



(10) **DE 10 2018 003 142 A1** 2019.06.13

(12)

Offenlegungsschrift

(21) Aktenzeichen: **10 2018 003 142.0**

(22) Anmeldetag: **17.04.2018**

(43) Offenlegungstag: **13.06.2019**

(51) Int Cl.: **G06F 8/70 (2018.01)**

(30) Unionspriorität:
17 290 160.5 **13.12.2017** **EP**

(71) Anmelder:
The MathWorks, Inc., Natick, Mass., US

(74) Vertreter:
**Müller-Boré & Partner Patentanwälte PartG mbB,
80639 München, DE**

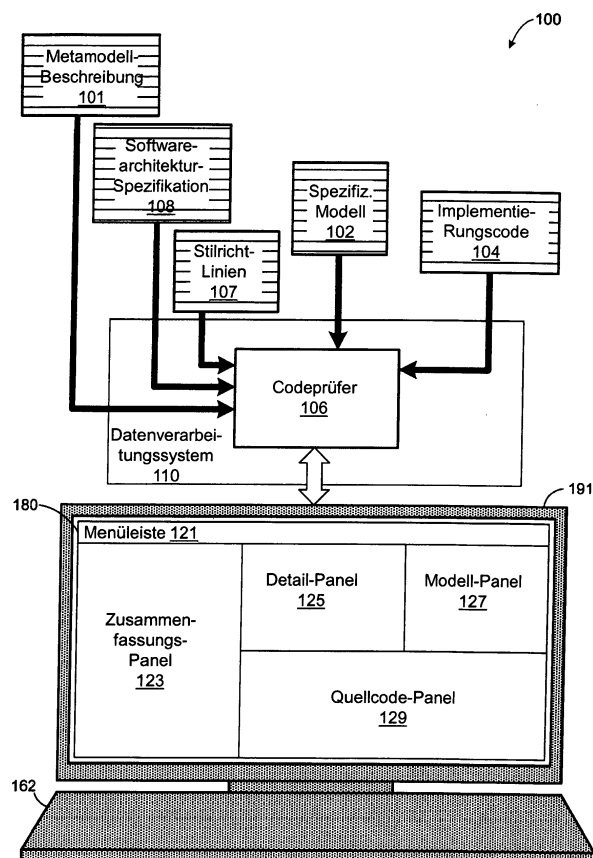
(72) Erfinder:
Bouissou, Olivier, Paris, FR

Prüfungsantrag gemäß § 44 PatG ist gestellt.

Die folgenden Angaben sind den vom Anmelder eingereichten Unterlagen entnommen.

(54) Bezeichnung: **Automatische Einstellung von Multitasking-Konfigurationen für ein Codeprüfsystem**

(57) Zusammenfassung: Es werden Verfahren und Systeme zum automatischen Einstellen von Multitasking-Konfigurationen beschrieben, die dazu verwendet werden, um durch ein Codeprüfsystem Implementierungscode zu prüfen, der auf einem dynamischen System bereitgestellt werden soll. Attribute von implementierten Aufgaben und Interrupt Service Routinen, die gleichzeitig auf dem dynamischen System laufen können, werden aus einer oder mehreren Spezifikationsdatenstrukturen eines spezifizierten Modells bestimmt und mit unabhängigen, implementierten, aus dem spezifizierten Modell vorbereiteten Rechen-Threads verknüpft. Konfiguriert mit Informationen, die für die gleichzeitigen Threads relevant sind, kann das Codeprüfsystem das Vorhandensein oder Fehlen von Defekten in dem Implementierungscode genauer bestimmen. Das spezifizierte Modell, der resultierende Implementierungscode und das dynamische System können komplex sein und eine standardisierte Softwarearchitektur, wie z. B. Automotive Open System Architecture (AUTOSAR) erfüllen.



Beschreibung

[0001] Die beschriebene Technologie betrifft eine automatisierte Konfiguration und Analyse eines Implementierungscodes durch ein Codeprüfsystem, der aus einem spezifizierten Modell erstellt wurde. Der Implementierungscode kann Rechen-Threads bzw. -stränge enthalten, die gleichzeitig auf zumindest einem Prozessor eines komplexen dynamischen Systems (z. B. einem Automobil oder einer anderen intelligenten Vorrichtung) ausgeführt werden, um eine automatisierte und/oder halbautomatische Steuerung bzw. Regelung des dynamischen Systems bereitzustellen. Die automatische Konfiguration des Codeprüfsystems kann das Konfigurieren von Einstellungen zur Multitasking-Codeanalyse des Implementierungscodes basierend auf Informationen umfassen, die von zumindest dem spezifizierten Modell erkannt werden. In Ausführungsformen ist das Codeprüfsystem angepasst, Defekte (wie Bugs und Laufzeitfehler) in den analysierten Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen, sowie Warnungen bezüglich Code bereitzustellen, der mögliche Laufzeitfehler enthalten kann.

[0002] Abschnitte des Implementierungscodes können eine Mehrzahl von Code-Threads bzw. -Strängen enthalten, die mit Aufgaben, Interrupt-Service-Routinen (ISRs) und Ressourcen verknüpft sind, die in dem spezifizierten Modell definiert werden können. Zumindest einige der Code-Threads können gleichzeitig in dem dynamischen System laufen. Um die Geschwindigkeit und Genauigkeit der Analyse des Implementierungscodes zu verbessern, können Attribute von Aufgaben, ISRs und/oder Ressourcen automatisch in dem spezifizierten Modell erkannt und mit entsprechenden Code-Threads in dem Implementierungscode verknüpft werden. Informationen, die für die Aufgaben, ISRs und Ressourcen relevant sind, können dem Codeprüfsystem zum automatischen Einstellen von Multitasking-Konfigurationen des Codeprüfsystems bereitgestellt werden.

[0003] Einige Ausführungsformen betreffen ein computerimplementiertes Verfahren zum Überprüfen des Implementierungscodes auf Fehler und zum automatischen Konfigurieren eines Codeprüfsystems basierend auf Informationen, die für die Parallelität bzw. Gleichzeitigkeit von Rechen-Threads in dem Implementierungscode relevant sind. Ein computerimplementiertes Verfahren kann umfassen: Vorgänge des Empfangens, durch einen Datenanalysator, einer oder mehrerer Datenstrukturen, die Informationen enthalten, die Aufgaben spezifizieren, die in Form von Teilen eines Implementierungscodes zu implementieren sind, der ein dynamisches System steuert bzw. regelt, wobei ein Teil der Aufgaben gleichzeitig auf einem Prozessor des dynamischen Systems unter Verwendung verschiedener Rechen-Threads innerhalb der Bestandteile des Implementierungscodes auszu-

führen sind; automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um spezifizierte Aufgabenkennungen für jede Aufgabe des Teils der Aufgaben zu identifizieren; automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um Aufgabenattribute zu identifizieren, die für die gleichzeitige Ausführung des Teils der Aufgaben auf dem Prozessor des dynamischen Systems relevant sind; und Bereitstellen von Informationen bezüglich Aufgabenidentität und der Aufgabenattribute des Teils der Aufgaben zum automatischen Durchführen einer Codeüberprüfung der Bestandteile des Implementierungscodes, um Defekte in den Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen. Ein Verfahren zum automatischen Einstellen von Multitasking-Konfigurationen für ein Codeprüfsystem kann ferner umfassen: Vorgänge zum Empfangen der einen oder mehreren Datenstrukturen durch das Codeprüfsystem; Empfangen eines Abschnitts des Implementierungscodes, der die Bestandteile des Implementierungscodes enthält, durch das Codeprüfsystem; Empfangen der Informationen über die Aufgabenidentität und die Aufgabenattribute durch das Codeprüfsystem; Empfangen der Informationen über die ISRs durch das Codeprüfsystem; Verarbeiten von zumindest einem Abschnitt der einen oder mehreren Datenstrukturen durch einen Codeverfolgungsprozessor, um automatisch Vergleichscode unter Verwendung eines oder mehrerer geänderter Aufgabenkennungen für die spezifizierten Aufgabenkennungen zu erzeugen; Analysieren des Vergleichscodes und des empfangenen Abschnitts des Implementierungscodes, um die Rechen-Threads in den Teilen des Implementierungscodes zu identifizieren, die den spezifizierten Aufgabenkennungen entsprechen; Verknüpfen der empfangenen Aufgabenattribute mit ihren entsprechenden Rechen-Threads; und statisches Analysieren der Bestandteile des Implementierungscodes durch das Codeprüfsystem basierend auf den verknüpften Aufgabenattributen und empfangenen Informationen bezüglich der ISRs, um Defekte in den Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen.

[0004] Einige Ausführungsformen betreffen ein Datenverarbeitungssystem, das zumindest einen Prozessor umfasst, der mit Computerprogrammanweisungen angepasst ist, um ein oder mehrere Verfahren zum automatischen Einstellen von Multitasking-Konfigurationen in einem Codeprüfsystem wie oben und weiter unten im Detail beschrieben auszuführen.

[0005] Einige Ausführungsformen betreffen ein computerlesbares Medium, das Computerprogrammanweisungen codiert, die, wenn sie von zumindest einem Prozessor ausgeführt werden, ein oder mehrere Verfahren zum automatischen Einstellen von Multitasking-Konfigurationen in einem Codeprüfsystem

wie oben und weiter unten im Detail beschrieben ausführen.

[0006] Die vorstehenden und andere Aspekte, Ausführungsformen und Merkmale der vorliegenden Lehren können vollständiger aus der folgenden Beschreibung in Verbindung mit den beiliegenden Zeichnungen verstanden werden.

Figurenliste

[0007] Der Fachmann wird verstehen, dass die hierin beschriebenen Figuren nur zu Veranschaulichungszwecken dienen. Es versteht sich, dass in einigen Fällen verschiedene Aspekte der Erfindung übertrieben oder vergrößert dargestellt werden können, um das Verständnis der Erfindung zu erleichtern. In den Zeichnungen beziehen sich gleiche Bezugszeichen im Allgemeinen auf gleiche Merkmale, funktionell ähnliche und/oder strukturell ähnliche Elemente in den verschiedenen Figuren. Die Zeichnungen sind nicht notwendigerweise maßstabsgetreu, stattdessen wird Wert darauf gelegt, die Prinzipien der Lehren zu veranschaulichen. Die Zeichnungen sollen den Umfang der vorliegenden Lehren in keiner Weise einschränken.

Fig. 1 zeigt im Überblick ein Beispiel eines Systems, das zum Analysieren von Implementierungscode angepasst ist;

Fig. 2 zeigt ein Beispiel einer mehrschichtigen Architektur, die gleichzeitige Aufgaben, Interrupt-Service-Routinen und Ressourcen enthalten kann;

Fig. 3 zeigt ein Beispiel einer Vorbereitung eines Implementierungscode aus einem spezifizierten Modell gemäß einigen Ausführungsformen;

Fig. 4A veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem eine Aufgabe definiert und mit einem Kurznamen verknüpft ist;

Fig. 4B veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem ein Ereignis für die in **Fig. 4A** identifizierte Aufgabe definiert ist;

Fig. 4C veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem ein Ereignis auf eine Aufgabe in einem spezifizierten Modell gemappt bzw.

abgebildet ist und ein vollständig qualifizierter Name für die Aufgabe bereitgestellt wird;

Fig. 5A veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem ein Alarm definiert ist;

Fig. 5B veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem ei-

ne mit dem Alarm in **Fig. 5A** verknüpfte Aufgabe referenziert ist;

Fig. 5C veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem eine mit dem Alarm in **Fig. 5A** verknüpfte Aufgabe definiert ist;

Fig. 6 veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem eine Interrupt-Service-Routine (ISR) definiert ist;

Fig. 7 veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem eine Ressource definiert ist;

Fig. 8 veranschaulicht einen exemplarischen Prozess, der verwendet werden kann, um Typen von Aufgaben basierend auf Aufgabenattributinformationen zu identifizieren, die aus einer oder mehreren Dateien eines spezifizierten Modells erkannt werden;

Fig. 9A veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem eine Aufgabe definiert ist und einen kurzen Namen erhält;

Fig. 9B veranschaulicht ein Beispiel eines Abschnitts eines spezifizierten Modells, in dem ein vollständig qualifizierter Name für die Aufgabe von **Fig. 9A** bereitgestellt ist;

Fig. 9C veranschaulicht ein Beispiel eines Abschnitts von Zwischencode, der aus dem spezifizierten Modell erzeugt wird, das den in **Fig. 9B** gezeigten Abschnitt des spezifizierten Modells enthält;

Fig. 9D veranschaulicht einen exemplarischen Teil von Implementierungscode, der aus Zwischencode erzeugt wird, der den in **Fig. 9C** gezeigten Abschnitt von Zwischencode enthält, und zwar gemäß einem Code-Vorverarbeitungsalgorithmus;

Fig. 9E veranschaulicht einen exemplarischen Teil von Implementierungscode, der aus Zwischencode erzeugt wird, der den Abschnitt des in **Fig. 9C** gezeigten Zwischencodes enthält, und zwar gemäß einem vorbestimmten Code-Vorverarbeitungsalgorithmus;

Fig. 10A veranschaulicht einen exemplarischen Abschnitt von Implementierungscode, der aus Zwischencode erzeugt wird, der eine Ressource enthält, und zwar gemäß einem Code-Vorverarbeitungsalgorithmus;

Fig. 10B veranschaulicht einen exemplarischen Abschnitt von Implementierungscode, der aus Zwischencode erzeugt wird, der eine Ressource enthält, und zwar gemäß einem vorbestimmten Code-Vorverarbeitungsalgorithmus;

Fig. 11 zeigt einige Komponenten, die in Ausführungsformen eines Codeprüfsystems **100** enthalten sein können, das für eine automatisierte Multitasking-Analyse angepasst ist;

Fig. 12 veranschaulicht exemplarische Vorgänge, die mit einem Verfahren zum automatischen Identifizieren von Aufgaben, ISRs und Ressourcen und ihrer verknüpften Attribute aus einem spezifizierten Modell gemäß einigen Ausführungsformen verknüpft sind;

Fig. 13 veranschaulicht exemplarische Vorgänge, die mit einem Verfahren zum automatischen Verknüpfen von Attributen mit spezifizierten Aufgaben, ISRs und Ressourcen zum Implementieren von Code-Threads gemäß einigen Ausführungsformen verknüpft sind;

Fig. 14 veranschaulicht exemplarische Vorgänge, die mit einem Verfahren zum automatischen Konfigurieren eines Codeprüfsystems für eine Multitasking-Analyse gemäß einigen Ausführungsformen verknüpft sind;

Fig. 15 zeigt eine exemplarische Konfigurationsbenutzerschnittstelle für ein Codeprüfsystem, die implementiert werden kann, um es einem Benutzer zu ermöglichen, Dateien eines spezifizierten Modells zu identifizieren, das für die Analyse von Implementierungscode relevant ist, ein anwendbares Metamodell zu identifizieren und eine manuelle oder automatische Konfiguration von Aufgaben und ISRs für die Multitasking-Analyse auszuwählen;

Fig. 16 zeigt ein Beispiel einer Ergebnis-Benutzerschnittstelle für ein Codeprüfsystem, die implementiert werden kann, um Codeanalyseergebnisse auszuwählen und anzuzeigen;

Fig. 17 zeigt ein Beispiel einer Task/ISR-Benutzerschnittstelle, die Informationen auflisten und bereitstellen kann, die für Aufgaben und ISRs relevant sind, die in einem spezifizierten Modell identifiziert sind;

Fig. 18 zeigt ein Beispiel von Multitasking-Analyseergebnissen, die von einem Codeprüfsystem angezeigt werden können, gemäß einigen Ausführungsformen;

Fig. 19 zeigt ein Beispiel von Multitasking-Ergebnisdetails, die für die in **Fig. 18** dargestellten Ergebnisse relevant sind, die auch von dem Codeprüfsystem angezeigt werden können, und zwar gemäß einigen Ausführungsformen;

Fig. 20 zeigt ein Beispiel einer Rechenumgebung, die speziell mit Computerprogrammierungsanweisungen angepasst sein kann, um Im-

plementierungscode gemäß den vorliegenden Ausführungsformen zu analysieren; und

Fig. 21 ist eine teilweise schematische Darstellung eines Beispiels einer Modellierungsumgebung.

[0008] Die Merkmale und Vorteile der vorliegenden Erfindung werden aus der folgenden detaillierten Beschreibung in Verbindung mit den Zeichnungen deutlicher.

DETAILLIERTE BESCHREIBUNG

[0009] Beispiele von dynamischen Systemen, die für die vorliegenden Ausführungsformen relevant sind, umfassen, ohne hierauf beschränkt zu sein, Automobile, Avionikinstrumente, Luft- und Raumfahrtmaschinen, Seefahrzeuge, Stromgeneratoren, Baumaschinen, Bergbaumaschinen, landwirtschaftliche Maschinen und Forstmaschinen. In einigen Fällen können die vorliegenden Ausführungsformen auch auf komplexe dynamische Mehrprozessor-Rechensysteme angewendet werden. In Ausführungsformen kann ein dynamisches System einen oder mehrere vernetzte Computer (z. B. eingebettete Computer) umfassen, die miteinander interagieren. Beispiele für dynamische Systeme umfassen Fahrzeuge (wie Automobile), physische bzw. technische Anlagen oder stationäre Maschinen und ein entsprechendes Steuer- bzw. Regelsystem (z. B. ein System, das eine beliebige Kombination aus einem oder mehreren Rückkopplungscontrollern, einem oder mehreren programmierbaren Logikcontrollern, einem oder mehreren Prozessoren (die Mikrocontroller und Mikroprozessoren umfassen), einem oder mehreren feldprogrammierbaren Gate-Arrays und einem oder mehreren digitalen Signalprozessoren umfasst), das eine automatisierte oder halbautomatische Steuerung bzw. Regelung des Fahrzeugs, der Anlage oder der Maschine durchführt. Es ist ersichtlich, dass die Integration moderner Diagnose- und Steuer- bzw. Regelelektronik in dynamische elektromechanische Systeme beispielsweise zu einer bemerkenswert komplexen automatisierten oder halbautomatischen Steuerung bzw. Regelung solcher Systeme führen kann. In einigen Fällen kann ein Systemsteuer- bzw. -regelcode Millionen von Codezeilen umfassen, die in Tausenden von Dateien verteilt sind und möglicherweise in unterschiedlichen Computersprachen geschrieben sind, wie es bei einigen modernen Automobilen der Fall ist. Ferner kann der Code in einer verteilten Verarbeitungsumgebung ausgeführt werden. Zum Beispiel kann sogar innerhalb eines einzelnen modernen Automobils die verteilte Verarbeitungsumgebung mehr als einhundert Mikrocontroller und Mikroprozessoren enthalten, die den Systemsteuer- bzw. regelcode ausführen.

[0010] Um bei der Entwicklung, Herstellung und Wartung solcher Systeme zu helfen, können Hersteller oder Vertreter von verwandten Industrien an ei-

ner Methodik zusammenarbeiten, durch die Systemsteuer- bzw. Regelcodes zur Implementierung in solchen komplexen Systemen strukturiert werden. Eine solche Zusammenarbeit besteht unter Automobilherstellern, deren Zulieferern und Unternehmen aus der Elektronik- und Softwareindustrie. Die Zusammenarbeit hat zur Entwicklung der Automotive Open System Architecture geführt, die als AUTOSAR bekannt ist. AUTOSAR wurde ursprünglich als eine Softwarearchitekturspezifikation entwickelt, die Beschränkungen auferlegt, wie Computercode bei der Systemsteuerung von Automobilen implementiert werden kann, obwohl die Spezifikation für andere oben aufgeführte Maschinentypen übernommen bzw. festgesetzt wurde.

[0011] Ein Ziel von AUTOSAR (und anderen Softwarearchitekturspezifikationen) besteht darin, die Entwicklung von Implementierungscode durch Systemsteuerungsingenieure im Wesentlichen unabhängig von der zugrundeliegenden Hardware-Infrastruktur zu machen, anstatt Implementierungscode zu haben, der für die zugrunde liegende Hardware hochspezialisiert ist. Dies ermöglicht die Übertragbarkeit des Codes auf jede Hardware, welche die relevante Softwarearchitekturspezifikation unterstützt. Durch die Einhaltung der Softwarearchitekturspezifikation kann ein Systemingenieur Code entwickeln, der das Verhalten verschiedener physikalischer Systemkomponenten (z. B. Sensoren, Aktoren, Mikrocontroller, Motoren usw.) mit wenig oder keinem Wissen über Kommunikations- und Betriebsdetails der physikalischen Komponenten bestimmt. Ein Systemingenieur kann z. B. ein spezifiziertes Modell gemäß Richtlinien erstellen, die den Systemaufbau Richtlinien entsprechen. In Ausführungsformen spezifiziert ein spezifiziertes Modell einer Softwarearchitekturspezifikation zumindest Verhalten und/oder Funktionalitäten von Codeabschnitten, die als Steuer- bzw. Regelalgorithmen auf dem dynamischen System auszuführen sind. Ein Codeabschnitt kann einen oder mehrere Codeteile enthalten, die mit Aufgaben und/oder Unterbrechungen verknüpft sind, die in einem spezifizierten Modell **102** definiert sind. Ein spezifiziertes Modell kann auch Attribute spezifizieren, die mit den Codeteilen verknüpft sind. Zumindest einige der Codeteile können, wenn sie implementiert werden, gleichzeitig und unabhängig als Rechen-Threads bzw. -Stränge auf derselben Hardware ausgeführt werden, wenn sie in einem dynamischen System eingesetzt werden. In einigen Implementierungen umfasst ein Rechen-Thread, der auch als ein Thread bezeichnet wird, eine Sequenz von Programmieranweisungen, die eine vordefinierte Funktionalität ausführt. Die Sequenz kann unabhängig von einem Scheduler zur Ausführung auf einem Prozessor gehandhabt werden. Zum Beispiel kann ein Thread eine Kennung aufweisen, die verwendet wird, um die Ausführung des Threads auf dem Prozessor aufzurufen. Ein Thread kann gleichzeitig (z. B. in ei-

ner Zeitmultiplex-Weise) mit anderen Threads auf demselben Prozessor oder auf verschiedenen Prozessoren ausgeführt werden und kann unterschiedliche Speicher, Variablen und/oder Adressraum mit den anderen Threads teilen oder verwenden. Ein oder mehrere Threads können als Teil eines größeren Computerprozesses ausgeführt werden. Zum Beispiel könnte ein größerer Rechenprozess einen aktuellen Kilometerbereich für ein Fahrzeug schätzen. Dieser Prozess kann einen „Kraftstoffreserve“-Thread umfassen, der wiederholt ausgeführt wird, um eine Kraftstoffmenge zu bestimmen, die in dem Kraftstofftank des Fahrzeugs verbleibt. Der Prozess kann auch einen „Verbrauchsmenge“-Thread enthalten, der gleichzeitig mit dem Kraftstoffreserve-Thread ausgeführt wird. Der Verbrauchsmenge-Thread kann beispielsweise eine Menge eines Kraftstoffverbrauchs aus Sensordaten oder Fahrzeuggeschwindigkeit bestimmen. Datenwerte, die von den beiden Threads in einen gemeinsamen Speicherplatz oder in unterschiedliche Speicherplätze geschrieben werden, können von dem Kilometerbereichsprozess verwendet werden, um eine Entfernung zu berechnen, die das Fahrzeug zurücklegen könnte, bevor kein Kraftstoff mehr zur Verfügung steht.

[0012] Ein Programmierer, der Implementierungscode aus dem spezifizierten Modell für ein dynamisches System entwickelt, muss zusätzliche Konformitätsebenen für den Implementierungscode adressieren. Neben dem Schreiben von Code, der frei von Syntaxfehlern und Laufzeitfehlern ist (z. B. Division durch Null, Stapelüberlauf, Assertionsfehler, Ausnahmen, nicht initialisierte Zeiger usw.), muss der Programmierer Code vorbereiten, der dem spezifizierten Modell nachkommt und bei Ausführung den Vorgaben der Systemaufbau Richtlinien entspricht. Obwohl Standard-Code-Compiler einige Laufzeitfehler erfassen können, haben Compiler keine Kenntnis von vollständigen Systemaufbau Richtlinien oder spezifizierten Modellen und können daher nicht feststellen, ob der Code frei von Fehlern ist, die zumindest das spezifizierte Modell betreffen. Standard-Compiler können auch nicht feststellen, ob Teile des Codes, die gleichzeitig ausgeführt werden, ohne Fehler ausgeführt werden. Da die Codebasis für komplexe dynamische Systeme sehr groß sein kann, ist die Aufgabe, des Prüfens des Implementierungscode auf fehlerfreie Ausführung hin für einen Programmierer unüberwindbar. Ein Versagen des Codes während des Systembetriebs kann inakzeptabel sein und zu einem unsicheren Betrieb von Maschinen führen, die große Kräfte erzeugen. Post-deployment Code Debugging kann möglicherweise keine Option sein.

[0013] Im Überblick und unter Bezugnahme auf **Fig. 1** betreffen die beschriebenen Ausführungsformen ein Codeprüfsystem **100**, das eine automatisierte Analyse des komplexen Implementierungscode **104** (z. B. etwa eine Million Zeilen von Code) in ver-

nünftigen Zeitspannen (z. B. einigen zehn Minuten oder weniger) durchführen kann, wobei der Implementierungscode mehrere Aufgaben oder Prozesse enthalten kann, die gleichzeitig auf einem oder mehreren Prozessoren eines dynamischen Systems ausgeführt werden. Sogar größere Mengen des Implementierungscode **104** können durch das Codeprüfsystem **100** analysiert werden. Das Codeprüfsystem kann unter anderem schnell und visuell einem Benutzer identifizieren bzw. aufzeigen, wo in der Implementierung Code-Fehlanpassungen zwischen dem Implementierungscode **104** und einem spezifizierten Modell **102** auftreten, Fehlerquellen und/oder Laufzeitfehler identifizieren, die sich beispielsweise auf die gleichzeitige und nicht gleichzeitige Ausführung von Codeteilen beziehen. In einigen Fällen kann ein Codeprüfsystem das Vorhandensein oder Nichtvorhandensein von Fehler identifizieren (z. B. Code-Nachweis-Funktionalität durchführen) und kann zusätzlich eine Ursache für jeden detektierten Fehler identifizieren.

[0014] In Ausführungsformen kann ein Codeprüfsystem **100** eine statische Codeanalyse von Segmenten des Implementierungscode **104** durchführen. Die statische Analyse kann das Verwenden von formalen Methoden zum Durchführen einer oder einer beliebigen Kombination aus semantischer Analyse, abstrakter Interpretation, Modellprüfung und Erfüllungslösung zum Verifizieren von Software-Interprozedural-, Kontroll- und Datenflussverhalten umfassen. Zumindest einige Codeteile innerhalb des Implementierungscode **104** umfassen Rechen-Threads bzw. -Stränge, die gleichzeitig ausgeführt werden können, wenn sie auf einem dynamischen System angewendet werden. Gleichzeitige Rechen-Threads können Tasks und ISRs entsprechen, die gleichzeitig auf einem Prozessor im dynamischen System ausgeführt werden sollen. Zum Beispiel kann ein gleichzeitiger Thread eine Sequenz von Anweisungen umfassen, die unabhängig von einem Betriebssystem-Scheduler für den Prozessor verwaltet werden können, auf dem der Thread gleichzeitig mit anderen Rechen-Threads ausgeführt wird. Das Codeprüfsystem **100** kann eine Codeprüffunktionalität betreffend die gleichzeitige Ausführung (auch als Multitasking bezeichnet) der Rechen-Threads ausführen und kann zusätzlich auf Laufzeitfehler überprüfen. Ein Codeprüfsystem **100** kann ein speziell angepasstes Datenverarbeitungssystem **110** umfassen, das mit Eingabe/Ausgabe-Vorrichtungen wie einer Anzeige **191**, auf der eine graphische Benutzerschnittstelle dargestellt werden kann, einer Tastatur **162** und einer Maus (nicht gezeigt) in Verbindung steht. In einigen Ausführungsformen kann die Anzeige **191** eine Berührungsbildschirmanzeige umfassen und eine Tastatur **162** muss nicht vorhanden sein. Das Codeprüfsystem **100** kann einen Codeprüfer **106** enthalten, der auf dem Datenverarbeitungssystem **110** ausgeführt wird.

[0015] In Ausführungsformen ist das Datenverarbeitungssystem **110** angepasst, ein spezifiziertes Modell **102** zu empfangen, das unter anderem eine oder mehrere Funktionalitäten identifiziert, die von zumindest einem Abschnitt eines Systemsteuer- bzw. -regelcodes durchzuführen sind, der eine automatisierte oder halbautomatisierte Steuerung bzw. Regelung eines dynamischen Systems bereitstellen soll, wie eines der oben beschriebenen dynamischen Systeme. Das empfangene spezifizierte Modell **102** kann ein Abschnitt oder die Gesamtheit eines vollständigen spezifizierten Modells sein, das für ein dynamisches System vorbereitet ist. Das Datenverarbeitungssystem **110** kann ferner Zwischencode (in **Fig. 3** gezeigt) und /oder Implementierungscode **104** empfangen, der von einem Programmierer vorbereitet wurde, um die der Funktionalität des spezifizierten Modells **102** auszuführen. In einigen Fällen empfängt das Datenverarbeitungssystem **110** auch Informationen oder wird mit Informationen über einen oder mehrere Teile eines Metamodells **101** programmiert, das für das spezifizierte Modell **102** relevant ist. Die Daten des Metamodells **101** können Informationen über eine Sprache und eine Syntax enthalten, die zum Bilden des spezifizierten Modells **102** verwendet werden. Das Datenverarbeitungssystem **110** kann ferner Informationen empfangen oder wird mit Informationen über Stilrichtlinien **107** und/oder Standards programmiert, die mit dem Metamodell **101** verwendet werden, beispielsweise wenn das spezifizierte Modell **102** vorbereitet wird. Exemplarische Stilrichtlinien können umfassen, ohne hierauf beschränkt zu sein, Formatierung des spezifizierten Modells **102**, Schema, das verwendet werden kann, um zumindest Abschnitte eines spezifizierten Modells **102** vorzubereiten, Dateiorganisationsrichtlinien und Dateibenennungskonventionen. Die Stilrichtlinien **107** können auch Steuer- bzw. Regelalgorithmusmodellierungsrichtlinien für MATLAB®, Simulink®, Stateflow®- und Embedded Coder®-Softwarepakete enthalten. Die Richtlinien werden von The MathWorks, Inc. aus Natick, Massachusetts, gehostet und in Zusammenarbeit mit Vertretern der Automobilindustrie entwickelt. Beispiele für aktuelle Richtlinien zur Modellierung von Steuer- bzw. Regelalgorithmen finden sich in „Control Algorithm Modelling Guidelines Using MATLAB®, Simulink®, and Stateflow®, Version 3.0“, die hierin durch Bezugnahme aufgenommen und unter <https://www.mathworks.com/solutions/automotive/standards/maab.html> am 13. Februar 2018 verfügbar sind. Die Stilrichtlinien **107** können auch Informationen aus Industriestandard-Richtlinien wie IEC 61508 und deren Ableitungen, ISO 26262 und MISRA C enthalten. Das Datenverarbeitungssystem **110** kann ferner Informationen empfangen oder mit Informationen über eine Softwarearchitekturspezifikation **108** programmiert werden. Daten der Softwarearchitekturspezifikation **108** können Informationen über eine Softwarearchitektur (von der ein Beispiel in **Fig. 2** gezeigt ist) enthalten, auf der Implementierungscode

104 ausgeführt werden kann. Informationen (wie Metamodell 101-Informationen, Stilrichtlinien **107** und Softwarearchitekturspezifikationen **108**), die von dem Datenverarbeitungssystem empfangen werden, können hierin kollektiv als Systemaufbau-Richtlinien bezeichnet werden.

[0016] In Ausführungsformen können zumindest das spezifizierte Modell **102** und der Implementierungscode **104** einem Codeprüfer **106** bereitgestellt werden, der zumindest statische Codeanalyse von Segmenten des Implementierungscode **104** durchführen kann, wie in der Europäischen Patentanmeldung Nr. 17290155.5, eingereicht am 30. November 2017 mit dem Titel „Systems and Methods for Evaluating Compliance of Implementation Code with a Software Architecture Specification“ beschrieben, die hierin durch Bezugnahme in ihrer Gesamtheit aufgenommen ist. Der Codeprüfer **106** kann Programmieranweisungen umfassen, die auf Hardware des Datenverarbeitungssystems **110** ausgeführt werden, das Logikgatter und Register umfasst, die als sequentielle Logikschaltungen angeordnet sind. Exemplarische Programmieranweisungen, die zur Verwendung als Teil eines Codeprüfers **106** in dem Codeprüfsystem **100** verwendet werden können, ist die Polyspace Code Checker™-Software, die von The MathWorks, Inc. aus Natick, Massachusetts, erhältlich ist.

[0017] Als ein weiteres Beispiel eines spezifizierten Modells **102** und zugehörigen Implementierungscode **104** kann ein spezifisches Modell, das für ein Automobilsystem vorbereitet ist, Funktionen definieren, die mit der Steuerung bzw. Regelung eines Fahrzeugmotors verknüpft sind (z. B. Kraftstoffeinspritzung, Lufteinlass, Drehzahlüberwachung, Temperatur, etc.). Das spezifizierte Modell **102** kann zum Beispiel von einem Fahrzeugsystem-Designer erstellt werden. In Ausführungsformen kann das spezifizierte Modell **102** eine oder mehrere Datenstrukturen (z. B. eine oder mehrere Dateien) umfassen, die manuell gemäß einem Schema erstellt werden, das zum Beispiel durch die Softwarearchitekturspezifikation **108** definiert ist.

[0018] Der Implementierungscode **104** kann entsprechender Softwarecode sein, der entwickelt wird, auf dem physikalischen System angewendet zu werden, um die gewünschten Funktionalitäten für die Motorsteuerung bzw. -regelung zu implementieren, und zwar nach dem obigen Automobilbeispiel, und er kann von einer anderen Entität als der Entität vorbereitet werden, die das spezifizierte Modell vorbereitet hat. Zum Beispiel kann die Vorbereitung des Implementierungscode **104** von einem Programmierer auf Anforderung eines Ingenieurs oder Systemdesigners erfolgen, der das spezifizierte Modell **102** entwickelt hat. Der Implementierungscode und jeglicher Zwischencode, falls er erzeugt wird, kann handgeschriebene Abschnitte enthalten und kann Abschnit-

te enthalten, die automatisch aus dem spezifizierten Modell **102** erzeugt werden (z. B. unter Verwendung eines Code-Vorbereitungstools wie DaVinci Developer, erhältlich von Vector Informatik GmbH, Stuttgart, Deutschland oder Embedded Coder®, erhältlich von The MathWorks, Inc. aus Natick, Massachusetts). In einigen Fällen kann der Implementierungscode vollständig oder teilweise unter Verwendung einer automatisierten Codegenerierung vorbereitet werden. In einigen Fällen kann zumindest ein Abschnitt des Implementierungscode **104** ein Code sein, der in einer technischen Datenverarbeitungsumgebung erzeugt wird, wie z. B. Code, der aus einer grafischen Programmierungsumgebung wie der Simulink-Umgebung von The MathWorks, Inc. aus Natick, Massachusetts, erzeugt wird. In zusätzlichen Ausführungsformen umfasst der Implementierungscode **104** Legacy-Code, der gemäß einer früheren Version eines spezifizierten Modells und/oder unter Bedingungen erzeugt wurde, die beispielsweise durch eine frühere Version von Systemaufbau-Richtlinien auferlegt wurden.

[0019] Aus Sicherheitsgründen kann es wünschenswert sein, den Implementierungscode **104** während oder nach der Entwicklung, aber vor der Bereitstellung auf dem Zielsystem schnell und genau auf Fehler zu überprüfen. Wenn der Implementierungscode **104** vor dem Start nicht angemessen geprüft wird, kann ein technisches Ergebnis ein unsachgemäßer Betrieb des ausführbaren Codes sein, der möglicherweise eine unsichere Situation für eine durch den Code gesteuerte bzw. geregelte Maschine erzeugt.

[0020] Der Implementierungscode **104** kann in einer Sprache geschrieben sein oder kann Dateien umfassen, die in mehreren unterschiedlichen Sprachen geschrieben sind. Exemplarische Sprachen umfassen, ohne hierauf beschränkt zu sein, C, C++, C#, Assemblercode, Java, Ada, strukturierten Text zur Ausführung auf einer programmierbaren Logiksteuerung (PLC) und Hardwarebeschreibungssprache (HDL). In einigen Fällen kann der Implementierungscode **104** verschlüsselt sein, wenn er aus proprietären und/oder Sicherheitsgründen empfangen wird, und anschließend durch das Codeprüfsystem **100** zur Analyse entschlüsselt werden. Ein Benutzer oder Kunde würde einen Schlüssel für die Entschlüsselung von verschlüsseltem Implementierungscode bereitstellen.

[0021] Während und/oder nach der Analyse des Implementierungscode **104** durch das Codeprüfsystem **100** können Informationen über die Analyse zur Anzeige bereitgestellt werden. Die Informationen über die Analyse können in einer oder mehreren graphischen Benutzerschnittstellen **180** auf einer Anzeige **191** wiedergegeben werden, wie es beispielsweise allgemein in **Fig. 1** dargestellt ist. Andere Anzeigeanordnungen können in anderen Aus-

führungsformen verwendet werden, um Ergebnisse von dem Codeprüfer **106** zu übermitteln. Als nur ein Beispiel können mehrere Panels auf einer Benutzerschnittstelle **180** angezeigt werden, die Details über das Vorhandensein oder die Abwesenheit von Defekten in dem Implementierungscode zusammenfassen und/oder bereitstellen sowie Informationen bereitstellen, die für die Multitasking-Analyse des Implementierungscode **104** relevant sind. Eine Benutzerschnittstelle **180** kann eine Menüleiste **121** zum Anavigieren und Auswählen verschiedener Funktionalität des Codeprüfsystems **100** enthalten. Die Benutzerschnittstelle **180** kann ein Zusammenfassungs-Panel **123**, das eine kurze Zusammenfassung des analysierten Umsetzungscode bereitstellt, ein Detail-Panel **125**, das Informationen über Systemaufbau-Richtlinien oder spezifizierte Modell- oder Codierungsverletzung bereitstellt, ein Modell-Panel **127**, das weitere erläuternde Informationen über den Teil der Systemaufbau-Richtlinien oder des spezifizierten Modells **102** bereitstellt, die für die Verletzung relevant sind, und ein Quellcode-Panel **129** enthalten, das einen Abschnitt von Quellimplementierungscode **104** anzeigt, der die Verletzungscodezeile oder Abschnitt enthält. Detailliertere Beispiele von Benutzerschnittstellen werden nachstehend in Verbindung mit **Fig. 15** bis **Fig. 18** beschrieben. Solche Benutzerschnittstellen **180** können eine leicht verständliche Zusammenfassung von Codeüberprüfungsergebnissen, ein einfaches Navigationstool und hilfreiche visuelle Anzeigen liefern, die für jeden Fehler und seine Ursachen relevant sind. Zum Beispiel können angezeigter Text und Informationen eine Verbindung zwischen einer Codezeile oder einem Abschnitt, der einen Fehler enthält oder nicht enthält, einem entsprechenden Modellteil aus dem spezifizierten Modell **102** und optional entsprechenden Systemaufbau-Richtlinien zeigen. Solche visuellen Hinweise können einem Programmierer dabei helfen, den Implementierungscode **104** (und das spezifizierte Modell **102**, falls erforderlich) schnell zu überdenken, zu überprüfen und zu überarbeiten, um Fehler oder Laufzeitfehler aus dem Implementierungscode **104** zu entfernen.

[0022] Um die Komplexität der Codeüberprüfung für Code, der auf dynamischen Systemen eingesetzt wird, weiter zu erklären und Aspekte der vorliegenden Erfindung weiter zu erläutern, wird nun Bezug auf **Fig. 2** genommen. In Ausführungsformen können Hardware und Software zum Steuern bzw. Regeln eines dynamischen Systems oder Rechensystems in einer mehrschichtigen Architektur implementiert werden, wie in der Zeichnung dargestellt. Im Allgemeinen kann eine niedrigere Ebene der Architektur, die hierin als eine Basisdienstschicht **290** bezeichnet wird, Hardware und einen Basissoftwaredienst zum Koppeln bzw. schnittstellenmäßigen Verbinden mit und Steuern bzw. Regeln der Systemhardware und jeglicher Peripheriegeräte umfassen. Basissoftwaredienste können Routinen zum Behan-

deln von Interrupts bzw. Unterbrechungen während der Codeausführung auf Hardwarekomponenten enthalten. Ein Code, der für die Basisdienstschicht **290** entwickelt wurde, kann zum Beispiel Maschinencode umfassen.

[0023] Eine obere Schicht der Architektur, die hierin als die Anwendungsschicht **210** bezeichnet wird, kann einen Softwarecode umfassen, der von Programmierern entwickelt wurde, der eine Systemsteuerung bzw. -regelung auf höherer Ebene gemäß den im Softwarecode definierten Funktionalitäten bereitstellt. Software in der Anwendungsschicht **210** kann eine oder mehrere Schnittstellen für einen Benutzer des Systems bereitstellen. In einigen Fällen kann eine Schnittstelle teilweise als On-Board-Diagnose(OBD)-Port bereitgestellt werden, auf den über einen Hardware-Mehrfachsteckverbinder zur Systemdiagnose zugegriffen werden kann. Software, die für die Anwendungsschicht **210** entwickelt wurde, kann einen Computercode umfassen, der unter Verwendung weithin bekannter Programmiersprachen vorbereitet wird, wie beispielsweise C, C++, Java und Python, ohne jedoch darauf beschränkt zu sein.

[0024] Eine Zwischenschicht **220** stellt eine Schnittstelle zwischen Anwendungen, die in der Anwendungsschicht **210** ausgeführt werden, und Operationen, die in der Basisdienstschicht **290** ausgeführt werden, bereit. Die Zwischenschicht **220** kann beispielsweise einem Betriebssystem in einem herkömmlichen Rechensystem entsprechen. Im Allgemeinen verwaltet und versendet die Zwischenschicht **220** Aufgaben oder Prozesse an die Basisdienstschicht **290** als Antwort auf Funktionen, Verfahren oder Unterroutinen, die in der Anwendungsschicht **210** ausgeführt werden, und empfängt Daten von der Basisdienstschicht **290**, die einer oder mehreren Anwendungen bereitgestellt werden können, die in der Anwendungsschicht **290** ausgeführt werden. Ausführungsformen der vorliegenden Erfindung, die sich auf die Automatisierung der Codeüberprüfungsanalyse beziehen, sind anwendbar auf dynamische Systeme und Rechensysteme mit mehrschichtigen Architekturen, wie sie oben in Verbindung mit **Fig. 2** beschrieben wurden, und für die Systemaufbau-Richtlinien für das Codeprüfsystem **100** verfügbar sind, wie es in Verbindung mit **Fig. 1** beschrieben wurde.

[0025] Im weiteren Detail zeigt **Fig. 2** eine dreischichtige Softwarearchitektur **200**, die AUTOSAR-spezifisch ist, um einen Systemsteuer- bzw. -regelcode zu implementieren. Die AUTOSAR-Architektur wird zum Erläutern von Aspekten der Erfindung verwendet, aber die Erfindung ist nicht auf AUTOSAR beschränkt. Als nur ein Beispiel einer Softwarearchitekturspezifikation umfasst die AUTOSAR-Dreischichtarchitektur **200** eine Anwendungsschicht **210**, eine Laufzeitumgebung **250** und eine Basis-

dienstschicht **290**. Bei AUTOSAR umfasst die Anwendungsschicht eine Anzahl von Softwarekomponenten **211**, **212**, **218**, die jeweils Funktionalitäten zum Zugreifen auf Komponenten (z. B. elektronische Steuer- bzw. Regeleinheiten (ECUs) **280** und/oder Sensoren) in der Basisdienstschicht eines Kraftfahrzeugsystems umfassen. Jede Softwarekomponente **211**, **212**, **218** kann mit einer Anzahl von ausführbaren Funktionen oder Runnables verknüpft sein. In Ausführungsformen ist das Verhalten der Softwarekomponenten in dem spezifizierten Modell **102** definiert, das gemäß den vorliegenden Ausführungsformen analysiert werden kann, um Aufgaben, ISRs, Ressourcen und verknüpfte Attribute zu identifizieren. Die Aufgaben können beispielsweise mit den ausführbaren Funktionen oder Runnables verknüpft sein und gleichzeitig in einem dynamischen System laufen. Eine Aufgabe oder ISR kann als ein Teil oder Teilsatz von Implementierungscode **104** (z. B. ein Rechen-Thread innerhalb des Implementierungscodes) implementiert werden, der auf die Implementierung der Funktionalität der Aufgabe gerichtet ist, wie in dem spezifizierten Modell **102** definiert. Eine Identifikation eines Teils des Implementierungscodes **104**, der mit einer Aufgabe verknüpft ist, kann aus einer Kennung in dem Implementierungscode **104** bestimmt werden, die zu einer Kennung für eine entsprechende Aufgabendefinition zurückverfolgt werden kann, die in dem spezifizierten Modell **102** bereitgestellt wird, und zwar unter Verwendung der hierin beschriebenen Methodik. Eine Aufgabe kann ein oder mehrere Runnables enthalten. Ein Runnable kann eine aufrufbare Befehlsfolge sein.

[0026] Eine Laufzeitumgebung **250** für eine mehrschichtige Architektur kann Code umfassen, der basierend auf der Kenntnis von Code in benachbarten oder anderen Schichten der Softwarearchitektur vorbereitet wird. Für AUTOSAR wird die Laufzeitumgebung **250** zumindest teilweise aus der Kenntnis der Softwarekomponenten **211**, **212**, **218** in der Anwendungsschicht **210** und aus der Kenntnis der Basissoftware **270** und der ECUs **280** in der Basisdienstschicht **290** erzeugt. Eine AUTOSAR-Laufzeitumgebung **250** kann Code enthalten, der gemäß einer durch die AUTOSAR-Systemaufbau-Richtlinien spezifizierten Methodik erzeugt wird.

[0027] Eine AUTOSAR-Laufzeitumgebung **250** kann eine Schnittstelle zwischen Softwarekomponenten in der Anwendungsschicht und Diensten in der Basisdienstschicht **290** bereitstellen. Die Laufzeitumgebung **250** kann mit Softwarekomponenten in der Anwendungsschicht **210** durch Softwareschnittstellen **240-1**, **240-2**, **240-3** interagieren (allgemein als Schnittstellen **240** bezeichnet). Die Laufzeitumgebung **250** kann auch mit der Basissoftware **270** durch Schnittstellen **260-1**, **260-2** interagieren. Die Schnittstellen können Datenanschlüsse (z. B. adressierbare Abschnitte in einem oder mehreren Datenspeicher-

elementen) umfassen, über die Informationen zwischen den verschiedenen Schichten der dreischichtigen Architektur **200** sowie zwischen Softwarekomponenten ausgetauscht werden. Auf diese Weise kann Funktionalität, die durch die Softwarekomponenten **211**, **212**, **218** definiert ist, auf der zugrundeliegenden elektronischen Steuer- bzw. Regeleinheit (ECU)-Hardware **280** in der Basisdienstschicht **290** implementiert werden (z. B. unter Verwendung einer oder mehrerer Aufgaben (die ein oder mehrere Runnables enthalten können), ISRs und Ressourcen), ohne dass ein Programmierer die speziellen Details über das Codieren für die ECU-Hardware **280** kennen muss.

[0028] Mit dem AUTOSAR-Beispiel fortfahrend, kann die Laufzeitumgebung **250** an der Ausführung von Softwarekomponenten **211**, **212**, **218** in der Anwendungsschicht **210** und ihren Runnables **211-1**, **211-2**, ... **211-18** teilnehmen. Zum Beispiel kann ein Betriebssystem, das in der Basissoftwareschicht **270** ausgeführt wird und auf die Laufzeitumgebung **250** zugreift, vorhandene Daten erfassen und/oder Daten an einer oder mehreren Schnittstellen **240** der Laufzeitumgebung platzieren und einen Aufruf durch die Laufzeitumgebung eines Runnables einer Softwarekomponente bewirken. In einigen Fällen können bestimmte Runnables routinemäßig von der Laufzeitumgebung **250** nur basierend auf der verstrichenen Zeit aufgerufen werden. Zum Beispiel kann die Laufzeitumgebung zyklisch eine „Geschwindigkeit“ aufrufen, die ausgeführt werden kann, um Geschwindigkeitsdaten abzurufen, die an einen Datenport einer Schnittstelle gesendet werden, die mit einem Geschwindigkeitssensor verknüpft ist. In einigen Implementierungen können Sequenzen von Runnables gemäß einer zulässigen Sequenz aufgerufen werden, die durch das spezifizierte Modell **102** spezifiziert ist und/oder durch die Softwarearchitekturspezifikation **108** erlaubt ist.

[0029] Obwohl AUTOSAR hierin als eine exemplarische Softwarearchitekturspezifikation beschrieben ist, auf welche die beschriebenen Ausführungsformen angewendet werden können, sind die Ausführungsformen nicht nur auf AUTOSAR beschränkt. Exemplarische Ausführungsformen können auch mit anderen Softwarearchitekturspezifikationen wie, unter anderem, Unified Modeling Language (UML), Systems Modeling Language (SysML), von der Motor Industry Software Reliability Association (MISRA) entwickelten Richtlinien, Common Object Request Broker Architecture (CORBA), Echtzeit-CORBA, Advanced Architecture Description Language (AADL), Software Communication Architecture, Modelling and Analysis of Real-Time and Embedded Systems (MARTE), ARINC-Standards, veröffentlicht von Aeronautical Radio, Incorporated, Cedar Rapids, Iowa, und Assembly Line Diagnostic Link (ALDL) implementiert werden. Systemaufbau-Richtlinien für

jede dieser Softwarearchitekturspezifikationen können vorgeben, wie entsprechende spezifizierte Modelle **102** für die verschiedenen Softwarearchitekturen gebildet werden, und können ferner vorgeben, wie zumindest ein Teil des Implementierungscodes **104** strukturiert ist, um das spezifizierte Modell und in einigen Fällen die Systemaufbau-Richtlinien zu erfüllen. Ausführungsformen, die sich auf die Codeüberprüfung beziehen, können auf andere Softwarearchitekturspezifikationen als AUTOSAR angewendet werden, die Einschicht- oder Mehrschicht-Softwarearchitekturen mit weniger oder mehr als drei Schichten erfordern können.

[0030] Das Prüfen des Implementierungscodes **104**, der mit einer Softwarearchitektur (wie der in **Fig. 2** dargestellten) verknüpft ist, auf Fehler kann kompliziert sein, sogar ohne das Vorhandensein von Rechen-Threads, die gleichzeitig ausgeführt werden können. Gleichzeitige Threads können Aufgaben und Interrupt Service Routinen (ISRs) entsprechen, die auf einer ECU innerhalb des dynamischen Systems ausgeführt werden. In Ausführungsformen kann eine Aufgabe oder ISR Ressourcen (z. B. Softwareobjekte) beim Ausführen verwenden, obwohl eine Ressource nur von jeweils einem ausführenden Thread zu einer Zeit genommen werden kann, aber unter verschiedenen Aufgaben und ISRs zu unterschiedlichen Zeiten geteilt werden kann. In Ausführungsformen kann eine Ressource Daten und/oder Codestücke enthalten, auf die zugegriffen werden kann und/oder die zur Ausführung des Implementierungscodes verwendet werden können. Aufgaben und/oder ISRs, die gleichzeitig in einem dynamischen System ausgeführt werden, können ein Teil der gesamten Aufgaben und/oder ISRs sein, die in dem dynamischen System ausgeführt werden. Zum Beispiel kann eine erste Gruppe von Aufgaben und/oder ISRs, die gleichzeitig ausgeführt werden, in dem Implementierungscode **104** zugewiesen werden, um auf demselben Prozessor des dynamischen Systems ausgeführt zu werden. In dem Implementierungscode können andere Aufgaben und ISRs anderen Prozessoren zugewiesen sein. Da die anderen Aufgaben und ISRs zur Ausführung auf anderen Prozessoren zugewiesen sind, werden sie nicht gleichzeitig auf demselben Prozessor wie die erste Gruppe von Aufgaben und/oder ISRs ausgeführt. In einigen Fällen können gleichzeitige Aufgaben und ISRs in einer Liste von Aufgaben und ISRs enthalten sein, bei denen es sich um planbare Elemente zur Ausführung auf einem Prozessor handelt.

[0031] Aufgrund des Vorhandenseins von Rechen-Threads, die gleichzeitig in dem Implementierungscode **104** ausgeführt werden können, ist eine Multitasking-Analyse des Codes schwieriger als eine Analyse von Code, der keine gleichzeitigen Threads enthält. Wenn es in einem solchen System gleichzeitige Threads gibt, ist vorher nicht bekannt, wel-

che Threads ausgeführt werden, da der Aufruf eines Threads von dem Zustand des dynamischen Systems abhängen kann, wie weiter unten erläutert wird. Dementsprechend können Informationen über die gleichzeitigen Threads (z. B. ob ein Thread ausgesetzt werden kann, welche Art von Auftreten einen Thread initiiert usw.) aus dem spezifizierten Modell **102** erfasst und dem Codeprüfsystem **100** bereitgestellt werden, um die Genauigkeit von Codeanalyse durch das Codeprüfsystem **100** gemäß den vorliegenden Ausführungsformen zu verbessern. Zum Beispiel kann ein Codeprüfsystem **100** ohne Informationen, die für Thread-Parallelität bzw. -Gleichzeitigkeit relevant sind, einen Thread-Code analysieren und ihn frei von Laufzeitfehlern und Standardcodierungsverletzungen finden bzw. befinden, die für die Softwarearchitektur relevant sind. Wenn der Thread jedoch gleichzeitig mit anderen Threads ausgeführt wird, kann der Thread in einen Wartezustand für Daten am Datenport einer Schnittstelle eintreten, der beispielsweise niemals erscheinen kann oder nie aufgerufen wird, beispielsweise auf Grund eines Nichtauftretens eines richtigen Typs von Aufrufvorgangs. Solche Fehler können durch herkömmliche statische Codeanalyse unentdeckt bleiben. Wenn das Codeprüfsystem **100** jedoch Kenntnis von den Thread-Attributen erhält, kann es das Vorhandensein oder Fehlen von Fehlern in Aufruf- und Ausführungssequenzen für gleichzeitige Threads erkennen.

[0032] **Fig. 3** ist eine vereinfachte Darstellung eines Beispiels einer Methodik **300** zum Überführen eines spezifizierten Modells **102** in einen Implementierungscode **104** auf hoher Ebene. Die zum Umwandeln des spezifizierten Modells in Implementierungscode verwendete Methodik **300** kann zumindest teilweise eine standardisierte Codevorbereitungsmethodik umfassen, die zumindest teilweise durch Systemaufbau-Richtlinien vorgegeben wird. Auf einer hohen Ebene kann ein spezifiziertes Modell **102**, das möglicherweise von einem oder mehreren Systementwicklern geschrieben wurde, einem ersten Implementierungsprozess **320** unterzogen werden, in dem für das veranschaulichte Beispiel Zwischencode **302** erzeugt wird. Der Zwischencode **302** kann eine oder mehrere Dateien umfassen, die zumindest etwas Computercode enthalten (z. B. C++ - oder C-Code, wie in **Fig. 3** dargestellt). Zusätzlicher Computercode, der in dem Zwischencode **302** vorhanden sein kann, umfasst unter anderem, ohne jedoch darauf beschränkt zu sein, Ada, Basic, C#, FORTRAN, Assemblercode und Hardware Description Language (HDL)-Code, wie beispielsweise VHDL, Verilog oder SystemC. In einigen Fällen kann ein automatisiertes Tool (wie DaVinci Developer, erhältlich von Vector Informatik GmbH aus Stuttgart, Deutschland) verwendet werden, um bei der Vorbereitung des Zwischen-codes **302** zu helfen. Der erste Implementierungsprozess **320** kann auch eine direkte Codierung durch einen oder mehrere menschliche Code-Implementie-

rer umfassen. Anschließend kann der Zwischencode **302** einer automatischen Codeproduktion durch einen Codeprozessor **330** (der auch als Vorprozessor bezeichnet werden kann) unterzogen werden, die zu dem Implementierungscode **104** führt. Der Codeprozessor **330** muss kein standardisierter Codeprozessor sein, sondern kann den Implementierungscode **104** in Übereinstimmung mit Beschränkungen, die durch das spezifizierte Modell **102** auferlegt werden, und Beschränkungen, die durch die entsprechenden Systemaufbau-Richtlinien auferlegt werden, generieren oder erzeugen. In einigen Fällen kann die Methodik zum Erzeugen des Implementierungskodes **104** mehrere Schritte des Vorbereitens von Zwischencode- und/oder Zwischendatenstrukturen in einem Aufbauprozess vor Erreichen des Implementierungskodes **104** umfassen. Einige dieser Schritte können das Erstellen von Zwischencode und/oder Datenstrukturen basierend beispielsweise auf Informationen über die Basisdienstschicht **290** umfassen.

[0033] In einigen Fällen kann die Codevorbereitungsmethodik **300** einen oder mehrere Codevorbereitungsschritte umfassen, für die ein oder mehrere Zwischencodes oder Sätze von Zwischencodewweisungen (nicht gezeigt) erzeugt werden, bevor der endgültige Implementierungscode **104** erreicht wird. Jeder Zwischencode kann teilweise auf zusätzlichen Informationen basieren, die für das dynamische System relevant sind, wie zum Beispiel Systemkonfigurationsinformationen und ECU-Konfigurationen. Andere Methoden beinhalten möglicherweise keine Zwischencodenvorbereitung. Stattdessen kann eine automatisierte oder halbautomatische Codevorbereitung, die einen Code-Implementierer unterstützt, ausgeführt werden, um den Implementierungscode **104** direkt aus einem spezifizierten Modell **102** vorzubereiten.

[0034] Der Erfinder hat erkannt und gewürdigt, dass ein technisches Problem auftreten kann, wenn versucht wird, den Implementierungscode **104** zu analysieren, der gemäß solchen Codebereitstellungsmethoden erzeugt wurde. Das technische Problem betrifft die Art und Weise, in der ausführbare Funktionen, die mit Aufgaben, ISRs und Ressourcen verknüpft sind, in dem Implementierungscode **104** implementiert sind. Kurz gesagt, können zumindest einige der ausführbaren Funktionen in einem spezifizierten Modell **102** als eine oder mehrere unterschiedliche Typen von Aufgaben und/oder ISRs (die mindestens eine Ressource enthalten können) definiert sein, die gleichzeitig auf einem Single-Core- oder Multi-Core-Prozessor (oder in einigen Fällen auf mehreren Prozessoren) eines dynamischen Systems laufen sollen. Jeder Aufgabe, ISR und Ressource kann ein Name in dem spezifizierten Modell **102** und zugeordneten Attributen gegeben werden. Attribute einer Aufgabe, ISR und Ressource können die gleichzeitige Ausführung des entsprechenden Rechen-Threads beein-

flussen, wenn sie auf einem Zielgerät eines dynamischen Systems ausgeführt werden. Gemäß einigen Softwarearchitekturspezifikationen **108** kann nur eine Aufgabe oder ISR zu einem gegebenen Zeitpunkt auf einem einzelnen Zielprozessor (beispielsweise einer ECU) ausgeführt werden. In einem komplexen Automobilsystem können beispielsweise Dutzende oder Hunderte von Aufgaben gleichzeitig auf einem Prozessor laufen. Dementsprechend muss ein Codeprüfer **106** zumindest einige Informationen über Aufgaben-, ISR- und Ressourcenattribute kennen, so dass er Segmente des Implementierungskodes **104**, die gleichzeitige Threads enthalten, auf das Vorhandensein oder Nichtvorhandensein von Fehlern in Bezug auf die gleichzeitige Thread-Ausführung überprüfen kann.

[0035] Ein vom Erfinder erkanntes Problem besteht darin, dass der Implementierungscode **104** allein keine ausreichenden Informationen enthält, um die Attribute einer Aufgabe, ISR oder Ressource zu bestimmen, die in Form eines implementierten, unabhängigen Code-Threads oder Codeteils vorliegt, der gleichzeitig mit anderen unabhängigen Code-Threads oder Codeteilen ausgeführt werden kann oder soll. Zusätzlich stimmt in einigen Fällen eine Kennung für den implementierten Code-Thread oder die implementierte Ressource nicht mit einer Kennung für die entsprechende Aufgabe, ISR oder Ressource in dem spezifizierten Modell **102** überein. Dies kann auftreten, wenn das spezifizierte Modell **102** und der Implementierungscode **104** vorbereitet werden, und kann auch auf Grund einer Codevorbereitungsmethodik **300** wie der in **Fig. 3** beschriebenen sein.

[0036] Informationen, die für ein Codeprüfsystem zur Multitasking-Codeanalyse nützlich sein können, umfassen einen Teil oder alle der folgenden Arten von Informationen für Rechen-Threads, die gleichzeitig auf Hardware eines dynamischen Systems ausgeführt werden können: Aufgabenidentitäten bzw. -kennungen, entsprechende Aufgabenattribute, Ereignisse und Alarmer, welche die entsprechenden Aufgaben auslösen können, zugehörige Ressourcen, die von den Aufgaben verwendet werden können, ISR-Identitäten, entsprechende ISR-Attribute, Ressourcenidentitäten bzw. -kennungen und entsprechende Ressourcenattribute. Eine Möglichkeit zum Bereitstellen von Informationen für den Codeprüfer **106** besteht in der manuellen Analyse des spezifizierten Modells **102**, eines durch die Codevorbereitungsmethodik **300** vorbereiteten Zwischencodes und Implementierungskodes **104**, um die relevanten Informationen zu bestimmen und dann das Codeprüfsystem **100** durch Eingabe von relevanten Informationen für jeden relevanten implementierten Code-Thread in dem Implementierungscode **104** beispielsweise über eine graphische Benutzerschnittstelle **180** zu konfigurieren. Bei kleinen Mengen an Implementierungscode

(z. B. weniger als 1000 Zeilen) kann dieser Ansatz zu einer korrekten Konfiguration des Codeprüfsystems **100** führen, ist jedoch anfällig für Fehler und zeitaufwendig.

[0037] Der Erfinder hat erkannt und gewürdigt, dass Fehler selbst für solche kleinen Mengen an Implementierungscode als ein Ergebnis der manuellen Konfiguration auftreten können und typischerweise auftreten, und dass der Prozess eine beträchtliche Zeit in Anspruch nehmen kann. Fehler können sich aus der Art und Weise ergeben, in der das spezifizierte Modell vorbereitet wird, sowie aus einer Codevorbereitungsmethodik **300**, in der zwei oder mehr unterschiedliche Namen für eine gleiche Aufgabe verwendet werden können, so dass Attribute für eine Aufgabe, die in dem spezifizierten Modell **102** spezifiziert ist, nicht korrekt mit dem entsprechenden implementierten Code-Thread in dem Implementierungscode **104** verknüpft werden.

[0038] Als ein Beispiel des Verwendens unterschiedlicher Namen für eine gleiche Aufgabe während einer Codevorbereitungsmethodik **300** (erneut unter Bezugnahme auf **Fig. 3**) kann eine Aufgabe einen ersten Namen oder Kennung **310** aufweisen, der bzw. die an einer ersten Stelle in dem spezifizierten Modell **102** definiert ist. Der erste Ort in dem spezifizierten Modell **102** kann eine erste abgegrenzte Datenstruktur sein (z. B. eine Gruppe von Einträgen in einer Datendatei, ein Container für eine AUTOSAR-ARXML-Datei, eine Mehrzahl von Einträgen zwischen komplementären Tags in einer Markup-Sprachdatei etc.). Eine zweite Stelle in dem gleichen spezifizierten Modell kann eine zweite Kennung **312** für die gleiche Aufgabe verwenden, die sich von der ersten Kennung **310** unterscheidet. In einigen Fällen kann die zweite Kennung **312** in dem Zwischencode **302** verwendet werden. Um die Sache noch komplizierter zu machen kann eine dritte Kennung **340**, die sich von der zweiten Kennung unterscheidet, für den entsprechenden implementierten Code-Thread in dem Implementierungscode **104** als ein Ergebnis der Aktion des Codeprozessors **330** verwendet werden. Das Problem der korrekten Identifizierung von Aufgaben, ISRs, Ressourcen und ihrer Attribute verschärft sich, wenn relevante Informationen für diese Elemente über mehrere disjunkte Orte in einer oder mehreren Datenstrukturen (z. B. einer oder mehreren Dateien) des spezifizierten Modells **102** verteilt sind, wie dies für AUTOSAR der Fall ist. Das Problem der Verwendung unterschiedlicher Namen für dieselbe Aufgabe oder denselben Prozess ist jedoch nicht auf AUTOSAR beschränkt. Die Verwendung unterschiedlicher Namen für Aufgaben und Prozesse kann in anderen spezifizierten Modellen auftreten, die mit anderen Softwarearchitekturen implementiert werden können. In anderen spezifizierten Modellen können Prozesse (aufrufbare Sequenzen von Anweisungen, wie beispielsweise Funktionen und Subrou-

tinien) anstelle von oder zusätzlich zu Aufgaben verwendet werden. Das Problem des korrekten Zuordnens von Aufgaben-, ISR- und Ressourcenattributen zu ihren entsprechenden implementierten Code-Threads kann unüberwindbar werden, wenn es Hunderte oder Tausende von Dateien mit Millionen von Einträgen für das spezifizierte Modell **102** und Millionen von Zeilen entsprechenden Implementierungscodes **104** in Tausenden von Dateien gibt.

[0039] Um bei der Erläuterung der vorliegenden Ausführungsformen zu helfen, werden nun weitere Details eines spezifizierten Modells **102**, Aufgaben, ISRs, Ressourcen und Implementierungscode **104** bereitgestellt. In Ausführungsformen kann ein spezifiziertes Modell **102** eine oder mehrere Datenstrukturen (z. B. die in **Fig. 3** gezeigten Dateien **spec1**, **spec2**, ... **specN**) umfassen, die unter anderem Funktionalität von einem oder mehreren Implementierungscode-Segmenten definieren, die aus dem spezifizierten Modell entwickelt werden können. Das spezifizierte Modell **102** kann von einem Systemingenieur vorbereitet werden, der allein oder in Zusammenarbeit mit anderen Systemsteuer- bzw. -regelcode für ein dynamisches System entwickelt. Ein spezifiziertes Modell **102** kann in irgendeiner geeigneten Weise vorbereitet werden, zum Beispiel durch direkte Codierung oder durch Ausfüllen von Vorlagen, die das Schreiben eines spezifizierten Modells leiten. Als ein Beispiel stehen Vorlagen für AUTOSAR zur Verfügung, die ausgefüllt werden können, um eine oder mehrere AUTOSAR Extensive Markup-Language-Dateien (ARXML-Dateien) für ein spezifiziertes Modell **102** zu erzeugen. Ein spezifiziertes Modell **102** kann so vorbereitet werden, dass es Systemaufbau-Richtlinien erfüllt, die Beschränkungen dahingehend auferlegen können, wie das spezifizierte Modell **102** gebildet wird (z. B. ein Schema für das spezifizierte Modell definieren).

[0040] Ein spezifiziertes Modell **102** kann auch Informationen über ausführbare Funktionen, Schnittstellen, Ports, ISRs, Aufgaben, Ressourcen und deren Attribute enthalten, die ein oder mehrere Implementierungscodeabschnitte, die aus dem spezifizierten Modell erzeugt werden, zum Ausführen von Funktionalität und Interagieren mit anderen Codesegmenten verwenden, die in dem dynamischen System ausgeführt werden. Mit einer Aufgabe verknüpfte Attribute können eine Art von Aufgabe und/oder eine Art von Vorkommnissen identifizieren, die eine Aufgabe aktivieren. Einige Softwarearchitekturspezifikationen, wie z. B. AUTOSAR und OSEK, sehen beispielsweise zwei Typen von Aufgaben vor, denen jeweils ein Prioritätswert zugewiesen werden kann: eine grundlegende Aufgabe und eine erweiterte Aufgabe. Eine grundlegende Aufgabe wird, wenn sie einmal aktiviert ist, normalerweise bis zur Beendigung ausgeführt, obwohl sie z. B. durch eine ISR oder Aufgabe mit höherer Priorität ausgesetzt werden kann.

Eine grundlegende Aufgabe kann, sobald sie aktiviert ist, nicht in einen Wartezustand eintreten, in dem sie darauf wartet, dass ein Ereignis ihre Ausführung beendet. Eine erweiterte Aufgabe kann, sobald sie aktiviert ist, in einen Wartezustand eintreten (z. B. Warten auf den Empfang von Daten) sowie durch ISRs und Aufgaben mit höherer Priorität ausgesetzt werden. Aufgaben und ISRs können Prioritätswerte in dem spezifizierten Modell **102** zugewiesen werden, die eine Ausführungsreihenfolge von zwei konkurrierenden Aufgaben oder ISRs bestimmen. In Ausführungsformen können Informationen über Aufgaben-, Ressourcen- und ISR-Kennungen und zugehörige Aufgaben-, Ressourcen- und ISR-Attribute zumindest teilweise aus dem spezifizierten Modell **102** bestimmt werden.

[0041] In Ausführungsformen können verschiedene Typen von ISRs existieren, die in einem dynamischen System ausgeführt werden können. Ein erster Typ von ISR kann bewirken, dass eine Ausführungsaufgabe auf einem Prozessor ausgesetzt wird, während die ISR ihre Ausführung auf dem Prozessor abschließt. Nach Abschluss des ersten ISR-Typs kann die ausführende Aufgabe dort fortgesetzt werden, wo sie unterbrochen wurde. Ein zweiter Typ von ISR kann auch bewirken, dass eine ausführende Aufgabe auf einem Prozessor ausgesetzt wird. Der zweite ISR-Typ kann jedoch einen Systemaufruf ausgeben, so dass, wenn die ISR die Ausführung beendet, eine Vermittlungsaufgabe auftreten kann, wenn während des Systemaufrufs eine Aufgabe mit höherer Priorität als die aktuell ausgesetzte Aufgabe aktiv wird. In Ausführungsformen können ISRs hardware-initiiert sein.

[0042] Elemente, die eine Aktivierung einer Aufgabe bewirken können, können Alarme und Ereignisse umfassen. Ein Alarm kann ein zyklischer Alarm, der mit einem wiederkehrenden Vorfall in einem dynamischen System verknüpft ist, oder ein einzelner Alarm sein, der mit einem einzelnen Vorfall in einem dynamischen System verknüpft ist. In einigen Fällen können Alarme mit elektrischen oder mechanischen Vorfällen in einem elektromechanischen System verknüpft sein (z. B. ausgelöst durch Hardwarevorfälle). Zum Beispiel kann ein sich wiederholender Vorfall für einen zyklischen Alarm eine Anzahl von Motorumdrehungen in einem Fahrzeug sein (z. B. alle 5000 Umdrehungen), und der zyklische Alarm kann nach jeweils 5000 Umdrehungen ausgelöst werden. Ein einzelner Vorfall für einen einzelnen Alarm kann beispielsweise ein schlüsselinitiiertes Start des Motors sein, und der einzelne Alarm kann nach dem durch den Schlüssel initiierten Start ausgelöst werden. In einigen Fällen können Alarme durch Softwarevorfälle ausgelöst werden. Zum Beispiel kann ein Alarm durch einen Zähler ausgelöst werden, der zählt, wie oft eine bestimmte Aufgabe ausgeführt wurde.

[0043] Die Aktivierung einer Aufgabe basierend auf einem Ereignis kann eine Zustandsänderung an einem Datenport umfassen, der mit der Aufgabe verknüpft ist, zum Beispiel das Auftreten von Daten an einem Port einer Schnittstelle. In einigen Fällen können die Daten auf Grund eines neuen Lesevorgangs von einem Sensor an einem Port erscheinen, so dass das Ereignis durch einen Hardwarevorfall ausgelöst wird. Zum Beispiel können Daten, die an einem Port erscheinen, der einem Sensor zugeordnet ist, die Aktivierung einer oder mehrerer Aufgaben bewirken, welche die Daten empfangen und die Daten verarbeiten. In einigen Fällen können die Daten als Ergebnis einer internen Berechnung durch das dynamische System an einem Port erscheinen, so dass das Ereignis durch Software ausgelöst wird. Es kann verschiedene Typen von Ereignissen geben, welche die Aktivierung von Aufgaben bewirken, wie zum Beispiel, ohne jedoch darauf beschränkt zu sein, Timing-Ereignisse, Fehlerereignisse, Beendigung des Sendens von Daten, Serveraufruf und Moduswechselergebnisse.

[0044] Wenn der Implementierungscode **104** aus einem spezifizierten Modell **102** gemäß einer Codevorbereitungsmethodik **300** erstellt wird, ist es wünschenswert, zumindest für Zwecke der genauen Multitasking-Codeanalyse zu wissen, welche Nebenläufigkeits- bzw. Gleichzeitigkeitselemente (z. B. Aufgaben, ISRs und Ressourcen) in dem Implementierungscode **104** vorhanden sind, und auch Attributinformationen über die vorhandenen Gleichzeitigkeitselemente zu kennen. Korrekte Informationen über die Gleichzeitigkeitselemente können von dem Codeprüfer **106** verwendet werden, um zum Beispiel die Planung und Multitasking-Ausführung von entsprechenden implementierten Code-Threads zu bewerten. Mit den korrekten Informationen kann eine statische Codeanalyse durch den Codeprüfer **106** durchgeführt werden, um das Vorhandensein oder Fehlen von Fehlern zu bestimmen, die sich auf die Multitasking-Ausführung von Aufgaben und ISRs in einem dynamischen System beziehen. Wie aus der obigen Beschreibung und der weiteren detaillierten Beschreibung unten ersichtlich ist, ist das korrekte Identifizieren von Aufgaben und ihren zugehörigen Informationen sowohl im spezifizierten Modell als auch im Implementierungscode und das korrekte Konfigurieren eines Codeprüfsystems **100** mit den relevanten Informationen über Aufgaben und ISRs, die dazu bestimmt sind, gleichzeitig zu laufen, ein technisch herausforderndes Problem.

[0045] In einigen Fällen kann der Implementierungscode **104** von einem Codeprüfer **106** analysiert werden, um zu bestimmen, ob irgendwelche Gleichzeitigkeitselemente, die in einem spezifizierten Modell **102** identifiziert werden, während der Ausführung gepoolt werden. Gepoolte Gleichzeitigkeitselemente können auf eine Ausführung nach Beendigung nach-

einander beschränkt werden, nachdem jedes vorherige Gleichzeitigkeitselement beendet wurde. In einem solchen Fall werden gespoolte Gleichzeitigkeitselemente nicht gleichzeitig ausgeführt. Solche Aufgaben, ISRs und Routinen, die nicht gleichzeitig ausgeführt werden, können markiert oder auf andere Weise als nicht gleichzeitige Elemente bezeichnet werden. Nicht gleichzeitige Elemente können dem Codeprüfer **106** vor Durchführung einer statischen Codeanalyse angezeigt werden, um das Vorhandensein oder Fehlen von Fehlern in dem Implementierungscode **104** zu bestimmen. Die Kenntnis nicht gleichzeitiger Aufgaben, ISRs und Routinen kann die Geschwindigkeit und Genauigkeit des Codeprüfers **106** verbessern.

[0046] Fig. 4A bis Fig. 4C zeigen Abschnitte **401**, **402**, **403** eines AUTOSAR-spezifizierten Modells **102**, die relevante Einträge enthalten, aus denen eine Aufgabe, ein Aufgabentyp und ein Ereignis, das mit der Aufgabe verknüpft ist, identifiziert werden können. Die relevanten Einträge des spezifizierten Modells zum Identifizieren einer Aufgabe und ihrer Attribute können in disjunkten Datenstrukturen (z. B. verschiedenen Dateien) oder an disjunkten Orten in einer Datenstruktur des spezifizierten Modells **102** erscheinen. In Ausführungsformen kann ein Datenanalysetool (dargestellt als Element **1110** in Fig. 11 unten) Datenstrukturen eines spezifizierten Modells **102** analysieren, z. B. parsen, um Einträge zu identifizieren, die für die Identifikation von Aufgaben, ISRs, Ressourcen und ihren zugeordneten Attributen relevant sind. Das Datenanalysetool **1110** kann als Programmierbefehle implementiert werden, die auf Hardware ausgeführt werden, die Logikgatter und Register umfasst, die als sequentielle Logikschaltungen angeordnet sind. Das Datenanalysetool kann programmiert werden mit, oder Informationen empfangen von, einer Softwarearchitekturspezifikation **108** oder Stilrichtlinien **107** über eine Methodik (z. B. ein Schema), die verwendet wird, um ein spezifiziertes Modell **102** vorzubereiten, so dass es bestimmen kann, welche Einträge in dem spezifizierten Modell für Aufgaben, ISRs, Ressourcen und ihre verknüpften Attribute relevant sind.

[0047] Mit Bezug auf Fig. 4A kann ein Datenanalysator **1110** einen ersten aufgabendefinierenden Eintrag **410a** in einem spezifizierten Modell **102** basierend auf einer vorbestimmten Kennung (z. B. einer oder mehreren Zeichensequenzen) identifizieren, die mit dem Eintrag verknüpft ist. In diesem Beispiel umfasst die vorbestimmte Kennung mindestens ein Extensible Markup Language (XML)-Tag, das den Text „DEFINITION-REF“ zusammen mit einem Teil von eingegebenem Text „Os/OsTask“ enthält, der sich neben dem Tag befindet. Ein anderer Teil des eingegebenen Textes kann eine Aufgabendefinitionskennung **412**, „MICROSAR/RH850_F1H“ in diesem Beispiel enthalten. Verbunden mit der Aufgabendefinition kann eine erste Kennung (auch als „Kurzname“

bezeichnet) der Aufgabe in einem Aufgabenbenennungseintrag **420a** zugeordnet sein, die beispielsweise basierend auf ihrem HTML-Tag von einem Datenanalysator erkannt werden kann. In diesem Beispiel ist die erste Kennung „T1_Speed“, obwohl eine andere Kennung für eine Aufgabe in einem Aufgabenbenennungseintrag **420a** verwendet werden kann. In Ausführungsformen kann der Aufgabenbenennungseintrag **420a** innerhalb eines Abschnitts des spezifizierten Modells **102** erscheinen, der den aufgabendefinierenden Eintrag **410a** enthält (z. B. zwischen den gleichen komplementären HTML-Tags, zwischen einschließenden Klammern etc.). In diesem Beispiel erscheinen die zwei Einträge zwischen komplementären <ECUC-CONTAINER-VALUE> HTML-Tags, und der entsprechende Abschnitt des spezifizierten Modells kann als „Container“ bezeichnet werden.

[0048] Gemäß einigen Ausführungsformen kann ein gleicher Abschnitt (z. B. Container) des spezifizierten Modells, der einen aufgabendefinierenden Eintrag **410a** enthält, eine Kennung für eine Hardwareressource enthalten, auf der die Aufgabe ausgeführt werden kann. Gemäß dem Beispiel von Fig. 4A kann die Aufgabendefinitionskennung **412** eine Kennung für einen Mikrocontroller (zum Beispiel RH850_F1H) enthalten, auf dem die Aufgabe ausgeführt werden soll. In Ausführungsformen kann ein Datenanalysator **1110** bestimmen, welche Aufgaben potentiell multitasking sind, wenn ihre entsprechenden implementierten Code-Threads ausgeführt werden, und zwar basierend auf Hardware-Ressourcen-Identifizierungsinformationen, die mit jeder Aufgabe verknüpft sind.

[0049] Unter Verwendung der Aufgabendefinitionskennung **412** kann der Datenanalysator das spezifizierte Modell **102** nach einem anderen bzw. weiteren Codeabschnitt durchsuchen (in diesem Beispiel einen anderen bzw. weiteren Abschnitt, der von ECUC-CONTAINER-HTML-Tags umschlossen ist), der die gleiche Aufgabendefinitionskennung **412** enthält. Ein weiterer Abschnitt **402** des spezifizierten Modells ist in Fig. 4B gezeigt. Gemäß einigen Implementierungen kann die Aufgabendefinitionskennung **412** an anderer Stelle in dem spezifizierten Modell **102** verwendet werden, um ein Ereignis oder einen Alarm zu definieren, das bzw. der mit der Aufgabe zum Aktivieren der Aufgabe verknüpft werden soll. Für das in Fig. 4B gezeigte Beispiel wird die Aufgabendefinitionskennung **412** in einem ereignisdefinierenden Eintrag **410b** verwendet, um ein Ereignis zu definieren, das die Aufgabe aktiviert. Der ereignisdefinierende Eintrag **410b** kann durch den Datenanalysator beispielsweise durch einen oder beide HTML-Tags (z. B. mit „DEFINITION-REF“) und einen entsprechenden Abschnitt von umschlossenem, eingegebenem Text (z. B. „Os/OsEvent“) identifiziert werden. Ein Ereignisbenennungseintrag **430a** kann in demselben Ab-

schnitt **402** des spezifizierten Modells **102** erscheinen und eine erste Kennung **432** für das Ereignis identifizieren, in diesem Beispiel „RteEvCyc2_T1_speed_100ms“.

[0050] Sobald eine erste Ereigniskennung **432** aus dem Ereignisbenennungseintrag **430a** bestimmt worden ist, kann der Datenanalysator das spezifizierte Modell **102** nach einem weiteren Auftreten derselben Ereigniskennung durchsuchen. Die erste Ereigniskennung **432** kann in einem Abschnitt **403** des spezifizierten Modells **102** auftreten, in dem ein Ereignis mit der Aufgabe verknüpft ist, wie in **Fig. 4C** dargestellt. Ein Abschnitt **403** des spezifizierten Modells, der das Ereignis-zu-Aufgabe-Mapping enthält, kann durch den Datenanalysator als ein Abschnitt identifiziert werden, der einen Ereignis-zu-Aufgabe-Mapping-Definitionseintrag **440** enthält und beispielsweise durch ECUC-CONTAINER HTML-Tags umschlossen ist. Ein Ereignis-zu-Aufgabe-Mapping-Definitionseintrag **440** kann zum Beispiel durch einen oder beide HTML-Tags (z. B. mit „DEFINITION-REF“) und einen entsprechenden Abschnitt des umschlossenen, eingegebenen Textes (z. B. „RteBswEventToTaskMapping“) identifiziert werden. Der Abschnitt **403** kann auch einen Ereignisbenennungseintrag **450** enthalten, der einen Kurznamen für das Ereignis bereitstellt, der auf die Aufgabe zu mappen ist. Innerhalb dieses Abschnitts des spezifizierten Modells **102** kann eine zweite Kennung **422** (auch als „voll qualifizierter Name“ bezeichnet) für das Ereignis in einem ersten referenzierenden Eintrag **430b** bereitgestellt werden, der die erste Ereigniskennung **432** enthält. Derselbe Abschnitt **403** des spezifizierten Modells **102** kann auch eine verknüpfte zweite Kennung (auch als „voll qualifizierter Name“ bezeichnet) für die Aufgabe enthalten, die zuerst in dem ersten Abschnitt **401** des spezifizierten Modells definiert wurde, das in **Fig. 4A** dargestellt ist. Die zugehörige zweite Kennung kann in einem zweiten referenzierenden Eintrag **420b** erscheinen. Zum Beispiel ist in diesem Fall die zweite Kennung **422** für die Aufgabe als „/ActiveEcuC/0s/T1_speed_100ms“ gezeigt, was sich von dem für die Aufgabe bereitgestellten Kurznamen „T1_speed“ unterscheidet. Obwohl in diesem Beispiel die zweite Kennung **422** für die Aufgabe den Kurznamen enthält, kann in der Praxis die zweite Kennung **422** sich vollständig von dem Kurznamen der Aufgabe unterscheiden, der in dem Aufgabenbenennungseintrag **420a** bereitgestellt wird. Die zweite Kennung **422** für die Aufgabe kann für die nachfolgende Codevorbereitungsmethodik **300** relevant sein, wie es weiter unten beschrieben wird.

[0051] Ein Datenanalysator kann aus dem Ereignis-zu-Aufgabe-Mapping-Abschnitt **403** des spezifizierten Modells **102** zusammen mit Informationen, die von anderen Abschnitten **401**, **402** des spezifizierten Modells erhalten werden, das in **Fig. 4A** und **Fig. 4B** gezeigt ist, mehrere Attribute einer Aufga-

be bestimmen, die anfänglich mit einem aufgaben-definierenden Eintrag **410a** definiert ist. Der Datenanalysator **1110** kann bestimmen, dass die Aufgabe durch ein Ereignis aktiviert wird, und kann den Aufgabentyp identifizieren. In Ausführungsformen kann eine Aufgabe, die durch ein Ereignis aktiviert wird, in einen Wartezustand für das Ereignis eintreten und ist ein erweiterter Aufgabentyp. Der Datenanalysator **1110** kann auch eine Kennung für das Ereignis bestimmen, das mit der Aufgabe zum Aktivieren der Aufgabe verknüpft werden soll. Der Datenanalysator **1110** kann auch eine zweite Kennung (vollständig qualifizierten Namen) für die Aufgabe bestimmen, die für die nachfolgende Codevorbereitungsmethodik relevant ist. Dementsprechend kann der Datenanalysator Informationen über die Aufgabe und ihre Attribute, die für die Multitasking-Codeanalyse relevant sind, im Speicher speichern und/oder an ein Codeprüfsystem **100** senden. Die Informationen können von dem Codeprüfsystem **100** empfangen oder abgerufen werden und dazu verwendet werden, seinen Codeprüfer **106** automatisch zu konfigurieren, um den entsprechenden implementierten Code-Thread für die bestimmte Aufgabe zu handhaben. Die Konfiguration des Codeprüfers **106** kann mindestens das Identifizieren eines Aufgabentyps (z. B. grundlegend oder erweitert) und des Aufgabennamens für jede Aufgabe für den Codeprüfer **106** umfassen.

[0052] In Ausführungsformen verwendet ein Codeprüfsystem **100** Informationen (z. B. eine Kennung für einen entsprechenden Rechen-Thread) über eine Aufgabe, um den Implementierungscode **104** nach dem entsprechenden Rechen-Thread zu durchsuchen und den Thread zu analysieren. Ohne die Kennung könnte der Rechen-Thread nicht analysiert werden und der Teil des Implementierungscode, der Fehler enthalten könnte, würde nicht überprüft werden. In einigen Fällen können Attributinformationen von dem Codeprüfsystem **100** verwendet werden, um die Analyse des Verhaltens eines Rechen-Threads zu verbessern. Zum Beispiel kann eine Aufgabe, die von einem Ereignis aufgerufen wird oder darauf wartet, dem Codeprüfsystem **100** zusammen mit Informationen über das Ereignis basierend auf einer Analyse des spezifizierten Modells **102** identifiziert werden. Das Codeprüfsystem kann dann den Implementierungscode **104** analysieren, um zu bestimmen, ob das Ereignis jemals auftritt und ob die Aufgabe jemals aufgerufen wird oder einen Wartezustand verlässt.

[0053] Ein Prozess zum Identifizieren von Aufgaben und deren Aktivierungsalarmen kann sich von dem Prozess zum Identifizieren von Aufgaben und ihren Aktivierungsereignissen unterscheiden. **Fig. 5A** zeigt einen Abschnitt **501** eines spezifizierten Modells **102**, das einen Container enthält, beginnend mit einem Containereintrag **510a** (z. B. einem erkennbaren Tag <ECUC-CONTAINER-VALUE>, in dem ein Alarm definiert ist. In Ausführungsformen kann das Identifizie-

ren einer Aufgabe und ihres Aktivierungsalarms damit beginnen, eine Definition für einen Alarm zu bestimmen und die Definition einer durch den Alarm aktivierten Aufgabe zuzuordnen.

[0054] Als ein Beispiel kann eine Alarmdefinition einen erkennbaren alarmdefinierenden Eintrag **510b** umfassen, der einen Schlüsselbegriff **512** (z. B. „Os/OsAlarm“) zwischen Definitions-Tags (z. B. <DEFINITION-REF ...>) enthält. Bei dem gleichen Container, der durch den Containereintrag **510a** gebunden ist, kann es Subcontainer **520a**, **530a** und zusätzliche Definitionen **520b**, **530b**, **530c** geben, die den Alarm betreffend. Ein erster Subcontainer **520a** kann eine Definition **520b** enthalten, die mit einer Alarmaktion verknüpft ist, und kann durch einen Schlüsselbegriff **522** (z. B. „Os/OsAlarm/OsAlarmAction“) zwischen Definitions-Tags erkennbar sein.

[0055] Der erste Subcontainer **520a** kann einen zweiten Subcontainer **530a** enthalten, in dem eine erste Definition **530b** vorhanden ist, die mit der Aufgabenaktivierung verknüpft ist. Die Definition **530b**, die mit der Aufgabenaktivierung verknüpft ist, kann durch einen Schlüsselbegriff **532** (z. B. „Os/OsAlarm/OsAlarmActivateTask“) erkannt werden, der sich zwischen Definitions-Tags befindet. Innerhalb desselben Subcontainers kann es einen identifizierenden Name **534** für eine Aufgabe geben, die durch den Alarm aktiviert wird. Der identifizierende Name **534** kann als ein Eintrag zwischen oder verknüpft mit identifizierenden Tags (z. B. <VALUE-REF ...>-Tags) erkennbar sein. In einigen Fällen enthält der identifizierende Name **534** ein Präfix (z. B. / rtk_cpu/OsRTA /), gefolgt von einem Aufgabennamen (z. B. „OsTsk_Check_Temp“). Der Aufgabename kann oder kann nicht derselbe Name sein, der verwendet wird, wenn zuerst die durch den Alarm aktivierte Aufgabe definiert wird.

[0056] Sobald der identifizierende Name **534** gefunden ist, kann der Datenanalysator **1110** das spezifizierte Modell **102** nach einer zweiten Instanz des Präfixes des identifizierenden Namens **534** durchsuchen. Die zweite Instanz kann in einem anderen Abschnitt **502** des spezifizierten Modells auftreten, was ist in **Fig. 5B** dargestellt ist. Der zweite Abschnitt **502** kann das Präfix **534a** enthalten, gefolgt von einem Kurznamen **552**. Der Kurzname **552** kann von dem Datenanalysator **1110** verwendet werden, um die ursprüngliche Definition der Aufgabe zu identifizieren, die durch den Alarm aktiviert wird, was in **Fig. 5C** dargestellt ist. Eine ursprüngliche Definition für die Aufgabe kann einen aufgabendefinierten Eintrag **540b** enthalten, der in einem Containereintrag **540a** vorkommt. Der aufgabendefinierende Eintrag kann einen Schlüsselbegriff (z. B. „/Os/OsTask“) enthalten, der von dem Datenanalysator **1110** erkannt wird. Ein nahe gelegener Kurzname **550** (z. B. innerhalb desselben Containers) kann den ursprünglichen Kurzna-

men **552** bereitstellen, der für die Aufgabe definiert ist. Dementsprechend können Informationen über ein Format oder Schema, das verwendet wird, um das spezifizierte Modell **102** zu bilden, und Informationen über Schlüsselbegriffe, die in dem spezifizierten Modell verwendet werden, verwendet werden, um Aufgaben und ihre aktivierenden Alarme oder Ereignisse zu identifizieren.

[0057] Interrupt Service Routinen (ISRs) und Softwareressourcen können auch aus Einträgen in einem spezifizierten Modell **102** identifiziert werden. **Fig. 6** zeigt einen Modellabschnitt **601** eines spezifizierten Modells, in dem eine ISR und zumindest einige ihrer Attribute definiert sind. Eine ISR kann auf einem Prozessor ausgeführt werden und auf einen Interrupt reagieren, der durch einen Hardware- oder Softwarevorfall verursacht werden kann, der sofortige Aufmerksamkeit erfordert (z. B. ein Betriebssystem einer ECU in einem Kraftfahrzeugsystem gibt ein Interrupt bei Erkennen einer Durch-Null-Teilen-Bedingung aus). In Ausführungsformen kann eine ISR mit einem ISR-definierenden Eintrag **610a** definiert sein, der zumindest einen erkennbaren Schlüsselbegriff **630** enthält (z. B. einen „Os/OsIsr“-Texteintrag zwischen HTML-Definitions-Tags <DEFINITION-REF ...>). Andere Schlüsselbegriffe und Tags können in anderen Ausführungsformen verwendet werden: In einigen Fällen kann der ISR ein Kurznameneintrag **620a** zugeordnet sein. In Ausführungsformen kann eine ISR in einem Container definiert sein, der durch Container-Tags **605a**, **605b** angegeben wird. Attribute einer ISR können innerhalb des Containers definiert sein. In dem dargestellten Beispiel umfassen definierte Attribute eine Kategorie von ISR (Kategorie **1**), einen Prioritätswert (**4**) für die ISR, eine Adresse und eine Stapelzuweisung für die ISR.

[0058] Die Identität eines entsprechenden Rechen-Threads in dem Implementierungscode für eine ISR kann in derselben Weise bestimmt werden, wie es unten in Verbindung mit **Fig. 9A** bis **Fig. 9E** zum Bestimmen der Identität von Rechen-Threads für Aufgaben beschrieben ist, die in dem spezifizierten Modell **102** definiert sind. In einigen Fällen werden alle Rechen-Threads in dem Implementierungscode **104**, die als mit ISRs verknüpft identifiziert werden, einem Codeprüfsystem während der Konfiguration als mit Interrupts verknüpft identifiziert. Wenn beispielsweise die Software Polyspace Code checker™ (modifiziert mit Code, wie in der europäischen Patentanmeldung Nr. 17290155.5, eingereicht am 30. November 2017 mit dem Titel „Systems and Methods for Evaluating Compliance of Implementation Code with a Software Architecture Specification“) zur Codeanalyse verwendet wird, kann die Identifizierung einer ISR verwendet werden, um die automatische Konfiguration einer „Interrupts“-Option für den entsprechenden Implementierungscode-Thread zur Multitasking-Analyse zu bewirken.

[0059] Fig. 7 zeigt einen Modellabschnitt **701** eines spezifizierten Modells **102**, in dem eine Ressource definiert ist. In Ausführungsformen kann eine Ressource ein Softwareobjekt sein, das von Aufgaben und ISRs verwendet und gemeinsam genutzt wird, obwohl eine Ressource nur zu einer bestimmten Zeit von einer Aufgabe oder ISR belegt werden kann. In einigen Fällen kann eine Ressource einen Teil des Codes, der einen Rechen-Thread enthält, enthalten oder damit verknüpft sein, so dass der Teil des Codes geschützt ist, wenn er innerhalb einer Aufgabe oder ISR ausgeführt wird (z. B. darf keine andere Aufgabe oder ISR auf den ausführenden Code zur gleichen Zeit zugreifen). Eine Ressource kann von einer Aufgabe oder einer ISR freigegeben werden, um die Ressource anderen Aufgaben und ISRs zur Verfügung zu stellen. In Ausführungsformen kann eine Ressource mit einem ressourcendefinierenden Eintrag **710a** definiert werden, der zumindest einen erkennbaren Schlüsselbegriff **730** enthält (z. B. einen „Os/OsResource“-Texteintrag zwischen HTML-Definitions-Tags <DEFINITION-REF ...>). Andere Schlüsselbegriffe und Tags können in anderen Ausführungsformen verwendet werden. In einigen Fällen kann der Ressource ein Kurznameneintrag **720a** zugeordnet sein. In Ausführungsformen kann eine Ressource innerhalb eines Containers definiert sein, der durch Container-Tags **705a**, **705b** angegeben ist. Die Identität einer entsprechenden implementierten Ressource in dem Implementierungscode **104** für eine Ressource kann in einer Weise bestimmt werden, wie sie unten in Verbindung mit Fig. 10A und Fig. 10B beschrieben ist.

[0060] Nach dem Identifizieren einer Aufgabe und ihrer verknüpften Attribute (wie etwa ob sie durch ein Ereignis oder einen Alarm aktiviert wird und Parameter oder Attribute des Aktivierungsereignisses oder des Alarms) kann der Datenanalysator **1110** eine Bezeichnung für die Aufgabe bestimmen, die bereitgestellt werden kann zum Konfigurieren des Codeprüfsystems zur Multitasking-Analyse des Implementierungscode **104**. Eine Bezeichnung für eine Aufgabe kann das Identifizieren eines Aufgabentyps (z. B. Kategorisieren der Aufgabe oder Auswählen einer Kategorisierungskennung für die Aufgabe) und/oder Bereitstellen oder Speichern von Informationen über die Aufgabe umfassen (z. B. Informationen zu Aufgabenattributen, z. B. Informationen zu einem Vorkommen, das eine Aufgabe aktiviert). Fig. 8 zeigt einen exemplarischen Entscheidungsalgorithmus **800**, den ein Datenanalysator **1110** oder ein anderes Analysetool implementieren kann, um eine Bezeichnung für aus einem spezifizierten Modell **102** identifizierte Aufgaben zu bestimmen. In einigen Fällen kann der Algorithmus **800** ausgeführt werden, wenn bzw. da beispielsweise jede Aufgabe durch den Datenanalysator **1110** identifiziert wird. In anderen Fällen kann der Entscheidungsalgorithmus **800** durch ein separates Aufgabenanalysetool ausgeführt werden, das mit rele-

vanten Informationen von dem Datenanalysator versehen ist. Das separate Aufgabenanalysetool kann Teil des Codeprüfsystems **100** oder ein eigenständiges Tool sein. Ein Aufgabenanalysetool kann als Programmieranweisungen implementiert werden, die geschrieben werden, um einen Algorithmus auszuführen, wie in Fig. 8 dargestellt, ausgeführt auf Hardware, die Logikgatter und Register umfasst, die als sequentielle Logikschaltungen angeordnet sind.

[0061] In Ausführungsformen kann ein Aufgabenbezeichnungsalgorithmus **800** beginnen, indem er eine Aufgabe von einem spezifizierten Modell identifiziert (Schritt **805**) (z. B. Identifizieren einer Definition einer Aufgabe von einem spezifizierten Modell **102** oder Empfangen einer Aufgabenkennung von einer Mehrzahl von Aufgaben, die von dem Datenanalysator identifiziert wurden). Der Prozess kann das Bestimmen (Schritt **820**) (z. B. unter Verwendung von Informationen und Verfahren, die oben in Verbindung mit Fig. 4A bis Fig. 5C beschrieben sind) umfassen, ob die Aufgabe durch einen Alarm oder ein Ereignis aktiviert wird. Wenn bestimmt wird (Schritt **820**), dass die Aufgabe durch ein Ereignis aktiviert wird, dann kann die Aufgabe als eine erste Art von Aufgabe (z. B. eine Aufgabe, die in einen Wartezustand wie einen erweiterten Aufgabentyp eintritt, zusätzlich oder alternativ eine Aufgabe, für eine Start- und Endzeit nicht a priori bekannt sind) bezeichnet werden (Schritt **824**). Wenn die Software Polyspace Code checker™ (modifiziert mit Code, wie in der europäischen Patentanmeldung Nr. 17290155.5, eingereicht am 30. November 2017 mit dem Titel „Systems and Methods for Evaluating Compliance of Implementation Code with a Software Architecture Specification“ beschrieben) für die Codeanalyse verwendet wird, kann eine erste Typenbezeichnung für eine Aufgabe verwendet werden, um eine automatische Konfiguration einer „Eintragspunkte“-Option für den entsprechenden implementierten Code-Thread für die Multitasking-Analyse zu bewirken. Der Prozess **800** kann dann bestimmen (Schritt **840**), ob verbleibende Aufgaben vorhanden sind (oder Abschnitte des spezifizierten Modells **102**, um nach Aufgaben zu suchen). Wenn noch Aufgaben vorhanden sind, kann der Prozess zu Schritt **805** zurückkehren. Wenn keine verbleibenden Aufgaben vorhanden sind, kann der Prozess enden.

[0062] Wenn bestimmt wird (Schritt **820**), dass die Aufgabe nicht durch ein Ereignis aktiviert wird, kann ein Schritt des Bestimmens (Schritt **830**), ob die Aufgabe durch einen Alarm aktiviert wird, ausgeführt werden. Wenn die Aufgabe nicht durch einen Alarm aktiviert wird, kann der Prozess die Aufgabe als ersten Typ von Aufgabe festlegen (Schritt **824**) und mit Schritt **840** fortfahren.

[0063] Wenn bestimmt wird (Schritt **830**), dass die Aufgabe durch einen Alarm aktiviert wird, kann ein Schritt zum Bestimmen (Schritt **842**), ob der Alarm

periodisch ist, ausgeführt werden. Wenn festgestellt wird (Schritt **842**), dass der Alarm nicht periodisch ist, kann der Prozess die Aufgabe als ersten Typ von Aufgabe festlegen (Schritt **824**) und mit Schritt **840** fortfahren. Wenn festgestellt wird (Schritt **842**), dass der Alarm periodisch ist kann der Prozess die Aufgabe als einen zweiten Typ von Aufgabe bezeichnen (Schritt **844**) (z. B. eine Aufgabe, die nicht in einen Wartezustand eintreten kann, wie eine Basisaufgabe, zusätzlich oder alternativ eine Aufgabe, für die eine Start- und Endzeit a priori bestimmt werden kann). In Bezug auf die oben erwähnte modifizierte Software Polyspace Code checker™ kann eine zweite Typenbezeichnung verwendet werden, um eine automatische Konfiguration einer „zyklischen Aufgaben“-Option für die Aufgabe in der Multitasking-Code-Analyse zu bewirken.

[0064] Weitere Informationen, die für die korrekte Konfiguration des Codeprüfsystems für die Multitasking-Analyse benötigt werden, umfassen die entsprechenden Identitäten von Rechen-Threads im Implementierungscode. Wie oben in Verbindung mit **Fig. 3** beschrieben tritt ein technisches Problem auf, wenn mehrere unterschiedliche Kennungen für eine gleiche Aufgabe, ISR oder Ressource in dem spezifizierten Modell **102** während des Codevorbereitungsprozesses verwendet werden, so dass sich ein entsprechender Name **340** für den implementierten Thread in dem Implementierungscode **104** von dem Namen **310** unterscheidet, der verwendet wird, um anfänglich die Aufgabe in dem spezifizierten Modell **102** zu definieren. **Fig. 9A** bis **Fig. 9E** betreffen ein Beispiel der Verwendung von unterschiedlichen Kennungen für eine Aufgabe und eine exemplarische Lösung zum korrekten Identifizieren von implementierten Rechen-Threads in dem Implementierungscode **104**.

[0065] **Fig. 9A** zeigt einen ersten Abschnitt **901** eines spezifizierten Modells **102**, in dem eine Aufgabe definiert ist und einen Kurznamen „Task_90_C“ mit einem Aufgabenbenennungseintrag **920a** erhält. Beispielsweise kann die Aufgabe eine Aufgabe sein, die ein Kühlsystem (z. B. Einschalten eines Kühlerlüfters) in einem Kraftfahrzeugsystem als Antwort auf ein Ereignis von einem Sensor initiieren soll, der einen Temperaturwert von 100°C oder höher an einen Datenport einer Schnittstelle postet. Gemäß der oben in Verbindung mit **Fig. 4A** - **Fig. 4C** beschriebenen Methodik kann eine zweite Kennung **922** („/ActiveEcuC/Os/eTask1“ in diesem Beispiel) aus einem zweiten Abschnitt **902** des spezifizierten Modells bestimmt werden, das in **Fig. 9B** gezeigt ist, die einen geeigneten aufgabenreferenzierenden Eintrag **920b** enthält.

[0066] Während der Vorbereitung des Implementierungscode **104** aus dem spezifizierten Modell **102** kann ein Code-Implementierer (ob Mensch oder ein automatisiertes Implementierungshilfsmittel) durch die

Systemaufbaurichtlinien eingeschränkt werden, einen Abschnitt der zweiten Kennung **922** zu verwenden, wenn er die Aufgabe für den Implementierungscode definiert. Als ein Beispiel kann es erforderlich sein, dass der Implementierer den letzten Abschnitt der zweiten Kennung „eTask1“ verwendet, wenn er den Zwischencode **302** aus dem spezifizierten Modell vorbereitet. Gemäß diesem Beispiel kann der Implementierer einen Zwischencodeabschnitt **903** erzeugen, wie in **Fig. 9C** dargestellt. Ein aufgabendefinierender Eintrag **960a** in dem Zwischencodeabschnitt **903** kann mit einem Schlüsselwort („TASK“ in diesem Beispiel) beginnen und einen erforderlichen Abschnitt des zweiten Kennung **922**, „eTask1“ in diesem Beispiel, enthalten.

[0067] Das Problem der Verwendung unterschiedlicher Aufgabennamen während der oben beschriebenen Codevorbereitung kann auftreten, wenn der Zwischencodeabschnitt **903** von einem Codeprozessor **330** verarbeitet wird, um den Implementierungscode **104** zu bilden. Obwohl der Codeprozessor **330** teilweise durch Anforderungen der Systemaufbaurichtlinien eingeschränkt sein kann, müssen nicht alle Details der Codevorbereitung eingeschränkt oder standardisiert sein. Zum Beispiel muss die Art, in der implementierte Thread-Kennungen in dem Implementierungscode **104** definiert sind, die Aufgaben in dem Zwischencode **302** entsprechen, nicht über alle Codeprozessoren **330** hinweg standardisiert sein. Ein Codeprozessor kann ein anderes Thread-Benennungsmakro als ein anderer Codeprozessor verwenden. Dementsprechend kann ein entsprechender Implementierungscodeteil **904**, wie er in **Fig. 9D** gezeigt ist, eine unterschiedliche und unbekannte entsprechende Kennung in einer Deklaration **960b** des entsprechenden implementierten Threads (z. B. ein anderer Funktionsname, der in einer Funktionsdeklaration verwendet wird) haben. Jede Kennung kann in einem Makro verwendet werden, das durch einen Codeprozessor **330** implementiert wird, das verwendet wird, um den entsprechenden Implementierungscodeteil **904** zu benennen.

[0068] Um dieses Problem zu überwinden, kann ein Verfolgungscodeprozessor **1120** (in **Fig. 11** gezeigt) verwendet werden, um die Identität und den Ort eines implementierten Rechen-Threads in dem Implementierungscode **104** zu bestimmen, der seiner verknüpften Aufgabe entspricht. Ein Verfolgungscodeprozessor **1120** kann als Programmieranweisungen implementiert sein, die auf Hardware ausgeführt werden, die Logikgatter und Register umfasst, die als sequentielle Logikschaltungen angeordnet sind. In Ausführungsformen kann ein Verfolgungscodeprozessor **1120** den Zwischencode **302** und den entsprechenden Implementierungscode **104** empfangen. Der Verfolgungscodeprozessor **1120** kann mit Informationen von Systemaufbaurichtlinien, die für die Codevorbereitung relevant sind, programmiert werden oder die-

se empfangen, so dass der Verfolgungscodeprozessor **1120** Vergleichsimplementierungscode **104a** erzeugt, der dem von einem Codeprozessor **330** erzeugten Implementierungscode **104** ähnlich ist. Jedoch kann ein gesteuertes Makro von dem Verfolgungscodeprozessor **1120** verwendet werden, um entsprechende Implementierungscodeteile in dem Vergleichsimplementierungscode **104a** zu bezeichnen, die Codeteilen in dem Implementierungscode **104** entsprechen, der von einem Codeprozessor **330** unter Verwendung eines unbekannten Makros erzeugt werden. Zum Beispiel kann das gesteuerte Makro eine geänderte Aufgabenkennung in dem Vergleichsimplementierungscode **104a** erzeugen, die identifiziert und verwendet werden kann, um die tatsächliche Kennung für den entsprechenden Codeteil in dem Implementierungscode **104** zu bestimmen.

[0069] Ein exemplarischer Vergleichsimplementierungscodeteil **905**, der mit einer Aufgabe verknüpft ist, ist in **Fig. 9E** gezeigt. In Ausführungsformen kann ein von dem Verfolgungscodeprozessor **1120** verwendetes Makro von einem Benutzer gesteuert werden und/oder ist dem Verfolgungscodeprozessor **1120** vorbestimmt und bekannt. In dem dargestellten Beispiel kann der Verfolgungscodeprozessor **1120** ein Benennungsmakro verwenden, das „_IDn“ an einen Aufgabennamen anfügt, der in dem Zwischencode **302** erscheint, um eine geänderte Aufgabenkennung zu erzeugen, obwohl irgendein Benennungsmakro verwendet werden kann. In der dargestellten Ausführungsform kann die numerische Bezeichnung „n“ in der Makroerweiterung verwendet werden, um Instanzen zu identifizieren, in denen ein gleicher Name für unterschiedliche Aufgaben in dem Zwischencode **302** erscheint. Wenn beispielsweise der Zwischencode **302** eine andere Instanz von „eTask1“ für eine andere Aufgabe enthielte, würde die Makroerweiterung auf „_ID2“ inkrementiert werden.

[0070] Mit internem Wissen über das Benennungsmakro, das von dem Verfolgungscodeprozessor **1120** verwendet wird, kann der Verfolgungscodeprozessor den Vergleichscode **104a** analysieren und mit dem empfangenen Implementierungscode **104** vergleichen, um die Kennung des entsprechenden implementierten Rechen-Threads einer Aufgabe in dem empfangenen Implementierungscode **104** zu bestimmen. Zum Beispiel kann der Verfolgungscodeprozessor **1120** oder ein anderes Analysetool den Vergleichscode **104a** nach beliebigen Instanzen zumindest eines Abschnitts des bekannten makroerweiterten Funktionsnamens (z. B. „eTask1“ oder „eTask1_ID“ in diesem Beispiel) schnell durchsuchen und bestimmen, ob es eine oder mehrere Instanzen des gesuchten Abschnitts gibt. Für jede Instanz kann das Analysetool indikative Codezeilennummern, nahegelegene Codierungseinträge und/oder Zeilen von nicht verarbeitetem Code identifizieren (wie beispielsweise

se „EventMaskType ev;“ und „*Schedule: Full“ in dem dargestellten Beispiel). Das Analysetool kann dann den Implementierungscode **104** nach ähnlichen und/oder gleichen Codezeilennummern, Codierungseinträgen und/oder Zeilen von unverarbeitetem Code und indikativen Codierungseinträgen durchsuchen, um die entsprechende implementierten Thread-Kennung der Aufgabe und den verknüpften Thread in dem Implementierungscode eindeutig zu identifizieren. Ein gleiches Verfahren, das ein bekanntes Benennungsmakro verwendet, kann verwendet werden, um Rechen-Threads in dem Implementierungscode **104** zu identifizieren, der mit ISRs verknüpft ist, die in dem spezifizierten Modell **102** definiert sind.

[0071] In einigen Implementierungen kann nur der Vergleichscode **104a** auf das Vorhandensein oder Fehlen von Fehlern überprüft werden. In solchen Fällen wird der Implementierungscode **104** möglicherweise nicht dem Verfolgungscodeprozessor **1120** bereitgestellt, und ein Vergleich des Implementierungscodes **104** mit dem Vergleichscode **104a** kann nicht durchgeführt werden. In solchen Fällen kann die Kenntnis des Benennungsmakros, das von dem Verfolgungscodeprozessor **1120** verwendet wird, genutzt werden, um Rechen-Threads in dem Vergleichscode **104a**, der mit Aufgaben und Interrupts verknüpft ist, direkt zu identifizieren.

[0072] Sobald die entsprechende implementierte Thread-Kennung einer Aufgabe oder ISR und der verknüpfte Thread in dem Implementierungscode **104** identifiziert sind, kann der implementierte Rechen-Thread mit Attributen verknüpft werden, die von dem Datenanalysator **1110** für die Aufgabe oder ISR aus dem spezifizierten Modell bestimmt werden und zur Verwendung in der automatischen Konfiguration des Codeprüfsystems **100** zur Multitasking-Analyse bereitgestellt werden. In einigen Fällen können Aufgabenattribute, eine oder mehrere Aufgabenkennungen (z. B. Kurzname, vollständig qualifizierter Name) und eine entsprechende implementierte Thread-Kennung (z. B. Funktionsname) und verknüpfter Thread-Code für jede identifizierte Aufgabe als Konfigurationsdaten **1160** für einen Zugriff durch das Codeprüfsystem **100** aufgezeichnet werden. In ähnlicher Weise können ISR-Attribute, eine oder mehrere ISR-Kennungen (z. B. Kurzname, vollständig qualifizierter Name) und eine entsprechende implementierte Thread-Kennung (z. B. Funktionsname) und verknüpfter Thread-Code für jede identifizierte ISR als Konfigurationsdaten **1160** für einen Zugriff durch das Codeprüfsystem **100** aufgezeichnet werden.

[0073] Die Identifikation eines Codeabschnitts in dem Implementierungscode **104**, der mit einer in dem spezifizierten Modell **102** definierten Ressource verknüpft ist, kann sich von dem Prozess unterscheiden, der zum Identifizieren von Rechen-Threads für Aufgaben und ISRs verwendet wird. Sobald eine

Kennung für eine Ressource für den Zwischencode **302** bestimmt wurde (z. B. „Resource_BasicAirbag“, der eine Variante des in **Fig. 7** gezeigten Kurznamens sein kann), kann ein Codeprozessor **330** ein Makro verwenden (das dem Codeprüfsystem **100** nicht bekannt ist), das den Namen der Ressource ändert, wenn ein entsprechender Abschnitt **1002** des Implementierungscodes **104** erzeugt wird, wie in **Fig. 10A** gezeigt. In Ausführungsformen kann ein anderes aber bekanntes Makro von einem Verfolgungscodprozessor **1020** verwendet werden, um einen entsprechenden Abschnitt **1003** des Vergleichsimplementierungscodes **104a** zu erzeugen, wie in **Fig. 10B** gezeigt.

[0074] In Ausführungsformen kann ein Ressourcenaufwurf mit einer implementierten Ressource in dem Implementierungscode verknüpft sein. Zum Beispiel kann ein erster Ressourcenaufwurf **1020b** angeben, dass ein Prozessor (z. B. ein Prozessor einer ECU in einem dynamischen System) eine identifizierte Ressource nehmen soll (in diesem Beispiel durch einen „TakeResource“-Aufruf angegeben). Ein zweiter Ressourcenaufwurf **1020c** kann angeben, dass der Prozessor die identifizierte Ressource freigeben kann (in diesem Beispiel durch einen „ReleaseResource“-Aufruf angegeben). Durch Analysieren des Vergleichscodes **104a** und des Implementierungscodes **104** kann ein Verfolgungscodprozessor **1120** oder ein Analysetool den Ort und jede zugehörige Kennung für eine implementierte Ressource in dem Implementierungscode **104** bestimmen, der einer in dem spezifizierten Modell **102** definierten Ressource entspricht. In Bezug auf die oben erwähnte modifizierte Polyspace Code checker™-Software kann die Identifikation der entsprechenden Initiierung einer implementierten Ressource und Freigabe in dem Implementierungscode verwendet werden, um eine automatische Konfiguration von „kritischer-Abschnitt-beginnt“ und „kritischer-Abschnitt-Ende“-Optionen für die implementierte Ressource in der Multitasking-Codeanalyse zu bewirken.

[0075] **Fig. 11** zeigt eine mögliche Ausführungsformanordnung eines Codeprüfsystems **100**, eines Datenanalysators **1110** und eines Verfolgungscodprozessors **1120**. In einigen Fällen kann jeder von einem Codeprüfer **106**, Datenanalysator **1110** und Verfolgungscodprozessor **1120** einen oder mehrere umfassen mehr Prozessoren. In einigen Implementierungen kann der Datenanalysator **1110** Teil des Codeüberprüfungssystems **100** sein. In einer solchen Implementierung kann die Funktionalität des Datenanalysators **1110** und des Codeüberprüfers **106** auf einem oder mehreren Prozessoren desselben Datenverarbeitungssystems **110** des Codeprüfsystems **100** ausgeführt werden. In einigen Fällen kann der Verfolgungscodprozessor **1120** Teil des Codeprüfsystems **100** sein. In ähnlicher Weise kann die Funktionalität des Verfolgungscodprozessors **1120** und des Code-

prüfers **106** einen oder mehrere Prozessoren desselben Datenverarbeitungssystems **110** des Codeprüfsystems ausführen. In einigen Implementierungen können der Datenanalysator **1110** und der Verfolgungscodprozessor **1120** in demselben Datenverarbeitungssystem verkörpert sein, das von dem Codeprüfsystem getrennt ist. Konfigurationsdaten **1160** können in einer oder mehreren Datenstrukturen gespeichert werden und/oder mit dem Codeprüfer **106** über eine oder mehrere Kommunikationsverbindungen an das Datenverarbeitungssystem **110** kommuniziert werden. In noch anderen Fällen können der Datenanalysator **1110** und der Verfolgungscodprozessor **1120** beide Teil des Codeprüfsystems **100** sein, und ihre jeweiligen Funktionalitäten können auf einem oder mehreren Prozessoren desselben Datenverarbeitungssystems **110** des Codeprüfsystems **100** ausgeführt werden.

[0076] **Fig. 12** zeigt exemplarische Schritte eines Verfahrens, die von einem Datenanalysator **1110** durchgeführt werden können. Ein Verfahren **1200** zum Bestimmen von Aufgabenkennungen und verknüpften Aufgabenattributen aus einem spezifizierten Modell **102** kann mit dem Empfangen (Schritt **1205**) einer oder mehrerer Datenstrukturen des spezifizierten Modells beginnen, welche Aufgaben definieren, die als Rechen-Threads in dem Implementierungscode **104** implementiert werden sollen. Das Verfahren **1200** kann das Analysieren (Schritt **1210**) der einen oder mehreren Datenstrukturen zum Identifizieren einer Aufgabenkennung für eine spezifizierte Aufgabe umfassen. Die Aufgabenkennung kann zumindest einen Abschnitt alphanumerischen Textes enthalten, der während der Vorbereitung des Implementierungscodes **104** aus dem spezifizierten Modell **102** verwendet wird. Beispielsweise kann der Abschnitt alphanumerischen Textes in einer Definition einer Aufgabe in dem spezifizierten Modell **102** verwendet werden und ferner als zumindest ein Abschnitt einer Referenz bzw. eines Verweises auf einen entsprechenden Code-Thread in dem Implementierungscode **104** verwendet werden. Die spezifizierte Aufgabe kann eine Aufgabe einer Mehrzahl von Aufgaben sein, die als Rechen-Threads zu implementieren sind, die gleichzeitig in einem dynamischen System laufen können. Das Verfahren **1200** kann das Analysieren (Schritt **1215**) der einen oder mehreren Datenstrukturen des spezifizierten Modells **102** zum Identifizieren von Attributen umfassen, die mit der spezifizierten Aufgabe verknüpft sind. Attribute einer Aufgabe können Informationen enthalten, die angeben, wie eine Aufgabe aktiviert wird, beispielsweise ob eine Aufgabe durch ein Ereignis oder durch einen Alarm aktiviert wird. Die Verfahren zum Analysieren (Schritt **1210** und Schritt **1215**) können Verfahren sein, wie sie oben in Verbindung mit **Fig. 4A** bis **Fig. 5C** beschrieben wurden. In einigen Fällen kann das Verfahren **1200** einen Prozess zum Identifizieren

eines Typs der spezifizierten Aufgabe (z. B. Verfahren **800**) umfassen.

[0077] Ein Verfahren **1200** zum Bestimmen von Aufgabenkennungen und verknüpften Aufgabenattributen aus einem spezifizierten Modell **102** kann das Bestimmen (Schritt **1220**) umfassen, ob das spezifizierte Modell **102** zusätzliche Aufgaben enthält. Wenn das spezifizierte Modell zusätzliche Aufgaben enthält, kann der Prozessablauf zur Analyse (Schritt **1210**) der einen oder mehreren Datenstrukturen des spezifizierten Modells zurückkehren, um eine andere Aufgabenkennung für eine andere spezifizierte Aufgabe zu identifizieren. Wenn es keine verbleibenden Aufgaben gibt, kann der Prozessablauf das Bereitstellen (Schritt **1225**) von Informationen bezüglich der Aufgabenidentität und ihrer verknüpften Aufgabenattribute zur automatischen Konfiguration eines Codeprüfsystems **100** umfassen (z. B. Bereitstellen der Informationen an ein Codeprüfsystem oder Speichern der Informationen als Konfigurationsdaten **1160** für einen Zugriff durch ein Codeprüfsystem). Gemäß einigen Ausführungsformen kann das Verfahren **1200** enden, nachdem jede Aufgabe und verknüpfte Aufgabenattribute aus der einen oder den mehreren empfangenen Datenstrukturen des spezifizierten Modells **102** identifiziert wurden.

[0078] In Ausführungsformen, in denen der Datenanalysator **1110** und ein Verfolgungscodetyp **1120** innerhalb desselben Datenverarbeitungssystems **110** wie der Codeprüfer **106** implementiert sind, kann das Verfahren **1200** ferner Schritte eines Verfahrens **1300** zum Verknüpfen von Aufgabenattributen jeder spezifizierten Aufgabe mit entsprechenden Rechen-Threads umfassen, wie es nachfolgend in Verbindung mit **Fig. 13** beschrieben wird. Zum Beispiel kann der Prozessablauf des Verfahrens **1200** mit Schritt **1305** des Verfahrens **1300** fortfahren. Nachdem das Verfahren **1300** abgeschlossen ist, kann der Prozessablauf Schritte eines Verfahrens **1400** zum automatischen Konfigurieren eines Codeprüfsystems zur Multitasking-Analyse umfassen, wie es nachfolgend in Verbindung mit **Fig. 14** beschrieben wird. Zum Beispiel kann der Prozessablauf des Verfahrens **1200** mit Schritt **1405** des Verfahrens **1400** fortfahren.

[0079] In Ausführungsformen, in denen der Datenanalysator **1110** in demselben Datenverarbeitungssystem **110** wie der Codeprüfer **106** implementiert ist, kann das Verfahren **1200** zum Bestimmen von Aufgabenkennungen und verknüpften Aufgabenattributen aus einem spezifizierten Modell **102** mit Schritten eines Verfahrens **1400** zum automatischen Konfigurieren eines Codeprüfsystems zur Multitasking-Analyse fortfahren, wie es nachfolgend in Verbindung mit **Fig. 14** beschrieben wird. Zum Beispiel kann der Prozessablauf des Verfahrens **1200** mit Schritt **1405** des Verfahrens **1400** fortfahren.

[0080] **Fig. 13** zeigt exemplarische Schritte eines Verfahrens **1300** zum Verknüpfen von Aufgabenattributen eines spezifizierten Modells **102** mit entsprechenden Rechen-Threads eines Implementierungscode **104** oder Vergleichscode **104a**, der aus dem spezifizierten Modell erstellt wird. Das Verfahren **1300** kann mit einem Verfolgungscodetyp **1120** ausgeführt werden und Schritte ausführen, wie sie oben in Verbindung mit **Fig. 9E** beschrieben wurden, gemäß einigen Ausführungsformen. Ein Verfahren **1300** kann mit dem Empfangen (Schritt **1305**) einer oder mehrerer Datenstrukturen des Zwischencodes **302** beginnen, die aus einem spezifizierten Modell **102** erstellt wurden. Das Verfahren **1300** kann auch ein Empfangen (Schritt **1310**) von Implementierungscode **104** umfassen, der aus dem Zwischencode erzeugt wurde. Das Verfahren **1300** kann auch das Empfangen (Schritt **1315**) von Informationen über spezifizierte Aufgaben, die in dem Zwischencode vorhanden sind, umfassen. Die empfangenen Informationen können die Kennung für jede spezifizierte Aufgabe enthalten und können zusätzlich Aufgabenattribute enthalten, die mit jeder spezifizierten Aufgabe verknüpft sind. In einigen Ausführungsformen kann der Schritt **1315** des Empfangens von Informationen über spezifizierte Aufgaben durch einen Schritt des Bestimmens spezifizierter Aufgabenkennungen aus dem empfangenen Zwischencode ersetzt werden (z. B. durch Suchen nach einem Schlüsselwort wie „TASK“ für das in **Fig. 9C** gezeigte Beispiel).

[0081] Ein Verfahren **1300** zum Verknüpfen von Aufgabenattributen zu entsprechenden Rechen-Threads kann ferner das Erzeugen (Schritt **1320**) des Vergleichscode **104a** aus dem empfangenen Zwischencode **302** umfassen. Der erzeugte Vergleichscode kann eine bekannte Variante jeder spezifizierten Aufgabenkennung enthalten, wie es oben in Verbindung mit **Fig. 9E** beschrieben ist. Ein Verfahren **1300** kann ferner das Analysieren (Schritt **1325**) des Vergleichscode **104a** und des Implementierungscode **104** zum Identifizieren einer Rechen-Thread-Kennung in dem Implementierungscode umfassen, die einer spezifizierten Aufgabenkennung in dem Zwischencode **302** entspricht, wie es oben in Verbindung mit **Fig. 9E** beschrieben ist. Das Verfahren **1300** kann ferner das Verknüpfen (Schritt **1330**) der implementierten Thread-Kennung und/oder ihres entsprechenden Rechen-Threads mit der spezifizierten Aufgabenkennung und/oder ihren verknüpften Aufgabenattributen umfassen. Der Schritt **1330** des Verknüpfens eines Rechen-Threads und seiner entsprechenden Aufgabenattribute kann verwendet werden, um den Typ der Aufgabe, mit der der Thread korrespondiert, für das Codeprüfsystem **100** korrekt zu bezeichnen. Der Schritt **1330** des Verknüpfens kann das Aufzeichnen einer Datenverbindung zwischen einer Aufgabenkennung und/oder ihren verknüpften Attributen und ihres entsprechenden implementierten Code-Threads

und/oder Thread-Kennung umfassen (z. B. Aufzeichnen eines Querverweises von Daten, die eine Aufgabe angeben, und Daten, die den entsprechenden Rechen-Thread der Aufgabe angeben). In einigen Implementierungen kann der Schritt **1330** des Verknüpfens das Übertragen von Daten an einen Codeprüfer **106** umfassen, der einen Typ einer Aufgabe identifiziert, die dem implementierten Code-Thread entspricht, und zwar basierend auf Informationen, die aus den Attributen der Aufgabe ermittelt werden.

[0082] Das Verfahren **1300** kann mit der Bestimmung (Schritt **1335**) fortfahren, ob weitere Aufgaben in dem empfangenen Zwischencode **302** verbleiben. Wenn in dem Zwischencode noch mehr Aufgaben verbleiben, kann das Verfahren **1300** zu dem Schritt des Analysierens (**1325**) des Vergleichscode **104a** und des empfangenen Implementierungscodes **104** zurückkehren. Wenn in dem Zwischencode **302** keine weiteren Aufgaben verbleiben, kann das Verfahren **1300** mit der Bereitstellung (**1145**) von Informationen bezüglich Thread-Identität und Aufgabenattributen zur automatischen Konfiguration eines Codeprüfsystems **100** fortfahren.

[0083] In einigen Fällen können Informationen bezüglich der Thread-Identität und entsprechenden Aufgabenattributen direkt an ein Codeprüfsystem **100** bereitgestellt werden oder als Konfigurationsdaten **1160** für den Zugriff durch das Codeüberprüfungssystem **100** aufgezeichnet werden. Wenn der Verfolgungscodeprozessor **1120** innerhalb desselben Datenverarbeitungssystems **110** implementiert ist wie der Codeprüfer **106**, kann das Verfahren **1300** mit Schritten eines Verfahrens **1400** zum automatischen Konfigurieren eines Codeprüfsystems zur Multitasking-Analyse fortfahren, wie es nachfolgend in Verbindung mit **Fig. 14** beschrieben ist. Andernfalls kann das Verfahren **1300** enden.

[0084] **Fig. 14** zeigt exemplarische Schritte eines Verfahrens zum automatischen Konfigurieren eines Codeprüfsystems zur Multitasking-Analyse. Ein Verfahren **1400** kann damit beginnen, auf eine oder mehrere Datenstrukturen eines bestimmten Modells **102** zuzugreifen (Schritt **1405**) und auf den Implementierungscode **104** oder den Vergleichscode **104a** zuzugreifen (Schritt **1410**), der aus dem spezifizierten Modell **102** erstellt wurde. Ein Verfahren **1400** kann ferner das Zugreifen (Schritt **1415**) auf Informationen umfassen, die Rechen-Threads und/oder ihre Kennungen in dem Implementierungscode und den entsprechenden Attributen identifizieren. In einigen Implementierungen können die entsprechenden Attribute mit ihren entsprechenden Rechen-Threads und/oder Thread-Kennungen verknüpft oder anderweitig mit ihnen identifiziert werden.

[0085] Ein Verfahren **1400** kann ferner das automatische Konfigurieren (Schritt **1420**) eines Code-

prüfsystems **100** zur Multitasking-Analyse basierend auf den implementierten Thread-Identitäten und ihren entsprechenden Attributen umfassen. Zum Beispiel kann jeder Rechen-Thread dem Codeprüfer **106** als ein Auftreten eines ersten Typs von Aufgabe (z. B. einer erweiterten Aufgabe) oder eines zweiten Typs von Aufgabe (z. B. einer grundlegenden Aufgabe) identifiziert werden. In einigen Implementierungen können dem Codeprüfer **106** auch Informationen über ein Ereignis oder einen Alarm, das bzw. der die Aufgabe aktiviert, bereitgestellt werden, wie zum Beispiel ein Ort in dem Implementierungscode **104** des Codes, der dem Ereignis oder dem Alarm entspricht. Ein Verfahren **1400** zum automatischen Konfigurieren eines Codeprüfsystems **100** zur Multitasking-Analyse kann dann mit der Durchführung (Schritt **1430**) einer statischen Codeanalyse zumindest eines Abschnitts des Implementierungscodes fortfahren und die Ergebnisse der statischen Codeanalyse anzeigen (Schritt **1440**). Nach der Analyse des Implementierungscodes kann das Verfahren **1400** enden.

[0086] Obwohl Schritte von Verfahren zum Analysieren eines spezifizierten Modells **102**, des Zwischencodes **302** und des Implementierungscodes **104** oben im Wesentlichen in Bezug auf Aufgaben und ihre verknüpften Attribute in Verbindung mit **Fig. 12** bis **Fig. 14** beschrieben worden sind, können ähnliche Verfahren zum Identifizieren von Interrupt Service Routinen (ISRs) und Ressourcen und ihren verknüpften Attributen aus einem spezifizierten Modell **102** verwendet werden. Zum Beispiel kann das Verfahren **1200** zum Bestimmen von Aufgabenkennungen und verknüpften Aufgabenattributen aus einem spezifizierten Modell **102**, das in **Fig. 12** gezeigt ist, für ISRs oder Ressourcen ausgeführt werden, wo in der Zeichnung das Wort „Aufgabe“ durch „ISR“ oder „Ressource“ ersetzt ist. In ähnlicher Weise kann das Verfahren **1300** zum Verknüpfen von Aufgabenattributen mit entsprechenden Rechen-Threads für ISRs und Ressourcen ausgeführt werden, wie in **Fig. 13** gezeigt, wo das Wort „Aufgabe“ durch „ISR“ oder „Ressource“ ersetzt ist.

[0087] In einigen Ausführungsformen kann es nur einen Typ von ISR in einem spezifizierten Modell **102** geben. In solchen Fällen können entsprechende Rechen-Threads in dem Implementierungscode **104** identifiziert werden und automatisch als ein Typ von ISR für das Codeprüfsystem bezeichnet werden. Wenn es mehr als einen Typ von ISR gibt, können Attribute der ISR (z. B. eine Kategorie der ISR, wie in **Fig. 6** angegeben) analysiert werden, um einen Typ von ISR zu bestimmen, der dem Codeprüfsystem **100** zugewiesen wird.

[0088] Benutzerschnittstellen für ein Codeprüfsystem **100** können auf einer Anzeige **191** in irgendeiner geeigneten Form wiedergegeben werden. Einige ex-

emplarische Benutzerschnittstellen sind in **Fig. 15** bis **Fig. 19** dargestellt. Eine exemplarische Konfigurationsbenutzerschnittstelle **1505** ist in **Fig. 15** gezeigt. Eine Konfigurationsbenutzerschnittstelle **1505** kann einem Benutzer ermöglichen, verschiedene Einstellungen des Codeprüfsystems **100** zu steuern, die mit der Multitasking-Codeanalyse verknüpft sind. In einer exemplarischen Ausführungsform kann eine Konfigurationsbenutzerschnittstelle **1505** ein Einstellungsnavi-gations-Panel **1510** und ein Multitasking-Konfigu-rations-Panel **1520** umfassen. Das Einstellungsnavi-gations-Panel **1510** kann einem Benutzer erlauben, verschiedene aufgelistete Optionen **1515** in dem Pa-nel auszuwählen, die mit Konfigurationseinstellungen verschiedener Merkmale des Codeprüfsystems **100** verbinden. In diesem Beispiel wurde eine „Multitas-king“-Option in den aufgelisteten Optionen **1515** zur Konfiguration ausgewählt.

[0089] Die Auswahl einer Multitasking-Option in dem Einstellungsnavi-gations-Panel **1510** kann die An-zeige eines Multitasking-Konfigurations-Panels **1520** bewirken. In Ausführungsformen kann das Multitas-king-Konfigurations-Panel **1520** einem Benutzer er-lauben, zwischen automatischen Konfigurationsop-tionen **1522** und manuellen Konfigurationsoptionen **1527** zu wählen. Die manuelle Konfigurationsoption **1527** kann ausgewählt werden, wenn ein Benutzer manuell Bezeichnungen für Aufgaben oder ISRs und ihre verknüpften Attribute eingeben möchte.

[0090] In einigen Fällen können mehrere Optionen für die automatische Konfiguration von Aufgaben und ISRs vorhanden sein, wie in dem veranschaulichten Beispiel von **Fig. 15** dargestellt. Eine erste Option kann ausgewählt werden, um zu ermöglichen, dass der Codeprüfer beim Prüfen oder Analysieren des Im-plementierungscodes **104** automatisch die Gleichzei-tigkeit von Aufgaben und ISRs erkennt. Eine zweite Option kann für eine automatische externe Konfigura-tion von Aufgaben und ISRs unter Verwendung exter-ner Information ausgewählt werden (z. B. Informatio-nen, die von einem Datenanalysator **1110** und einem Verfolgungscodprozessor **1120** bereitgestellt wer-den, der außerhalb des normalen Betriebs des Code-prüfers **106** bestimmt wurde. Die erste und zweite Option der automatischen Konfiguration können auch eine Softwarearchitektureintragsoption **1524** enthal-ten, die es einem Benutzer ermöglicht, einen Typ von Softwarearchitekturspezifikation zu identifizieren, die für den zu analysierenden Code gilt (in diesem Bei-spiel AUTOSAR ausgewählt).

[0091] Eine exemplarische Ergebnisbenutzer-schnittstelle **1605**, die Codeanalyseergebnisse von einem Codeprüfsystem **100** anzeigen kann, ist in **Fig. 16** dargestellt. Die Ergebnisbenutzerschnittstel-le **1605** kann ein Zusammenfassungs-Panel **123**, ein Detail-Panel **125** und ein Quellcode-Panel **129** enthal-ten. Das Zusammenfassungs-Panel **123** kann eine

Liste von Codeüberprüfungen bereitstellen, die von dem Codeprüfsystem **100** beispielsweise an einem Abschnitt des Codes durchgeführt wurden. In dem veranschaulichten Beispiel fasst das Zusammenfas-sungs-Panel **123** Codeüberprüfungen zusammen, die für eine Funktion (identifiziert als „erste ()“) des Implementierungscodes durchgeführt werden. Die Auflistung kann gemäß einigen Ausführungsformen verschiedene Arten von Codeüberprüfungen ange-ben, die an jeder Zeile des Quellcodes durchgeführt wurden. Jedes Element in der Liste kann farbcodiert sein und/oder ein bestimmtes Symbol enthalten, das anzeigt, ob die Codezeile eine Prüfung bestanden hat (in diesem Beispiel als ein Häkchen gezeigt), eine Prüfung nicht bestanden hat (angezeigt durch ein „X“ oder ein „?“ in diesem Beispiel) oder es bekannt ist, dass sie zu einem schwerwiegenden Laufzeitfehler führt (in diesem Beispiel durch ein „!“ angezeigt). An-dere Arten von Zusammenfassungen können in dem Zusammenfassungs-Panel **123** in anderen Ausführungsformen bereitgestellt werden, wie beispielswei-se in der am 30. November 2017 eingereichten euro-päischen Patentanmeldung Nr. 17290155.5 mit dem Titel „Systems and Methods for Evaluating Complian-ce of Implementation Code with a Software Architec-ture Specification“ beschrieben.

[0092] Für das veranschaulichte Beispiel von **FIG. 16** kann ein Benutzer einen Eintrag in dem Zusammenfassungs-Panel **123** auswählen. Das ausgewähl-te Ergebnis **1610** kann hervorgehoben oder auf ande-re Weise durch die Benutzerauswahl betont werden. Das ausgewählte Ergebnis **1610** kann das Codeprüf-system **100** veranlassen, verknüpfte Informationen über das ausgewählte Ergebnis **1610** anzuzeigen. Zum Beispiel kann das Codeprüfsystem **100** weite-re Details des bestimmten Codierungsfehlers in dem Detail-Panel **125** anzeigen, wie dargestellt. Die Infor-mationen können eine Erklärung liefern, warum die Codezeile als fehlerhaft identifiziert wurde. In dem veranschaulichten Beispiel überschreitet eine Anzahl von Iterationen einer while-Schleife eine maximale Anzahl von Iterationen (**200**), die entweder durch das spezifizierte Modell **102** oder die Systemaufbauricht-linien erlaubt sind. Das Detail-Panel kann in einer Ab-tast- bzw. Beispielzeile des Codes Maximalwerte für Variablen identifizieren, die in dem entsprechenden Implementierungscode verwendet werden, so dass ein Benutzer den Grund für einen möglichen Laufzeit-fehler schnell bestimmen kann.

[0093] In Ausführungsformen kann ein Codeprüfsys-tem **100** auch eine Zeile oder einen Abschnitt des Implementierungscodes in dem Quellcode-Panel **129** anzeigen, der die Codezeile enthält, die als einen Codierungsfehler aufweisend identifiziert wird und dem ausgewählten Ergebnis **1610** entspricht. In eini-gen Fällen kann das Codeprüfsystem den Codeab-schnitt **1620** mit dem Fehler hervorheben, farbcodie-ren oder anderweitig betonen, so dass die Codezei-

le schnell ermittelt werden kann. Durch Bereitstellen einer Verbindung zwischen einem ausgewählten Ergebnis **1610** in dem Zusammenfassungs-Panel **123**, Informationen, die für das ausgewählte Ergebnis in dem Detail-Panel **125** relevant sind, und einen entsprechenden Abschnitt des Implementierungscodes **104** in dem Quellcode-Panel **129**, kann ein Benutzer des Codeprüfsystems **100** Fehler, die in dem Implementierungscode **104** gefunden werden, schnell verstehen, zu diesen navigieren und diese korrigieren.

[0094] Fig. 17 zeigt eine weitere exemplarische Benutzerschnittstelle, die von einem Codeprüfsystem **100** angezeigt werden kann, um Informationen bereitzustellen, die für die Multitasking-Codeanalyse für Aufgaben und ISRs relevant sind. In Ausführungsformen kann eine Aufgaben/ISR-Zusammenfassungsschnittstelle **1705** in irgendeiner geeigneten Weise auf einer Anzeige **191** des Codeprüfsystems **100** angezeigt werden. Die Aufgaben/ISR-Zusammenfassungsschnittstelle **1705** kann eine erste Auflistung **1720** von ISRs, die in einem spezifizierten Modell **102** identifiziert sind, und eine zweite Auflistung **1740** von Aufgaben enthalten, die in dem spezifizierten Modell identifiziert sind. Die Auflistungen können einen oder mehrere der Namen, die mit jeder ISR und jeder Aufgabe verknüpft sind, und/oder eine entsprechende Rechen-Thread-Kennung enthalten (z. B. einen entsprechenden Funktionsdeklarationsnamen, der in dem Implementierungscode **104** verwendet wird).

[0095] Jeder Eintrag in der Aufgaben/ISR-Zusammenfassungsschnittstelle **1705** kann ein oder mehrere aktive Elemente enthalten, die Links zu dem Implementierungscode **104** bereitstellen. Zum Beispiel kann eine implementierte Thread-Kennung **1743a** als aktiver Text gerendert werden und kann von einem Benutzer ausgewählt werden, um einen Link zu einem Abschnitt des Implementierungscodes **104** bereitzustellen und diesen anzuzeigen, der die implementierten Thread-Kennung **1743a** und den zugehörigen Quellcode des implementierten Threads enthält. Das Auswählen des aktiven Texts kann zum Beispiel die Anzeige des Quellcodes des implementierten Threads in einem Quellcode-Panel **129** bewirken.

[0096] Jeder Eintrag in der Aufgaben/ISR-Zusammenfassungsschnittstelle **1705** kann auch Aktivierungsinformationen über die ISR oder die Aufgabe enthalten. Die Aktivierungsinformationen **1743b** können auch zumindest teilweise als ein aktives Element gerendert werden, das einen Link zu einem Abschnitt des Implementierungscodes **104** und eine Anzeige desselben bereitstellt, der für die Aktivierung der ISR oder der Aufgabe relevant ist. Auf diese Weise kann ein Benutzer des Codeprüfsystems **100** Abschnitte einer großen Codebasis, die für ISRs oder Aufgaben und deren Aktivierungscode spezifisch sind, schnell und visuell bewerten.

[0097] Ein Codeprüfsystem **100** kann auch verschiedene Arten von Zusammenfassungenbenutzerschnittstellen anzeigen, die für die Multitasking-Analyse relevant sind, um einem Benutzer beim Verstehen von Ergebnissen aus der statischen Codeanalyse des Implementierungscodes **104** zu helfen. Ein Beispiel von angezeigten Ergebnissen, die für Multitasking relevant sind, ist in Fig. 18 Gezeigt. In Ausführungsformen kann das Codeprüfsystem **100** eine Liste von Variablen („x“, „y“, „a“) erfassen und anzeigen, die von mehreren Aufgaben verwendet werden, die bei der Ausführung gleichzeitig laufen. Der Codeüberprüfer **106** kann eine Aufzeichnung von Variablen, die von gleichzeitigen Aufgaben verwendet werden, führen und bestimmen, ob ein und dieselbe Variable von mehreren Aufgaben geteilt wird und ob eine geteilte Variable geschützt oder ungeschützt ist. Geteilte Variablen, die nicht geschützt sind, können von einem gleichzeitig ausgeführten Thread überschrieben oder beschädigt werden, wenn sich ein erster Thread, die die Variable verwendet, in einem Wartezustand befindet, und dies kann zu einem falschen Ergebnis des ersten Threads führen, wenn er beispielsweise wieder ausgeführt wird.

[0098] Gemäß dem veranschaulichten Beispiel von Fig. 18 kann ein Benutzer einen Geteilte- bzw. Gemeinsam-Genutzte-Variable-Eintrag **1810**, um weitere Informationen über die geteilte bzw. gemeinsam genutzte Variable in einem Detail-Panel **125** zu erhalten, von dem ein Abschnitt in Fig. 19 gezeigt ist, sowie mindestens einen entsprechenden Abschnitt des Implementierungscodes **104** oder Vergleichscodes **104a** in einem Quellcode-Panel **129** auswählen (ein in Fig. 16 gezeigtes Beispiel), das einen Rechen-Thread für eine Aufgabe enthält, welche die geteilte Variable verwendet. Das Detail-Panel kann Informationen über die gemeinsame Nutzung der Variablen und eine Aufgabenaufstellung **1910** bereitstellen, die für eine gemeinsam genutzte Variable relevant ist. In diesem veranschaulichten Beispiel wird die Variable „x“ von zwei Aufgaben auf verschiedene Arten geteilt. Eine erste Aufgabe „act10ms“ kann die Variable lesen und einen Wert in die Variable schreiben. Eine zweite Aufgabe „init“ kann auch einen Wert in die Variable schreiben. Da die Variable nicht geschützt ist, kann ein Wert, der von einem Thread der beiden Aufgaben geschrieben wird, durch einen zweiten Thread der anderen Aufgabe überschrieben werden. Einträge in dem Detail-Panel **125**, das in Fig. 19 gezeigt ist, können auch aktive Elemente enthalten, die für jede Aufgabe in der Aufgabenaufstellung **1910** einen Link zu dem entsprechenden Quellcode bereitstellen. In einigen Fällen kann das Codeprüfsystem Informationen über Aufgaben-, ISR- oder Ressourcenkennungen in einem spezifizierten Modell **102**, eine entsprechende Kennung im Zwischencode **302** und eine entsprechende Kennung für einen Rechen-Thread oder Codeteil im Implementierungscode **104** oder Vergleichscode **104a** verwenden, um eine Ver-

bindung zwischen Modellabschnitten und ihren entsprechenden Codeteilen aufrechtzuerhalten. Auf diese Weise können Informationen, die für eine Aufgabe, ISR oder Ressource relevant sind, leicht aus einer Modelldatei und/oder Codedatei abgerufen werden, beispielsweise zur Anzeige und Überprüfung durch einen Benutzer, um das Modell oder den Code zu prüfen und/oder zu bearbeiten.

[0099] Ergebnisse einer zusätzlichen Code-Analyse des spezifizierten Modells **102** und des Implementierungscodes **104** können Konsistenzdaten zwischen spezifizierten Aufgaben in dem spezifizierten Modell **102** und ihren entsprechenden implementierten Code-Threads enthalten. Zum Beispiel kann eine Konsistenzprüfung das Bestimmen umfassen, dass es einen entsprechenden implementierten Code-Thread in dem Implementierungscodes **104** für jede in dem spezifizierten Modell **102** definierte Aufgabe gibt. Das Erkennen einer Diskrepanz in der Anzahl zwischen spezifizierten Aufgaben und entsprechenden implementierten Code-Threads kann fehlende Aufgaben von fehlenden Code-Threads angeben. Eine weitere Konsistenzprüfung kann das Bestimmen umfassen, dass ein implementierter Code-Thread das Verhalten seiner entsprechenden definierenden Aufgabe zeigt. Zum Beispiel kann ein implementierter Code-Thread, der einen Befehl enthält, der bewirkt, dass der Thread in einen Wartezustand eintritt und der in dem spezifizierten Modell **102** als ein Basisaufgabentyp definiert wurde, einen Fehler bei der Multitasking-Codierung anzeigen.

[0100] Benutzerschnittstellen wie diejenigen, die in **Fig. 16** bis **Fig. 19** gezeigt sind, können von einem Benutzer des Codeprüfsystems **100** betrachtet und verwendet werden, um Ergebnisse, die von dem Codeprüfsystem erzeugt werden, leicht zu überprüfen und zu verstehen. Anzeigen, die von dem Codeprüfsystem **100** bereitgestellt werden, sind nicht auf die in **Fig. 16** bis **Fig. 19** gezeigten beschränkt und Ergebnisse der Multitasking-Codeanalyse können auf eine Vielzahl von verschiedenen Arten gerendert werden.

[0101] Ausführungsformen der beschriebenen Technologie betreffen das Identifizieren von Rechen-Threads i Implementierungscodes, die mit Gleichzeitigkeitselementen (z. B. Aufgaben, ISRs und Ressourcen) und ihren Attributen verknüpft sind, die in einem spezifizierten Modell **102** spezifiziert sind. Das spezifizierte Modell kann eine Metamodellbeschreibung **101** einhalten und zumindest teilweise in Übereinstimmung mit der Metamodellbeschreibung und anderen Aspekten von Systemaufbaurichtlinien vorbereitet werden. Die Identifikation von Gleichzeitigkeitselementen kann auf der Identifizierung von Schlüsselbegriffen oder -einträgen in dem spezifizierten Modell basieren, die mit einem bestimmten Gleichzeitigkeitselement verknüpft sind (z. B. Defi-

nition einer Aufgabe, ISR oder Ressource und ihrer Attribute), wie oben in exemplarischen Ausführungsformen beschrieben. Das Identifizieren von Rechen-Threads Implementierungscodes **104** oder dem Vergleichscodes **104a** kann wenigstens teilweise eine automatische Codevorbereitung in Übereinstimmung mit Systemaufbaurichtlinien des Implementierungscodes **104** und/oder des Vergleichscodes **104a** umfassen. Ausführungsformen betreffen auch die Verwendung von Informationen, die für Gleichzeitigkeitselemente und entsprechende implementierte Rechen-Thread-Identitäten relevant ist, um automatisch ein Codeprüfsystem **100** zur automatisierten Multitasking-Codeanalyse zu konfigurieren, um Fehler in den Teilen des Implementierungscodes zu erkennen oder zu beweisen.

[0102] Obwohl die Beschreibung oben in Verbindung mit **Fig. 3** bis **Fig. 10B** zum automatischen Identifizieren von Gleichzeitigkeitselementen zur Verwendung beim Konfigurieren eines Codeprüfers **106** spezifizierte Modelle **102** und Implementierungscodes **104** betrifft, die bzw. der für eine AUTOSAR-Anwendung entwickelt wurde, können Aspekte der vorliegenden Ausführungsformen auf verschiedene Architekturspezifikationen und Sprachspezifikationen wie die hier erwähnten angewendet werden. In Ausführungsformen kann ein spezifiziertes Modell **102**, das für eine Mehrebenen-Systemarchitektur entwickelt wurde, wie beispielsweise eine Systemarchitektur mit drei Ebenen, die in Verbindung mit **Fig. 2** beschrieben wurde, analysiert werden, um das Vorhandensein und Identitäten von Gleichzeitigkeitselementen in dem spezifizierten Modell **102** zu bestimmen. Die Analyse kann die Verwendung von Informationen (z. B. Schemaregeln, Schlüsselbegriffen etc.) umfassen, die aus einer Architekturspezifikation **108**, einem Metamodell **101**, und/oder Stilrichtlinien **107** erhalten werden, die Datenstrukturen (wie etwa Container oder Codezeilen) innerhalb des spezifizierten Modells **102** identifizieren können, die zum Definieren von Gleichzeitigkeitselementen verwendet werden und aus denen Attribute über Gleichzeitigkeitselemente erhalten werden können. Die Analyse kann das Verwenden der Systemaufbaurichtlinien zum Verfolgen und Identifizieren eines selben Gleichzeitigkeitselements in verschiedenen Datenstrukturen umfassen, wo eine unterschiedliche Kennung für dasselbe Gleichzeitigkeitselement verwendet werden kann, und um Attributinformationen zu sammeln, die in verschiedenen Datenstrukturen für das Gleichzeitigkeitselement bereitgestellt werden. Die Analyse kann auch die Verwendung von Systemaufbaurichtlinien umfassen, welche die Bildung von Implementierungscodes aus dem spezifizierten Modell **102** leiten, um Rechen-Threads in dem Implementierungscodes zu identifizieren, die Gleichzeitigkeitselementen entsprechen, die aus dem spezifizierten Modell **102** identifiziert werden. Auf diese Weise kann Code, der mit einem Gleichzeitigkeitselement

verknüpft ist, automatisch für bzw. über verschiedene Schichten einer mehrschichtigen Softwarearchitektur **200** für ein dynamisches System oder Rechensystem identifiziert und aufgelöst werden, und Attribute, die für das Gleichzeitigkeitsselement relevant sind, können verwendet werden, um einen Codeprüfer **106** automatisch zu konfigurieren.

[0103] Fig. 20 zeigt detailliertere Komponenten eines Datenverarbeitungssystems **110**, das zum Betrieb als Codeprüfsystem **100** angepasst werden kann. Einige oder alle der gezeigten Komponenten können beispielsweise in einem Codeprüfsystem **100** vorhanden sein. In einer verteilten Rechenumgebung können sich einige Komponenten auf einem Server befinden und einige Komponenten können sich auf einer Clientvorrichtung **1140** befinden, siehe Fig. 12. In einigen Ausführungsformen kann eine Vorrichtung zum Implementieren automatisierter Konfigurationen für eine Multitasking-Codeanalyse des implementierten Codes **104** eine Rechenvorrichtung **2010** umfassen, die als ein Desktop-Computer, eine Workstation oder ein Laptop-Computer verkörpert sein kann. Geeignete Workstations umfassen unter anderen Dell Precision Workstations von Dell, Inc. aus Round Rock, Texas, die HP **Z400**, **Z600** und **Z800** Workstations von Hewlett Packard Co. aus Palo Alto, Kalifornien. Andere Rechenvorrichtungen, die verwendet werden können, umfassen Palmcomputer und andere tragbare Rechenvorrichtungen, z. B. Smartphones.

[0104] Komponenten der Rechenvorrichtung **2010** können eine Verarbeitungseinheit **2020**, einen Speicher **2030** und einen Bus **2021**, der verschiedene Komponenten einschließlich des Speichers mit der Verarbeitungseinheit **2020** koppelt, umfassen, sind jedoch nicht darauf beschränkt. Exemplarische Verarbeitungseinheiten **2020** umfassen unter anderem, ohne darauf beschränkt zu sein, Single- oder Multicore-Prozessoren wie die Core™ Pentium®- oder Celeron®-Prozessoren von Intel Corp. aus Santa Clara, Kalifornien, oder die Phenom-, AMD Athlon- oder AMD Opteron-Prozessorfamilien von Advanced Micro Devices aus Sunnyvale, Kalifornien.

[0105] Der Bus **2021** kann irgendeiner von mehreren Arten von Busstrukturen sein, einschließlich eines Speicherbusses oder Speichercontrollers, eines peripheren Busses und eines lokalen Busses, der irgendeine einer Vielzahl von Busarchitekturen verwendet. Als Beispiel und nicht als Einschränkung umfassen solche Architekturen einen Industriestandardarchitektur (ISA)-Bus, einen Mikrokanalarchitektur (MCA)-Bus, einen Enhanced ISA (EISA)-Bus, einen Video Electronics Standards Association (VESA) lokalen Bus und einen Peripheral Component Interconnect (PCI)-Bus, auch als Mezzanine-Bus bekannt.

[0106] Der Computer **2010** kann eine oder mehrere Arten von maschinenlesbaren Medien umfassen. Maschinenlesbare Medien können beliebige verfügbare Medien sein, auf die der Computer **2010** zugreifen kann, und sie umfassen sowohl flüchtige als auch nichtflüchtige, hergestellte Speichermedien, entfernbare und nicht entfernbare hergestellte Speichermedien. Als Beispiel und nicht als Einschränkung können maschinenlesbare Medien Informationen wie etwa computerlesbare Anweisungen, Datenstrukturen, Programmmodule oder andere Daten umfassen. Maschinenlesbare Medien umfassen, sind jedoch nicht beschränkt auf, RAM; ROM, EEPROM, Flash-Speicher oder andere Speichervorrichtungstechnologie, CD-ROM, Digital Versatile Disks (DVD) oder andere optische Plattenspeicher, Magnetkassetten, Magnetband, Magnetplattenspeicher oder andere Magnetspeichervorrichtungen oder irgendeine andere hergestellte Datenspeichervorrichtung, die zum Speichern der gewünschten Information verwendet werden kann und auf die der Computer **2010** zugreifen kann.

[0107] Der Speicher **2030** kann Computerspeichermedien in Form von flüchtigem und/oder nichtflüchtigem Speicher, wie zum Beispiel einen Nur-Lese-Speicher (ROM) **2031** und einen Direktzugriffsspeicher (RAM) **2032**, umfassen. Ein grundlegendes Eingabe/Ausgabe-System **2033** (BIOS), das die Grundroutinen enthält, die helfen, Informationen zwischen Elementen innerhalb des Computers **2010** zu übertragen, wie während des Starts, können im ROM **2031** gespeichert werden. Der RAM **2032** kann Daten und/oder Programmmodule enthalten, auf die durch die Verarbeitungseinheit **2020** unmittelbar zugegriffen werden kann und/oder die gegenwärtig betrieben werden. Beispielfhaft und nicht einschränkend, veranschaulicht Fig. 20 ein Betriebssystem **2034**, Anwendungsprogramme **2035**, andere Programmmodule **2036** und Programmdaten **2037**.

[0108] Der Computer **2010** kann auch andere entfernbare/nicht entfernbare, flüchtige/nichtflüchtige maschinenlesbare Medien umfassen. Nur als Beispiel zeigt Fig. 20 ein Festplattenlaufwerk **2041**, das von nicht entfernbaren, nichtflüchtigen magnetischen Medien liest oder auf diese schreibt, ein Magnetplattenlaufwerk **2051**, das von einer entfernbaren nichtflüchtigen Magnetplatte **2052** liest oder auf diese schreibt, und ein optisches Plattenlaufwerk **2055**, das von einer entfernbaren, nichtflüchtigen optischen Platte **2056**, wie einer CD-ROM oder einem anderen optischen Medium liest oder auf diese schreibt. Andere entfernbare/nicht entfernbare, flüchtige/nichtflüchtige maschinenlesbare Medien, die in der exemplarischen Betriebsumgebung verwendet werden können, umfassen, ohne darauf beschränkt zu sein, Magnetbandkassetten, Flash-Speicherkarten, Digital Versatile Disks, digitales Videoband, Festkörper-RAM Festkörper-ROM und dergleichen. Das

Festplattenlaufwerk **2041** kann mit dem Systembus **2021** über eine nicht entfernbare Speicherschnittstelle wie die Schnittstelle **2040** verbunden sein, und das Magnetplattenlaufwerk **2051** und das optische Plattenlaufwerk **2055** können mit dem Systembus **2021** über eine entfernbare Speicherschnittstelle, wie der Schnittstelle **2050** verbunden sein.

[0109] Die Laufwerke und ihre zugehörigen maschinenlesbaren Medien, die oben erörtert und in **Fig. 20** dargestellt sind, bieten eine Speicherung von maschinenlesbaren Anweisungen, Datenstrukturen, Programmmodulen und anderen Daten für den Computer **2010**. In **Fig. 20** ist beispielsweise das Festplattenlaufwerk **2041** als das Betriebssystem **2044**, Anwendungsprogramme **2045**, andere Programmmodule **2046** und Programmdateien **2047** speichernd dargestellt. Diese Komponenten können entweder dieselben oder andere sein als das Betriebssystem **2034**, Anwendungsprogramme **2035**, andere Programmmodule **2036** und Programmdateien **2037**. Das Betriebssystem **2044**, die Anwendungsprogramme **2045**, die anderen Programmmodule **2046** und die Programmdateien **2047** sind hier mit unterschiedlichen Nummern versehen, um zu veranschaulichen, dass es zumindest verschiedene Kopien sind.

[0110] Ein Benutzer kann Befehle und Informationen in den Computer **2010** durch Eingabevorrichtungen eingeben, wie eine Tastatur **2062** und eine Zeigevorrichtung **2061**, die üblicherweise als Maus, Trackball oder Touchpad bezeichnet werden. Andere Eingabevorrichtungen (nicht gezeigt) können ein Mikrofon, einen Joystick, eine Spielkonsole, eine Satellitenschüssel, einen Scanner oder dergleichen umfassen. Diese und andere Eingabevorrichtungen können mit der Verarbeitungseinheit **2020** über eine Benutzereingabeschnittstelle **2060** verbunden sein, die mit dem Systembus gekoppelt ist, aber durch andere Schnittstellen- und Busstrukturen, wie etwa eine parallele Schnittstelle, eine Spielschnittstelle oder eine universelle serielle Schnittstelle (USB) verbunden sein. Ein Monitor **2091** oder eine andere Art von Anzeigevorrichtung kann auch mit dem Systembus **2021** über eine Schnittstelle, wie etwa eine Videoschnittstelle **2090** verbunden sein. Zusätzlich zu dem Monitor kann eine Recheneinheit **2010** auch andere periphere Ausgabevorrichtungen wie Lautsprecher **2097** und Drucker **2096** umfassen, die über eine Ausgabeperipherieschnittstelle **2095** verbunden sein können.

[0111] Der Computer **2010** kann in einer vernetzten Umgebung unter Verwendung logischer Verbindungen zu einer oder mehreren entfernten Vorrichtungen arbeiten, wie zum Beispiel ein entfernter Computer **2080**. Der entfernte Computer **2080** kann ein Personal Computer, ein Server, ein Router, ein Netzwerk-PC, ein Peer-Gerät oder anderer gemeinsamer Netzwerkknoten sein, und kann viele oder alle der oben

beschriebenen Elemente relativ zu dem Computer **2010** umfassen, obwohl nur eine Speichereinrichtung **2081** in **Fig. 20** dargestellt wurde. Die logischen Verbindungen, die in **Fig. 20** gezeigt sind, umfassen ein lokales Netzwerk (LAN) **2071** und ein Weitbereichsnetzwerk (WAN) **2073**, können jedoch auch andere Netzwerke umfassen. Solche Netzwerkumgebungen können in Büros, unternehmensweiten Computernetzen, Intranets und dem Internet üblich sein. Netzwerkverbindungen können drahtgebunden, auf Lichtleitfasern basiert oder drahtlos sein.

[0112] Wenn er in einer LAN-Netzwerkumgebung verwendet wird, kann der Computer **2010** über eine Netzwerkschnittstelle oder einen Adapter **2070** mit dem LAN **2071** verbunden sein. Wenn er in einer WAN-Netzwerkumgebung verwendet wird, kann der Computer **2010** ein Modem **2072** oder andere Mittel zum Herstellen von Kommunikationen über das WAN **2073**, wie das Internet enthalten. Das Modem **2072**, das intern oder extern sein kann, kann über die Benutzereingabeschnittstelle **2060** oder einen anderen geeigneten Mechanismus mit dem Systembus **2021** verbunden sein. In einer Netzwerkumgebung können Programmmodule, die in Bezug auf den Computer **2010** oder Teile davon gezeigt sind, in einer entfernten Speichervorrichtung gespeichert sein. Beispielfähig und nicht einschränkend, zeigt **Fig. 20** entfernte Anwendungsprogramme **2085**, wie sie sich auf einer Speichervorrichtung **2081** befinden. Es versteht sich, dass die gezeigten Netzwerkverbindungen beispielhaft sind und andere Mittel zum Herstellen einer Kommunikationsverbindung zwischen den Computern verwendet werden können.

[0113] Einige Modelle, die sich auf die vorliegenden Ausführungsformen beziehen, können in einer Modellierungsumgebung **2100** entwickelt und simuliert werden, und ein Beispiel davon ist in **Fig. 21** gezeigt. Die Modellierungsumgebung **2100** kann eine Benutzerschnittstellen(UI)-Engine **2102**, einen Modelleditor **2104**, eine oder mehrere Modellelementbibliotheken **2106**, einen Codegenerator **2108** und eine Simulations-Engine **2112** umfassen. Die UI-Engine **2102** kann eine oder mehrere Benutzerschnittstellen (UIs), wie z. B. grafische Benutzerschnittstellen (GUIs) und/oder Befehlszeilenschnittstellen (CLIs), auf einer Anzeige **191** einer Datenverarbeitungsvorrichtung, wie einer Workstation, einem Laptop, einem Tablet etc. erzeugen und darstellen. Die GUIs und CLIs können eine Benutzerschnittstelle zu der Modellierungsumgebung **2100** bereitstellen, beispielsweise ein Modellbearbeitungsfenster. Der Modelleditor **2104** kann ausgewählte Operationen an einem Modell ausführen, z. B. Öffnen, Erstellen, Bearbeiten und Speichern, als Antwort auf Benutzereingaben oder programmatisch.

[0114] Die Simulations-Engine **2112** kann einen Interpreter **2116**, einen Parser **2118**, einen Modell-

Compiler **2120** und einen oder mehrere Solver bzw. Löser, wie zum Beispiel Solver bzw. Löser **2122a-c**, enthalten. Der Modell-Compiler **2120** kann einen oder mehrere Zwischendarstellungs- bzw. Intermediate Representation(IR)-Builder, wie den IR-Builder **2124** enthalten. In einigen Implementierungen können ein oder mehrere IR-Builder enthalten sein oder den Solvern **2122** zugeordnet sein. Die Simulations-Engine **2112** kann computergenerierte, ausführbare Modelle unter Verwendung eines oder mehrerer der Solver **2122a-c** ausführen, z. B. kompilieren und laufen lassen oder interpretieren. Zum Beispiel können die Solver **2122** einen Satz von Gleichungen für ein Modell generieren und können den Satz von Gleichungen lösen. Die Solver **2122** können auch eine Lösung für eine speicherinterne Zwischendarstellung (intermediate representation, IR) eines Modells generieren, das einen Satz von Gleichungen darstellt. Die Solver **2122** können die Lösung für die IR unter Verwendung numerischer Techniken erzeugen. Exemplarische Solver umfassen einen oder mehrere Fixed-Step-Continuous-Time-Solver, die numerische Integrationstechniken verwenden können, und einen oder mehrere Variable-Step-Solver, die zum Beispiel auf dem Runge-Kutta- oder Dormand-Prince-Paar basieren können. Bei einem Fixed-Step-Solver bleibt die Schrittweite während der Simulation des Modells konstant. Bei einem Variable-Step-Solver kann die Schrittweite von Schritt zu Schritt variieren, um beispielsweise Fehlertoleranzen zu erfüllen. Eine nicht erschöpfende Beschreibung geeigneter Solver findet sich im Simulink Benutzerhandbuch von The MathWorks, Inc. (Ausgabe vom März 2017).

[0115] Der Codegenerator **2108** kann auf ein Modell, wie etwa ein spezifiziertes Modell **102**, zugreifen und kann Code für das Modell generieren. In einigen Ausführungsformen kann der generierte Code ein Quellcode sein, der von dem Modell-Compiler **2120** kompiliert und von einem oder mehreren Prozessoren außerhalb der Modellierungsumgebung **2100** ausgeführt werden kann. Der generierte Code kann somit eigenständiger Code relativ zu der Modellierungsumgebung **2100** sein. Beispiele für generierten Code umfassen unter anderem Ada, Basic, C, C++, C#, FORTRAN, Assemblercode und Hardware Description Language (HDL)-Code, wie beispielsweise VHDL, Verilog oder Systeme, die dazu verwendet werden können, eine programmierbare Logikvorrichtung zu synthetisieren.

[0116] Exemplarische Modellierungsumgebungen umfassen die MATLAB® technische Rechenumgebung (TCE) und die Simulink® modellbasierte Designumgebung, beide von The MathWorks, Inc. Aus Natick, MA, als auch das Simscape™ physikalische Modellierungssystem, das SimEvent® Modellierungstool für diskrete Ereignisse und das Stateflow® Zustandsdiagrammtool, ebenfalls von The MathWorks, Inc., Embedded Coder®, ebenfalls erhältlich von The

MathWorks, Inc. aus Natick, Massachusetts, DaVinci Developer, erhältlich von Vector Informatik GmbH aus Stuttgart, Deutschland, das MapleSim physikalische Modellierungs- und Simulationstool von Waterloo Maple Inc. Aus Waterloo, Ontario, Kanada, die GTSUITE Modellierungs- und Simulationsumgebung von Gamma Technologies, LLC aus Chicago, IL, die Ricardo WAVE und WAVE RT Modellierungs- und Simulationstools von Ricardo Software aus Chicago, IL, eine Tochtergesellschaft von Ricardo plc, das AVL Boost Modellierungs- und Simulationstool der AVL GmbH aus Graz, Österreich, das LabVIEW virtuelles Instrumentenprogrammiersystem und das NI MatrixX modellbasierte Model Designprodukt, beide von National Instruments Corp. aus Austin, TX, das Visual Engineering Environment (VEE)-Produkt von Keysight Technologies Inc., Santa Rosa, CA, das System Studio modellbasierte Signalverarbeitungsalgorithmus-Design- und Analysetool und das SPW Signalverarbeitungsalgorithmustool von Synopsys, Inc. aus Mountain View, CA, ein Unified Modeling Language (UML)-System, ein Systems Modeling Language (SysML)-System, das Systemgeneratorsystem von Xilinx, Inc. aus San Jose, CA, und die Rational Rhapsody Design Manager Software von IBM Corp. Aus Somers, NY. Modelle, die in der High-Level-Modellierungsumgebung erstellt werden, können weniger Implementierungsdetails enthalten und daher auf einer höheren Ebene als bestimmte Programmiersprachen arbeiten, z. B. die Programmiersprachen C, C++, C# und Systeme.

[0117] Mit einer Modellierungsumgebung **2100** kann eine simulierte Ausführung eines Modells ausgeführt werden, z. B. um den Betrieb eines dynamischen Systems zu approximieren. Die simulierte Ausführung eines Modells kann auch als Simulieren des Modells bezeichnet werden. Modelle, die innerhalb der Modellierungsumgebung **2100** konstruiert werden, können Textmodelle, graphische Modelle, wie etwa Blockdiagramme, zustandsbasierte Modelle, Modelle für diskrete Ereignisse, physikalische Modelle und Kombinationen davon umfassen. Ein graphisches Modell kann Icons oder Blöcke enthalten, die Berechnungen, Funktionen oder Operationen darstellen, und verbindende Linien oder Pfeile zwischen den Blöcken können Daten, Signale oder Beziehungen zwischen jenen Berechnungen, Funktionen oder Operationen darstellen. Die Icons oder Blöcke können darüber hinaus von dem Benutzer aus einer oder mehreren der Bibliotheken oder Paletten **2106** ausgewählt werden, die Icons oder Blöcke für die von der Modellierungsumgebung **2100** unterstützten Blöcke enthalten. Eine Modelleditor-GUI kann eine Ausführen bzw. Run-Schaltfläche enthalten, die von dem Benutzer ausgewählt werden kann. Die Modellierungsumgebung **2100** kann auch konfiguriert sein, einen Laufbefehl zu empfangen, der von dem Benutzer eingegeben wird, z. B. in der GUI oder in einer Befehlszeilenschnittstelle (CLI). Als Antwort darauf,

dass der Benutzer die Run-Schaltfläche auswählt oder den Befehl Ausführen bzw. Run eingibt, kann die Simulations-Engine **2112** das Modell ausführen und kann die Ergebnisse der Ausführung des Modells einem Benutzer präsentieren. Exemplarische grafische Modelle umfassen unter anderem Simulink Modelle, Simscape physikalische Modelle, SimEvent Modelle, Stateflow Grafiken, LabVIEW Blockdiagramme, MatrixX Modelle, Scade Modelle und VEE Diagramme. Andere Formen von Quellprogramm umfassen unter anderem Modelica Modelle von Modelica Association, Uniform Modeling Language (UML) Modelle und Systems Modeling Language (SysML) Modelle.

[0118] Das MATLAB® TCE ist eine mathematisch orientierte Textprogrammierungsumgebung für digitale Signalverarbeitung (DSP) Design, neben anderen Verwendungen. Die Simulink® modellbasierte Designumgebung ist ein Modellierungstool für die Modellierung und Simulation dynamischer und anderer Systeme. Die MATLAB® und Simulink® Umgebungen bieten eine Reihe von High-Level-Features, welche die Entwicklung und Erforschung von Algorithmen erleichtern und modellbasiertes Design unterstützen. Exemplarische High-Level-Features umfassen unter anderem dynamische Typisierung, Array-basierte Operationen, Datentyp-Inferencing, Sample-Time-Inferencing und Execution-Order-Inferencing.

[0119] In einigen Ausführungsformen kann die Modellierungsumgebung **2100** eine deklarative Sprache implementieren. Eine deklarative Sprache ist eine Sprache, welche die Logik einer Berechnung ausdrückt, ohne ihren Kontrollfluss zu beschreiben. Eine deklarative Sprache kann beschreiben, was ein Programm in Bezug auf die Problemdomäne erreichen muss, anstatt zu beschreiben, wie es als eine Sequenz der Programmiersprachengrundelemente zu erreichen ist. In einigen Fällen kann eine deklarative Sprache eine einzelne Zuweisung implementieren, in der Variablen nur einmal zugewiesen werden. Beispiele für Sprachen, welche die deklarative Modellierung unterstützen, umfassen unter anderem die modellbasierte Simulink® Designumgebung, die eine zeitbasierte Sprache ist, die Modelica Modellierungssprache und das LabVIEW grafische Programmiersystem, Hardware Description Language (HDL), die Prolog-Sprache und die Haskell Sprache. Verhalten zumindest einiger der Modellelemente und Verbindungselemente eines Modells können Rechenimplementierungen umfassen, die implizit durch eine deklarative Sprache definiert sind.

[0120] Es sollte verstanden werden, dass die Modellierungsumgebung **2100** zu illustrativen Zwecken gedacht ist und dass die vorliegende Offenbarung mit anderen Modellierungsumgebungen verwendet werden kann. Zum Beispiel können der Codegenerator **2108** und/oder der Compiler **2120** in einigen Imple-

mentierungen von der Modellierungsumgebung **2100** getrennt sein.

[0121] Eine oder mehrere der Benutzerschnittstellen-Engine **2102**, des Modelleditors **2104**, des Codegenerators **2108**, des Modell-Compilers **2120** und der Simulations-Engine **2112** können durch ein oder mehrere Softwaremodule und/oder Bibliotheken implementiert sein, die Programmanweisungen enthalten, welche die hier beschriebenen Verfahren durchführen, wenn sie auf einer Logikschaltung eines oder mehrerer Prozessoren ausgeführt werden. Die Softwaremodule können in einem Speicher wie einem Hauptspeicher, einem persistenten Speicher und/oder einem computerlesbaren Medium einer Workstation, eines Servers oder einer anderen Datenverarbeitungsmaschine oder -vorrichtung gespeichert sein und von einem oder mehreren Prozessoren ausgeführt werden. Andere computerlesbare Medien können ebenfalls verwendet werden, um diese Programmanweisungen zu speichern und auszuführen, wie beispielsweise nicht flüchtige computerlesbare Medien einschließlich optischer, magnetischer oder magnetooptischer Medien. In einigen Ausführungsformen können eine oder mehrere der Benutzerschnittstellen-Engine **2102**, des Modelleditors **2104**, des Codegenerators **2108**, des Modell-Compilers **2120** und der Simulations-Engine **2112** Hardwareregister und Kombinationslogik umfassen, die konfiguriert und angeordnet sind, sequentielle Logikschaltungen zu erzeugen. In einigen Ausführungsformen können verschiedene Kombinationen von Software und Hardware, einschließlich Firmware, verwendet werden, um die beschriebenen Verfahren zu implementieren.

[0122] Geeignete Codegeneratoren zur Verwendung mit der vorliegenden Offenbarung umfassen, ohne darauf beschränkt zu sein, die MATLAB® Coder™-, die Simulink Coder-, die Embedded Coder- und die Simulink HDL Coder-Produkte von The MathWorks, Inc. aus Natick, MA, und das TargetLink-Produkt von dSpace GmbH aus Paderborn, Deutschland. Geeignete Zielsprachen-Compiler umfassen das xPC Target™-Tool von The MathWorks, Inc., und einen C-Sprachen-Compiler. Andere Codeerzeugungssysteme und andere Compiler können jedoch zusätzlich oder alternativ zu den für die Modellierungsumgebung **2100** beschriebenen verwendet werden.

[0123] Exemplarische Verfahren der beschriebenen Technologie umfassen Kombinationen von Prozessen (1) bis (27), wie sie nachstehend beschrieben sind. Beispiele für ein computerlesbares Medium, das gemäß der beschriebenen Technologie implementiert werden kann, sind durch die Konfiguration (28) angegeben. Ein exemplarisches System, das gemäß der beschriebenen Technologie implementiert werden kann, ist durch die Konfiguration (30) angegeben.

(1) Ein computerimplementierter Prozess zum Prüfen von Computercode, wobei der Prozess umfasst: Empfangen, durch einen Datenanalysator, einer oder mehrerer Datenstrukturen, die Informationen enthalten, die Aufgaben spezifizieren, die in Form von Teilen von Implementierungscode zu implementieren sind, der ein dynamisches System steuert bzw. regelt, wobei ein Teil der Aufgaben gleichzeitig auf einem Prozessor des dynamischen Systems unter Verwendung verschiedener Rechen-Threads innerhalb der Bestandteile des Implementierungs-codes auszuführen sind; automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um spezifizierte Aufgabenkennungen für jede Aufgabe des Teils der Aufgaben zu identifizieren; automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um Aufgabenattribute zu identifizieren, die für die gleichzeitige Ausführung des Teils der Aufgaben auf dem Prozessor des dynamischen Systems relevant sind; und Bereitstellen von Informationen bezüglich der Aufgabenidentität und der Aufgabenattribute des Teils der Aufgaben zum automatischen Durchführen einer Codeprüfung der Bestandteile des Implementierungs-codes, um Defekte in den Teilen des Implementierungs-codes zu detektieren oder deren Abwesenheit zu beweisen.

(2) Der Prozess von (1), wobei die Aufgabenattribute über mehrere disjunkte Datenstrukturen oder an disjunkten Stellen innerhalb einer Datenstruktur verteilt sind.

(3) Der Prozess von (1) oder (2), wobei die Aufgabenattribute Informationen über Typen von Vorkommnissen umfassen, die jede Aufgabe in dem Teil von Aufgaben initiieren.

(4) Der Prozess von (3), wobei ein erster Typ von Vorkommnis ein Ereignis ist, das mit einer Zustandsänderung an einem Datenport verknüpft ist, der mit einer Aufgabe des Teils von Aufgaben verknüpft ist, und ein zweiter Typ von Vorkommnis ein Alarm ist, der mit einer Aufgabe des Teils von Aufgaben verknüpft ist.

(5) Der Prozess von (3) oder (4), ferner umfassend das Bestimmen einer Bezeichnung für zumindest eine Aufgabe des Teils von Aufgaben basierend auf einem Typ von Vorkommnis, das die Aufgabe initiiert.

(6) Der Prozess nach einem von (1) bis (5), wobei die Aufgabenattribute Informationen über Softwareressourcen umfassen, die von dem Teil von Aufgaben verwendet werden.

(7) Der Prozess nach einem von (1) bis (6), ferner umfassend das automatische Analysieren, durch den Datenanalysator, der einen oder

mehreren Datenstrukturen, um Interrupt Service Routinen (ISRs) zu identifizieren, die gleichzeitig mit den Aufgaben auf dem Prozessor des dynamischen Systems ausgeführt werden können, und Bereitstellen von Informationen bezüglich der ISRs zum automatischen Durchführen einer statischen Codeanalyse.

(8) Der Prozess nach einem von (1) bis (7), ferner umfassend das automatische Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um Ressourcen zu identifizieren, die von dem Teil von Aufgaben verwendet werden können, und Bereitstellen von Informationen bezüglich der Ressourcen zum automatischen Durchführen einer statischen Codeanalyse.

(9) Der Prozess nach (7) oder (8), ferner umfassend: Empfangen, durch ein Codeprüfsystem, der einen oder mehreren Datenstrukturen; Empfangen, durch das Codeprüfsystem, eines Abschnitts des Implementierungs-codes, der die Bestandteile des Implementierungs-codes enthält; Empfangen, durch das Codeprüfsystem, der Informationen über die Aufgabenidentität und die Aufgabenattribute; Empfangen, durch das Codeprüfsystem, der Informationen über die ISRs; Verarbeiten, durch einen Codeverfolgungsprozessor, zumindest eines Abschnitts der einen oder mehreren Datenstrukturen, um automatisch Vergleichscode unter Verwendung einer oder mehrerer geänderter Aufgabenkennungen für die spezifizierten Aufgabenkennungen zu erzeugen; Analysieren des Vergleichs-codes und des empfangenen Abschnitts des Implementierungs-codes, um die Rechen-Threads in den Teilen des Implementierungs-codes zu identifizieren, der den spezifizierten Aufgabenkennungen entspricht; Verknüpfen der empfangenen Aufgabenattribute mit ihren entsprechenden Rechen-Threads; und statisches Analysieren, durch das Codeprüfsystem, der Bestandteile des Implementierungs-codes basierend auf den verknüpften Aufgabenattributen und empfangenen Informationen bezüglich der ISRs, um Defekte in den Teilen des Implementierungs-codes zu detektieren oder deren Abwesenheit zu beweisen.

(10) Der Prozess von (9), wobei die Defekte Laufzeitfehler umfassen.

(11) Der Prozess nach (9) oder (10), ferner umfassend das automatische Konfigurieren des Codeprüfsystems basierend auf den empfangenen Aufgabenattributen, um zumindest einen Rechen-Thread als einen grundlegenden Aufgabentyp oder einen erweiterten Aufgabentyp zu erkennen.

(12) Der Prozess nach einem von (9) bis (11), ferner umfassend das Empfangen, durch das

Codeprüfsystem, zumindest eines Attributs für eine Aufgabe, die ausgewählt ist aus der Gruppe bestehend aus: einem Aufgabentyp, einem Prioritätswert für die Aufgabe und einem Typ von Vorkommnis, das die Aufgabe aktiviert.

(13) Der Prozess nach einem von (9) bis (12), ferner umfassend das automatische Konfigurieren des Codeprüfsystems basierend auf den empfangenen Informationen über die ISRs, um zumindest einen Rechen-Thread als eine Interrupt Service Routine zu erkennen.

(14) Der Prozess nach einem von (9) bis (13), ferner umfassend das Empfangen, durch das Codeprüfsystem, zumindest eines Attributs für eine ISR, die ausgewählt ist aus der Gruppe bestehend aus: einem ISR-Typ, einem Prioritätswert für die ISR und einem Typ von Vorkommnis, das die ISR aktiviert.

(15) Der Prozess nach einem von (9) bis (14), ferner umfassend das automatische Konfigurieren des Codeprüfsystems basierend auf empfangenen Informationen über Ressourcen, um in zumindest einem Rechen-Thread einen Beginn eines Abschnitts eines Codes mit geschütztem Zugriff zu erkennen.

(16) Der Prozess nach einem von (9) bis (15), ferner umfassend das Empfangen, durch das Codeprüfsystem, zumindest eines Attributs für eine Ressource, die ausgewählt ist aus der Gruppe bestehend aus: einer Kennung, die einen Anfang eines Abschnitts von Code mit geschütztem Zugriff markiert, und einer Kennung, die ein Ende des Abschnitts von Code mit geschütztem Zugriff markiert.

(17) Der Prozess nach einem von (9) bis (16), ferner umfassend das Identifizieren von ungeschützten Variablen, die von mehreren Aufgaben des Teils von Aufgaben geteilt bzw. gemeinsam genutzt werden.

(18) Der Prozess nach einem von (9) bis (17), wobei die eine oder die mehreren Datenstrukturen eine oder mehrere Datenstrukturen von Zwischencode umfassen.

(19) Der Prozess nach einem von (9) bis (18), ferner umfassend das Bereitstellen, durch das Codeprüfsystem, von Konsistenzinformationen, die zumindest ein Maß an Konsistenz zwischen den spezifizierten Aufgabenkennungen in der einen oder den mehreren Datenstrukturen und den Rechen-Threads in den Teilen des Implementierungscodes identifizieren.

(20) Der Prozess von (19), wobei ein erstes Maß an Konsistenz eine Konsistenz zwischen einer ersten Anzahl der spezifizierten Aufgabenkennungen und einer zweiten Anzahl der Re-

chen-Threads oder Rechen-Thread-Kennungen für die Rechen-Threads umfasst.

(21) Der Prozess von (19) oder (20), wobei ein zweites Maß an Konsistenz eine Konsistenz zwischen Typen von Aufgaben, die für die spezifizierten Aufgabenkennungen aus der einen oder den mehreren Datenstrukturen bestimmt werden, und Typen der Rechen-Threads umfasst.

(22) Der Prozess nach einem von (9) bis (21), ferner umfassend das Bestimmen aus den Aufgabenattributen, ob jede Aufgabe des Teils von Aufgaben durch ein Ereignis oder einen Alarm aktiviert wird; und Bereitstellen von Informationen, die für die Aktivierung jeder Aufgabe des Teils von Aufgaben zum automatischen Durchführen einer statischen Codeanalyse der Bestandteile des Implementierungscodes relevant sind.

(23) Der Prozess von (22), ferner umfassend das Speichern im Speicher der Informationen über die Aufgabenidentität und die Aufgabenattribute, der Informationen über die ISRs, implementierte Rechen-Thread-Kennungen und Informationen über die Aktivierung jeder Aufgabe als Konfigurationsdaten für ein Codeprüfsystem.

(24) Der Prozess nach einem von (1) bis (23), wobei die eine oder die mehreren Datenstrukturen Datenstrukturen eines spezifizierten Modells sind und die Bestandteile des Implementierungscodes aus der einen oder den mehreren Datenstrukturen gemäß einer standardisierten Methodik vorbereitet werden.

(25) Der Prozess nach einem von (1) bis (24), wobei die eine oder die mehreren Datenstrukturen ARXML-Dateien eines AUTOSAR-kompatiblen spezifizierten Modells sind.

(26) Der Prozess nach einem von (1) bis (24), wobei das spezifizierte Modell ein spezifiziertes Modell ist, das mit OSEK, UML, SysML, AADL, MARTE oder ARINC kompatibel ist.

(27) Der Prozess nach einem von (1) bis (26), ferner umfassend das Durchführen einer statischen Analyse der Bestandteile des Implementierungscodes, wobei die statische Analyse eine oder eine Kombination von abstrakter Interpretation, Modellprüfung und Erfüllungslösung umfasst.

(28) Computerlesbares Medium, das Anweisungen enthält, die, wenn sie von zumindest einem Prozessor ausgeführt werden, ein System, das den zumindest einen Prozessor enthält, dahingehend anpassen, Schritte nach einem der Prozesse (1) bis (27) durchzuführen.

(29) Ein computerimplementierter Prozess zum Prüfen von Computercode, wobei der Prozess umfasst: Empfangen, durch einen Datenanaly-

sator, einer oder mehrerer Datenstrukturen, die Informationen enthalten, die Interrupt Service Routinen (ISRs) spezifizieren, die in Form von Bestandteilen des Implementierungscodes zu implementieren sind, der ein dynamisches System steuert bzw. regelt, wobei eine erste ISR der ISRs mit einem ersten Rechen-Thread innerhalb der Bestandteile des Implementierungscodes verknüpft ist, der gleichzeitig auf einem Prozessor des dynamischen Systems mit zumindest einem zweiten anderen Rechen-Thread ausgeführt wird, automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um eine ISR-Kennung für die erste ISR zu identifizieren; und Bereitstellen von Informationen bezüglich der Identität der ersten ISR zum automatischen Durchführen einer Codeprüfung der Bestandteile des Implementierungscodes, um Defekte in den Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen.

(30) Ein Codeanalysesystem, umfassend zumindest einen Prozessor, der mit ausführbaren Anweisungen angepasst ist, um eine oder mehrere Datenstrukturen zu empfangen, die Informationen enthalten, die Aufgaben spezifizieren, die in Form von Teilen eines Implementierungscodes zu implementieren sind, der ein dynamisches System steuert bzw. regelt, wobei ein Teil der Aufgaben gleichzeitig auf einem Prozessor des dynamischen Systems unter Verwendung verschiedener Rechen-Threads innerhalb der Bestandteile des Implementierungscodes auszuführen sind; automatisches Analysieren der einen oder mehreren Datenstrukturen, um für jede Aufgabe des Teils der Aufgaben spezifizierte Aufgabenkennungen zu identifizieren; automatisches Analysieren der einen oder mehreren Datenstrukturen, um Aufgabenattribute zu identifizieren, die für die gleichzeitige Ausführung des Teils der Aufgaben auf dem Prozessor des dynamischen Systems relevant sind; und Bereitstellen von Informationen bezüglich der Aufgabenidentität und der Aufgabenattribute des Teils der Aufgaben zum automatischen Durchführen einer Codeprüfung der Bestandteile des Implementierungscodes, um Defekte in den Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen.

[0124] Die gesamte Literatur und ähnliches Material, die in dieser Anmeldung zitiert werden, einschließlich, aber nicht beschränkt auf Patente, Patentanmeldungen, Artikel, Bücher, Abhandlungen und Webseiten, ungeachtet des Formats solcher Literatur und ähnlicher Materialien, werden ausdrücklich durch Bezugnahme in ihrer Gesamtheit aufgenommen. Für den Fall, dass eine oder mehrere der aufgenommenen Literatur und ähnlichen Materialien von dieser Anmeldung abweichen oder ihr widersprechen, ein-

schließlich, aber nicht beschränkt auf definierte Begriffe, Begriffsverwendung, beschriebene Techniken oder dergleichen, steuert diese Anwendung.

[0125] Die hier verwendeten Abschnittsüberschriften dienen nur zu organisatorischen Zwecken und sind nicht so zu verstehen, dass sie den beschriebenen Gegenstand in irgendeiner Weise beschränken.

[0126] Während die vorliegenden Lehren in Verbindung mit verschiedenen Ausführungsformen und Beispielen beschrieben wurden, ist es nicht beabsichtigt, dass die vorliegenden Lehren auf solche Ausführungsformen oder Beispiele beschränkt sind. Im Gegenteil umfassen die vorliegenden Lehren verschiedene Alternativen, Modifikationen und Äquivalente, wie sie für den Fachmann ersichtlich sind.

[0127] Die Ansprüche sollten nicht so verstanden werden, dass sie auf die beschriebene Reihenfolge oder die beschriebenen Elemente beschränkt sind, sofern nicht in diesem Sinne angegeben. Es ist ersichtlich, dass verschiedene Änderungen an Form und Detail von einem Durchschnittsfachmann vorgenommen werden können, ohne von dem Wesen und dem Umfang der beigefügten Ansprüche abzuweichen. Alle Ausführungsformen, die in das Wesen und den Umfang der folgenden Ansprüche und Äquivalente dazu fallen, werden beansprucht.

ZITATE ENTHALTEN IN DER BESCHREIBUNG

Diese Liste der vom Anmelder aufgeführten Dokumente wurde automatisiert erzeugt und ist ausschließlich zur besseren Information des Lesers aufgenommen. Die Liste ist nicht Bestandteil der deutschen Patent- bzw. Gebrauchsmusteranmeldung. Das DPMA übernimmt keinerlei Haftung für etwaige Fehler oder Auslassungen.

Zitierte Patentliteratur

- EP 17290155 [0016, 0058, 0061, 0091]

Zitierte Nicht-Patentliteratur

- ISO 26262 [0015]

Patentansprüche

1. Computerimplementiertes Verfahren zum Prüfen von Computercode, wobei das Verfahren umfasst:

Empfangen, durch einen Datenanalysator, einer oder mehrerer Datenstrukturen, die Informationen enthalten, die Aufgaben spezifizieren, die in Form von Bestandteilen von Implementierungscode zu implementieren sind, der ein dynamisches System steuert bzw. regelt, wobei ein Teil der Aufgaben gleichzeitig auf einem Prozessor des dynamischen Systems unter Verwendung verschiedener Rechen-Threads innerhalb der Bestandteile des Implementierungscodes auszuführen sind;

automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um spezifizierte Aufgabenkennungen für jede Aufgabe des Teils der Aufgaben zu identifizieren;

automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um Aufgabenattribute zu identifizieren, die für die gleichzeitige Ausführung des Teils der Aufgaben auf dem Prozessor des dynamischen Systems relevant sind; und

Bereitstellen von Informationen bezüglich der Aufgabenidentität und der Aufgabenattribute des Teils der Aufgaben zum automatischen Durchführen einer Codeprüfung der Bestandteile des Implementierungscodes, um Defekte in den Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen.

2. Verfahren nach Anspruch 1, wobei die Aufgabenattribute über mehrere disjunkte Datenstrukturen oder an disjunkten Stellen innerhalb einer Datenstruktur verteilt sind.

3. Verfahren nach Anspruch 1, wobei die Aufgabenattribute Informationen über Typen von Vorkommnissen umfassen, die jede Aufgabe in dem Teil von Aufgaben initiieren.

4. Verfahren nach Anspruch 3, wobei ein erster Typ von Vorkommnis ein Ereignis ist, das mit einer Zustandsänderung an einem Datenport verknüpft ist, der mit einer Aufgabe des Teils von Aufgaben verknüpft ist, und ein zweiter Typ von Vorkommnis ein Alarm ist, der mit einer Aufgabe des Teils von Aufgaben verknüpft ist.

5. Verfahren nach Anspruch 3, ferner umfassend Bestimmen einer Bezeichnung für zumindest eine Aufgabe des Teils von Aufgaben basierend auf einem Typ von Vorkommnis, das die Aufgabe initiiert.

6. Verfahren nach Anspruch 1, wobei die Aufgabenattribute Informationen über Softwareressourcen umfassen, die von dem Teil von Aufgaben verwendet werden.

7. Verfahren nach einem der Ansprüche 1 bis 6, ferner umfassend:

automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um Interrupt Service Routinen (ISRs) zu identifizieren, die gleichzeitig mit den Aufgaben auf dem Prozessor des dynamischen Systems ausführen können; und Bereitstellen von Informationen bezüglich der ISRs zum automatischen Durchführen einer statischen Codeanalyse.

8. Verfahren nach einem der Ansprüche 1 bis 6, ferner umfassend:

automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um Ressourcen zu identifizieren, die von dem Teil von Aufgaben verwendet werden können; und Bereitstellen von Informationen bezüglich der Ressourcen zum automatischen Durchführen einer statischen Codeanalyse.

9. Verfahren nach Anspruch 7 oder 8, ferner umfassend:

Empfangen, durch ein Codeprüfsystem, der einen oder mehreren Datenstrukturen;

Empfangen, durch das Codeprüfsystem, eines Abschnitts des Implementierungscodes, der die Bestandteile des Implementierungscodes enthält;

Empfangen, durch das Codeprüfsystem, der Informationen über die Aufgabenidentität und die Aufgabenattribute;

Empfangen, durch das Codeprüfsystem, der Informationen über die ISRs; Verarbeiten, durch einen Codeverfolgungsprozessor, zumindest eines Teils der einen oder mehreren Datenstrukturen, um automatisch Vergleichscode unter Verwendung einer oder mehrerer geänderter Aufgabenkennungen für die spezifizierten Aufgabenkennungen zu erzeugen;

Analysieren des Vergleichscodes und des empfangenen Abschnitts des Implementierungscodes, um die Rechen-Threads in den Teilen des Implementierungscodes zu identifizieren, der den spezifizierten Aufgabenkennungen und ISRs entspricht;

Verknüpfen der empfangenen Aufgabenattribute mit ihren entsprechenden Rechen-Threads; und

statisches Analysieren, durch das Codeprüfsystem, der Bestandteile des Implementierungscodes basierend auf den verknüpften Aufgabenattributen und empfangenen Informationen bezüglich der ISRs, um Defekte in den Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen.

10. Verfahren nach Anspruch 9, wobei die Defekte Laufzeitfehler umfassen.

11. Verfahren nach Anspruch 9, ferner umfassend automatisches Konfigurieren des Codeprüfsystems basierend auf den empfangenen Aufgabenattributen, um zumindest einen Rechen-Thread als einen grund-

legenden Aufgabentyp oder einen erweiterten Aufgabentyp zu erkennen.

12. Verfahren nach Anspruch 11, ferner umfassend Empfangen, durch das Codeprüfsystem, zumindest eines Attributs für eine Aufgabe, die ausgewählt ist aus der Gruppe bestehend aus: einem Aufgabentyp, einem Prioritätswert für die Aufgabe und einem Typ von Vorkommnis, das die Aufgabe aktiviert.

13. Verfahren nach Anspruch 9, ferner umfassend automatisches Konfigurieren des Codeprüfsystems basierend auf den empfangenen Informationen über die ISRs, um zumindest einen Rechen-Thread als eine Interrupt Service Routine zu erkennen.

14. Verfahren nach Anspruch 13, ferner umfassend Empfangen, durch das Codeprüfsystem, zumindest eines Attributs für eine ISR, die ausgewählt ist aus der Gruppe bestehend aus: einem ISR-Typ, einem Prioritätswert für die ISR und einem Typ von Vorkommnis, das die ISR aktiviert.

15. Verfahren nach Anspruch 9, ferner umfassend automatisches Konfigurieren des Codeprüfsystems basierend auf empfangenen Informationen über Ressourcen, um in zumindest einem Rechen-Thread einen Beginn eines Abschnitts eines Codes mit geschütztem Zugriff zu erkennen.

16. Verfahren nach Anspruch 15, ferner umfassend Empfangen, durch das Codeprüfsystem, zumindest eines Attributs für eine Ressource, die ausgewählt ist aus der Gruppe bestehend aus: einer Kennung, die einen Anfang eines Abschnitts von Code mit geschütztem Zugriff markiert, und einer Kennung, die ein Ende des Abschnitts von Code mit geschütztem Zugriff markiert.

17. Verfahren nach Anspruch 9, ferner umfassend Identifizieren von ungeschützten Variablen, die von mehreren Aufgaben des Teils von Aufgaben geteilt bzw. gemeinsam genutzt werden.

18. Verfahren nach Anspruch 9, wobei die eine oder die mehreren Datenstrukturen eine oder mehrere Datenstrukturen von Zwischencode umfassen.

19. Verfahren nach Anspruch 9, ferner umfassend Bereitstellen, durch das Codeprüfsystem, von Konsistenzinformationen, die zumindest ein Maß an Konsistenz zwischen den spezifizierten Aufgabekennungen in der einen oder den mehreren Datenstrukturen und den Rechen-Threads in den Teilen des Implementierungscodes identifizieren.

20. Verfahren nach Anspruch 19, wobei ein erstes Maß an Konsistenz eine Konsistenz zwischen einer ersten Anzahl der spezifizierten Aufgabekennungen und einer zweiten Anzahl der Rechen-Threads oder

Rechen-Thread-Kennungen für die Rechen-Threads umfasst.

21. Verfahren nach Anspruch 19, wobei ein zweites Maß an Konsistenz eine Konsistenz zwischen Typen von Aufgaben, die für die spezifizierten Aufgabekennungen aus der einen oder den mehreren Datenstrukturen bestimmt werden, und Typen der Rechen-Threads umfasst.

22. Verfahren nach Anspruch 9, ferner umfassend: Bestimmen aus den Aufgabenattributen, ob jede Aufgabe des Teils von Aufgaben durch ein Ereignis oder einen Alarm aktiviert wird; und Bereitstellen von Informationen, die für die Aktivierung jeder Aufgabe des Teils von Aufgaben zum automatischen Durchführen einer statischen Codeanalyse der Bestandteile des Implementierungscodes relevant sind.

23. Verfahren nach Anspruch 22, ferner umfassend Speichern im Speicher der Informationen über Aufgabenidentität und die Aufgabenattribute, der Informationen über die ISRs, implementierte Rechen-Thread-Kennungen und Informationen über die Aktivierung jeder Aufgabe als Konfigurationsdaten für ein Codeprüfsystem.

24. Verfahren nach einem der Ansprüche 1 bis 23, wobei die eine oder die mehreren Datenstrukturen Datenstrukturen eines spezifizierten Modells sind und die Bestandteile des Implementierungscodes aus der einen oder den mehreren Datenstrukturen gemäß einer standardisierten Methodik vorbereitet werden.

25. Verfahren nach Anspruch 24, wobei die eine oder die mehreren Datenstrukturen ARXML-Dateien eines AUTOSAR-kompatiblen spezifizierten Modells sind.

26. Verfahren nach Anspruch 24, wobei das spezifizierte Modell ein spezifiziertes Modell ist, das mit OSEK, UML, SysML, AADL, MARTE oder ARINC kompatibel ist.

27. Verfahren nach Anspruch 1, ferner umfassend Durchführen einer statischen Analyse der Bestandteile des Implementierungscodes, wobei die statische Analyse eine oder eine Kombination von abstrakter Interpretation, Modellprüfung und Erfüllungslösung umfasst.

28. Computerlesbares Medium, das Anweisungen enthält, die, wenn sie von zumindest einem Prozessor ausgeführt werden, ein System, das den zumindest einen Prozessor enthält, dahingehend anpassen, Vorgänge nach einem der Ansprüche 1 bis 27 durchzuführen.

29. Computerimplementiertes Verfahren zum Prüfen von Computercode, wobei das Verfahren umfasst:

Empfangen, durch einen Datenanalysator, einer oder mehrerer Datenstrukturen, die Informationen enthalten, die Interrupt Service Routinen (ISRs) spezifizieren, die in Form von Bestandteilen des Implementierungscodes zu implementieren sind, der ein dynamisches System steuert bzw. regelt, wobei eine erste ISR der ISRs mit einem ersten Rechen-Thread innerhalb der Bestandteile des Implementierungscodes verknüpft ist, der gleichzeitig auf einem Prozessor des dynamischen Systems mit zumindest einem zweiten anderen Rechen-Thread ausführt; automatisches Analysieren, durch den Datenanalysator, der einen oder mehreren Datenstrukturen, um eine ISR-Kennung für die erste ISR zu identifizieren; und

Bereitstellen von Informationen bezüglich der Identität der ersten ISR zum automatischen Durchführen einer Codeprüfung der Bestandteile des Implementierungscodes, um Defekte in den Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen.

30. Codeanalysesystem, umfassend zumindest einen Prozessor, der mit ausführbaren Anweisungen angepasst ist zum:

Empfangen einer oder mehrerer Datenstrukturen, die Informationen enthalten, die Aufgaben spezifizieren, die in Form von Teilen eines Implementierungscodes zu implementieren sind, der ein dynamisches System steuert bzw. regelt, wobei ein Teil der Aufgaben gleichzeitig auf einem Prozessor des dynamischen Systems unter Verwendung verschiedener Rechen-Threads innerhalb der Bestandteile des Implementierungscodes auszuführen sind;

automatisches Analysieren der einen oder mehreren Datenstrukturen, um für jede Aufgabe des Teils der Aufgaben spezifizierte Aufgabenkennungen zu identifizieren;

automatisches Analysieren der einen oder mehreren Datenstrukturen, um Aufgabenattribute zu identifizieren, die für die gleichzeitige Ausführung des Teils der Aufgaben auf dem Prozessor des dynamischen Systems relevant sind; und

Bereitstellen von Informationen bezüglich der Aufgabenidentität und der Aufgabenattribute des Teils der Aufgaben zum automatischen Durchführen einer Codeprüfung der Bestandteile des Implementierungscodes, um Defekte in den Teilen des Implementierungscodes zu detektieren oder deren Abwesenheit zu beweisen.

Es folgen 23 Seiten Zeichnungen

Anhängende Zeichnungen

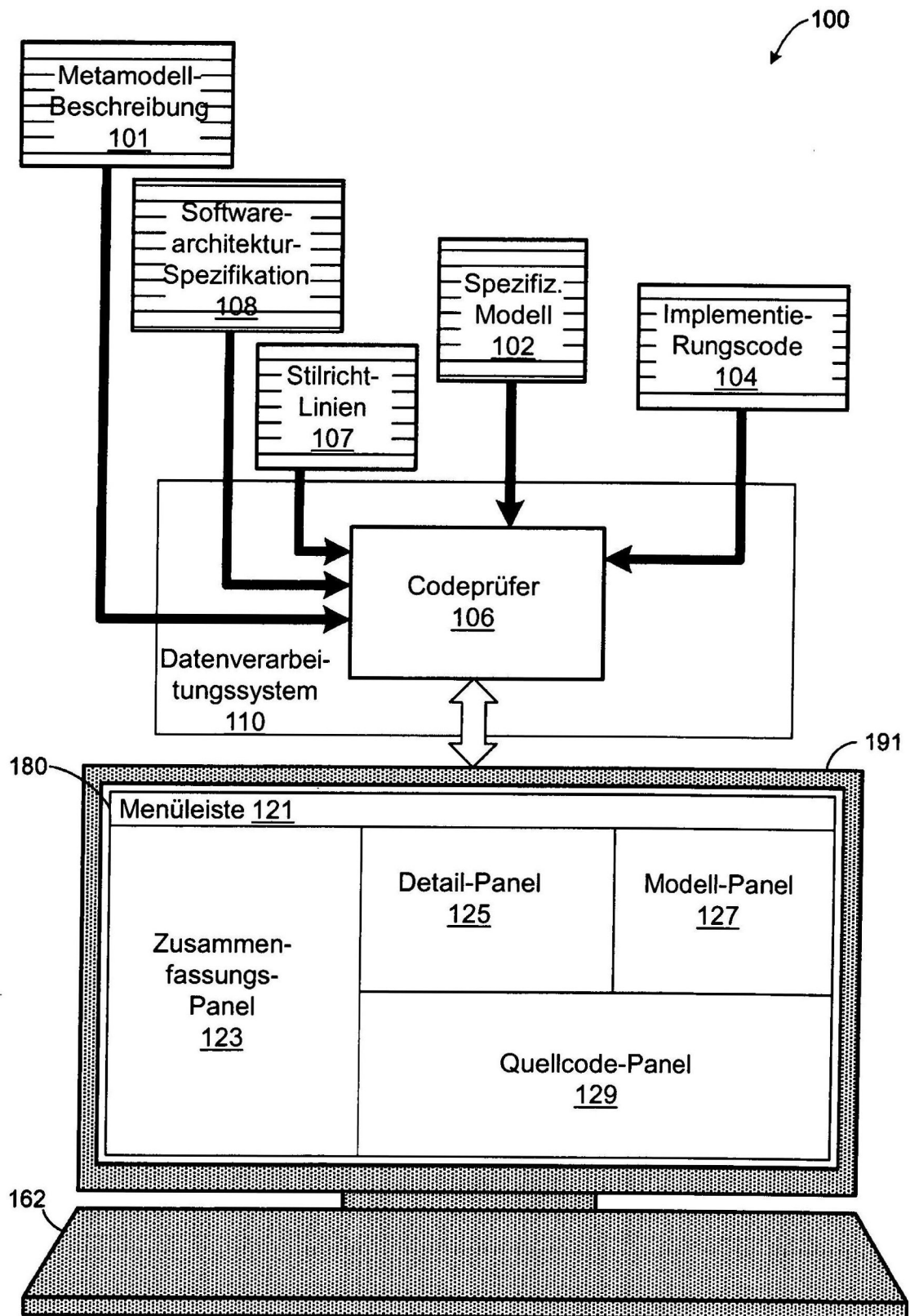


FIG. 1

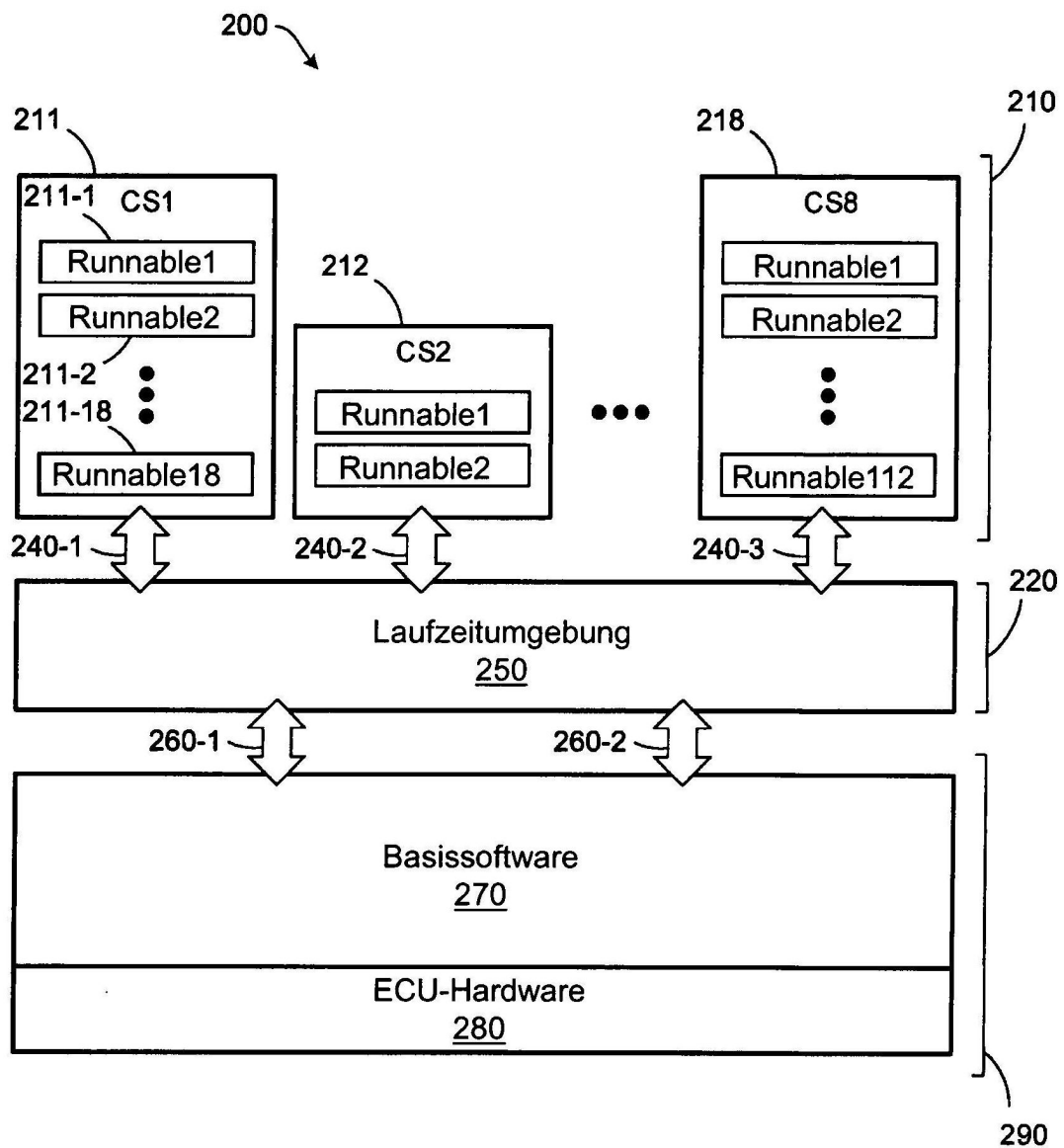


FIG. 2

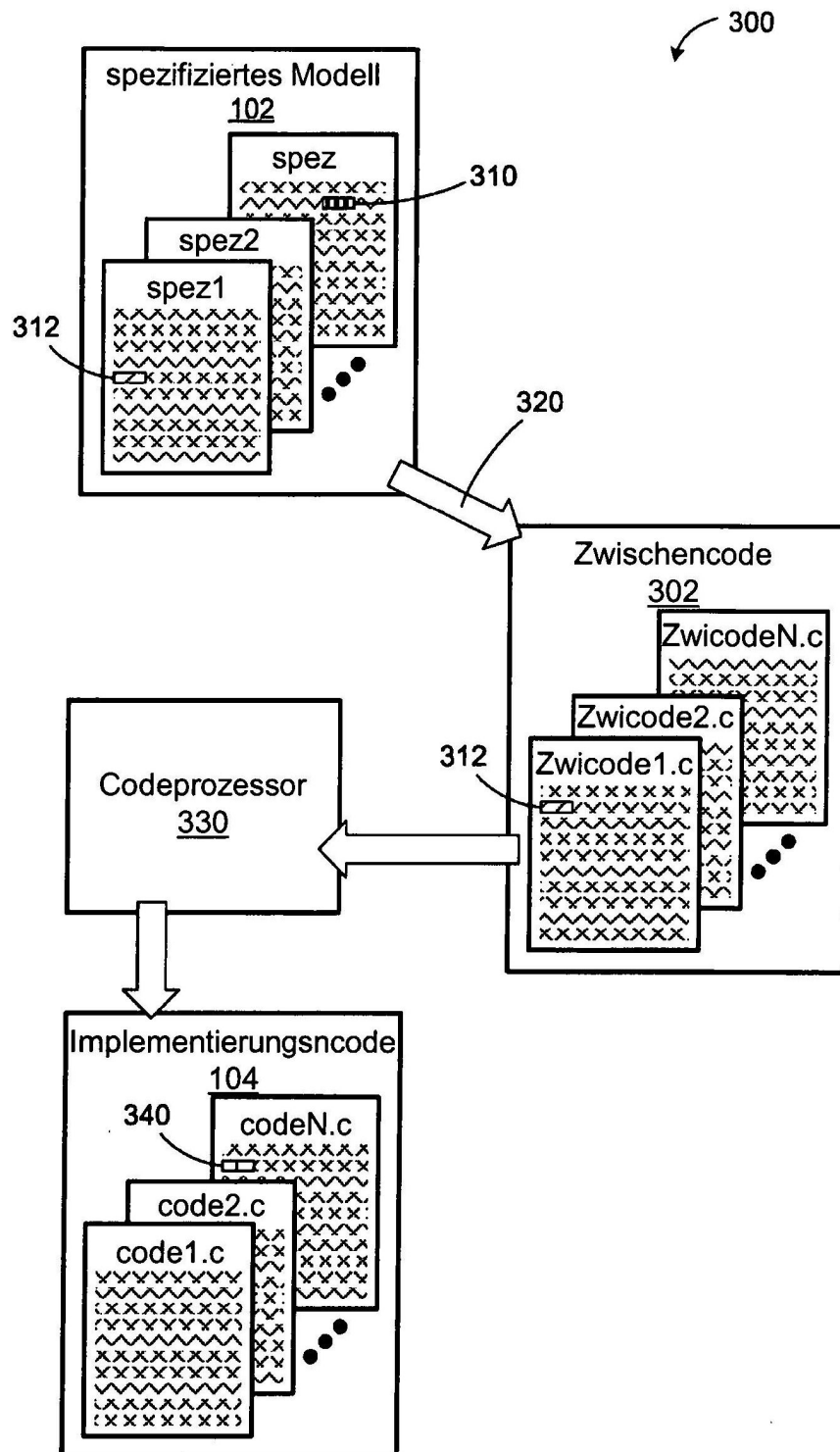


FIG. 3

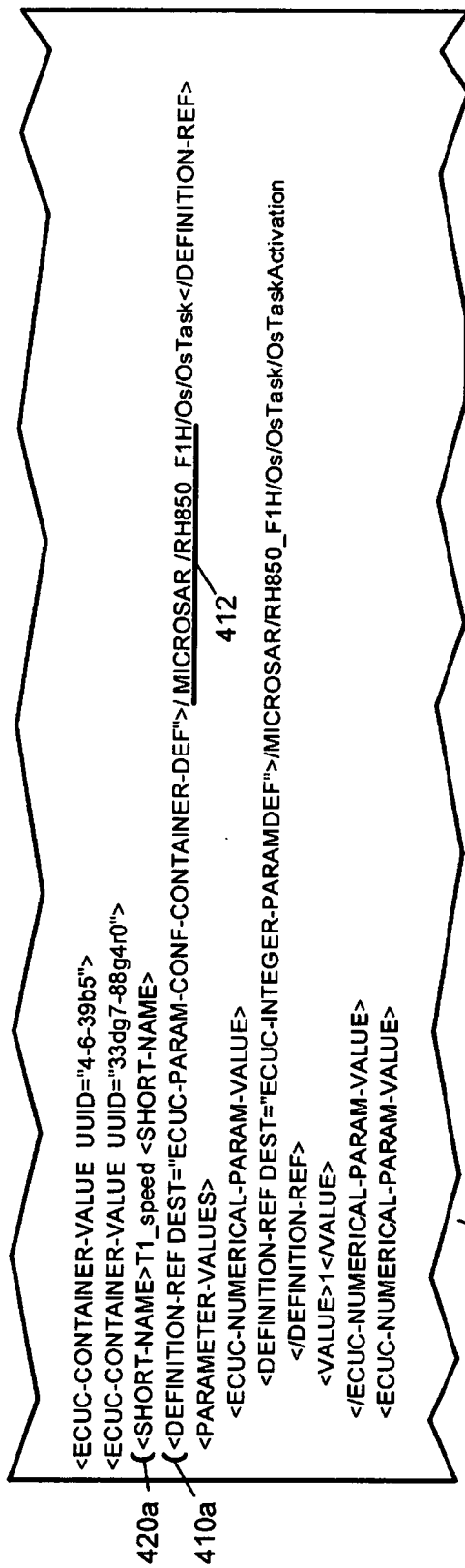


FIG. 4A

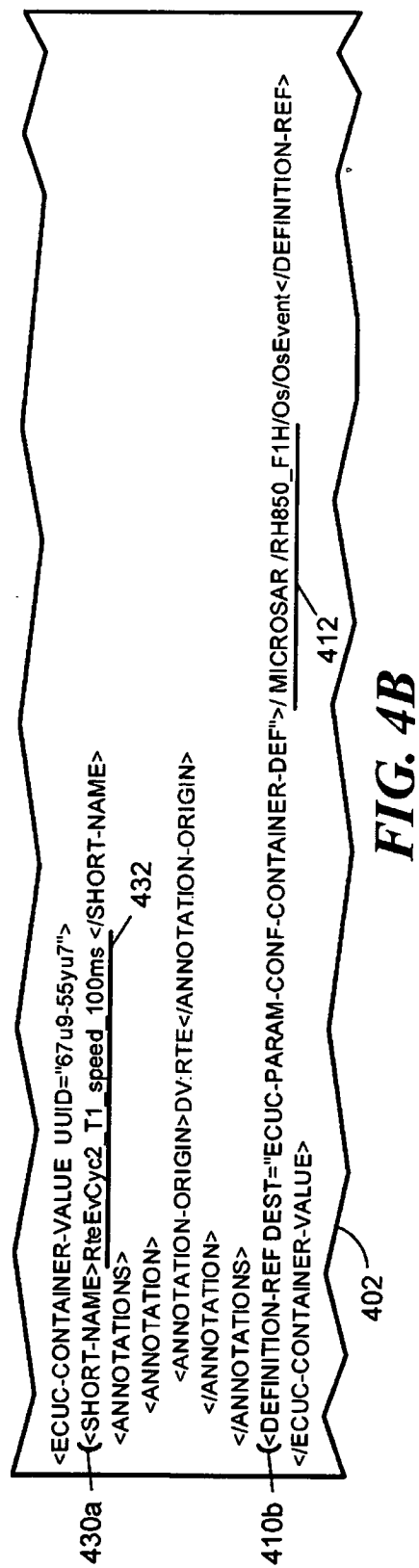


FIG. 4B

```

450 <ECUC-CONTAINER-VALUE UUID="6f8bf641-d96d-46c7-b515-b04436e81845">
440 (<SHORT-NAME>Cpod_MainFunc_Timing1</SHORT-NAME>
    (<DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/MICROSAR/Rte/RteBswModuleInstance/
      RteBswEventToTaskMapping</DEFINITION-REF>
    <PARAMETER-VALUES>
      <ECUC-NUMERICAL-PARAM-VALUE>
        <DEFINITION-REF DEST="ECUC-BOOLEAN-PARAM-DEF">/MICROSAR/Rte/RteBswModuleInstance/RteBswEventToTaskMapping/
          <VALUE>false</VALUE>
      </ECUC-NUMERICAL-PARAM-VALUE>
      <ECUC-NUMERICAL-PARAM-VALUE>
        <DEFINITION-REF DEST="ECUC-INTEGGER-PARAM-DEF">/MICROSAR/Rte/RteBswModuleInstance/RteBswEventToTaskMapping/
          <VALUE>7</VALUE>
      </ECUC-NUMERICAL-PARAM-VALUE>
    </PARAMETER-VALUES>
    <REFERENCE-VALUES>
      <ECUC-REFERENCE-VALUE>
        <DEFINITION-REF DEST="ECUC-FOREIGN-REFERENCE-DEF">/MICROSAR/Rte/RteBswModuleInstance/
          <VALUE-REF DEST="BSW-TIMING-EVENT">/MICROSAR/Cpod_ib_bswmd/BswModuleDescriptions/Cpod/CpodBehavior/
            </ECUC-REFERENCE-VALUE>
            <ECUC-REFERENCE-VALUE>
              <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/MICROSAR/Rte/RteBswModuleInstance/RteBswEventToTaskMapping/
                <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/ActiveEcuC/Os/T1_speed_100ms</VALUE-REF>
              </ECUC-REFERENCE-VALUE>
              <ECUC-REFERENCE-VALUE>
                <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/MICROSAR/Rte/RteBswModuleInstance/RteBswEventToTaskMapping/
                  <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/ActiveEcuC/Os/RteEvCyc2_T1_speed_100ms</VALUE-REF>
                </ECUC-REFERENCE-VALUE>
              </ECUC-REFERENCE-VALUE>
            </ECUC-REFERENCE-VALUE>
          </ECUC-REFERENCE-VALUE>
        </ECUC-REFERENCE-VALUE>
      </ECUC-REFERENCE-VALUE>
    </REFERENCE-VALUES>
  </ECUC-CONTAINER-VALUE>

```

FIG. 4C

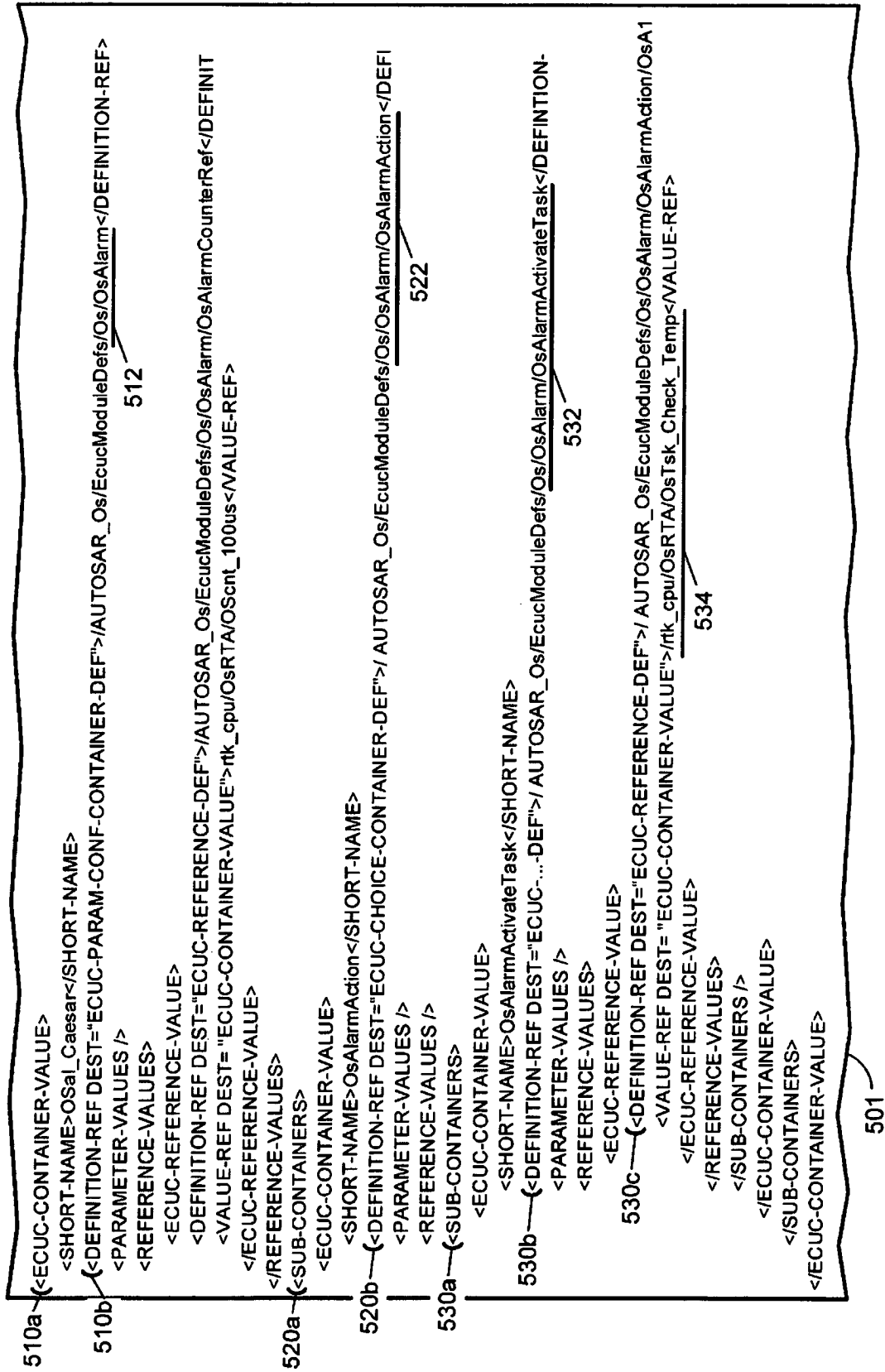


FIG. 5A

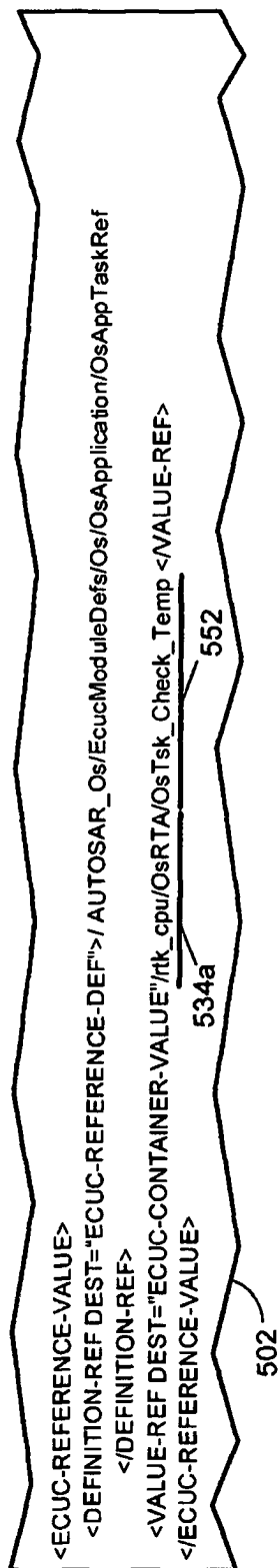


FIG. 5B

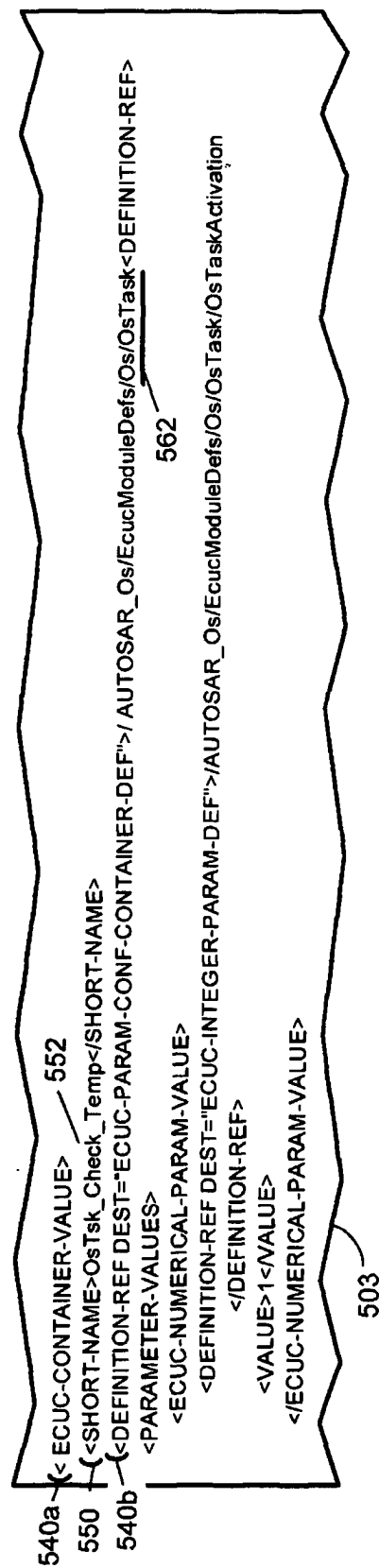
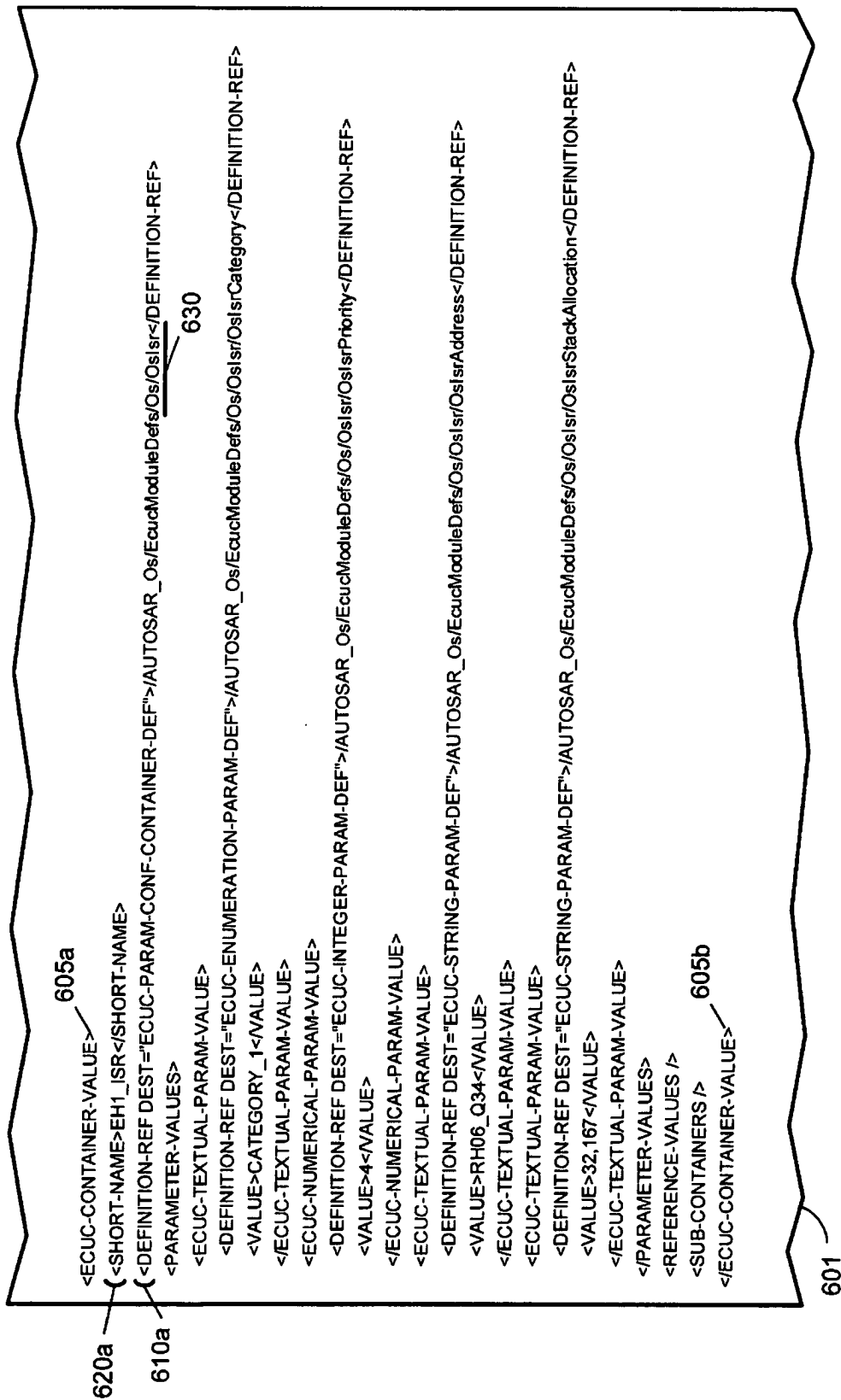


FIG. 5C

**FIG. 6**

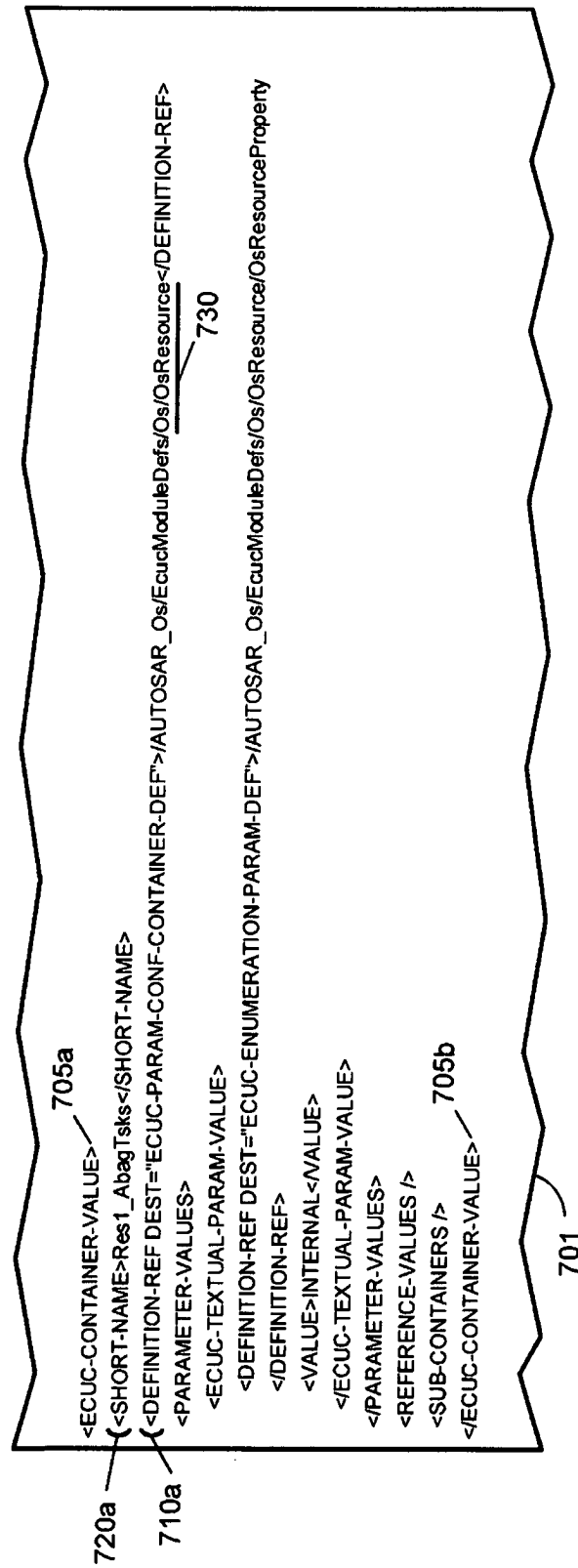
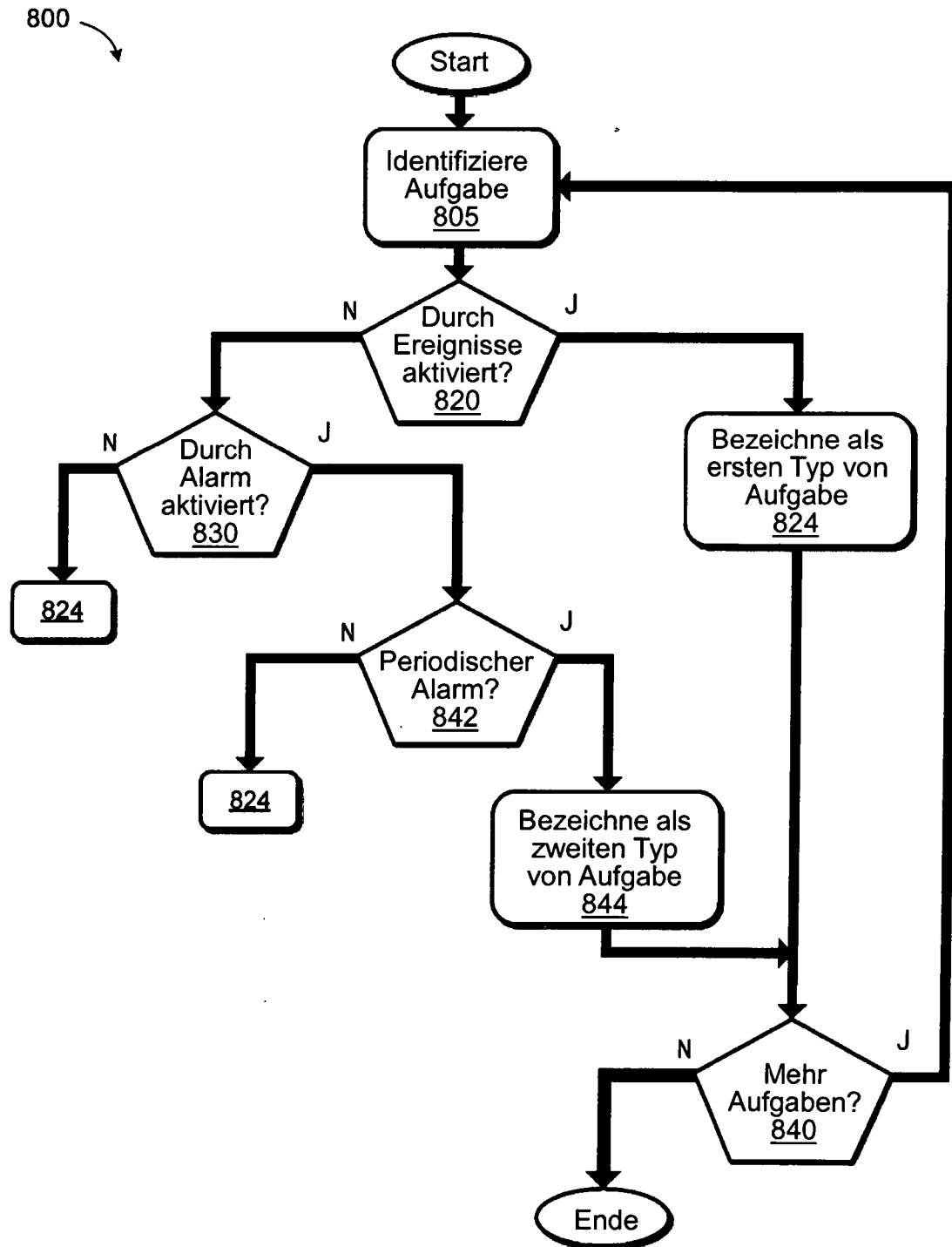


FIG. 7

**FIG. 8**

920a — `<ECUC-CONTAINER-VALUE UUID="3375441c-981f-54bec3bb0d46">`
`<SHORT-NAME>Task_90_C</SHORT-NAME>`
`<DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">`
`/MICROSAR/RH850_F1H/Os/OsTask</DEFINITION-REF>`
`</ECUC-CONTAINER-VALUE>`

901

FIG. 9A

920b — `<ECUC-CONTAINER-VALUE UUID="6f8bf641-d96d-b04436e81845">`
`<SHORT-NAME>IpduM_MainFunctionTimingEvent0</SHORT-NAME>`
`<VALUE-REF DEST=ECUC-CONTAINER-VALUE>/ActiveEcuC/Os/eTask1`
`</VALUE-REF>`
`</ECUC-CONTAINER-VALUE>`

922

902

FIG. 9B

960a — `(TASK (eTask1) /* PRQA S 3408, 1503 */ /* MD_Rte_3408, MD_MSR_14.1 */`
`{`
`EventMaskType ev;`
`/*****`
`*        << Start manually changes LK >>`
`*****/`
`SdsHdl_Init();`
`}`

903

FIG. 9C

```

/*****
* Task:          Task_9ms
* Priority:       80
* Schedule:      FULL
*****/
960b (void eventTaskfunc1(void)
{
    EventMaskType ev;
/*****
*      <<Start manually changes LK
*****/
    SdsHdl_Init();
}
904

```

FIG. 9D

```

/*****
* Task:          Task_9ms
* Priority:       80
* Schedule:      FULL
*****/
960c (void eTask1_ID1(void)
{
    EventMaskType ev;
/*****
*      <<Start manually changes LK
*****/
    SdsHdl_Init();
}
905

```

FIG. 9E

```

*****
* TASK:      Airbag1
* Priority:   8
* Schedule:  Full
*****/
void airbag1func1(void)
{
    SdsHdl_Init();
1020b TakeResource(BasicAirbagRes1)
    /* begin protected code */
1020c /* end protected code */
    ReleaseResource(BasicAirbagRes1)
}
1002

```

FIG. 10A

```

*****
* TASK:      Airbag1
* Priority:   8
* Schedule:  Full
*****/
void airbag1_ID1(void)
{
    SdsHdl_Init();
1030b TakeResource_BasicAirbag_R1()
    /* begin protected code */
1030c /* end protected code */
    ReleaseResource_BasicAirbag_R1()
}
1003

```

FIG. 10B

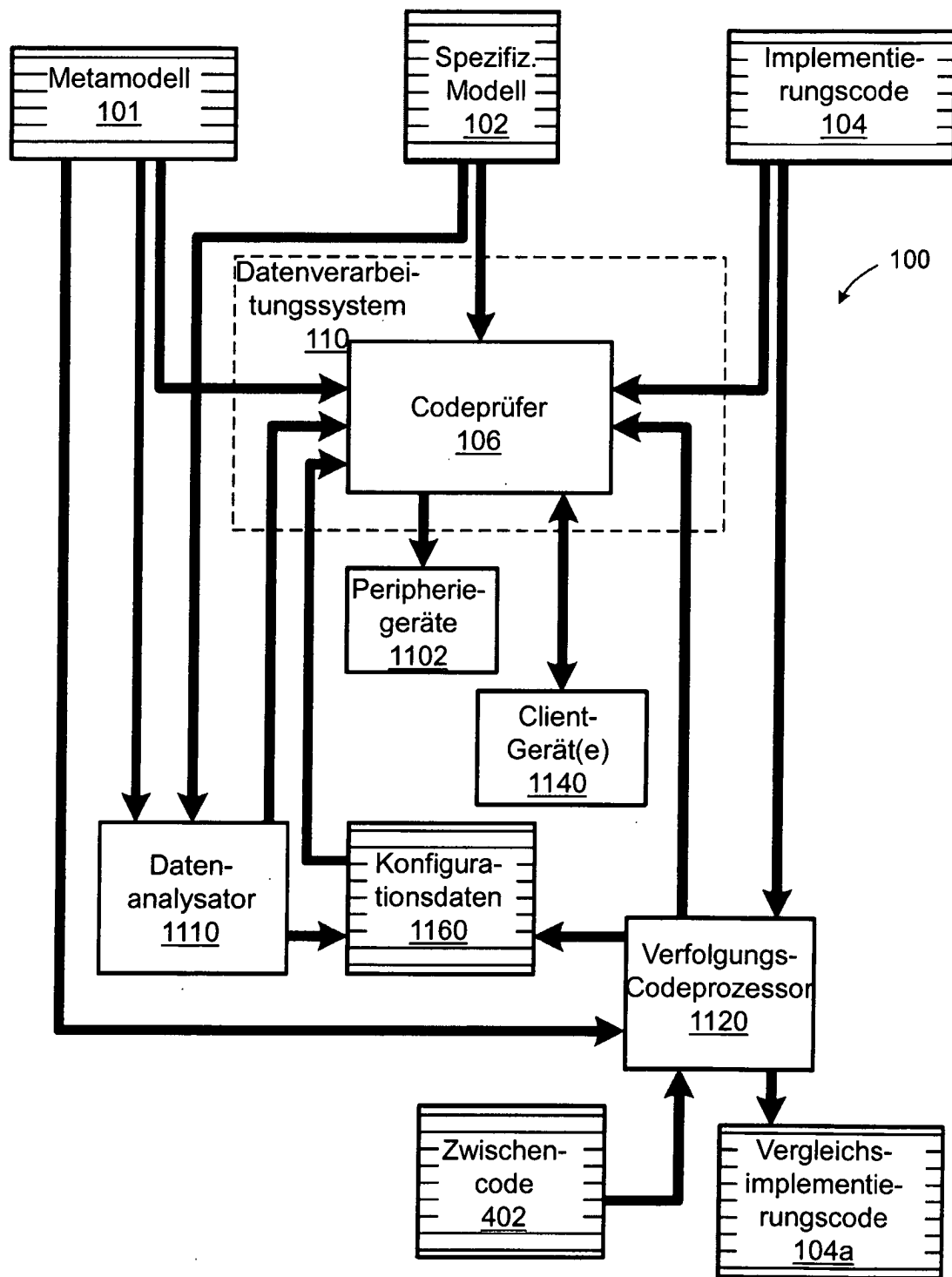
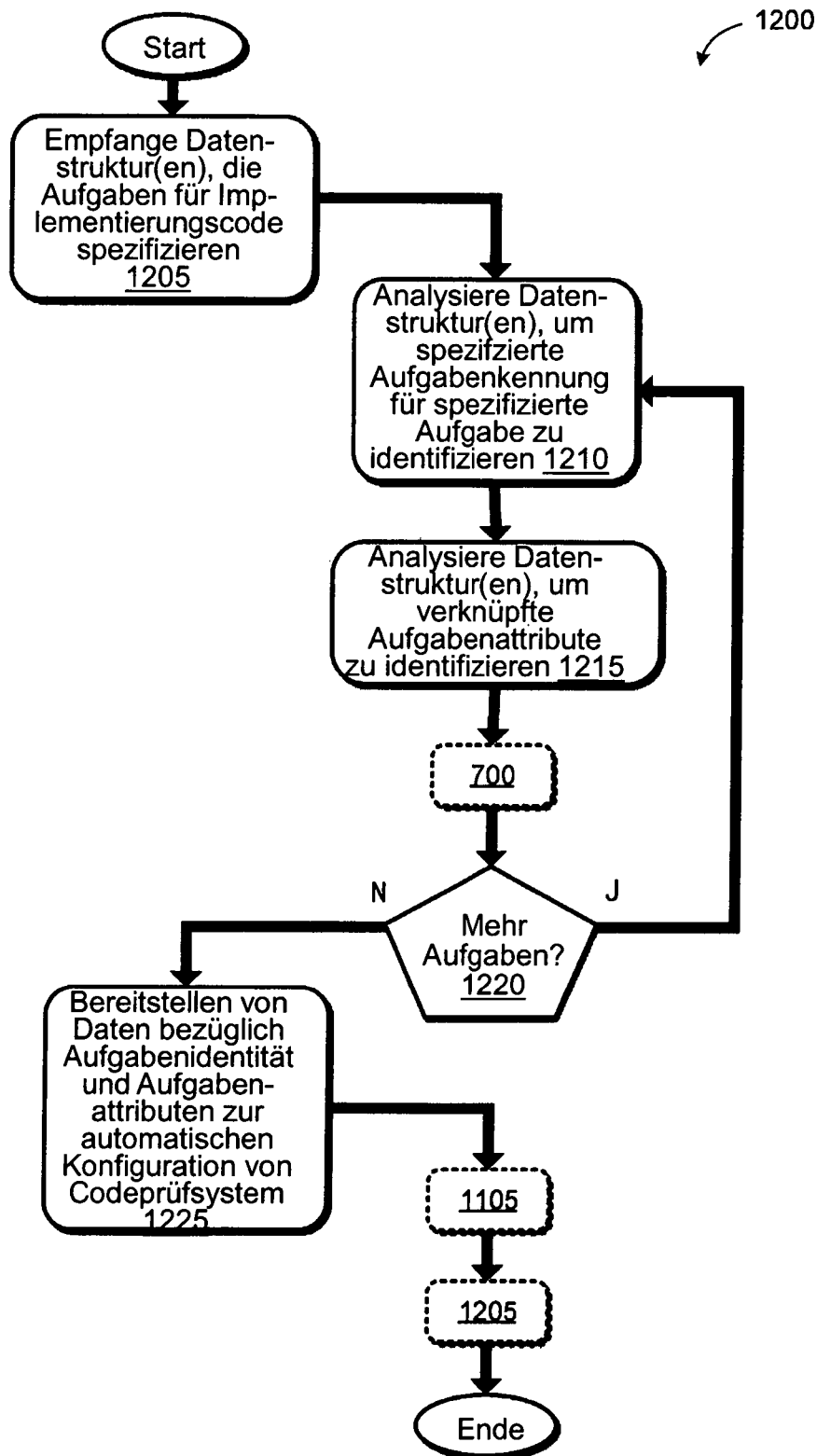
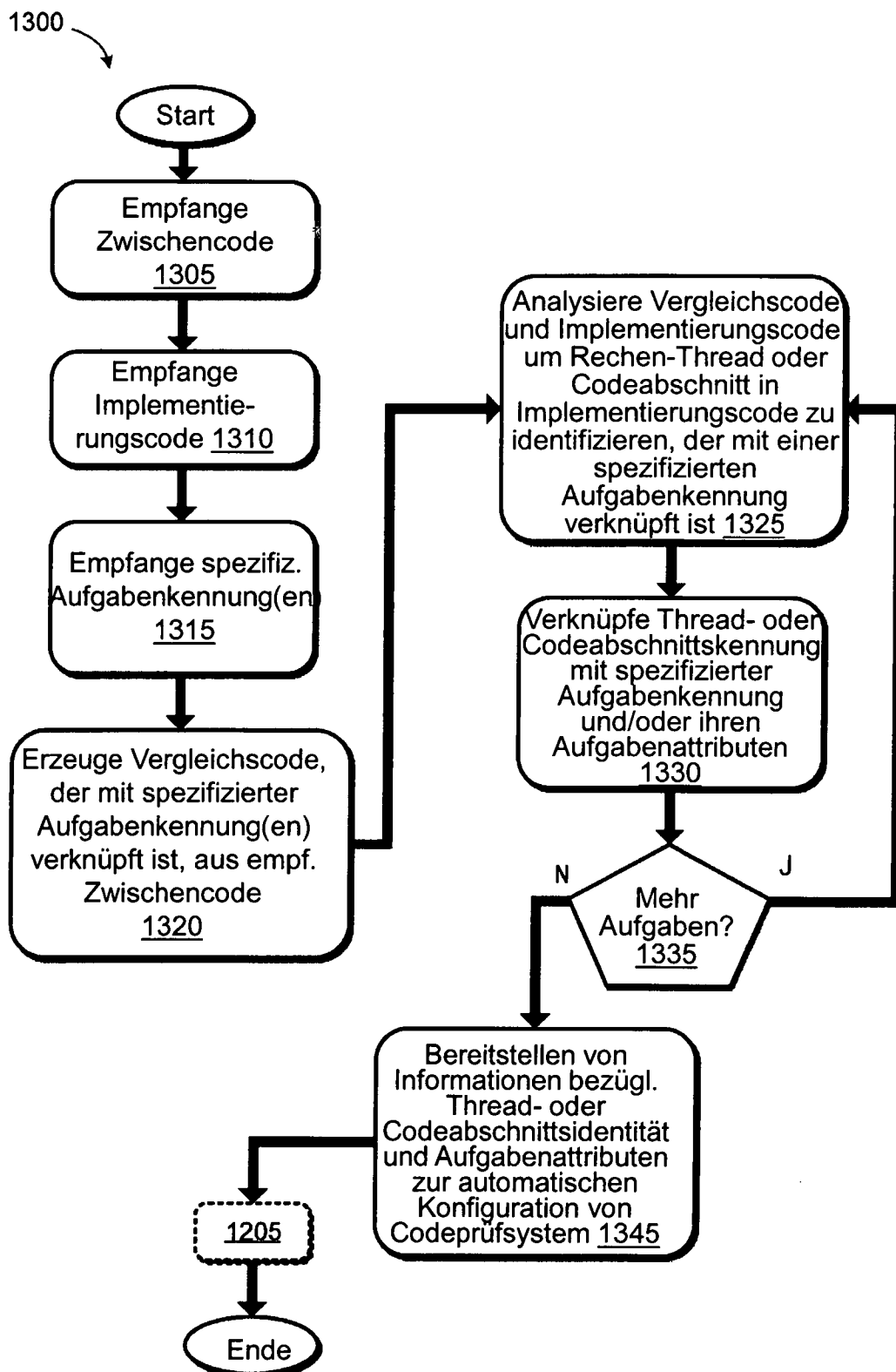
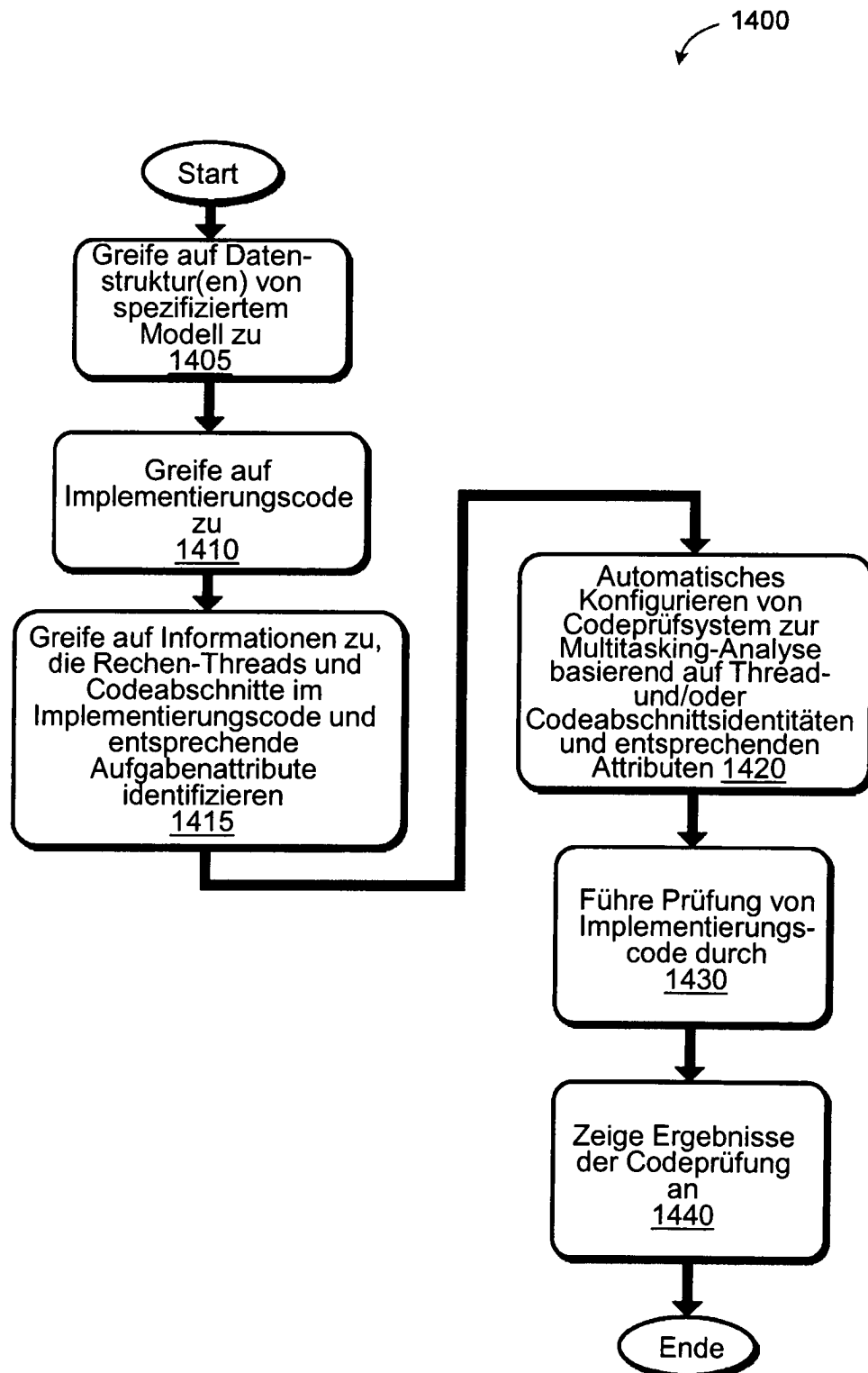


FIG. 11

**FIG. 12**

**FIG. 13**

**FIG. 14**

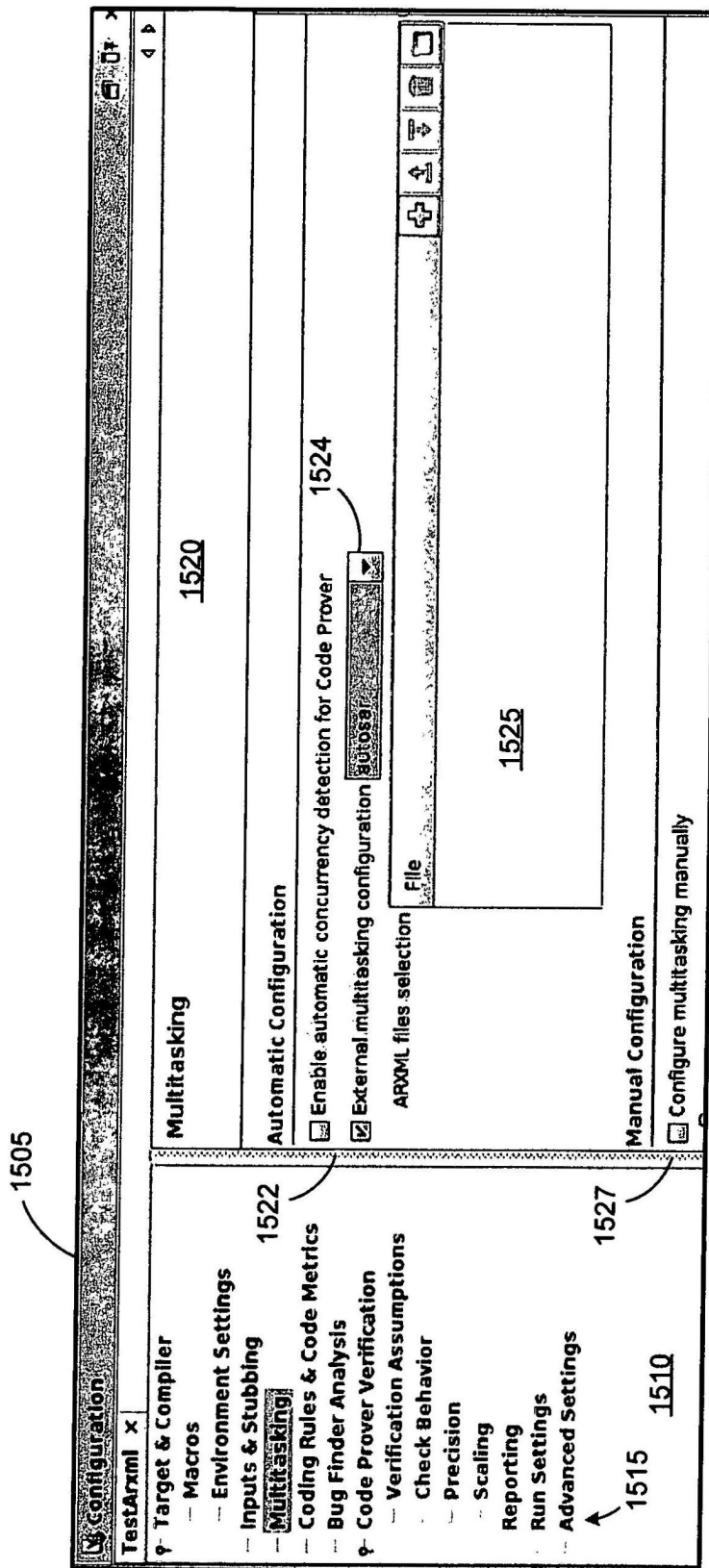


FIG. 15

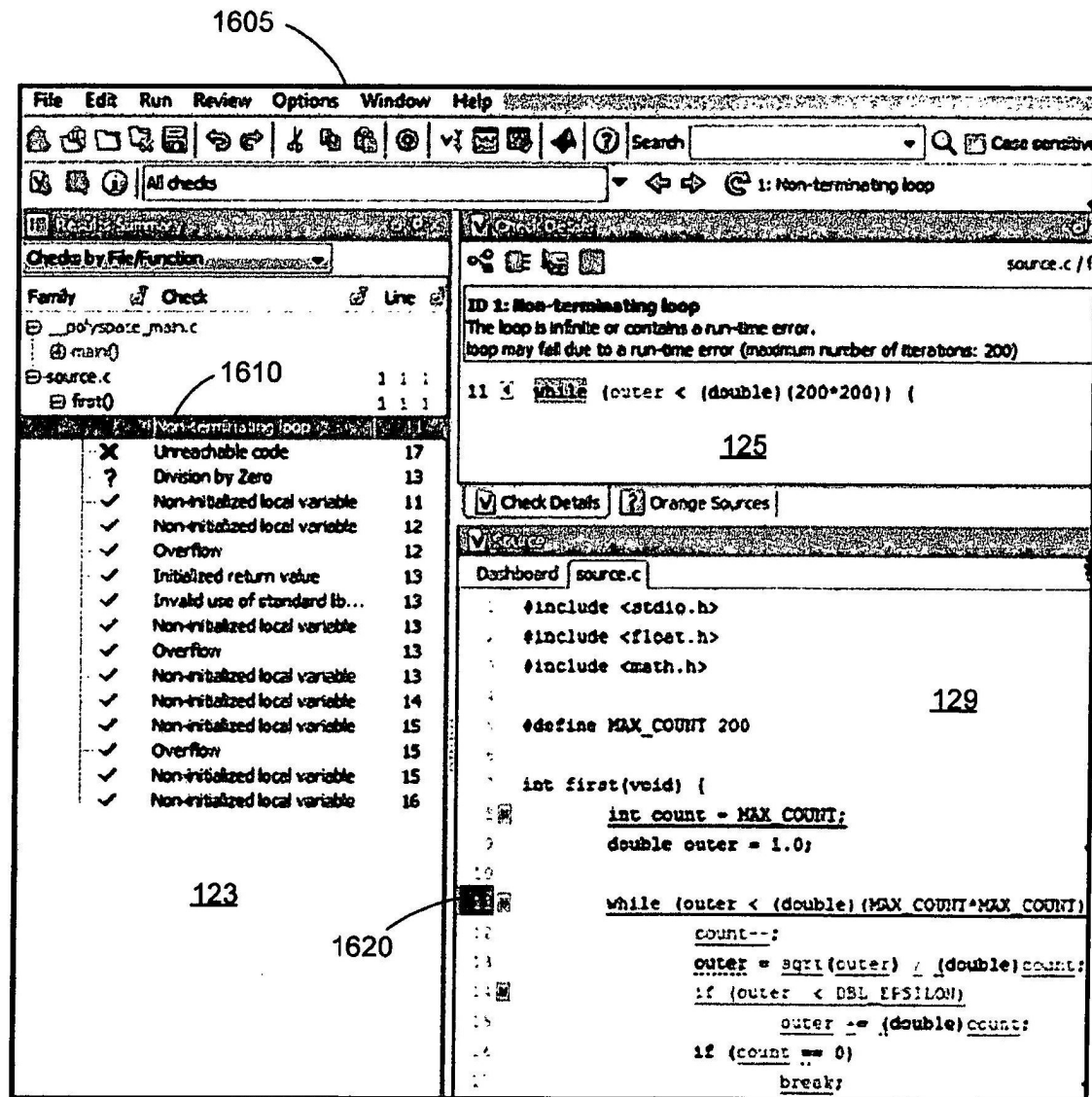


FIG. 16

1705		
1720	Entry point	Set by
	Interrupts (1)	
	φ <u>Interrupt()</u>	
	Executes repeatedly after the main entry point completes Externally declared	
1740	Tasks (8)	
	φ <u>Cyclic task()</u>	
	Executes repeatedly after the main entry point completes Externally declared	
	φ <u>act10ms()</u>	
1743a	Starts after the main entry point completes	Externally declared
	φ <u>afterinit()</u>	
	Starts in <u>init at line 36</u>	Externally declared
1743b	φ <u>ev1_task()</u>	
	Starts in <u>internally_cyclic_task at line 69</u>	Externally declared
	φ <u>ev2_task()</u>	
	Starts in <u>internally_cyclic_task at line 71</u>	Externally declared
	φ <u>init()</u>	
	Starts after the main entry point completes	Externally declared
	φ <u>internally_cyclic_task()</u>	
	Starts after the main entry point completes	Externally declared
	φ <u>not_in_code()</u>	
	Starts after the main entry point completes	Externally declared

FIG. 17

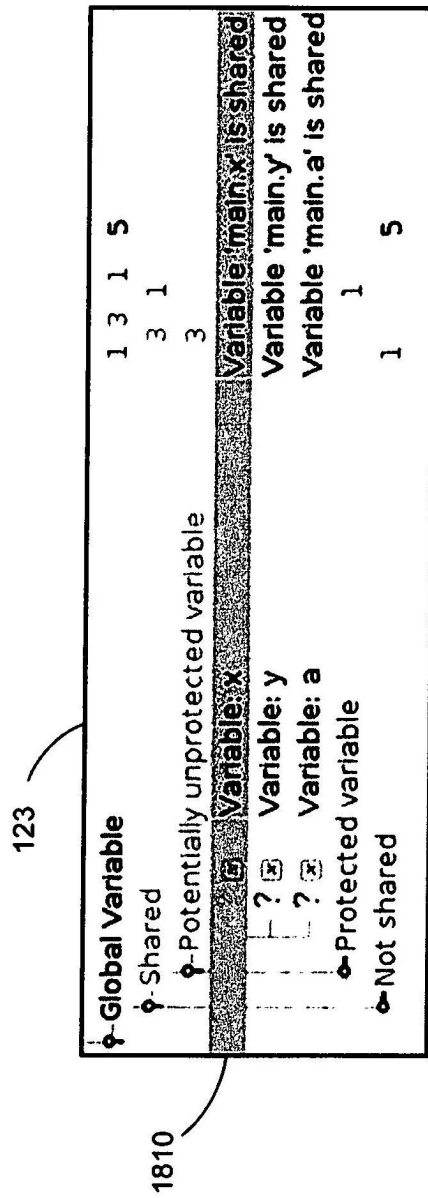


FIG. 18

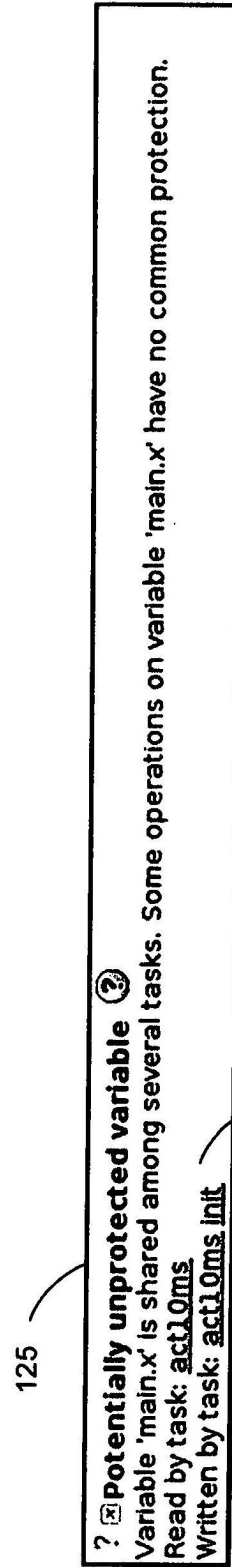


FIG. 19

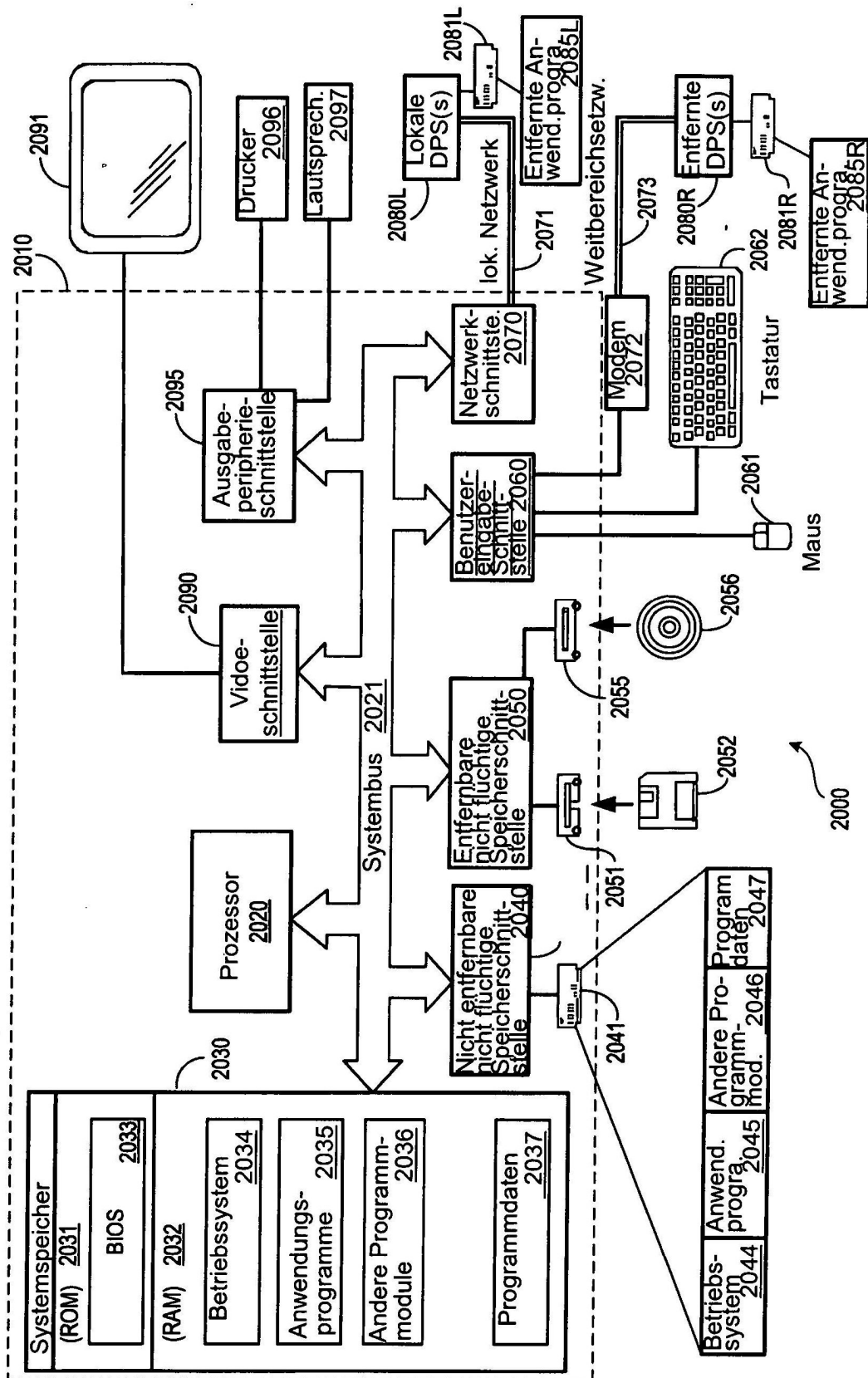


FIG. 20

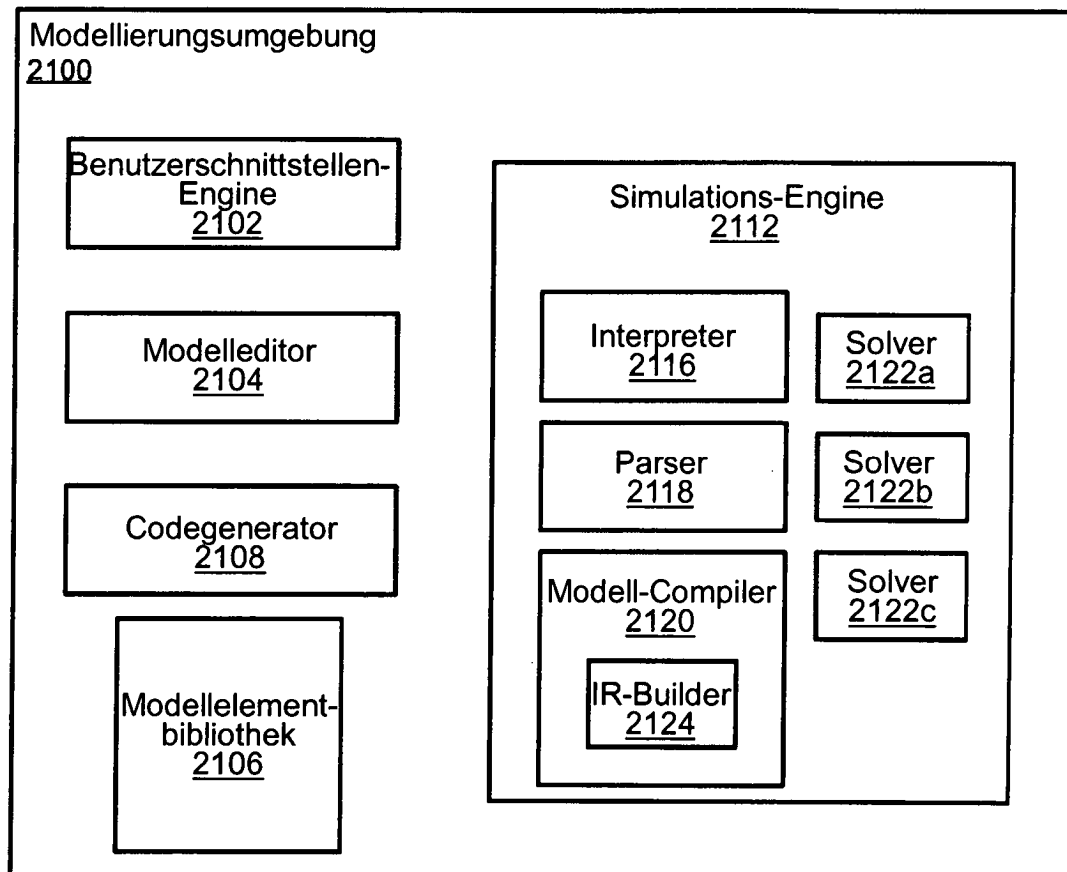


FIG. 21