingebettete Speicherprüfung Prüfeinheiten für Analog- und Mischsignal erfordern.

[0007] Daher ist es erwünscht, ein Prüfgerät bereitzustellen, das in Abhängigkeit von Prüfungsanforderungen umstrukturiert werden kann. Außerdem ist es erwünscht, die Apparatur eines anderen Herstellers in Verbindung mit automatischen Prüfgeräten (ATE) anzuschließen und zu nutzen. Wegen der spezialisierten Beschaffenheit normaler Prüfsysteme und der gesetzlich geschützten Beschaffenheit des Datenformats in jeder Her-

stellerapparatur ist es jedoch häufig unmöglich, Zubehör von einem anderen Hersteller anzuschließen und zu nutzen.

[0008] EP-A-0 388 107 offenbart eine Fahrzeugsteuervorrichtung mit einer ersten und zweiten Einstelleinrichtung zum jeweiligen Einstellen von ersten und zweiten Regelparametern eines Fahrzeugs; einer ersten und zweiten Steuereinrichtung zum Steuern der jeweils ersten und zweiten Einstelleinrichtung; und einer Übertragungseinrichtung zum Übertragen eines Signals von der ersten Steuereinrichtung zu der zweiten Steuereinrichtung, wobei das Signal durch die erste Steuereinrichtung bestimmt wird und einen zu regelnden Zustand darstellt; bei der die erste Steuereinrichtung eine Verarbeitung ausführt zum Berechnen des ersten Regelparameters auf der Basis von gelieferten Daten, auf der Basis des so berechneten ersten Regelparameters beurteilt, ob die zweite Steuereinrichtung in Kooperation mit der ersten Steuereinrichtung arbeiten muss oder nicht, und, wenn die kooperative Operation erforderlich ist, ein Ausgangssignal einschließlich einer Anweisung, die eine kooperative Anforderung an den Kommunikationspfad angibt, erzeugt; und die zweite Steuereinrichtung eine Verarbeitungsoperation zum Berechnen des zweiten gesteuerten Parameters als Reaktion auf das Ausgangssignal einschließlich der Anweisung ausgibt, die die durch die erste Steuereinrichtung erzeugte, kooperative Anforderung angibt; und bei der die erste und die zweite Einstelleinrichtung die ersten und zweiten Regelparameter steuern, die damit gleichzeitig und entsprechend den Verarbeitungsergebnissen der jeweiligen ersten und zweiten Steuereinrichtung zugeordnet sind.

[0009] Ferner beschreibt US-A-2002/0073375 A1 eine Vorrichtung, um den Entwurf eines integrierten Schaltkreises in funktioneller Hinsicht nachzuprüfen. Es wird eine auf Hardware orientierte, prüfungsspezifische, objektorientierte Programmiersprache genutzt, um Prüfungstests auszuarbeiten und kundengerecht anzufertigen. Die Sprache ist erweiterbar und wird geformt, um Elemente zum Stimulieren und Beobachten von Hardwarebauelementmodellen zur Verfügung zu stellen. Die Vorrichtung ist plattform- und simulatorunabhängig und wird zur Integration mit Funktionen von Verilog, VHDL und C angepasst. Eine modulare Systemumgebung sichert das Zusammenwirken mit jedem Simulator durch eine vereinheitlichte Systemschnittstelle, die mehrere externe Typen unterstützt. Ein Prüfgeneratormodul erzeugt aus einer Funktionsbeschreibung automatisch Überprüfungstests. Eine Testreihe kann eine beliebige Kombination von statistisch und dynamisch erzeugten Tests umfassen. Eine gerichtete Erzeugung schränkt erzeugte Tests auf spezifische Funktionalitäten ein. Prüfparameter werden an jedem Punkt während einer Entwicklung verändert, und die stochastische Stabilität wird unterstützt. Ein Prüfmodul kann eine beliebige Kombination von statischen und dynamischen Prüfungen durchführen. Differenzielles Prüfen erlaubt eine stufenweise Entwicklung von Testreihen während des Prozesses der Entwurfsentwicklung.

[0010] US-A-6 028 439 zeigt ein modulares Prüfgerät für integrierte Schaltkreise, das einen Satz von Prüfgerätmodulen umfasst, um eine Testreihe an einem integrierten Schaltkreis-Prüfobjekt (DUT) durchzuführen. Jedes Modul enthält einen Speicher zum Speichern von Befehlssätzen, die angeben wie das Modul für jeden Test der Reihenfolge zu konfigurieren ist. Vor dem Start jedes Testes führt ein Programmschaltwerk in jedem Modul einen Befehlssatz aus, um das Modul für den Test entsprechend zu konfigurieren. Danach sendet das Programmschaltwerk in jedem Modul ein Bereitschaftssignal an eine Start-Logikschaltung in jedem anderen Modul, das anzeigt, dass es bereit ist, den Test durchzuführen. Wenn die Programmschaltwerke aller an dem Test teilnehmenden Module signalisiert haben, dass sie bereit sind, signalisiert die Start-Logikschaltung in jedem Modul ihrem Programmschaltwerk, den Test zu beginnen. Die Module führen dann den Test mit ihren, einem Zeitgebersignal zugeordneten Aktivitäten aus. Der Konfigurierungsprozess der Module, das Erzeugen der Bereitschaftssignale und das Beginnen eines Tests werden für jeden Test der Folge wiederholt.

[0011] Schließlich offenbart US-A-2003/0005375 eine Prüfstation für ein Speicherprüfgerät, das einen oder mehrere Messplätze aufweist, die jeweils einzeln algorithmisch steuerbar sind, die sich jeweils mit nicht weniger als vierundsechzig Kanälen befassen können und die miteinander verbunden werden können, um eine Mehrpositions-Prüfstation von zwei oder mehreren Messplätzen zu bilden. Es können miteinander bis zu neun Messplätze als eine Einzelpositions-Prüfstation verbunden werden.

ABRISS DER ERFINDUNG

[0012] Die Erfindung berücksichtigt die Nachteile des oben erwähnten Standes der Technik und wird verwirklicht durch ein verteiltes System, das die Merkmale von Anspruch 1 aufweist. Bevorzugte Ausführungen dieses Systems sind in den Unteransprüchen genau erklärt.

[0013] Das Prüfsystem der offenen Architektur erlaubt nach einer Ausführung der Erfindung die Verwendung von Modulen Dritter. Das Hardware- und Software-Framework des Prüfsystems enthält Standardschnittstellen,

mit denen Module von unterschiedlichen Herstellern in einer Plug-and-Plag-Weise zusammenwirken können. Die Module können Hardware sein, wie beispielsweise eine Funktionseinheit, digitale Pinkarte, Analogkarte oder Bauelement-Stromversorgung oder Software wie ein Tool oder Zusatzgerät einschließlich eines Testausführungstools, Systemüberwachungs- oder Zulassungstools, Einheitsniveau-Reglers (z. B. Basisinstrument, Steuerung eines allgemeinen Interface-Bus), Datenbank oder Software zur Steuerung von anderer Ausrüstung.

[0014] In einer Ausführung ist die Architektur eine verteilte Objektumgebung unter dem Rechnerbetriebssystem Microsoft Windows. Das Prüfgerät ist ein modularisiertes System mit Modulsteuersoftware und einer Backplane-Kommunikationsbibliothek, die sowohl eine Kommunikation von Modul zu Modul als auch eine Kommunikation von Regler zu Modul ermöglicht. Die Module umfassen zum Beispiel Digitalbausteine, Module der Gerätestromversorgung (DPS), beliebige Wellengeneratormodule (AWG), Digitalisiermodule und anwendungsspezifische Software.

[0015] In einer Ausführung umfasst ein Modulverbindungs-Aktivierungsprogramm ein Schaltmatrixnetzwerk, das einen Mehrmodulverbindungs- und Synchronisierungsmechanismus zur Verfügung stellt. Wenn mehrere Prüfobjekte (DUT) des gleichen Typs getestet werden, ermöglicht dieses Schaltmatrixnetzwerk außerdem die Aufteilung von gemeinsamen Prüfdaten und Ressourcen zwischen mehreren Steuereinheiten und Messplätzen.

[0016] Wegen der Stationssteuervorrichtungen, die je Standort unabhängig sind, können alle Messplätze asynchron in Betrieb sein. Dies erleichtert eigentlich den Mehrfachtest von Prüfobjekten. Diese Modularisierung und Konfiguration mehrerer Standorte bewirkt außerdem eine Skalierbarkeit für das System. In einer Ausführung des Systems kann ein einzelner Regler zum Steuern und Testen mehrerer DUT konfiguriert werden.

[0017] Das Konzept von Plug-and-Plag-Modulen oder austauschbaren Modulen wird durch Verwendung von Standardschnittstellen sowohl auf dem Niveau von Hardware als auch dem Niveau von Software erleichtert. Bei Software werden Framework-Klassen verwendet, um die Module freizugeben, zu aktivieren, zu steuern und zu überwachen. Das Framework ist ein Satz von Klassen und Methoden, die gemeinsame testbezogene Operationen ausführen. Dieses umfasst Klassen für Stromversorgung und Fortschaltung von Pinelektronik, das Einstellen von Strom-/Spannungspegel und von Zeitablaufbedingungen, das Erhalten von Messungen, das Steuern eines Prüfablaufes, usw.. Das Framework liefert außerdem Methoden für Laufzeitdienste und Fehlerbeseitigung. Framework-Objekte können durch das Implementieren von Standardschnittstellen nach einer Ausführung der Erfindung arbeiten. Es ist eine auf C++ basierende Referenzimplementierung der Framework-Klassen vorgesehen. Ein Anwender kann außerdem seine eigenen spezifischen Framework-Klassen entwickeln.

[0018] Schnittstellenrealisierung von Hardware-Software und Übertragung werden durch eine Backplane-Kommunikationsbibliothek erhalten. Eine offene Backplane-Kommunikationsbibliothek, auf die man über ein auf C++-Sprache basierendes Prüfprogramm und eine Prüfprogrammierschicht der grafischen Benutzeroberfläche (GUI) über C++ Zugriff hat, stellt eine generalisierte Anwenderschnittstelle für das Prüfsystem bereit. Das Verfahren zum Erzeugen eines Prüfprogramms unter Verwendung von C/C++ Konstrukten ist in der US-Patentanmeldung Nr. 60/447 839 offenbart. Die Kommunikationsbibliothek bewirkt den Mechanismus zum Kommunizieren mit den Stationssteuervorrichtungen in einer Weise, die für Benutzeranwendungen und Prüfprogramme transparent ist. Von entscheidender Bedeutung ist, dass die Backplane-Kommunikationsbibliothek die Schnittstelle zur Verfügung stellt, die für Übertragungen über der Prüfgerät-Rückwandplatine (in diesem Zusammenhang ist „Backplane" eine abstrakte, nicht zwangsläufig physikalische Hardware-Rückverdrahtungsleiterplatte) beabsichtigt ist, wodurch die Funktionen bereitgestellt werden, die zum Kommunizieren mit den an speziellen Standorten angeschlossenen Modulen notwendig sind. Eine Verwendung dieser Bibliothek schließt die Notwendigkeit aus, dass Modulhersteller ihre eigenen Treiber (wie Treiber auf der Ebene von MS-Windows) schaffen. Dies ermöglicht es, dass herstellerspezifische Modulsoftware den normalen Backplane-Treiber nutzt, um mit den entsprechenden Hardware-Modulen zu kommunizieren. In einer Ausführung nutzt das Backplane-Kommunikationsprotokoll ein paketbasiertes Format.

[0019] Ein Vorteil der offenen Architektur ist, dass sie die gesamte Nutzung des Prüfgerätes vereinfacht. Sie bewirkt einen Mechanismus zum Entwickeln von Lösungen Dritter und zur Wiederverwendung dieser Lösungen ohne größere neue Bearbeitung. Für einen beliebigen vorgegebenen Messplatz können je nach Bedarf geeignete Module ausgewählt und genutzt werden. Weil Module auswechselbar sind, kann jeder Messplatz umkonfiguriert werden, um die optimale Prüfung eines Prüfobjekts (DUT) zu erreichen. Sie vereinfacht außerdem das Ergebnis plattformübergreifender Unverträglichkeit. Alle diese Vereinfachungen führen zu reduzier-

tem Aufwand, schnellerer Durchlaufzeit und folglich verringerten Prüfkosten.

[0020] Die vorliegende Erfindung stellt ein System zum Prüfen von mindestens einem Prüfobjekt (DUT) bereit. Das System umfasst mindestens eine Stationssteuervorrichtung zum Steuern von zumindest einem Prüfmodul, um mindestens einen Test (der Teil eines Prüfplanes sein kann) für mindestens ein DUT anzuwenden. Ein Systemregler steuert die zumindest eine Stationssteuervorrichtung.

[0021] Eine Prüfmodul-Schnittstelle legt die Prüfmodulfunktionen zur Schnittstellenrealisierung einer Stationssteuervorrichtung mit einem ersten Prüfmodul fest, wobei die Prüfmodul-Schnittstelle erweiterbar ist, um die Stationssteuervorrichtung an ein zweites Prüfmodul anzupassen, die nicht erweiterte Prüfmodul-Schnittstelle unfähig ist, um die Stationssteuervorrichtung an das zweite Prüfmodul anzupassen.

[0022] Das System umfasst außerdem eine erweiterbare Prüffunktion wie beispielsweise eine benutzerdefinierbare Testklasse, die von DUT-spezifischen Eigenschaften unabhängig ist. Ein Test ist eine praktische Ausführung der erweiterbaren Prüffunktion.

[0023] Ein Prüfmodul kann mit dem eine Prüfgerätepin-Schnittstelle nutzenden Prüfobjekt, die von DUT-spezifischen Eigenschaften unabhängig sein kann, kommunizieren. Die Prüfmodul-Schnittstelle kann eine Prüfmodul-Schnittstellenklasse und die Prüfgerätepin-Schnittstelle eine Prüfgerätepin-Schnittstellenklasse aufweisen.

[0024] Ein verteiltes System einer Ausführung der Erfindung umfasst ein Host-Betriebssystem, um die Steuerung von mindestens einer Stationssteuervorrichtung durch eine Systemsteuervorrichtung zu ermöglichen, und mindestens ein lokales Betriebssystem, das mit jeder Stationssteuervorrichtung assoziiert ist, um die Steuerung von mindestens einem Prüfmodul durch eine assoziierte Stationssteuervorrichtung zu ermöglichen. Zumindest ein Prüfmodul führt das Prüfen an einem entsprechenden DUT aus.

[0025] Das Host-Betriebssystem kann die Operation der zumindest einen Stationssteuervorrichtung synchronisieren, die Kommunikation zwischen der Systemsteuervorrichtung und der zumindest einen Stationssteuervorrichtung entscheiden und die Operation der zumindest einen Stationssteuervorrichtung überwachen. Eine Stationssteuervorrichtung kann die Operation des zumindest einen Prüfmoduls, das mit der Stationssteuervorrichtung assoziiert ist, überwachen.

[0026] Das Host-Betriebssystem weist mindestens eine Host-Schnittstelle zum Kommunizieren mit der zumindest einen Stationssteuervorrichtung auf. Eine Prüfmodulschnittstelle legt die Prüfmodulfunktionen zur Schnittstellenrealisierung einer Stationssteuervorrichtung mit einem ersten Prüfmodul fest, wobei die Prüfmodulschnittstelle erweiterbar ist, um die Stationssteuervorrichtung an ein zweites Prüfmodul anzupassen, wobei die nicht erweiterte Prüfmodulschnittstelle unfähig ist, die Stationssteuervorrichtung an das zweite Prüfmodul anzupassen.

[0027] Das Host-Betriebssystem kann zumindest eine einheitliche Framework-Kiasse umfassen, die in einer Standardcomputersprache (z. B. C/C++) entwickelt werden kann, um es einem Benutzer zu ermöglichen, anwendungsspezifische Klassen zum Steuern der zumindest einen Stationssteuervorrichtung zu entwickeln.

[0028] Jedes lokale Betriebssystem kann zumindest eine lokale Framework-Klasse enthalten, die in einer Standardcomputersprache (C/C++) entwickelt werden kann, um es einem Benutzer zu ermöglichen, anwendungsspezifische Klassen zum Steuern des zumindest einen Prüfmoduls zu entwickeln.

[0029] Die Anzahl von Modulen, die durch jede Stationssteuervorrichtung gesteuert werden, ist skalierbar. Das lokale Betriebssystem, das mit der entsprechenden Stationssteuervorrichtung assoziiert ist, ermöglicht es, den Typ von Prüfmodulen, die durch die Stationssteuervorrichtung gesteuert werden, umzustrukturieren. Das Host-Betriebssystem ermöglicht es, dass die Anzahl von durch die Systemsteuervorrichtung gesteuerten Stationssteuervorrichtungen skalierbar ist und die Anzahl von durch das Prüfsystem getesteten DUT skalierbar ist.

[0030] Ein Emulator kann die Nutzung des Prüfmoduls eines Bewerbers mit dem Prüfsystem simulieren, um das Modul des Bewerbers als kompatibel mit dem Prüfsystem nachzuprüfen.

KURZE BESCHREIBUNG DER ZEICHNUNGEN

[0031] Fig. 1 veranschaulicht eine herkömmliche Prüfgeräte-Architektur;

[0032] Fig. 2 veranschaulicht eine Systemarchitektur nach einer Ausführung der vorliegenden Erfindung;

[0033] Fig. 3 veranschaulicht eine Software-Architektur nach einer Ausführung der vorliegenden Erfindung;

[0034] Fig. 4 veranschaulicht die Verwendung von Testklassen nach einer Ausführung der Erfindung;

[0035] Fig. 5 ist eine schematische Darstellung der Vereinheitlichten Modellierungssprache (UML), die das Zusammenwirken eines Prüfgerätesystems und unterschiedliche, von dem Hersteller gelieferte Modulressourcen nach einer Ausführung der Erfindung veranschaulicht;

[0036] Fig. 6 stellt eine Ausführung von Stationssteuervorrichtungsobjekten zur Leitung eines Benutzertests dar, wie er durch eine Stationssteuerungsvorrichtung geführt wird;

[0037] Fig. 7 veranschaulicht die Ausführung eines Objektersatzes auf der Seite der Systemsteuervorrichtung, der die in Fig. 6 gezeigte Stationssteuervorrichtung darstellt;

[0038] Fig. 8 veranschaulicht eine Prüfumgebung nach einer Ausführung der Erfindung.

AUSFÜHRLICHE BESCHREIBUNG BEISPIELHAFTER AUSFÜHRUNGEN

[0039] Fig. 1 stellt die generalisierte Architektur eines herkömmlichen Prüfgerätes dar, die zeigt, wie ein Signal erzeugt und auf ein Prüfobjekt (DUT) angewendet wird. Jeder DUT-Eingangspin ist mit einem Treiber **2** verbunden, der Testdaten anwendet, während jeder DUT-Ausgangspin mit einem Komparator **4** verbunden ist. In den meisten Fällen werden Dreifach-Treiber-Komparatoren verwendet, so dass jeder Prüfgerätepin (Kanal) entweder als ein Eingangspin oder ein Ausgangspin wirksam werden kann. Die einem einzelnen DUT zugeordneten Prüfgerätepins bilden zusammen einen Messplatz, der mit einem verknüpften Zeitsignalgenerator **6**, Wellenformgenerator **8**, Strukturspeicher **10**, Zeitdatenspeicher **12**, Wellenformspeicherdaten **14** und Block **16**, der die Datenrate festlegt, arbeitet.

[0040] Fig. 2 stellt die Systemarchitektur **100** nach einer Ausführung der vorliegenden Erfindung dar. Ein Systemregler (SysC) **102** ist mit mehreren Stationsreglern (SiteCs) **104** verbunden. Der Systemregler kann auch mit einem Netzwerk gekoppelt werden, um auf zugeordnete Datensätze zuzugreifen. Durch eine Modulverbindungs-Aktivierungseinrichtung **600** ist jeder Stationsregler gekoppelt, um ein oder mehrere an einem Messplatz **110** örtlich festgelegte Prüfmodule **108** zu steuern. Die Modulverbindungs-Aktivierungseinrichtung **106** ermöglicht den Austausch von verbundenen Hardwaremodulen **108** und dient außerdem als Bus zur Datenübertragung (zum Laden von Strukturdaten, Sammeln von Antwortdaten, Bereitstellung von Steuerung, usw.) Außerdem kann durch die Modulverbindungs-Aktivierungseinrichtung ein Modul an einer Station auf ein Modul an einer anderen Station zugreifen. Die Modulverbindungs-Aktivierungseinrichtung **106** ermöglicht es, dass unterschiedliche Messplätze gleiche oder unterschiedliche Modulkonfigurationen aufweisen. Mit anderen Worten kann jeder Messplatz unterschiedliche Anzahl und Typen von Modulen einsetzen. Mögliche Hardwareausführungen umfassen zugeordnete Verbindungen, Schalteranschlüsse, Busanschlüsse, Ringanschlüsse und Sternschaltungen. Die Modulverbindungs-Aktivierungseinrichtung **106** kann zum Beispiel durch eine Schaltmatrix implementiert werden. Jeder Messplatz **110** ist einem DUT **112** zugeordnet, der durch ein Lastmodul **114** mit den Modulen der entsprechenden Station verbunden ist. In einer Ausführung kann eine einzelne Stationssteuervorrichtung mit mehreren DUT-Stationen verbunden sein.

[0041] Der Systemregler **102** dient als Manager des Gesamtsystems. Er koordiniert die Aktivitäten der Stationssteuervorrichtung, leitet zur Systemebene parallele Teststrategien und sorgt zusätzlich sowohl für die Steuerung von Handhabungsprogrammen/Messköpfen als auch die Datenerfassung auf Systemebene und Unterstützung bei Fehlerbehandlung. In Abhängigkeit von der betrieblichen Einstellung kann der Systemregler **102** an einer CPU eingesetzt werden, die getrennt ist von der Funktion der Stationssteuervorrichtungen **104**. Alternativ dazu kann eine gemeinsame CPU durch die Systemsteuervorrichtung **102** und die Stationssteuervorrichtungen **104** geteilt werden. Entsprechend kann jede Stationssteuervorrichtung **104** an ihrer eigenen zugeordneten CPU (zentrale Verarbeitungseinheit) oder als ein getrennter Prozess oder Befehlsfolge innerhalb der gleichen CPU eingesetzt werden.

[0042] Die Systemarchitektur kann man sich konzeptionell als das in Fig. 2 gezeigte, verteilte System mit dem Verständnis vorstellen, dass die einzelnen Systemkomponenten auch als logische Komponenten eines integrierten monolithischen Systems und nicht zwangsläufig als physikalische Komponenten eines verteilten Systems betrachtet werden könnten.

[0043] Fig. 3 veranschaulicht eine Software-Architektur **200** nach einer Ausführung der vorliegenden Erfindung. Die Software-Architektur **200** stellt ein verteiltes Betriebssystem dar, das Elemente für den Systemregler **220** aufweist, zumindest eine Stationssteuervorrichtung **240** und zumindest ein Modul **260** in Übereinstimmung mit Elementen **102**, **104**, **108** des zugeordneten Hardwaresystems. Zusätzlich zu dem Modul **260** umfasst die Architektur **200** ein entsprechendes Element zur Modulemulation **280** in Software.

[0044] Als eine beispielhafte Wahl kann die Entwicklungsumgebung für diese Plattform auf Microsoft Windows basieren. Die Nutzung dieser Architektur besitzt nebenbei Vorteile hinsichtlich der Übertragbarkeit von Programmen und Unterhaltung (z. B. könnte ein Kundendienstingenieur einen Laptop anschließen, der auf dem Rechnerbetriebssystem des Prüfgerätes läuft, um eine moderne Diagnose durchzuführen). Für umfassende berechnungsintensive Operationen (wie beispielsweise Zusammenstellungen von Prüfmustern) kann die relevante Software jedoch als unabhängige separate Informationseinheit hergestellt werden, die in der Lage ist, unabhängig zu laufen, um eine Jobbelegung über verteilte Plattformen zu ermöglichen. So sind zugeordnete Software-Tools für Stapeljobs in der Lage, auf mehreren Plattformtypen zu laufen.

[0045] Als beispielhafte Wahl kann die C++ nach ANSI/ISO Standard als Muttersprache für die Software genutzt werden. Natürlich ist eine Vielfalt von Optionen verfügbar (um eine Schicht über den nominalen C++-Schnittstellen zur Verfügung zu stellen), die es Dritten erlauben, sie mit einer alternativen Sprache ihrer eigenen Wahl in das System zu integrieren.

[0046] Fig. 3 veranschaulicht eine schattierte Darstellung von Elementen entsprechend ihrer Organisierung durch nominelle Quelle (oder kollektive Entwicklung als ein Teilsystem) einschließlich der Schnittstellen **290** des Prüfgerät-Rechnerbetriebssystems, Benutzerkomponenten **292** (z. B. geliefert durch einen Benutzer für Testzwecke), Systemkomponenten **294** (z. B. geliefert als Softwareinfrastruktur für grundlegende Vernetzungsmöglichkeit und Kommunikation), Modulentwicklungskomponenten **296** (z. B. geliefert durch einen Modulentwickler), und externe Komponenten **298** (z. B. geliefert durch externe Quellen außer Modulentwickler).

[0047] Aus der Perspektive einer quellen-basierten Organisation umfassen die Schnittstellen **290** des Prüfgerät-Betriebssystems (TOS): Systemregler an Stationssteuervorrichtungsschnittstellen **222**, Framework-Klassen **224**, Stationssteuervorrichtung an Modulschnittstellen **245**, Framework-Klassen **246**, vorgegebene Modulniveau-Schnittstellen, Backplane-Kommunikationsbibliothek **249**, Aufbauplattenschlitz IF (Schnittstelle) **262**, Lastmodul-Hardware IF **264**, Backplane-Simulations-IF **283**, Lastmodul-Simulations-IF **285**, DUT-Simulations-IF **287**, Verilog PLI (Programmiersprachen-Schnittstelle) **288** für Verilog-Modell des DUT und C/C++ Spracheunterstützung **289** für C/C++ Modell des DUT.

[0048] Benutzerkomponenten **292** umfassen: einen Benutzerprüfplan **242**, Benutzertestklassen **243**, Hardware-Lastmodul **265** und DUT **266**, ein Verilog-Modell **293** des DUT und ein C/C++ Modell **291** des DUT.

[0049] Systemkomponenten **294** umfassen: Systemtools **226**, Kommunikationsbibliothek **230**, Testklassen **244**, einen Backplane-Treiber **250**, HW-Backplane **261**, Simulations-Framework **281**, Backplane-Emulation **282** und Lastmodul-Simulation **286**.

[0050] Modulentwicklungskomponenten **296** umfassen: Modulbefehlsausführung **248**, Modulhardware **263** und Modulemulation **284**.

[0051] Die externen Komponenten **298** umfassen externe Tools **225**.

[0052] Der Systemregler **220** umfasst Schnittstellen **222** zu Stationssteuervorrichtung, Framework-Klassen **224**, Systemtools **226**, externe Tools **225** und eine Kommunikationsbibliothek **230**. Die Software des Systemreglers ist die primäre Übergabestelle für den Benutzer. Sie bewirkt den Netzkoppler für die Stationssteuervorrichtungen der Erfindung und eine Synchronisierung der Stationssteuervorrichtungen in einer Mehrstations-/DUT-Umgebung, wie es in der US-Patentanmeldung Nr. 60/449 622 durch den gleichen Rechtsnachfolger beschrieben ist. Benutzeranwendungen und Tools auf Basis der grafischen Benutzeroberfläche (GUI) oder anderweitig laufen auf dem Systemregler. Der Systemregler kann außerdem als der Verwahrungsort für alle auf den Prüfplan bezogenen Informationen einschließlich Prüfpläne, Prüfmuster und Prüfparameter-Datensätze wirksam werden. Ein Prüfparameter-Datensatz enthält Parametrierungsdaten für eine Testklasse in der objektorientierten Umgebung einer Ausführung der Erfindung.

[0053] Drittentwickler können Tools zusätzlich zu den normalen Systemtools **226** (oder als Ersetzungen dafür) bereitstellen. Die Standardschnittstellen **222** auf dem Systemregler **220** umfassen Schnittstellen, die die

Tools zum Zugriff auf das Prüfgerät und Prüfobjekte nutzen. Die Tools (Anwendungen) **225**, **226** ermöglichen eine interaktive Steuerung und Stapelsteuerung der Prüf- und Prüfgerätobjekte. Die Tools umfassen Anwendungen zur Bereitstellung von Automatisierungsfähigkeiten (zum Beispiel durch die Verwendung von SECS/TSEM, usw.)

[0054] Die auf dem Systemregler **220** zu findende Kommunikationsbibliothek **230** bewirkt den Mechanismus zum Kommunizieren mit den Stationssteuervorrichtungen **240** in einer Weise, die für Benutzeranwendungen und Prüfprogramme transparent ist.

[0055] Die Schnittstellen **222**, die in dem Speicher resident sind, der dem Systemregler **220** zugeordnet ist, liefern offene Schnittstellen für die Framework-Objekte, die auf dem Systemregler ausführen. Enthalten sind Schnittstellen, die es der Modulsoftware auf Basis der Stationssteuervorrichtungen erlauben, auf Strukturdaten zuzugreifen und wiederzugewinnen. Enthalten sind auch sowohl Schnittstellen, die Anwendungen und Tools nutzen, um Zugriff auf das Prüfgerät und Prüfobjekte zu haben, als auch Scripting-Schnittstellen, die die Möglichkeit schaffen, durch eine Scripting-Engine auf das Prüfgerät und Testkomponenten zuzugreifen und zu manipulieren. Dies ermöglicht einen gemeinsamen Mechanismus für interaktive Anwendungen, Stapelanwendungen und Fernanwendungen, um ihre Funktionen auszuführen.

[0056] Die dem Systemregler **220** zugeordneten Framework-Klassen **224** bewirken einen Mechanismus, um mit diesen oben erwähnten Objekten in Wechselwirkung zu treten, indem eine Referenzimplementierung einer Standardschnittstelle bewirkt wird. Zum Beispiel liefert die Stationssteuervorrichtung **240** nach der Erfindung ein Funktionstestobjekt. Die Framework-Klassen des Systemreglers können einen entsprechenden Funktionstest-Stellvertreter als einen auf dem Systemregler basierenden entfernten Ersatz des Funktionstestobjekts bewirken. So wird die Funktionstest-Standardschnittstelle den Tools auf dem Systemregler **220** verfügbar gemacht. Das System, Modulentwicklungs- und Schnittstellenkomponenten **294**, **296** und **290** können jeweils als ein Rechnerbetriebssystem betrachtet werden, das zwischen dem Systemregler und den Stationsteuervorrichtungen verteilt ist. Die Framework-Klassen bewirken effektiv eine mit dem Host-Systemregler verbundene Betriebssystem-Schnittstelle. Sie stellen außerdem die Softwareelemente dar, die den Netzkoppler zu den Stationssteuervorrichtungen bewirken und eine Synchronisierung der Stationssteuervorrichtungen in einer Mehrstellen-/DUT Umgebung zur Verfügung stellen. Diese Schicht liefert somit ein Objektmodell in einer Ausführung nach der Erfindung, das zum Manipulieren und Zugreifen von Stationssteuervorrichtungen geeignet ist, ohne sich direkt mit der Kommunikationsschicht befassen zu müssen.

[0057] Die Stationssteuervorrichtung **240** richtet einen Benutzerprüfplan **242**, Benutzertestklassen **243**, Standardtestklassen **244**, Standardschnittstellen **245**, Stationssteuervorrichtungs-Framework-Klassen **246**, Modulschnittstellen **247** mit Hochpegel-Befehl (d. h. vorgegebene Modulpegel-Schnittstellen), Modulbefehlsausführung **248**, Backplane-Kommunikationsbibliothek **249** und einen Backplane-Treiber **250** aus. Vorzugsweise wird der größte Teil der Testfunktionstüchtigkeit durch die Stationssteuervorrichtungen **104**/**240** gehandhabt, was folglich eine unabhängige Arbeitsweise der Messplätze **110** erlaubt.

[0058] Ein Prüfplan **242** wird durch den Benutzer geschrieben. Der Plan kann direkt in einer Standardcomputersprache wie C++ geschrieben werden oder in einer Prüfprogrammiersprache mit höherem Niveau beschrieben werden, um einen C++ Code zu erzeugen, der anschließend in das ausführbare Prüfprogramm kompiliert werden kann.

[0059] Der Prüfplan erzeugt Prüfobjekte durch Nutzung der Framework-Klassen **246** und/oder normaler oder dem Benutzer gelieferter Testklassen **244**, die den Stationssteuervorrichtungen zugeordnet sind, konfiguriert die Hardware mit Standardschnittstellen **245** und definiert den Prüfplanablauf. Er stellt außerdem eine beliebige zusätzliche Logik zur Verfügung, die während einer Ausführung des Prüfplans erforderlich ist. Der Prüfplan unterstützt einige grundlegende Dienste und bewirkt eine Schnittstelle für die Dienste der zugrunde liegenden Objekte wie Fehlersuchdienste (z. B. Zwischenstoppen) und Zugriff auf zugrunde liegende Framework- und Standardklassen.

[0060] Die den Stationssteuervorrichtungen zugeordneten Framework-Klassen **246** sind ein Satz von Klassen und Methoden, die gemeinsame testbezogene Operationen durchführen. Das Framework auf Stationssteuervorrichtungsniveau umfasst zum Beispiel Klassen für Stromversorgung und Fortschaltung von Pinelektronik, das Einstellen von Bedingungen für Pegel und zeitliche Steuerung, das Erhalten von Messungen und Steuern eines Prüfablaufes, usw.. Das Framework liefert außerdem Methoden für Laufzeitdienste und Fehlerbeseitigung. Die Framework-Objekte können durch Ausführung der Standardschnittstellen arbeiten. Zum Beispiel wird die Ausführung der Prüfgerätepin-Framework-Klasse standardisiert, um eine allgemeine Prüfgeräte-

pin-Schnittstelle zu implementieren, die die Testklassen nutzen können, um mit Hardwaremodulpins in Wechselwirkung zu treten.

[0061] Es können bestimmte Framework-Objekte implementiert werden, die mit der Unterstützung der Modulpegel-Schnittstellen **247** arbeiten, um mit den Modulen zu kommunizieren. Die Framework-Klassen der Stationssteuervorrichtungen sind effektiv wie eine lokale Rechnerbetriebssystem-Schnittstelle wirksam, die jede Stationssteuervorrichtung unterstützt.

[0062] Im Allgemeinen sind mehr als neunzig Prozent des Programmcodes Daten für den Gerätetest und die restlichen zehn Prozent des Codes realisieren die Prüfmethodik. Die Geräteprüfdaten sind vom Prüfobjekt abhängig (z. B. Bedingungen der Stromversorgung, Bedingungen der Signalspannung, Zeitsteuerungsbedingungen, usw.). Der Prüfcode besteht aus Methoden zum Laden speziell festgelegter Gerätebedingungen in ATE Hardware (Hardware automatischer Prüfeinrichtungen) und auch derjenigen, die benötigt werden, um benutzerspezifische Entwurfsziele (wie beispielsweise Datenerfassung) zu realisieren. Das Framework einer Ausführung der Erfindung stellt ein hardwareunabhängiges Modell für Prüf- und Prüfgeräteobjekte bereit, das es dem Benutzer erlaubt, die Aufgabe von DUT-Prüfprogrammierung durchzuführen.

[0063] Zur Erhöhung der Wiederverwertbarkeit eines Prüfcodes kann ein solcher Code von beliebigen gerätespezifischen Daten (z. B. Pinname, Auslöseimpuls-Daten, usw.) oder von gerätetestspezifischen Daten (z. B. Bedingungen für Gleichspannungsgeräte, Messpins, Anzahl von Zielpins, Strukturdatensatzname, Adressen von Strukturprogrammen) unabhängig gemacht werden. Falls ein Code für eine Prüfung mit Daten dieses Typs kompiliert wird, würde die Wiederverwertbarkeit des Prüfcodes abnehmen. Daher können dem Prüfcode nach einer Ausführung der Erfindung beliebige gerätespezifische Daten oder gerätetestspezifische Daten als Eingaben während der Codeausführungszeit extern zur Verfügung gestellt werden.

[0064] In einer Ausführung der Erfindung realisiert eine Testklasse, die die Ausführung einer hier als ITest bezeichneten Standardtestschnittstelle ist, die Trennung von Prüfdaten und Code (und damit die Code-Wiederverwertbarkeit) für einen speziellen Testtyp. Eine solche Testklasse kann als „Dokumentvorlage" für ihre getrennten Fälle betrachtet werden, die nur auf der Basis von gerätespezifischen und/oder gerätetestspezifischen Daten voneinander abweichen. Testklassen sind im Einzelnen in dem Prüfplan-Datensatz angegeben. Jede Testklasse implementiert typischerweise einen spezifischen Typ von Gerätetest oder Einstellung für Gerätetest. Zum Beispiel kann eine Ausführung der Erfindung eine spezielle Ausführung der Schnittstelle ITest, zum Beispiel FunctionalTest, als die Basisklasse für alle Funktionstests für DUT zur Verfügung stellen. Sie liefert die grundlegende Funktionalität des Einstellens von Prüfbedingungen, das Ausführen von Strukturen und Bestimmen des Status der Prüfung basierend auf dem Vorhandensein von Messmarken, die ihren Zweck verfehlt haben. Andere Typen von Ausführungen können Wechsel- und Gleichstrom-Testklassen umfassen, die hier als ACParametricTests und DCParametricTests bezeichnet sind.

[0065] Alle Testtypen können Standardausführungen einiger virtueller Methoden (z. B. init(), preExec() und postExec()) bewirken. Diese Methoden werden die Einsprungstellen des Prüfingenieurs zum Übersteuern von vorgegebenem Verhalten und Einstellen beliebiger testspezifischer Parameter. In den Prüfplänen können jedoch auch kundenspezifische Testklassen verwendet werden.

[0066] Testklassen ermöglichen es dem Benutzer, Klassenverhalten durch Bereitstellung von Parametern zu konfigurieren, die genutzt werden, um die Optionen für einen besonderen Fall dieses Tests im Einzelnen festzulegen. Zum Beispiel kann ein Funktionstest die Parameter PList und TestConditions annehmen, um jeweils im Einzelnen die Strukturliste zum Abarbeiten und die Pegel- und Zeitsteuerungsbedingungen für den Test festzulegen. Das Bestimmen von unterschiedlichen Werten für diese Parameter (durch die Verwendung unterschiedlicher „Testblocks" in einem Testplan-Beschreibungsdatensatz) ermöglicht es dem Benutzer, unterschiedliche Fälle eines Funktionstests zu erzeugen. **Fig. 4** veranschaulicht, wie unterschiedliche Testfälle von einer einzelnen Testklasse abgeleitet werden können. Eine Dokumentvorlage-Bibliothek kann als die Universalbibliothek von generischen Algorithmen und Datenstrukturen eingesetzt werden. Diese Bibliothek kann für einen Benutzer des Prüfgerätes sichtbar sein, so dass der Benutzer zum Beispiel die Ausführung einer Testklasse zum Erzeugen einer benutzerdefinierten Testklasse modifizieren kann.

[0067] Hinsichtlich benutzerentwickelter Testklassen unterstützt eine Ausführung des Systems die Integration solcher Testklassen zu dem Framework dadurch, dass sich alle Testklassen von einer einzelnen Testschnittstelle, z. B. ITest, ableiten, so dass das Framework sie in der gleichen Weise manipulieren kann wie den Standardsatz von Systemtestklassen. Benutzern steht es frei, in ihre Testklassen eine zusätzliche Funktionstüchtigkeit einzubeziehen, mit dem Verständnis, dass sie in ihren Prüfprogrammen einen Code nach Kunden-

wunsch verwenden müssen, um aus diesen zusätzlichen Leistungsmerkmalen Nutzen zu ziehen.

[0068] Jedem Messplatz **110** ist das Prüfen von einem oder mehreren DUT **106** zugeordnet und funktioniert durch eine konfigurierbare Sammlung von Prüfmodulen **112**. Jedes Prüfmodul **112** ist eine separate Informationseinheit, die eine spezielle Prüfaufgabe durchführt. Zum Beispiel könnte ein Prüfmodul **112** die Stromversorgung eines DUT, eine Pinkarte, eine Analogkarte, usw. sein. Diese modulare Lösung bewirkt einen hohen Grad von Flexibilität und Konfigurierbarkeit.

[0069] Die Modulbefehl-Ausführungsklassen **248** können durch Hersteller von Modulhardware bereitgestellt werden und implementieren entweder die Modulpegel-Schnittstellen für Hardwaremodule oder bewirken modulspezifische Ausführungen von Standardschnittstellen in Abhängigkeit von dem von einem Hersteller gewählten Befehlsausführungsverfahren. Die externen Schnittstellen dieser Klassen werden durch vorbestimmte Anforderungen der Modulpegel-Schnittstellen und Anforderungen an die Backplane-Kommunikationsbibliothek definiert. Diese Schicht sorgt außerdem für eine Erweiterung des Standardsatzes von Prüfbefehlen, die die Ergänzung von Methoden (Funktionen) und Datenelementen ermöglicht.

[0070] Die Backplane-Kommunikationsbibliothek **249** erzeugt die Schnittstelle für Standardkommunikation über die Backplane, wodurch die Funktionen bereitgestellt werden, die zum Kommunizieren mit den an den Messplatz angeschlossenen Modulen notwendig sind. Dies ermöglicht es, dass herstellerspezifische Modulsoftware einen Backplane-Treiber **250** zum Kommunizieren mit den entsprechenden Hardware-Modulen nutzt. Das Backplane-Kommunikationsprotokoll kann ein auf Paket beruhendes Format verwenden.

[0071] Prüfgerätepin-Objekte stellen physikalische Prüfgerätekanäle dar und leiten sich von einer Prüfgerätepin-Schnittstelle ab, die hier als ITesterPin bezeichnet ist. Der Softwareentwicklungs-Bausatz (SDK) einer Ausführung der Erfindung stellt eine Standardausführung von ITesterPin bereit, die TesterPin genannt werden kann, die in Form einer vorgegebenen Modulpegel-Schnittstelle, IChannel, implementiert wird. Herstellern steht es frei, von TesterPin Gebrauch zu machen, wenn sie die Funktionsvielfalt ihrer Module in Form von IChannel implementieren können; andererseits müssen sie eine Ausführung von ITesterPin bewirken, um mit ihrem Modul zu arbeiten.

[0072] Die hier als IModule bezeichnete Standardmodul-Schnittstelle, die durch das Prüfgerätesystem nach der Erfindung vorgesehen ist, stellt allgemein ein Hardwaremodul des Herstellers dar. Vom Hersteller gelieferte, modulspezifische Software für das System kann in Form von Executables wie beispielsweise Dateien für Betriebssystemroutinen (DLL) bereitgestellt werden. Software für jeden Modultyp von einem Hersteller kann in einer einzelnen DLL eingebettet sein. Jedes derartige Softwaremodul ist verantwortlich für die Bereitstellung von herstellerspezifischen Ausführungen für die Modulschnittstellenbefehle, die die API (Anwendungsprogrammierschnittstelle) für Modulsoftwareentwicklung aufweisen.

[0073] Es gibt zwei Aspekte der Modulschnittstellenbefehle: Erstens dienen sie als Schnittstelle für Benutzer zum Kommunizieren (indirekt) mit einem speziellen Hardwaremodul in dem System, und zweitens stellen sie die Schnittstellen bereit, von denen die Drittentwickler Nutzen ziehen können, um ihre eigenen Module in das Stationssteuervorrichtungspegel-Framework zu integrieren. Folglich werden die durch das Framework zur Verfügung gestellten Modulschnittstellenbefehle in zwei Typen eingeteilt:
Die ersten und deutlichsten sind diejenigen „Befehle", die dem Benutzer durch die Framework-Schnittstellen gezeigt werden. So liefert eine Prüfgerätepin-Schnittstelle (ITesterPin) Methoden, um Werte von Pegeln und zeitlicher Steuerung zu erhalten und einzustellen, während eine Stromversorgungsschnittstelle (IPowerSupply) zum Beispiel Methoden zum Netzeinschalten und Netzausschalten liefert.

[0074] Außerdem stellt das Framework die spezielle Kategorie der vorgegebenen Modulpegel-Schnittstellen bereit, die zum Kommunizieren mit den Modulen genutzt werden können. Diese sind die von Framework-Klassen genutzten Schnittstellen (d. h. „Standardausführungen" von Standardschnittstellen) zum Kommunizieren mit Herstellermodulen.

[0075] Die Verwendung des zweiten Aspektes, die Modulpegel-Schnittstellen, ist jedoch optional. Der Vorteil so vorzugehen ist, dass Hersteller dann von den Ausführungen von Klassen wie beispielsweise ITesterPin und IPowerSupply, usw. Nutzen ziehen können, wenn sie sich auf den Inhalt spezieller Mitteilungen, die an ihre Hardware durch Implementieren der Modulpegel-Schnittstellen gesendet werden, konzentrieren. Wenn diese Schnittstellen für den Hersteller ungeeignet sind, können sie jedoch wählen, um ihre kundenspezifischen Ausführungen der Framework-Schnittstellen (z. B. Herstellerausführungen von ITesterPin, IPowerSupply, usw.) zur Verfügung zu stellen. Diese würden dann die kundenspezifische Funktionsvielfalt, die für ihre Hardware geeig-

net ist, bereitstellen.

[0076] Die Integration von modulspezifischer Herstellersoftware kann somit durch zwei unterschiedliche Mittel erreicht werden: kundenspezifische Ausführung relevanter Framework-Klassen und Schnittstellen oder kundenspezifische Ausführung der speziellen Kategorie von Modulpegelschnittstellen.

[0077] Als Nächstes wird eine beispielhafte Anwendung beider Methoden mit Hilfe von Fig. 5 gezeigt, die eine grafische Darstellung der Universellen Modellierungssprachenklasse (UML) ist, die die Wechselwirkung des Prüfgerätesystems nach einer Ausführung der Erfindung und vom Hersteller gelieferten Modulen darstellt.

[0078] Der Hersteller eines neuen Digitalbausteins, Dritthersteller A (TPA), stellt ein Softwaremodul zum Kommunizieren mit seinem Hardwaremodul zur Verfügung. Dieses Softwaremodul wird die Standardschnittstelle IModule implementieren. Wir wollen dieses Modulobjekt TPAPinModule nennen. Der Hersteller TPA ist in der Lage, von der Standardsystemausführung der hier als TesterPin bezeichneten ITesterPin-Schnittstelle Gebrauch zu machen, indem die relevante vorgegebene Modulpegel-Schnittstelle, in diesem Fall IChannel, in seinen Modul implementiert wird. Dies wird durch die Tatsache möglich gemacht, dass TesterPin normale vorgegebene Modulpegel-Schnittstellen wie beispielsweise IChannel nutzt, um mit Modulen zu kommunizieren. Deshalb stellt TPAPinModule Pins zur Verfügung, indem einfach TesterPin-Objekte erzeugt und gezeigt werden.

[0079] Wir betrachten jetzt einen anderen Hersteller, Dritthersteller B (TPB), der feststellt, dass die IChannel-Schnittstelle mit seiner Hardware nicht gut arbeitet. Deshalb muss TPB nicht nur seine eigene IModule-Implementierung (TPBPinModule) vorsehen, sondern auch eine Implementierung der ITesterPin-Schnittstelle TPBTesterPin.

[0080] Diese Lösung gibt Drittentwicklern ziemlich viel Flexibilität beim Wählen, wie ihre Hardware und Unterstützungssoftware zu entwickeln ist. Während sie die IModule-Schnittstelle implementieren müssen, können sie wählen, Modulpegel-Schnittstellen zu implementieren oder Objekte wie TesterPins zu implementieren, wenn sie es für angebracht halten.

[0081] Tatsächlich können Hersteller wählen, Prüfgerätepins zu implementieren, um Erweiterungen zu bewirken, die nicht in der Schnittstelle ITesterPin unterstützt werden. Das Framework wird Benutzern einen Mechanismus zum Wiedergewinnen einer speziellen Schnittstelle oder einer Ausführungs-Hinweisadresse zur Verfügung stellen. Das bedeutet, dass wenn ein Benutzercode eine ITesterPin-Hinweisadresse besitzt, wird das Framework in der Lage sein zu bestimmen, ob sie sozusagen auf ein Objekt TPBTesterPin hinweist, wenn sie das muss. (Es ist anzumerken, dass dieses Merkmal über Laufzeittyp-Erkennung (RTTI) des C++ Standards bereitgestellt werden kann). Mit anderen Worten, wenn der Prüfplan die Schnittstelle ITesterPin aufruft, kann die Schnittstelle ihrerseits die Prüfgerätepin-Ausführung des Herstellers der Prüfgerätepin-Klasse, die modulspezifische Informationen einschließt (z. B. Adressen von zu setzenden Registern, um einen speziellen DUT Auslöseimpuls zur Verfügung zu stellen) direkt anrufen.

[0082] Zusammenfassend, während der Framework-Code immer die ITesterPin Schnittstelle nutzen wird, steht es Benutzern frei, bei Bedarf Gebrauch von durch Modulhersteller vorgesehenen speziellen Merkmalen und Erweiterungen zu machen. Mit anderen Worten, ein Modulhersteller kann zum Beispiel Methoden (Funktionen) zu der normalen Systemausführung der Klasse hinzufügen. Der Kompromiss für den Benutzer ist, dass das Nutzenziehen aus speziellen Herstellererweiterungen den Prüfcode mit anderen Herstellermodulen weniger brauchbar macht.

[0083] Auf dem modularen Pegel besitzt das System **100** nominell zwei Betriebsarten. In einer Online-Betriebsart werden Modulelemente **260** (z. B. Hardwareelemente) genutzt, und in einer Offline-Betriebsart wird eine Modulemulation in Software **280** genutzt.

[0084] Für die Online-Betriebsart umfasst das Modulelement **260** Hardware-Backplane **261**, Aufbauplattenschlitz IF (Schnittstelle) **262**, Modul-Hardware **263**, Lastmodul-Hardware-IF **264**, Hardware-Lastmodul **265** und DUT **266**.

[0085] Für die Offline-Betriebsart umfasst die Modulemulation in Software **280** Simulations-Framework **281**, Backplane-Emulation **282**, Backplane-Simulation IF **283**, Modulemulation **284**, Lastmodul-Simulation IF **285**, Lastmodul-Simulation **286** und DUT Simulation IF **287**. Für DUT Simulation sind zwei Modelle gezeigt. Ein Verilog nutzendes Modell enthält die Verilog PLI (Programmiersprachenschnittstelle) **288** und ein DUT Veri-

log-Modell **293**. Ein C/C++ nutzendes Modell enthält C/C++ Spracheunterstützung **289** und ein DUT C/C++ Modell **291**. Anzumerken ist, dass die Simulation auf jedem Computer, z. B. einem PC, durchgeführt werden kann.

[0086] Im Online-Modus liefern die Modulhersteller physikalische Hardwarekomponenten zur Unterstützung des Prüfens, wie digitale Prüfgerätekanäle, DUT Stromversorgungen oder Gleichspannungsmesseinheiten. Die Module koppeln durch die Aufbauplattenschlitz-Schnittstelle **262** mit der Hardware-Backplane **261**.

[0087] Für Offline-Arbeit würde eine auf PC basierende oder andere Umgebung, die das Äquivalent des Systemreglers abarbeitet, zusätzlich alle Verantwortlichkeiten sowohl der Bereitstellung von Stationssteuervorrichtungspegel-Framework und Laufzeitumgebung für die unteren Softwareschichten als auch das Nachahmen von Hardware übernehmen.

[0088] Die Backplane-Emulation **282** stellt einen Software-Ersatz für die physikalische Backplane **261** bereit. Dieser kommuniziert durch die Backplane-Simulationsschnittstelle **283** mit der (vom Hersteller gelieferten) Modulemulationssoftware **284**.

[0089] Die Modulemulationssoftware **284** wird vorzugsweise vom Modulhersteller zur Verfügung gestellt und ist typischerweise eng mit einer speziellen Herstellerausführung eines Moduls **263** verbunden. Folglich wird die Modulemulationssoftware über Module, die von unterschiedlichen Herstellern geliefert sind, in den Details typischerweise abweichen. In diesem Fall erlaubt die Modulsimulation dem Hersteller, Funktionsvielfalt von Hardware durch ein Software-Modell (z. B. die Modulemulationssoftware **284**) zu zeigen, Stimulationssignale an das simulierte Lastmodul **286** zu senden und Antwortsignale des DUT von dem simulierten Lastmodul **286** zu empfangen und zu verarbeiten, das durch die DUT Simulationsschnittstelle **287** mit der DUT Modelliersoftware **291**, **293** verbunden ist. In einigen Fällen können Hersteller es vorteilhaft finden, eine einfache funktionale Simulation der Modul- und Umgehungsemulation der Modul-Firmware bereitzustellen. Die Modul-Emulationssoftware vergleicht die Antwort des simulierten DUT auf die Stimulationssignale des simulierten Moduls mit einer bekannten guten DUT Antwort. Basierend auf diesem Vergleich bestimmt die Software, ob der durch das Modul ausgeführte Test sein Ziel, das DUT wie gewünscht zu prüfen, erfüllt und unterstützt den Benutzer, das Modul vor seiner Nutzung in einem integrierten Schaltkreis (physikalisches DUT) auf dem physikalischen Online-Prüfgerät auszuprüfen.

[0090] Die Lastmodul-Simulationsschnittstelle **285** dient als Kanal für Signale zu der und von der Modul-Emulationsschicht und dem simulierten Lastmodul **286**. Die Lastmodul-Simulationskomponente **286** unterstützt Gerätebuchsenzuordnung und Signalfortleitung zu der und von der DUT Simulationsschnittstelle **287**.

[0091] Die DUT Simulation kann ein systemeigener Code (d. h. C/C++) Simulation **291** oder eine Verilog-Programmiersprachen-Schnittstelle (PLI) für ein Funktionsmodell des Zielprüfobjektes **293** sein. Das Modell koppelt mit dem simulierten Lastmodul durch die DUT Simulationsschnittstelle **287**.

[0092] Zu beachten ist, dass die gesamte Steuerung dieser Schichten durch das Simulations-Framework **281** bewirkt wird. Das Simulations-Framework misst die simulierte DUT Antwort auf bekannte Stimulationssignale. Die Methode der Systememulation ist in der US-Patentanmeldung Nr. 10/403 817 offenbart.

Kommunikation und Steuerung

[0093] Kommunikation und Steuerung werden durch das Management von in Zusammenhang gebrachten Softwareobjekten durchgeführt. Vorzugsweise ist ein Kommunikationsmechanismus hinter einem Objektmodell auf dem Systemregler verborgen. Dieses Objektmodell bewirkt eine Vertretung für die auf der Stationssteuervorrichtung gefundenen Klassen und Objekte und stellt dadurch ein geeignetes Programmiermodul zur Anwendungsentwicklung (z. B. Prüfen von integrierten Schaltkreiselementen) zur Verfügung. Dies ermöglicht Anwendungsentwicklern (z. B. Benutzer des ATE Systems) unnötige Einzelheiten, die auf die Spezifik von Kommunikationen zwischen der Anwendung und der Stationssteuervorrichtung/dem Systemregler bezogen sind, zu vermeiden.

[0094] Fig. 6 veranschaulicht die spezielle Ausführung eines Stationssteuervorrichtungsobjektes wie es durch eine Stationssteuervorrichtung **104** in Stationssteuervorrichtungssoftware **240** gewartet wird. Das Stationssteuervorrichtungsobjekt umfasst CmdDispatcher **602**, FunktionalTestMsgHandler **604** und FunctionalTest **606**. Schnittstellen enthalten IMsgHandler **608** und ITest **610**.

[0095] Die Stationssteuervorrichtungs-Software **240** enthält vorzugsweise alle der Funktionsklassen, die eine Anwendung zum Zugriff benötigen kann. Diese Klassen können zum Beispiel Tests, Module, Pins, usw. umfassen. Weil die Test- und Softwaretools des Benutzers typischerweise auf unterschiedlichen Computern liegen, werden Mitteilungen von den Tools auf dem Systemregler zu einem Server auf der Stationssteuervorrichtung gesendet. Dieser Server wird ein Verfahren auf einem Befehlsabfertigungsobjekt aufrufen.

[0096] Das Befehlsabfertigungsobjekt (CmdDispatcher) **602** wartet einen Plan von Objekten des Mitteilungsabarbeitungsprogramms, die die IMsgHandler Schnittstelle **608** implementieren. Fig. 6 stellt eine spezielle Ausführung von IMsgHandler, Functional Test MsgHandler **604** dar. Mitteilungen, die durch das Objekt **602** CmdDispatcher empfangen werden, enthalten eine Kennzeichnung des Objektes, mit dem zu kommunizieren ist. Diese Kennzeichnung wird in einem internen Plan gefunden, der sich zu der speziellen Ausführung auflöst, in diesem Fall das gezeigte Objekt **604** FunctionalTestMsgHandler.

[0097] In diesem Beispiel besteht IMsgHandler **608** aus einem einzelnen Verfahren, handleMessage(). Dieses Verfahren wird vorzugsweise als eine einzelne Ausführungsklasse implementiert. In dem dargestellten Fall wird das FunctionalTestMsgHandler **604** die Mitteilung in Abhängigkeit von der genauen Beschaffenheit der ankommenden Mitteilung an eines von sechs Verfahren weiterleiten. Der Dateianfangssatz der eingehenden Mitteilung enthält eine Mitteilung id, die es dem Mitteilungsabarbeitungsprogramm erlaubt zu entscheiden, wie die Mitteilung zu interpretieren und wohin weiterzuleiten ist.

[0098] Die entsprechende Kommunikationsumgebung an dem Systemregler **102** bezieht sich auf den Abschnitt Tools **225**, **226** der Systemreglersoftware **220**. Fig. 7 veranschaulicht eine Ausführung eines auf dem Systemregler **102** in Systemreglersoftware **220** gewarteten Tool-Objekts (oder Systemreglerobjekt) in Übereinstimmung mit dem in Fig. 6 gezeigten Stationssteuervorrichtungsobjekt. Das Tool-Objekt umfasst Objekt CmdDispatcher **702**, FunctionalTestMsgHandler **704** und FunctionalTestProxy **706**. Schnittstellen umfassen IMsgHandler **708**, ITestClient **710** und IDispatch **712**. Außerdem ist eine Dienstleistungsprogrammanwendung **714** enthalten.

[0099] Für dieses Beispiel sind die Klassen CmdDispatcher **702**, IMsgHandler **708** und FunctionalTestMsgHandler **704** die gleichen wie in Fig. 6. Jedoch werden keine konkreten Darstellungen von FunctionalTest **606** (oder eine beliebige andere Stationssteuervorrichtungsklasse verwendet. Stattdessen besitzt das Tool-Objekt Proxy-Klassen zur Kommunikation mit jedem Objekt an der Stationssteuervorrichtung **104**. Daher umfasst zum Beispiel das Tool-Objekt die Klasse FunctionalTestProxy **706** anstelle von FunctionalTest **606**. Entsprechend ist ITestClient **710** in dem Tool-Objekt nicht das Gleiche wie ITest **610** in dem Stationssteuervorrichtungsobjekt. Im Allgemeinen werden auf dem Systemregler **102** laufende Anwendungen nicht die exakten Schnittstellen, wie auf der Stationssteuervorrichtung **104** vorgesehen, verwenden. In diesem Fall werden drei Verfahren von ITest **610** (nämlich preExec(), execute() und postExec() durch ein einzelnes Verfahren in ITestClient **710** (nämlich runTest()) ersetzt. Außerdem ist ITestClient **710** vorzugsweise eine Doppelschnittstelle, das heißt, sie erbt von IDispatch **712**, das als ein Komponentenobjektmodell (COM) von Microsoft implementiert wird. Es bewirkt eine Schnittstelle, die eine Scripting-Engine aktiviert, um auf das diese Schnittstelle implementierende Objekt zuzugreifen. Dies ermöglicht es, das System auf der Plattform Windows von Microsoft scriptable zu machen.

[0100] Als ein Beispiel zur Abarbeitung der in den Fig. 6–Fig. 7 gezeigten Ausführungen kann eine auf dem Systemregler **102** laufende Anwendung (z. B. in einem der Tool-Abschnitte **226**, **228**) mit einer Stationssteuervorrichtung **104** kommunizieren, bei der ein Prüfplan **242** ein oder mehrere FunctionalTest Objekte **606** enthält. Während der Initialisierung des Prüfplans **242** auf der Stationssteuervorrichtung **104** wird ein entsprechendes Prüfplanobjekt in die Stationssteuervorrichtung **104** geladen, die ein Objekt TestPlanMessageHandler konstruiert und es mit dem CmdDispatcher Objekt **602** erfasst. Dieses weist dem Mitteilungsabarbeitungsprogramm eine einzelne ID zu. Ähnliche Abarbeitungen treten mit anderen Testelan Objekten auf, die den Prüfplan **242** aufstellen.

[0101] Die Anwendung (z. B. in Tools **226**, **228**) auf dem Systemregler **103** initialisiert die Kommunikationsbibliothek **230**, verbindet über einen Kommunikationskanal mit der Stationssteuervorrichtung **104** und erhält einen Kennungscode (ID) für das TestPlan Objekt. Die Bibliothek konstruiert ein TestPlanProxy Objekt und initialisiert es mit diesem Kennungscode. Während der Initialisierung bestimmt dieses Proxy-Objekt, wie viele Tests es enthält und ihre Typen und Kennungscodes. Es lädt die geeigneten Dateien für Betriebssystemroutinen (DLL) für jeden Typ (in diesem Fall nur ein Typ) und konstruiert die Proxy-Objekte für sie, indem sie mit ihren ID-Werten initialisiert werden.

[0102] Die Prüf-Proxy-Objekte initialisieren ihrerseits ebenfalls. Um dies zu tun, konstruieren sie geeignete

Mitteilungen, um ihre Namen zu erhalten (indem sie ihre ID-Werte nutzen) und senden sie zu einem Verbindungsabnehmer in der Stationssteuervorrichtung **104**, die die Mitteilung an den CmdDispatcher **602** weitergibt. Dieses Objekt sucht die Zielkennungscodes in ihrem internen Plan auf und leitet die Mitteilung weiter an die handleMessage() Verfahren der FunctionalTestMsgHandler Objekte **604**. Wenn die Mitteilung zum Beispiel eine Anforderung zum Erhalten von Testnamen war, erhalten diese Objekte ihre entsprechenden Testnamen und antworten den Test-Proxy-Objekten der Anwendung mit den entsprechenden Namenzeichenketten.

[0103] Nachdem die Initialisierung beendet ist, besitzt die Anwendung einen Fernzugriff auf ein TestPlan Objekt und durch ihn auf beide Testobjekte. Der Benutzer kann jetzt zum Beispiel auf einen Knopf „Prüfplan ablaufen" auf der Anwendung drücken. Infolgedessen ruft die Anwendung das Verfahren RunTestPlan() auf dem Prüfplan-Proxyobjekt auf. Dieses Verfahren konstruiert eine RunTestPlan Mitteilung mit der Ziel-ID des Prüfplanobjekts und ruft die sendMessage() Funktion auf dem RPC-Proxy auf, die die Mitteilung an die Stationssteuervorrichtung sendet.

[0104] Der Verbindungsabnehmer auf der Stationssteuervorrichtung **104** ruft das handleMessage() Verfahren auf dem CmdDispatcher Objekt **602** auf, das ihm den Kennungscode des Prüfplanobjektes weitergibt. Das CmdDispatcher Objekt **602** sucht diesen ID in seinem internen Plan auf, findet das Mitteilungsabarbeitungsprogramm für das TestPlan Objekt und ruft das handleMessage() Verfahren an diesem Objekt auf, das wiederum das RunTestPlan() Verfahren an dem TestPlan Objekt aufruft. In ähnlicher Weise kann die Anwendung die Namen und den letzten Betriebszustand der Prüfobjekte erhalten.

Verfahren zur Verwendung der Kommunikationsbibliothek

[0105] Das Folgende ist eine beispielhafte Verwendung der Kommunikationsbibliothek **230**.

[0106] Die Kommunikationsbibliothek **230** ist vorzugsweise eine statische Bibliothek. Eine Anwendung kann diese Kommunikationsbibliothek durch einen Datensatz CommLibrary.h nutzen. Eine Anwendung, die die Klassen der Kommunikationsbibliothek exportieren muss, sollte die Vorverarbeitungsprozessor-Definitionen COMMLIBRARY_EXORTS, COMMLIBRARY_FORCE_LINKAGE aufweisen, die zusätzlich einschließlich des oben enthaltenen Datensatzes definiert sind. Eine Anwendung, die die Kommunikationsbibliothek exportiert, muss Vorverarbeitungsprozessor-Definitionen nicht definieren. Wenn die Kommunikationsbibliothek als Server genutzt wird, muss die Anwendung die folgende statische Funktion von CcmdDispatcher: InitializeServerunsigned long portNo) aufrufen.

[0107] Die portNo ist diejenige portNo, auf die der Server hören sollte. Der dem Server entsprechende Befehlsdispatcher kann wieder gefunden werden, indem die statische Funktion: getServerCmdDispatcher auf der CcmdDispatcher Klasse aufgerufen wird.

[0108] Wenn die Kommunikationsbibliothek als Verbindungsprogramm genutzt wird, sollte die Anwendung die folgende statische Funktion von CcmdDispatcher aufrufen:
InitializeClientconst OFCString serverAddress,
unsigned long serverPortNo,
CcmdDispatcher **pCmdDispatcher,
OFCString serverId)
serverAddress und ServerPortNo, mit denen das Verbindungsprogramm Kontakt herstellen muss. Diese Funktion initialisiert den Befehlsdispatcher-Zeiger für das Verbindungsprogramm und serverId, zu denen es Kontakt hergestellt hat. Außerdem kann das Verbindungsprogramm zu einem späteren Zeitpunkt den dem serverId entsprechenden Befehlsdispatcher wieder finden, indem die statische Funktion getClientCmdDispatcher aufgerufen wird.

[0109] Wenn die Kommunikationsbibliothek kompiliert ist, wurde die Form in den Files ClientInterface.idl und ServerInterface.idl ausgeschlossen. Die bevorzugte Ausführung wendet die bereits erzeugten Stub- und Proxy-Datensätze für diese Schnittstellendefinition-Datensätze an, um die Proxy- und Stub-Ausführungsdatensätze in die gleiche Bibliothek zu verketten. Daher können Server und Verbindungsprogramm im gleichen Adressenraum konkret dargestellt werden. Die folgenden Änderungen in den Schnittstellen-Definitionsdatensätzen und Stub-Datensätzen werden vorzugsweise gemacht, um die Kommunikationsbibliothek als Server und Verbindungsprogramm im gleichen Adressenraum arbeiten zu lassen.

Änderungen in Schnittstellen-Definitions-Datensätzen

[0110] Die folgenden Namensraumvereinbarungen werden vorzugsweise in jedem der Schnittstellen-Definitions-Datensätze ergänzt. Dies soll die Namenskollision zwischen den Proxy-Implementierungsfunktionen und unserer eigenen Implementierung der Schnittstellenfunktionen vermeiden. Die folgenden Namensraumvereinbarungen werden in der serverinterface.idl ergänzt.

```
cpp_quote("#ifdef_cplusplus")
cpp_quote("namespace COMM_SERVER")
cpp_quote("{")
cpp_quote("#endif")
cpp_quote("}")
```

[0111] Die Funktionen in dem Stub-Ausführungsdatensatz werden geändert, um unsere eigenen Implementierungsfunktionen für die Funktionen aufzurufen, die in den Schnittstellen vereinbart sind, d. h. wir haben eine anders benannte Funktion, die jeder der Funktionen, die in den Schnittstellen vereinbart sind, entspricht.

[0112] Um den Konflikt beim Funktionsaufruf zu vermeiden, ist es vorzuziehen, den Namen der Implementierungsfunktionen „COMM"_string vorzusetzen. So wird der Code in den Stub-Funktionen geändert, um „COMM_functionalName" anstelle von „functionName" aufzurufen.

[0113] Damit dieses Verfahren arbeitet, sollten alle Funktionsklassen, die vorhanden sind, auch ihr entsprechendes Mitteilungsabarbeitungsprogrammobjekt und Proxy-Klassen besitzen. Alle Mitteilungsabarbeitungsprogrammobjekte sollten sich von der durch die Kommunikationsbibliothek zur Verfügung gestellte Klasse IMsgHandler ableiten. Die IMsgHandler Klasse ist eine abstrakte Klasse. Sie ist vorzugsweise die Verbindlichkeit des Herstellers des Mitteilungsabarbeitungsprogramms, um eine Definition für handleMessage, setObjectId, handleError zur Verfügung zu stellen. Alle Mitteilungstypen sollten von Eins starten (wir reservieren Null für handleError). Die Funktionsklassen besitzen vorzugsweise ihr entsprechendes Mitteilungsabarbeitungsprogramm als ihre Elementvariable. Im Konstruktor der Funktionsklasse sollte die Funktionsklasse selbst mit dem Mitteilungsabarbeitungsprogramm registriert werden, indem eine durch ihr Mitteilungsabarbeitungsprogramm erzeugte Funktion aufgerufen wird. Als Nächstes sollte das Mitteilungsabarbeitungsprogrammobjekt mit Befehlsdispatcher registriert werden, indem die Funktion addMsgHandler auf dem Befehlsdispatcher mit dem Mitteilungsabarbeitungsprogramm als Parameter aufgerufen wird. Die Funktion addMsgHandler wird dem Mitteilungsabarbeitungsprogramm und der Funktionsklasse einen Kennungscode zuweisen. Der Destruktor der Funktionsklasse soll die Funktion removeMsgHandler auf dem Befehlsdispatcher aufrufen, indem die Funktionsklassenbezeichnung als Parameter gesendet wird. Proxy-Klassen sollen sich auch nach dem gleichen Verfahren der Eintragung richten, wie es für die Funktionsklassen erläutert ist.

[0114] Die folgende CTestPlan Klasse zeigt, wie eine typische Funktionsklasse entsprechend der bevorzugten Ausführung der vorliegenden Erfindung aussehen wird:

```
File:-TestPlan.h

Class CTestPlan
```

```
{
private:
        unsigned long m_Id;
        CTestPlanMsgHandler m_tplMsgHandler;
}
File:-TestPlan.cpp
extern CcmdDispatcher *g_pCmdDispatcher;
CTestPlan::CTestPlan
{
     m_tplMsgHandler.setTestPlan(this);
     g_pCmdDispatcher.AddMsgHandler(&m_tplMsgHandler)
}
CTestPlan::~CTestPlan
{
     g_pCmdDispatcher.removeMsgHandler(m_Id)
}
```

[0115] Das Objekt g_pCmdDispatcher soll initialisiert werden durch Aufrufen von getCmdDispatcher(), das von den Kommunikationsdateien für Betriebsroutinen gezeigt wird. Die folgende Klasse CTestPlanMsgHandler stellt dar, wie ein typisches Mitteilungsabarbeitungsprogramm aussehen wird.

```
File:-TestPlanMsgHandler.h
Class CtestPlanMsgHandler: public IMsgHandler
{
public:
       setTestPlan(CTestPlan *pTestPlan);
       setTestPlanProxy(CTestPlanProxy *pTestPlanProxy);
       void handleMessage(unsigned long msgType,
                                  unsigned long senderId
                                  unsigned long senderMsgLen,
                                  byte *pSenderMsg)
   void handleSetName(unsigned long senderId,
                                  unsigned long senderMsgLen,
                                  byte *pSenderMsg);
```

```
   void handleGetName(unsigned long senderId,
                                    unsigned long senderMsgLen,
                                    byte *pSenderMsg);
private:
       CTestPlan m_pTestPlan;
       CTestPlanProxy m_pTestPlanProxy;
       typedef void (CFuncTestMsgHandler::*handlerFn)(unsigned
long, unsigned long, byte*);
       std::map<int, handlerFn>m_handlers;
}
File:-TestPlanMsgHandler.cpp
CTestPlanMsgHandler::CtestPlanMsgHandler
{
     m_handlers[HandleError] = handleError;
     m_handlers[GetName] = handleGetName;
     m_handlers[SetName] = handleSetName;
}
void
CTestPlanMsgHandler::handleMessage(unsigned long msgType,
                          unsigned long senderId,
                          unsigned long senderMsgLen,
                          byte *pSenderMsg)
{
     if(msgType == 0)
     {
          handleError(senderId, senderMsgLen, pSenderMsg);
     }
     else
     {
          handlerFn fn = NULL
          h_Iter_t fIter;
          fIter = m_handlers.find(msgType);
          if(fIter == m_handlers.end() )
          {
               return;
```

```
            }
            fn = fIter->second;
            if(NULL ! = fn)
            {
                (this->*fn)(senderId, senderMsgLen, pSenderMsg);
            }
        }
    }
void
CTestPlanMsgHandler::handleSetName(unsigned long senderId,
                                   unsigned long senderMsgLen,
                                   byte *pSenderMsg)
{
    if(m_pTEstPlanProxy ! = NULL)
    {
        OFCString tplName = ByteToString(senderMsgLen,
pSenderMsg)
        m_pTplProxy->setName(tplName);
    }
}
void
CTestPlanMsgHandler::handleGetName(unsigned long senderId,
                                   unsigned long senderMsgLen,
                                   byte *pSenderMsg)
{
    OFCString testName;
    If(m_pTestPlan ! = NULL)
    {
        unsigned long l_destId
        unsigned long l_msgType;
        unsigned long l_senderId;
        unsigned l_senderMsgLen;
        byte *l_senderMsg = NULL;
        if(m_pTestPlan->getName(testName) ! = true)
        {
```

```
            //Wenn ein Ausfall aufgetreten ist, sende
            Fehlermeldung char *errorString =
            "Fehlerwiedergewinnungsname";
        l_destId = senderId;
        l_msgType = HandleError;
        l_senderId = m_Id;
        l_senderMsgLen = strlen(errorString);
        l_senderMsg = StringToByte(errorString);
        sendMsg(l_destId,
                    l_msgType,
                    l_senderId,
                    l_senderMsgLen,
                    l_senderMsg);
              return;
        }
        l_destId = senderId;
        l_msgType = SetName;
        long l_senderId = m_Id;
        l_senderMsgLen = testName.length();
        l_senderMsg = NULL;
        StringToByte(testName, &l_senderMsg);
        sendMsg(l_destId,
                    l_msgType,
                    l_senderId,
                    l_senderMsgLen,
                    l_senderMsg);
            DELETE-BYTE(l_senderMsg);
    }
}
void
CTestPlanMsgHandler::handleError(unsigned long senderId,
                                 unsigned long senderMsgLen,
                                 byte *pSenderMsg)
{
   OFCString errorString;
```

```
   ByteToString(senderMsgLen, pSenderMsg, errorString);
   m_pTestPlanProxy->setError(errorString);
}
```

[0116] Die folgende Klasse CTestPlanProxy zeigt, wie eine typische Proxy-Klasse aussehen wird.

```
File: - TestPlanProxy.h
Class CTestPlanProxy
{
   CTestPlanProxy(unsigned long serverId);
   ~CTestPlanProxy();
private:
       CTestPlanProxy();
       unsigned long m_Id;
       unsigned long m_serverId;
       CTestPlanMsgHandler m_tplMsgHandler;
}
File: - TestPlanProxy.cpp
extern CcmdDispatcher *g_pCmdDispatcher;
CTestPlanProxy::CTestPlanProxy(unsigned long serverId)
{
         m_serverId = serverId;
         m_tplMsgHandler.setTestPlanProxy(this);
         g_pCmdDispatcher.AddMsgHandler(&m_tplMsgHandler)

         /// initialisiere Proxy mit seinem Namen.
         unsigned long msgType;
         unsigned long senderMsgLen;
         byte *pSenderMsg = NULL;
         msgType = GetName;
         senderMsgLen = 0;
         pSenderMsg = NULL;
         sendMsg(m_clientId,
         msgType,
```

```
        m_Id,
        senderMsgLen,

              pSenderMsg);
    //Prüfen, ob die Fehlerdatenfolge durch das
    Mitteilungsabarbeitungsprogramm eingestellt worden
    ist.
    If(m_errorString.length() !=0)
    {
        OFCString errorString = m_errorString;
        m_errorString = "";
        throw exception(errorString.c_str() );
    }
}
CTestPlanProxy::~CTestPlanProxy
{
  g_pCmdDispatcher.removeMsgHandler(m_Id)
}
```

[0117] Das Objekt g_pCmdDispatcher sollte initialisiert werden, indem getCmdDispatcher() aufgerufen wird.

Systemkonfigurationen und Prüfen

[0118] Fig. 8 veranschaulicht eine nominelle Prüffolge **800** nach einer Ausführung der vorliegenden Erfindung. Die Prüffolge **800** umfasst die Installation **802** von Modulen in einer Prüfumgebung **804**, die eine Testvorbereitung **806** und Systemprüfung **808** umfasst. Am Anfang wird ein neues Modul **810** (Hardware oder Software oder eine Kombination davon) bestätigt **812** (durch ein bestimmtes externes Verfahren, etwa auf Basis einer Qualitätskontrolle des Herstellers). Die Installation **802** erfordert zuerst eine Testvorbereitung **806** einschließlich einer Installation von Hardwaremodul-Emulation für Offline-Simulation **810**, die Installation von Modulressourcen-Datensätzen und Schnittstellen zur Prüfprogrammentwicklung **812** und Installation von modulspezifischen Strukturkompilern zur Strukturkompilierung **814**. Als nächstes wird eine Systemprüfung **808** mit Eingaben von Kalibrierung **816**, Diagnose **818** und Konfigurierung **320** durchgeführt. Anschließend wird eine Systemprüfung **808** für das neue Modul durchgeführt, die umfasst: (1) Schnittstellensteuerung, (2) Synchronisierung, Ablaufsteuerung und Wiederholbarkeit, (3) Fehler/Warnungsbehandlung, (4) Mehrstellensteuerung und (5) Mehrinstrument-Modulsteuerung.

Patentansprüche

1. Verteiltes System für ein Halbleiter-Prüfsystem zum Prüfen zumindest einer geprüften Vorrichtung (**112**), welches verteilte System aufweist:
ein Hostoperationssystem zum Ermöglichen der Steuerung durch eine Systemsteuervorrichtung (**102**);
zumindest ein lokales Operationssystem zum Ermöglichen der Steuerung von zumindest einem Prüfmodul (**108**),
wobei zumindest ein Prüfmodul (**108**) eine Prüfung bei einer entsprechenden geprüften Vorrichtung (**112**) durchführt; und
zumindest eine Stationssteuervorrichtung (**104**), die durch die Systemsteuervorrichtung (**102**) gesteuert wird, wobei das zumindest eine lokale Operationssystem mit jeder Stationssteuervorrichtung (**104**) assoziiert ist, um eine Steuerung des zumindest einen Prüfmoduls (**108**) durch eine assoziierte Stationssteuervorrichtung (**104**)

zu ermöglichen,
dadurch gekennzeichnet, dass
die assoziierte Stationssteuervorrichtung (**104**) zumindest ein Prüfmodul (**108**) mit assoziierter Stationssteuervorrichtung (**104**) in einer Plug-and-Plag-Weise steuert.

2. Verteiltes System nach Anspruch 1, bei dem das Hostoperationssystem die Operation der zumindest einen Stationssteuervorrichtung (**104**) synchronisiert.

3. Verteiltes System nach Anspruch 1, bei dem das Hostoperationssystem über die Kommunikation zwischen der Systemsteuervorrichtung (**102**) und der zumindest einen Stationssteuervorrichtung (**104**) entscheidet.

4. Verteiltes System nach Anspruch 1, bei dem die Systemsteuervorrichtung (**102**) die Operation der zumindest einen Stationssteuervorrichtung (**104**) überwacht.

5. Verteiltes System nach Anspruch 1, bei dem eine Stationssteuervorrichtung (**104**) die Operation des zumindest einen Prüfmoduls (**108**), das mit der Stationssteuervorrichtung (**104**) assoziiert ist, überwacht.

6. Verteiltes System nach Anspruch 1, bei dem das Hostoperationssystem zumindest eine Hostschnittstelle für die Kommunikation mit der zumindest einen Stationssteuervorrichtung (**104**) aufweist.

7. Verteiltes System nach Anspruch 6, bei dem die zumindest eine Hostschnittstelle zum Kommunizieren mit zumindest einem Prüfmodul (**108**) mit einer Stationssteuervorrichtung (**104**) assoziiert ist.

8. Verteiltes System nach Anspruch 1, weiterhin aufweisend eine Prüfmodul-Schnittstelle zum Definieren von Prüfmodulfunktionen für die Bildung einer Schnittstelle zwischen einer Stationssteuervorrichtung (**104**) und einem ersten Prüfmodul (**108**), wobei die Prüfmodul-Schnittstelle erweiterbar ist zur Bildung einer Schnittstelle zwischen der Stationssteuervorrichtung (**104**) und einem zweiten Prüfmodul (**108**), wobei die nicht erweiterte Prüfmodul-Schnittstelle nicht ausreichend ist für die Bildung einer Schnittstelle zwischen der Stationssteuervorrichtung (**104**) und dem zweiten Prüfmodul (**108**).

9. Verteiltes System nach Anspruch 1, bei dem das Hostoperationssystem zumindest eine Hostgerüstklasse enthält.

10. Verteiltes System nach Anspruch 9, bei dem die zumindest eine Hostgerüstklasse in einer Standardcomputersprache entwickelt ist, um einem Benutzer zu ermöglichen, anwendungsspezifische Klassen zum Steuern der zumindest einen Stationssteuervorrichtung (**104**) zu entwickeln.

11. Verteiltes System nach Anspruch 10, bei dem die Standardcomputersprache C oder C++ ist.

12. Verteiltes System nach Anspruch 1, bei dem jedes lokale Operationssystem zumindest eine lokale Gerüstklasse enthält.

13. Verteiltes System nach Anspruch 12, bei dem die zumindest eine lokale Gerüstklasse in einer Standardcomputersprache entwickelt ist, um einem Benutzer zu ermöglichen, anwendungsspezifische Klassen zum Steuern des zumindest einen Prüfmoduls (**108**) zu entwickeln.

14. Verteiltes System nach Anspruch 13, bei dem die Standardcomputersprache C oder C++ ist.

15. Verteiltes System nach Anspruch 1, bei dem die Anzahl von von jeder Stationssteuervorrichtung (**108**) gesteuerten Modulen (**108**) skalierbar ist.

16. Verteiltes System nach Anspruch 1, bei dem das lokale Operationssystem, das mit einer entsprechenden Stationssteuervorrichtung (**104**) assoziiert ist, ermöglicht, dass der Typ von Prüfmodulen (**108**), die durch die Stationssteuervorrichtung (**104**) gesteuert werden, rekonfiguriert wird.

17. Verteiltes System nach Anspruch 1, bei dem das Hostoperationssystem ermöglicht, dass die Anzahl von Stationssteuervorrichtungen (**104**), die von der Systemsteuervorrichtung gesteuert werden, skalierbar ist.

18. Verteiltes System nach Anspruch 1, bei dem das Hostoperationssystem ermöglicht, dass die Anzahl

der DUT (**112**), die von dem Prüfsystem geprüft werden, skalierbar ist.

19. Verteiltes System nach Anspruch 1, bei dem das zumindest eine Prüfmodul (**18**) Hardware und/oder Software aufweist.

20. Verteiltes System nach Anspruch 1, weiterhin aufweisend einen Emulator zum Simulieren des Gebrauchs eines Kandidatenprüfmoduls mit dem Prüfsystem, um das Kandidatenmodul als kompatibel mit dem Prüfsystem zu verifizieren.

21. Verteiltes System nach Anspruch 1, bei dem ein erster Satz von Modulen (**108**) in einer ersten Prüfstation (**110**) unterschiedlich gegenüber einem zweiten Satz von Modulen (**108**) in einer zweiten Prüfstation ausgebildet ist.

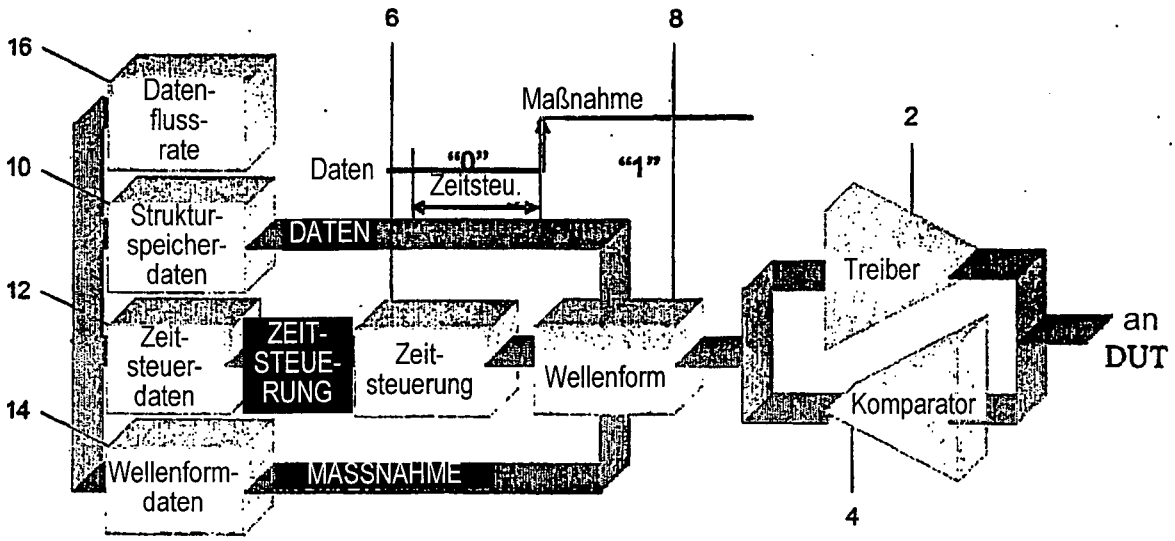
22. Verteiltes System nach Anspruch 1, bei dem eine erste Prüfstation (**110**) eine erste Konfiguration zum Prüfen einer ersten DUT (**112**) hat und eine zweite Prüfstation (**110**) eine zweite Konfiguration zum Prüfen einer zweiten DUT (**112**) hat, wobei die erste und die zweite Prüfstation (**110**) rekonfigurierbar sind, um zusammen eine dritte Prüfstation (**110**) zu bilden, die stattdessen eine dritte DUT (**112**) prüft.

23. Verteiltes System nach Anspruch 1, bei dem ein erstes Modul (**108**) in einer ersten Prüfstation (**110**) zu einem zweiten Modul (**108**) in einer zweiten Prüfstation (**110**) zugreifen kann.

24. Verteiltes System nach Anspruch 1, weiterhin aufweisend eine Kommunikationsbibliothek mit einem vorbestimmten Satz von Funktionen und Schnittstellen zur Verwendung mit Prüfmodulen (**108**).

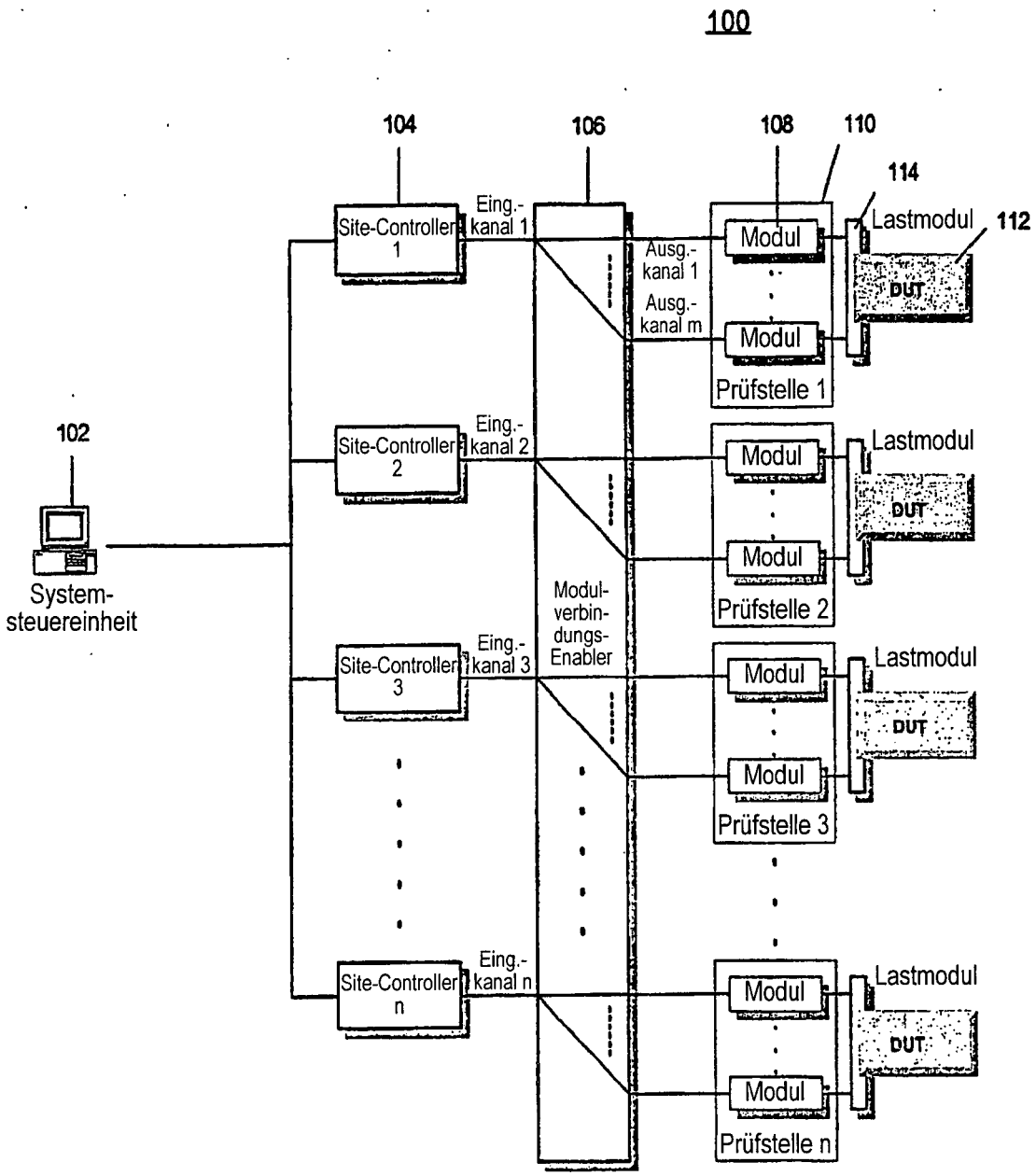
Es folgen 8 Blatt Zeichnungen

Anhängende Zeichnungen

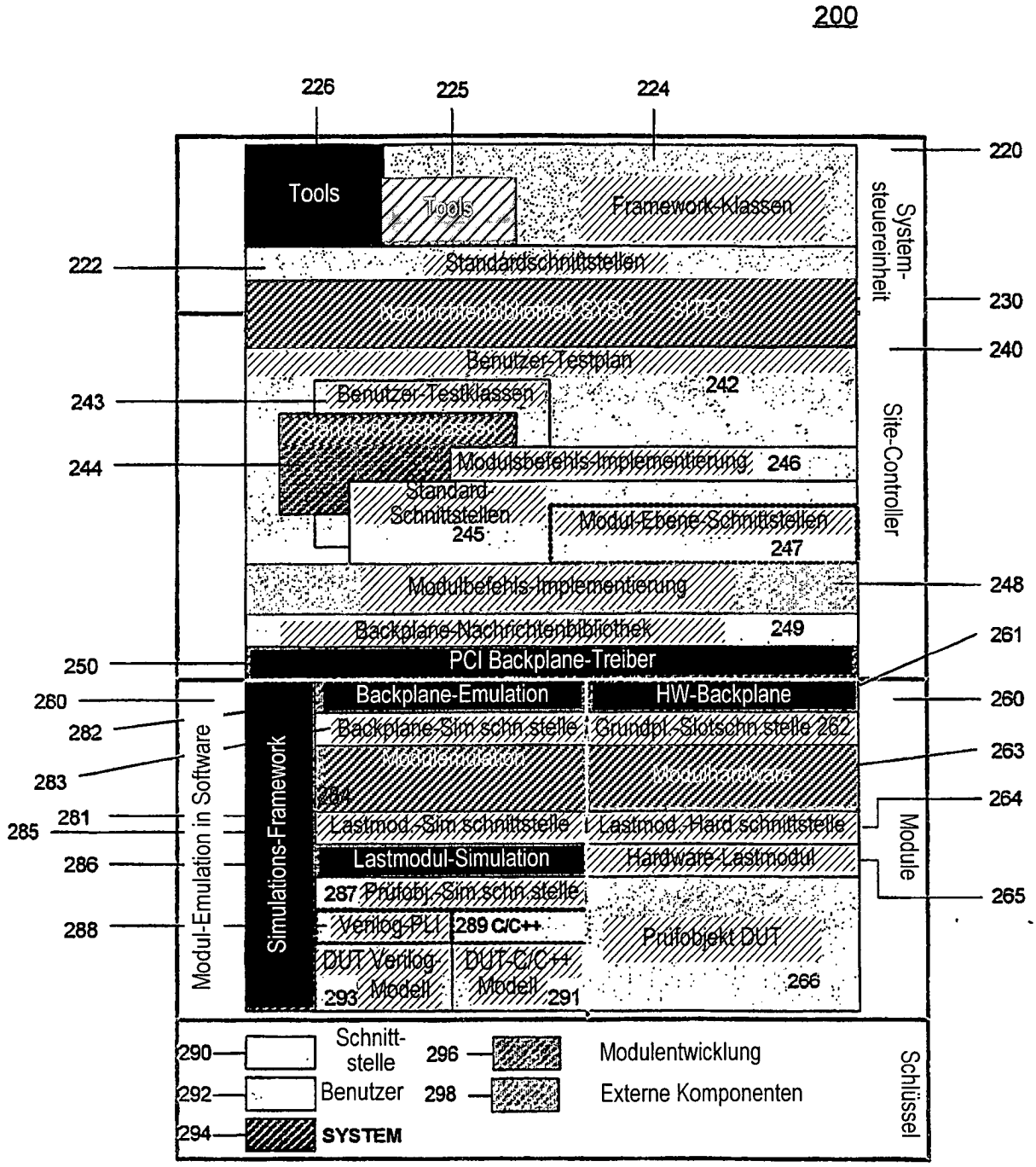


Bedienungsmaßnahme = Strukturdaten * Zeitsteuerdaten (begrenzt durch Zeiteinstellung) * Wellensatz * Ansteuerung

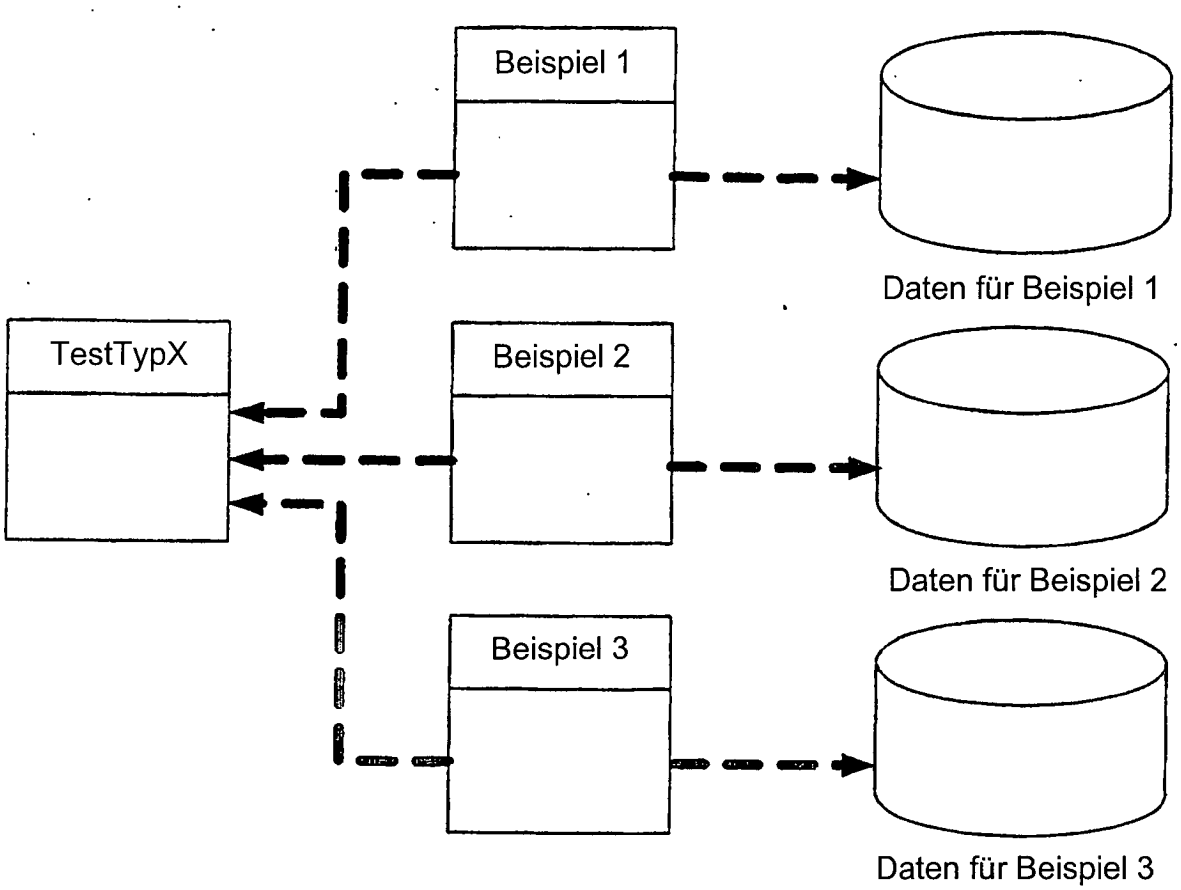
Figur 1


100
104
106
108
110
114
112
102
Site-Controller 1
Site-Controller 2
Site-Controller 3
Site-Controller n
Eing.-kanal 1
Eing.-kanal 2
Eing.-kanal 3
Eing.-kanal n
Ausg.-kanal 1
Ausg.-kanal m
Modul
Lastmodul
DUT
Prüfstelle 1
Prüfstelle 2
Prüfstelle 3
Prüfstelle n
Modul-verbin-dungs-Enabler
System-steuereinheit

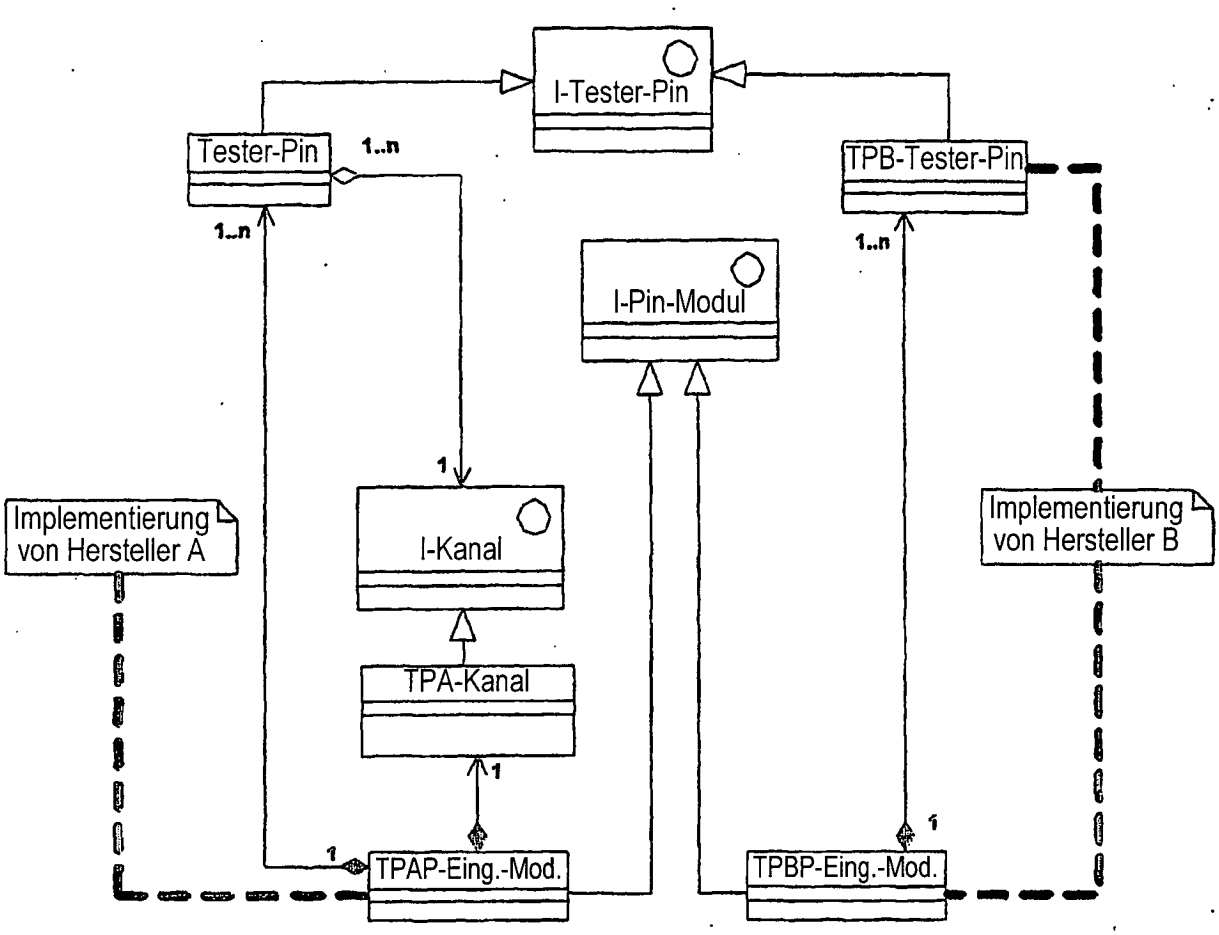
Figur 2


200
226
225
224
220
Tools
Tools
Framework-klassen
System-steuereinheit
222
Standardschnittstellen
Kernnachrichtenbibliothek
230
Benutzer-Testplan
240
242
243
Benutzer-Testklassen
Site-Controller
244
Modulbefehls-Implementierung
246
Standard-Schnittstellen
245
Modul-Ebene-Schnittstellen
247
Modulbefehls-Implementierung
248
Backplane-Nachrichtenbibliothek
249
261
250
PCI Backplane-Treiber
260
Backplane-Emulation
HW-Backplane
260
282
Backplane-Sim schn stelle
Grundpl.-Slotschn stelle 262
263
283
Modulsimulation
Modulhardware
284
264
281
285
Lastmod.-Sim schnittstelle
Lastmod.-Hard schnittstelle
Module
Modul-Emulation in Software
Simulations-Framework
286
Lastmodul-Simulation
Hardware-Lastmodul
287
Prüfobj.-Sim schn stelle
265
288
Verilog-PLI
289 C/C++
Prüfobjekt DUT
DUT Verilog-Modell
293
DUT-C/C++-Modell
291
266
290
Schnitt-stelle
296
Modulentwicklung
292
Benutzer
298
Externe Komponenten
294
SYSTEM
Schlüssel

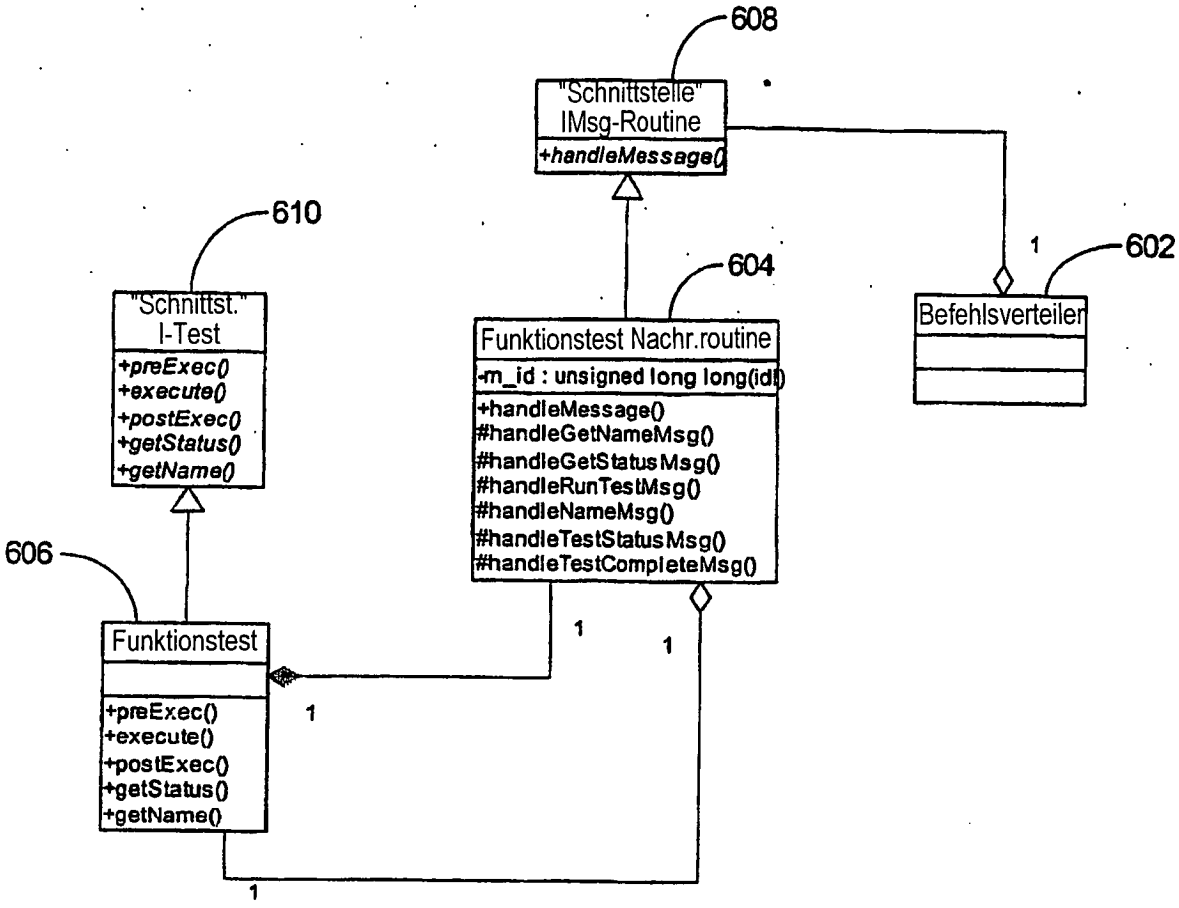
Figur 3


Beispiel 1
Daten für Beispiel 1
TestTypX
Beispiel 2
Daten für Beispiel 2
Beispiel 3
Daten für Beispiel 3

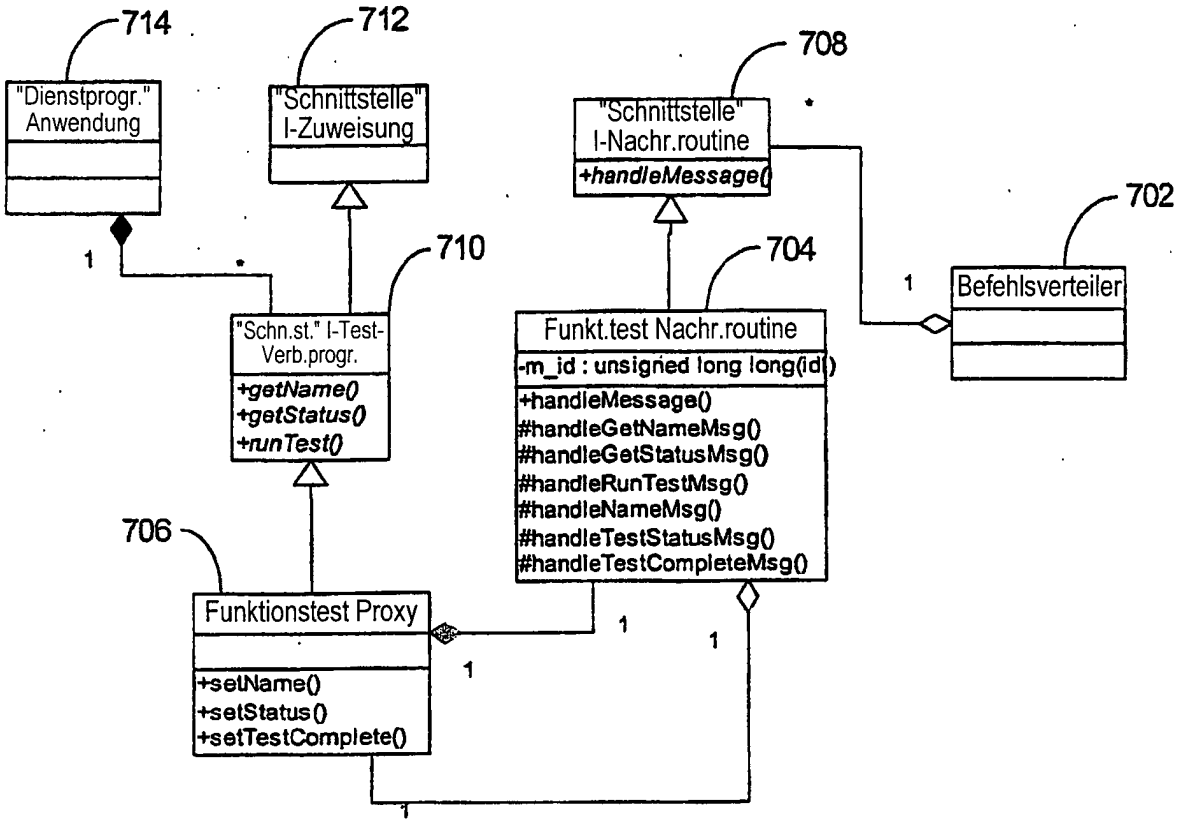
Figur 4


I-Tester-Pin
Tester-Pin
1..n
1..n
TPB-Tester-Pin
1..n
I-Pin-Modul
1
Implementierung von Hersteller A
I-Kanal
Implementierung von Hersteller B
TPA-Kanal
1
1
1
TPAP-Eing.-Mod.
TPBP-Eing.-Mod.

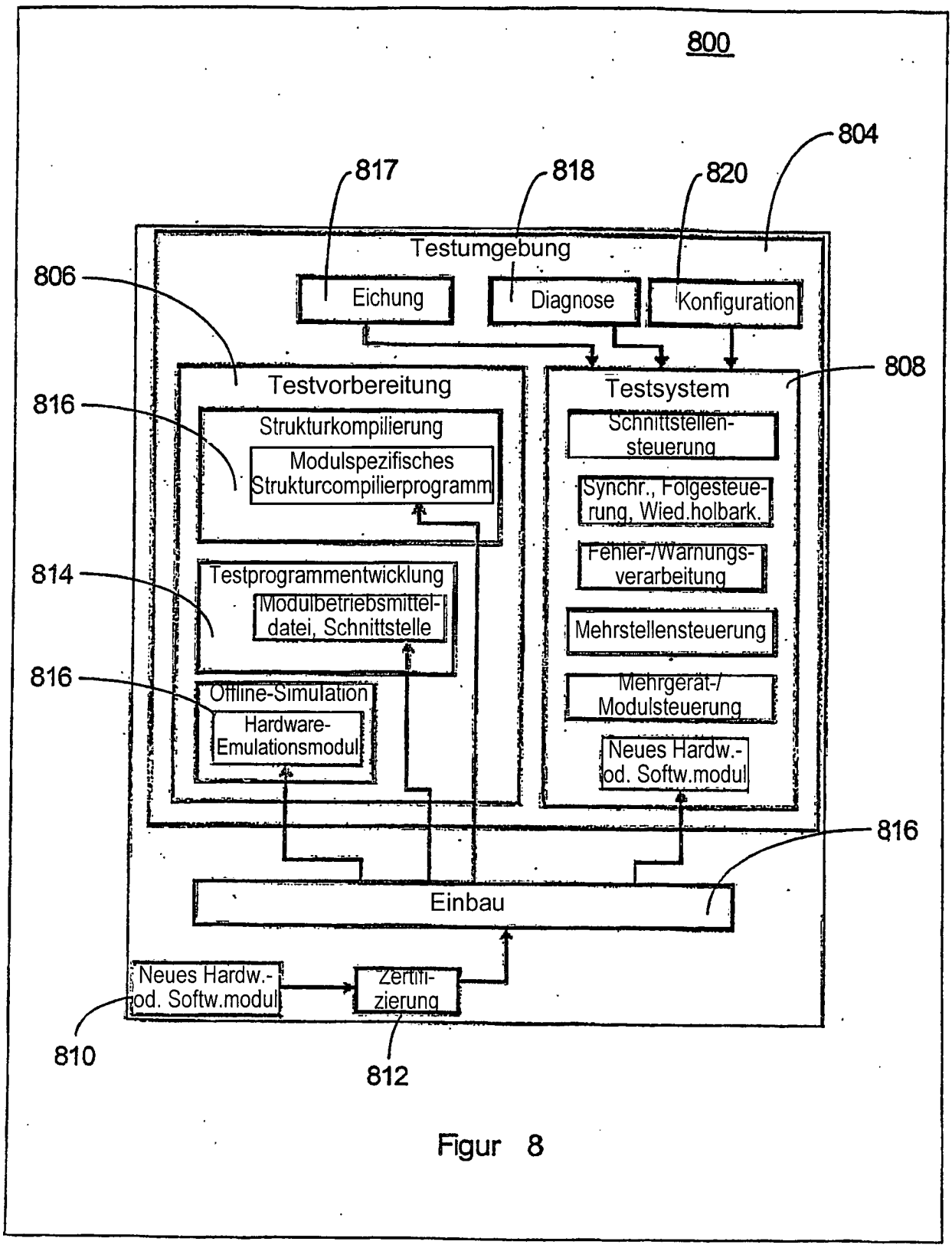
Figur 5


608
"Schnittstelle"
IMsg-Routine
+handleMessage()
610
"Schnittst."
I-Test
+preExec()
+execute()
+postExec()
+getStatus()
+getName()
604
Funktionstest Nachr.routine
-m_id : unsigned long long(id)
+handleMessage()
#handleGetNameMsg()
#handleGetStatusMsg()
#handleRunTestMsg()
#handleNameMsg()
#handleTestStatusMsg()
#handleTestCompleteMsg()
1
602
Befehlsverteiler
606
Funktionstest
+preExec()
+execute()
+postExec()
+getStatus()
+getName()
1
1
1
1

Figur 6


714
712
708
"Dienstprogr." Anwendung
"Schnittstelle" I-Zuweisung
"Schnittstelle" I-Nachr.routine
+handleMessage()
702
1
*
710
704
1
Befehlsverteiler
"Schn.st." I-Test-Verb.progr.
+getName()
+getStatus()
+runTest()
Funkt.test Nachr.routine
-m_id : unsigned long long(id)
+handleMessage()
#handleGetNameMsg()
#handleGetStatusMsg()
#handleRunTestMsg()
#handleNameMsg()
#handleTestStatusMsg()
#handleTestCompleteMsg()
706
Funktionstest Proxy
+setName()
+setStatus()
+setTestComplete()
1
1
1
1
1

Figur 7


800
804
817
818
820
Testumgebung
806
Eichung
Diagnose
Konfiguration
808
Testvorbereitung
Testsystem
816
Strukturkompilierung
Modulspezifisches Strukturcompilierprogramm
Schnittstellen-steuerung
Synchr., Folgesteue-rung, Wied.holbark.
814
Testprogrammentwicklung
Modulbetriebsmittel-datei, Schnittstelle
Fehler-/Warnungs-verarbeitung
Mehrstellensteuerung
816
Offline-Simulation
Hardware-Emulationsmodul
Mehrgerät-/ Modulsteuerung
Neues Hardw.-od. Softw.modul
816
Einbau
Neues Hardw.-od. Softw.modul
Zertifi-zierung
810
812

Figur 8