Zerberus System - Ein Entwicklungsmodell für sichere und zuverlässige Computersysteme

Christian Buckl
TU München
Lehrstuhl für Eingebettete Systeme und Robotik
buckl@in.tum.de

Abstract

Computersysteme dringen in erhöhten Maße in sicherheitskritische Bereiche vor und dadurch steigen auch die Anforderungen an die Zuverlässigkeit und Sicherheit an die entwickelten Computersysteme. Standardisierte Modelle zur Erstellung von solchen Computersystemen fehlen jedoch. Ziel des Zerberus Projektes ist es ein Entwicklungsmodell mit kompletter Werkzeugkette für die Entwicklung entsprechender Systeme anzubieten. Grundlage ist die Wiederverwendung von entwickelten Fehlertoleranzmechanismen, so dass sich der Entwickler wieder auf die Entwicklung der eigentlichen Funktionalität der Anwendung konzentrieren kann. Das Zerberus System bietet eine Sprache zur plattformunabhängigen Modellierung sicherer und zuverlässiger Systeme, einen Codegenerator sowie Laufzeitsysteme für diverse Plattformen als Prototypen an. In einer nächsten Projektphase sollen die entwickelten Ansätze durch die Anwendung des Zerberus Systems in Industrieprojekten evaluiert werden.

1. Einleitung

Der bislang zu erbringende Aufwand bei der Erstellung sicherheitskritischer Applikationen steht in größer werdendem Widerspruch zu den steigenden Anforderungen an Produktqualität komplexer Software bei gleichzeitig zu minimierenden Entwicklungszeiten in kürzesten Produktzyklen bei höchstem Kostendruck. Erhebliche Entlastung könnte hier von einer geeigneten Werkzeugunterstützung kommen, die den zusätzlichen Aufwand für die Ausstattung des Systems mit Mechanismen zur Fehlertoleranz zwecks Erreichung von Zuverlässigkeit und Sicherheit minimiert.

Die vorhandenen Entwicklungswerkzeuge für die Umsetzung von Fehlertoleranzmechanismen beschränken sich auf spezielle Probleme und Anwendungen. Ein standardisiertes Modell zur Erstellung von Hardware und Software für sicherheitskritische Produkte fehlt ebenso wie darauf aufsetzende Werkzeuge. In der Regel findet stattdessen eine Verkettung der Modellierung von Funktionalität und Sicherheitsaspekten statt, die die Implementierung nur ausgewiesenen Fachleuten in Zusammenarbeit mit den Applikationsspezialisten erlaubt. Gängige Praxis ist dabei zumeist die Implementierung nach internen Erfahrungswerten innerhalb eines Unternehmens und die nach der Implementierung stattfindende Zertifizierung durch eine akkreditierte Stelle. Bestenfalls wird bei der Festlegung der Hard- und Softwarearchitektur die später zertifizierende Stelle mit einbezogen, um auch deren Erfahrungspotential zu nutzen. Dieses Verfahren ist gerade für kleine und mittlere Unternehmen wirtschaftlich nicht durchführbar, weil eine vorherige Zeit- und Kostenabschätzung unmöglich ist.

Aus den genannten Gründen besteht ein erheblicher Bedarf an einem einheitlichen Entwicklungsmodell für sicherheitskritische Applikationen. Innerhalb des Zerberus Projektes wurde deshalb ein solches Modell, sowie geeignete Werkzeuge entwickelt, um den Entwicklungsprozess zu beschleunigen, die Fehlerrate zu senken und die Kosten für die Entwicklung zu reduzieren. Zudem wurde ein Augenmerk auf eine erleichterte Zertifizierung durch Kontrollstellen, wie z.B. den TÜV, gelegt.

Zu Beginn dieses Berichts werden die Grundlagen des Zerberus Systems erläutert. Die einzelnen Schritte des vorgeschlagenen Entwicklungsprozesses sind in Abbildung 1 dargestellt und werden jeweils in einzelnen Kapiteln beschrieben. Die vorgeschlagenen Entwicklungsschritte sind die plattformunabhängige Spezifizierung des funktionalen

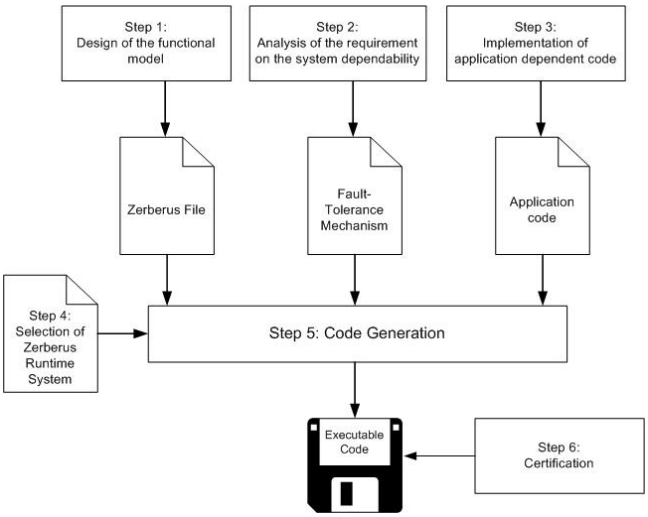

Abbildung 1. Schritte des Entwicklungsprozesses

Modells (Kap.3.1), die Wahl geeigneter Fehlertoleranzmechanismen (Kap.3.2), die Implementierung des rein anwendungsabhängigen Codes (Kap.3.3), die Wahl einer Laufzeitumgebung (Kap.3.4) und die Codegenerierung(Kap.3.5). Falls benötigt unterstützt das Zerberus System den Entwickler auch bei der Zertifizierung (Kap.3.6) der Applikation durch Kontrollstellen.

Zum Ende dieses Berichts werden schließlich noch die geplanten Weiterentwicklungen innerhalb des Zerberus Projektes erläutert.

2. Grundlagen des Zerberus Systems

Die derzeitige Basis der Fehlertoleranzmechanismen ist strukturelle Hardwareredundanz: mindestens drei gleichwertige Rechner berechnen parallel die Applikationsalgorithmen. Die Synchronisations-, Votierungs- und Entscheidungsalgorithmen über die Ausgabe der Ergebnisse werden durch Protokolle festgelegt, durch die Laufzeitsysteme realisiert und müssen daher nicht mehr durch den Anwendungsentwickler implementiert werden. Durch diese Wiederverwendung der Fehlertoleranzmechanismen muss nur der rein anwendungsabhängige Code implementiert werden. Fehlerquellen können dadurch vermieden und der Entwicklungsprozess beschleunigt werden.

Die einzelnen Rechner sind mit Standardkomponenten aufgebaut. Dadurch können einerseits die Kosten reduziert werden, andererseits kann stets auf die leistungsstärkste Hardware zurückgegriffen werden. Alle Rechner sind zudem jeweils über Sensorik und Aktorik mit der zu kontrollierenden Umgebung verbunden. Dadurch können kritische Ausfallspunkte im Sinne eines single point of failure vermieden werden.

Um auch Entwurfsfehler tolerieren zu können wird neben der strukturellen Redundanz auch Diversität der Hardware und der Software unterstützt. Dazu werden durch das Zerberus System Laufzeitsysteme für unterschiedliche Plattformen angeboten. Unter Plattform wird dabei die Hardware, das Betriebssystem und die Programmiersprache verstanden. Die Diversität der Hardware wird durch die Verwendung von Standardhardware möglich: es sind Alternativen verschiedener Hersteller vorhanden und die Kosten für eine unterschiedliche Ausstattung der einzelnen Rechner unterscheiden sich kaum gegenüber der identischen Ausstattung. Eine Entwicklung von unterschiedlichen Versionen der Applikationssoftware kann aufgrund der Beschränkung der Implementierung auf den rein anwendungsabhängigen Code mit vertretbaren Mehraufwand angewandt werden.

3. Entwicklungsschritte

Die einzelnen Entwicklungsschritte wurden bereits in Abbildung 1 dargestellt und werden nun in diesem Kapitel ausführlicher erläutert.

3.1. Schritt 1: Spezifizierung des Formalen Modells

Im ersten Schritt muss der Entwickler das formale Modell der Applikation plattformunabhängig spezifizieren. Dabei werden die funktionalen Einheiten, die Interaktionen dieser Einheiten, sowie die zeitlichen Rahmenbedingungen identifiziert. Das formale Modell dient als Spezifikation der Applikation und wird als Basis für die Ausführung der Fehlertoleranzmechanismen verwendet. Die zeitliche Synchronisation, der Vergleich der Zustände der einzelnen Einheiten, Entscheidungen über auszugliedernde bzw. ausgebende Einheiten und die Integration einer vormals ausgegliederten Einheit während des laufenden Betriebs müssen unterstützt werden.

Aus dieser Forderung ergeben sich folgende Bedingungen:

1. Determinismus der redundanten Einheiten: Die einzelnen redundanten Einheiten müssen sich deterministisch verhalten [7], so dass eine Votierung über den Zuständen der einzelnen Einheiten möglich ist. Da die Forderung nach striktem Determinismus die Möglichkeiten der N-Versions-Programmierung stark einschränken würde, wird diese Forderung jedoch abgeschwächt. Es müssen vielmehr Zeitpunkte existieren an denen Votierungsalgorithmen durchgeführt werden können, die die Menge der Einheiten korrekt in eine fehlerfreie Menge und eine fehlerbehaftete Menge unterteilen.

2. Existenz von deterministischen Zeitpunkten: Die in Punkt 1 geforderten Zeitpunkte müssen deterministisch und vom jedem Rechner eindeutig bestimmbar sein. Nur dadurch ist es möglich eine Synchronisation [6] und Votierung durchführen zu können.

3. Separierung des Zustandes: Um eine Votierung durchzuführen und eine vormals ausgegliederte Einheit wieder zu integrieren, ist eine Separierung des Zustandes von der Funktionalität nötig. Dies ist die Voraussetzung für die Möglichkeit einer applikationsunabhängige Realisierung von Votierungs- und Integrationsalgorithmen.

Die Spezifizierung des Modells erfüllt durch die Verwendung der Zerberus Sprache diese Bedingungen. Grundlage der Erfüllung ist das zeitgesteuerte Paradigma [5] der Zerberus Sprache die eine deterministische Ausführung der Applikation erlaubt und zugleich Zeitpunkte spezifiziert an denen eine Votierung sinnvoll ist. Zweites Merkmal der Zerberus Sprache ist die Trennung der Funktionalität, des Zustandes und der Plattform voneinander. Basis der Zerberus Sprache war Giotto [3] [2], eine Sprache zur Spezifizierung eines formalen Modells für verteilte Echtzeitsysteme. Diese Sprache wurde an die Bedürfnisse zur Unterstützung von Fehlertoleranzmechanismen und dem Einsatz von diversen Plattformen angepasst.

Im Folgenden wird die Zerberus Sprache knapp beschrieben, eine ausführlichere Abhandlung ist in Form eines technischen Berichts [1] verfügbar.

Tasks: Hauptkonzept der Zerberus Sprache sind periodisch aufgerufenen Funktionen, sogenannte Tasks. Tasks stellen die eigentliche Funktionalität der Anwendung dar und müssen deshalb durch den Enwickler implementiert werden. Ebenso wird die Aufruffrequenz durch den Entwickler festgelegt. Während der Ausführung agieren Tasks eigenständig: Synchronisationspunkte zwischen Tasks sind verboten und der Code der Funktionen ist auf sequentiellen Code beschränkt. Aufgrund dieser Beschränkung ist die Implementierung für den Entwickler stark vereinfacht.

Die tatsächliche physikalische Ausführung der Tasks auf der CPU wird durch das Laufzeitsystem realisiert und ist transparent für den Entwickler. Dieser kann also davon ausgehen, dass Tasks zu Beginn des Intervals gestartet werden und zum Ende des Intervals ihre Berechnung beenden.

Ports: Zur Kommunikation zwischen Tasks und zur Separierung des Zustandes von der Funktionalität der Anwendung werden Ports eingeführt. Ports sind Speicherplätze von spezifizierter Größe und Repräsentation (nötig aufgrund der Unterstützung unterschiedlicher Plattformen), die von Tasks auf deterministische Art und Weise gelesen und geschrieben werden können. Der Lesezugriff findet dabei immer zu Beginn, die Schreibzugriffe zum Ende des Aufrufintervals des entsprechenden Tasks statt. Gleichzeitige Schreibzugriffe mehrerer Tasks auf einen Port sind nicht erlaubt und werden vom Codegenerator bzw. dem Laufzeitsystem abgefangen. Jedoch muss der Entwickler garantieren, dass eine Einhaltung der zeitlichen Schranken möglich ist, also die längsten Ausführungszeiten der Tasks nicht den Aufruffrequenzen widersprechen.

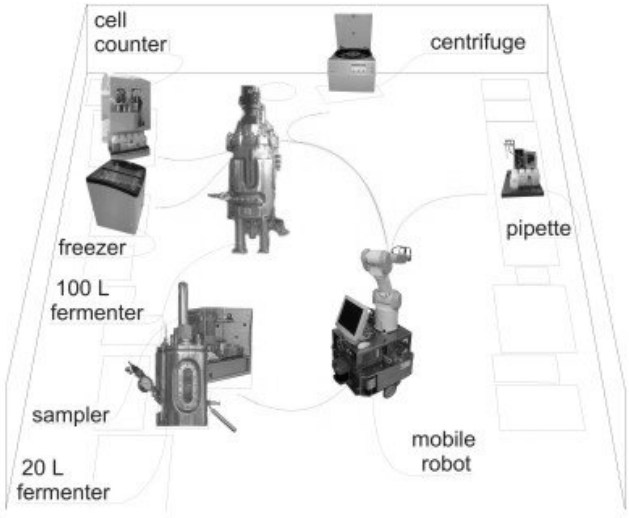

Abbildung 2. Beispielsapplikation

Sensoren und Aktoren: Zur Kommunikation mit der Umgebung werden Sensoren und Aktoren eingeführt. Dabei handelt es sich um die softwaretechnische Realisierung der Ein- und Ausgabe. Sensoren und Aktoren sind ähnlich wie Tasks periodisch aufgerufenen sequentielle Funktionen, jedoch wird die Ausführung dieser Funktionen als instantan angenommen. Sensoren werden dabei immer zu Beginn ihres Intervals ausgeführt, Aktoren zum Ende des Intervals. Aufgrund der vorangegangenen Annahme dürfen Sensoren und Aktoren keine langwierigen Berechnungen durchführen und auch keine Synchronisationspunkte mit der Umwelt besitzen. Die Ergebnisse von Sensoren werden in, durch den Entwickler angegebene, Ports geschrieben. Auch hier gilt die vorangegangenen Forderung nach der Absenz von Kollisionen bei Schreibzugriffen. Die Ausgabe des Systems durch Aktoren basiert wiederum auf den Werten einzelner Ports, die durch den Aktor gelesen werden können.

Modi und Guards: Um die Ausführung des Systems flexibler zu gestalten und auch verschiedenen Betriebsmodi zuzulassen, können Modi und Guards verwendet werden. Ein Mode ist eine Teilmenge der deklarierten Tasks, Sensoren und Aktoren, die simultan auf dem System ausgeführt werden. Zwischen verschiedenen Modes kann durch die Benutzung von Moduswechseln umgeschaltet werden. Ein Moduswechsel ist dabei eine auf Portwerten basierende binäre Funktion, die die Entscheidung eines Wechsels evaluiert. Modes sollten vor allem für verschiedene Betriebsmodi verwendet werden. Um eine feinere Steuerung des Systems zu erreichen können Guards verwendet werden. Ein Guard ist ähnlich wie ein Moduswechsel eine auf Portwerten basierende binäre Funktion. Allerdings werden Guards direkt Tasks zur Kontrolle zugewiesen. Vor dem Start eines Tasks wird dann der entsprechende Guard evaluiert und nur bei einem positiven Ergebnis der Task auch wirklich gestartet.

Zum besseren Verständnis wird die Zerberus Sprache im Folgenden anhand eines Beispiels erläutert. Die Beispielsapplikation ist in der Abbildung 2 dargestellt. Es soll ein System für die Steuerung eines Roboters in einem biotechnologischen Labors entwickelt werden. Der Roboter soll in dem Labor zwischen verschiedenen Arbeitsstationen navigieren, Objekte transportieren und einzelne Arbeitsschritte erledigen. Wichtigstes Ziel ist das Vermeiden von Kollisionen. Der Roboter ist für diese Anwendung mit unterschiedlicher Sensorik ausgestattet: via Wireless LAN werden Aufträge empfangen, über Funk kann der Roboter seine Position bestimmen und eine Kamera wird für die Objekterkennung benutzt.

Die Aufgabe kann nun zuerst in zwei unterschiedliche Betriebsmodi unterteilt werden: das Navigieren des Roboters von einer Arbeitsstation zur nächsten und das Greifen bzw. Positionieren von Objekten. In beiden Modi müssen Funktio-

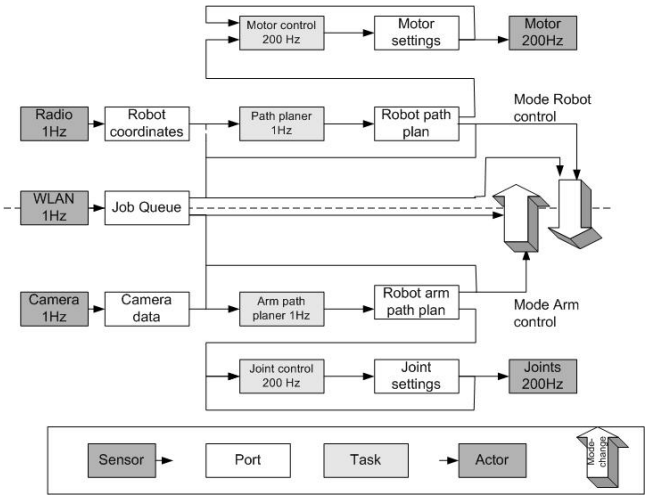

Abbildung 3. Formales Modell

nen implementiert werden, die den Pfad des Roboters (Task path planer) bzw. des Roboterarmes (Task arm path planer) planen und Kollisionen vermeiden. Diese Tasks besitzen aufgrund ihrer Komplexität lange Berechnungszeiten. Um dennoch eine gleichmäßige Bewegung des Roboters zu erhalten werden zwei weitere Tasks (motor control, joint control) eingeführt, die die Ergebnisse der Pfadplaner verwenden, um die Werte zur Motoransteuerung zu berechnen. Aufgrund der niedrigeren Komplexität ist es möglich eine deutlich höhere Ausführungsfrequenz für diese Tasks zu verwenden.

Abbildung 3 zeigt die graphische Darstellung des Modells für unsere Beispielsapplikation. Es wird deutlich, dass eine graphische Darstellung der Anwendung gut geeignet ist, um eine Applikation zu veranschaulichen. An der Entwicklung eines graphischen Editors für die Zerberus Sprache wird deshalb gearbeitet.

3.2. Schritt 2: Wahl der anzuwendenden Fehlertoleranzmechanismen

Im zweiten Schritt muss der Entwickler die anzuwendenden Fehlertoleranzmechanismen wählen. Derzeit wird die strukturelle Redundanz als Bedingung vorausgesetzt, lediglich die Anzahl der redundanten Einheiten kann variiert werden. So sind alle Mechanismen auf eine Art und Weise ausgelegt, dass der Redundanzgrad beliebig gewählt werden kann.

Zusätzlich zur strukturellen Redundanz kann auch Diversität in Hardware und Software eingesetzt werden. Dazu werden verschiedene Laufzeitsysteme angeboten.

In naher Zukunft soll das Zerberus System auch um weitere Fehlertoleranzmechanismen wie z.B. Rückwärtsbehebung erweitert werden, um die Erzwingung von struktureller Hardwareredundanz zu vermeiden und die Entwicklung der Systeme zu flexibilisieren.

3.3. Schritt 3: Implementierung des anwendungsabhängigen Codes

Nachdem das formale Modell und die verwendeten Fehlertoleranzmechanismen spezifiziert wurden, muss der anwendungsabhängige Code erstellt werden. Dies beinhaltet den Code für die Taskfunktionen, die Aktor- und Sensorfunktionen sowie den Evaluierungsfunktionen für Moduswechsel und Guards. Alle genannten Funktionen sind ausschließlich als sequentieller Code zu implementieren, insbesondere sind keinerlei Interaktionspunkte erlaubt.

Die restliche Funktionalität der Anwendungen, die Realisierung des formalen Modells, sowie der Fehlertoleranzmechanismen, werden durch die Laufzeitsysteme erfüllt. Dadurch wird ein Maximum der Wiederverwendbarkeit von Code erreicht und die Gefahr der Fehlerentstehung minimiert. Lediglich die Algorithmen zur Reaktion auf Fehlern, die rein anwendungsspezifisch sind, müssen zusätzlich implementiert werden.

Um die Anwendungsentwicklung weiter zu vereinfachen ist es zudem angedacht, für typische Sensorik und Aktorik Funktionen bereitzustellen, die vom Anwendungsentwickler wiederverwendet werden kann. Auch dies würde zu einer weiteren Verbesserung der Fehlersicherheit führen.

3.4. Schritt 4: Wahl der Laufzeitumgebung

Wie bereits erwähnt werden für diverse Plattformen Laufzeitsysteme angeboten. Derzeit werden die Programmiersprachen C und C++ jeweils für VxWorks 5.5 unterstützt. Eine Portierung der Laufzeitsysteme auf VxWorks 6.0 unter Ausnutzung der vorhandenen Verbesserungen (vor allem im Bereich Speicherschutz) ist derzeit in Arbeit. Zudem wird die Entwicklung von Laufzeitsystemen für RTLinux geprüft.

Allerdings besteht für den Entwickler auch die Möglichkeit das Zerberus System zu verwenden, wenn für die gewünschte Plattform kein Laufzeitsystem angeboten wird. Aus diesem Grund werden die Protokolle zur Votierung, Synchronisierung und Integration angeboten, so dass der Entwickler ein Laufzeitsystem auch eigenständig entwickeln kann. Um eine mehrfache Implementierung von Laufzeitsystemen zu vermeiden, existiert eine weitere Sprache, die sogenannten Zerberus Tags, die eine anwendungsunabhängige Entwicklung eines Laufzeitsystems erlauben. Zerberus Tags markieren dabei Stellen im Code, die durch Applikationsdaten ersetzt werden müssen. Diese Ersetzung erfolgt durch den Codegenerator und wird im nächsten Abschnitt erläutert. Durch die Einführung der Zerberus Tags wird eine Wiederverwendung der Laufzeitsysteme unterstützt und die Erweiterbarkeit des Zerberus Systems garantiert.

Während dem Entwurf der Protokolle für die zeitliche Synchronisation und Votierung wurde darauf geachtet, dass die Protokolle möglichst universell zu verwenden sind. Schwerpunkt lag auf der Einfachkeit der Protokolle, so dass die Umsetzung der Protokolle auch einfach verifiziert werden kann.

Votierung Die Votierung erfolgt im Zerberus System in zwei Runden, um auch unzuverlässige Kommunikationsmedien tolerieren zu können. In einer ersten Runde werden Zustandsdaten (basierend auf Portwerten) ausgetauscht, in der zweiten Runde erfolgt der Austausch von Zustandsnachrichten. Mittels dieser Nachrichten ist es möglich, einzelne verlorene Nachrichten zu rekonstruieren und eine konsistente Sicht der Rechner zu erreichen.

Synchronisation Zur Vermeidung von zusätzlichen Overhead basieren die Synchronisationsalgorithmen auf den ohnehin nötigen Votierungsalgorithmen. Durch Kenntnisse über den erwarteten Zeitpunkt der Ankunft von Votierungsnachrichten (möglich durch das zeitgesteuerte Paradigma), den zu erwartenden Nachrichtenverzögerungen (abhängig vom Kommunikationsmedium) und eine Festlegung einer maximalen Verzögerung können die einzelnen Rechner Rückschlüsse über die globale Synchronisation erlangen.

3.5. Schritt 5: Codegenerierung

Die Abbildung eines anwendungsunabhängigen Laufzeitsystems auf anwendungsabhängigen lauffähigen Code wird durch den Zerberus Codegenerator vorgenommen. Bild 4 zeigt den Codegenerationsprozess.

In einem ersten Schritt wird das formale Modell in Form von Zerberus Code geparst und einer syntaktischen und semantischen Prüfung unterzogen. Im Erfolgsfall werden nun die Laufzeitsystemdateien ebenfalls geparst und die Zerberus Tags durch anwendungsabhängige Daten ersetzt. Zudem werden die entsprechenden Fehlertoleranzmechanismen ausgewählt und der vom Entwickler implementierte Code übernommen.

Das Ergebnis der Codegenerierungsprozesses ist ausführbarer Code für die entsprechende Plattform.

3.6. Schritt 6: Zertifizierung

Der Entwicklungsprozess und der hohe Grad an Wiederverwendung soll auch im Zertifizierungsprozess genutzt werden. Prinzipiell können, wie in Abbildung 5 dargestellt, drei Konzepte identifiziert werden, die zertifiziert werden müssen: das Zerberus System, die Laufzeitsysteme sowie die Anwendung.

Das Zerberus System besteht dabei aus dem Sprachkonzept, dem Konzept der Zerberus Tags, den Protokollen zur Synchronisation, Votierung und Integration, sowie dem Zerberus Codegenerator. Während der Zertifizierung muss dabei aufgezeigt werden, dass das Konzept korrekt ist, die Protokolle den Anforderungen entsprechen und der Zerberus Codegenerator fehlerfrei arbeitet.

In einem zweiten Schritt müssen die Laufzeitsysteme getestet werden. Diese umfassen neben der prinzipiellen Korrektheit vor allem die Übereinstimmung mit den geforderten Protokollen zur Synchronisation, Votierung und Integration, sowie in der korrekten Ausführung des formalen Modells.

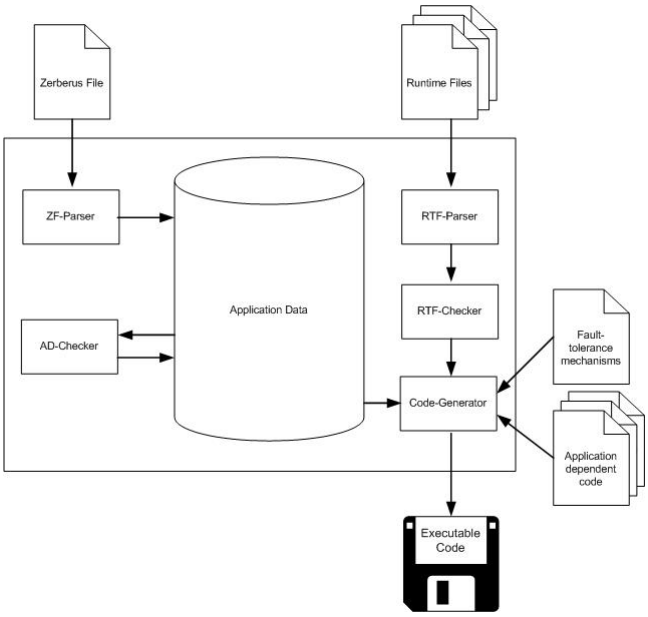

Abbildung 4. Code Generation

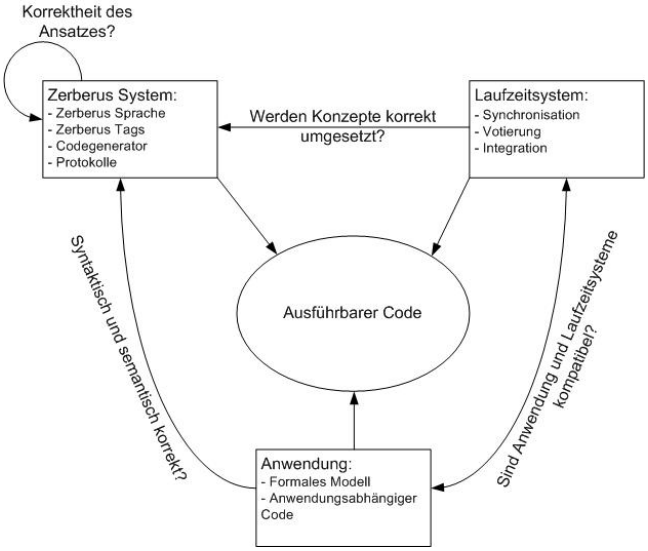

Abbildung 5. Zertifizierung

Erst im letzten Schritt muss die Anwendung selber zertifiziert werden. Dabei muss der vom Entwickler implementierte Code überprüft und die Kompatibilität der Anwendung mit den verwendeten Laufzeitsystemen zum Beispiel auf die Unterstützung der von der Anwendung geforderten Regelungszeiten an das Laufzeitsystem getestet werden. Zudem erfolgt eine Überprüfung des Codes an die vom Zerberus System gestellten Bedingungen, z.B. eine Überprüfung der maximalen Ausführungszeiten.

Der dem Zerberus System zugrundeliegende Ansatz sieht nun vor, dass bei einer Verwendung eines zertifizierten Zerberus Systems und von zertifizierten Laufzeitsystemen für eine Zertifizierung der Applikation der dritte Schritt ausreichend ist oder zumindest Resultate aus vorangegangenen Zertifizierungsprozessen wiederverwendet werden können. Dazu wird in einem Pilotprojekt die Zertifizierung des Zerberus Systems an einer Beispielsanwendung aus der Medizin [4] durch den TÜV angestrebt.

4. Zusammenfassung und Ausblick

Mit dem Zerberus System wird den Entwicklern von sicherheitskritischen Computersystemen ein Entwicklungsmodell mit integrierter Werkzeugunterstützung zur Seite gestellt, das eine schnelle und kostengünstige Realisierung von zuverlässigen und sicheren Systemen unterstützt. Alle Werkzeuge wurden in Form von Prototypen entwickelt und stehen zur Verfügung.

Beispielanwendungen konnten aufzeigen, dass mit dem bestehenden System schnell sichere und zuverlässige Anwendungen mit Regelungszeiten im Bereich von wenigen Millisekunden (AMDk6-Rechner, Kommunikation: Switched Ethernet) entwickelt werden können. Eine Verbesserung der erreichbaren Regelungszeiten kann leicht durch den Einsatz von leistungsfähigerer Hardware erreicht werden.

Der nächste wichtige Schritt ist nun die Durchführung von Pilotprojekten in der Industrie, um einerseits die Anwendbarkeit und die Vorzüge des Ansatzes zu demonstrieren und um andererseits noch intensiver auf die Wünsche der Industrie eingehen zu können, z.B. in Form einer Unterstützung bei der Dokumentengenerierung. Aus diesem Grund sind zwei Projekte mit Industriepartner geplant deren Ergebnisse schnellstmöglich veröffentlicht werden sollen. Da beide Projekte in Form einer Parallelentwicklung (konventionelle Entwicklung+ Entwicklung unter Zuhilfenahme des Zerberus Systems) geplant sind, lassen sich direkte Rückschlüsse in Bezug auf Entwicklungszeiten und -kosten gewinnen.

Ein weiteres wichtiges Ziel ist eine Unterstützung von weiteren Fehlertoleranzmechanismen wie z.B. Rückwärtsbehebung. Durch den Verzicht auf eine zwingende Verwendung von struktureller Hardwareredundanz kann das Anwendungsfeld des Zerberus Systems erweitert werden. Dadurch würde auch eine Entwicklung von Echtzeitsystemen ohne Bedarf an Fehlertoleranzmechanismen unter Zuhilfenahme des Zerberus Systems für Applikationsspezialisten attraktiv, da auf diese Art und Weise nahezu keine Kenntnisse in Bezug auf Echtzeitprogrammierung nötig sind.

Um den Erfolg des Zerberus Systems zu garantieren, ist schließlich noch eine erfolgreiche Zertifizierung durch den TÜV nötig. Wie bereits erwähnt wird dazu eine Beispielanwendung aus der Medizin entwickelt, die letztendlich durch den TÜV zertifiziert werden soll mit einem Augenmerk auf zukünftige Zertifizierungsprozese von mit dem Zerberus System entwickelten Anwendungen.

Literatur

[1] C. Buckl. Zerberus Language Specification Version 1.0. Technical Report TUM-I0501, TU München, Jan. 2005.

[2] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded control systems development with giotto. *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 64 – 72, 2001.

[3] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, pages 166 – 184, 2001.

[4] Kai Schlüter. Adaptiv lernendes System zur cardio-vaskulären Regelung, Apr. 2004.

[5] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126, Jan. 2003.

[6] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, 1985.

[7] S. Poledna, A. Burns, A. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers*, 49:100–110, Feb. 2000.