CIP model-checking

Author(s):
Moglestue, Andreas

Publication Date:
2004

Permanent Link:

https://doi.org/10.3929/ethz-a-004878149 →

DISS. ETH Nr. 15578

CIP Model-Checking

A dissertation submiited to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY

for the degree of

Doctor of Technical Sciences

presented by

Andreas Hubert Moglestue
dipl. El.-Ing ETH Zürich

born 06.09.1971

citizen of
the United Kingdom

accepted on the recommendation of

Prof. Dr. Lothar Thiele
Prof. Dr. Albert Kündig
Prof. Dr. Armin Biere

2004

Zusammenfassung

Verifikation ist ein wichtiger Bestandteil im Entwicklungszyklus jedes Produktes. Um Software vollständig und wiederholbar verifizieren zu können, müssen klar definierte Methoden angewandt werden. *Model Checking* ist eine derartige Methodik. Ein *Model Checker* kann eine passend eingegebene. Systembeschreibung analysieren und Aussagen über das System machen. Einige Beispiele sind Signalsysteme bei der Bahn oder Überwachungssysteme bei der Luftfahrt, bei denen es unerlässlich ist, dass sich überschneidende Trajektorien nie gleichzeitig freigegeben werden können. Oder, im kleineren Maßstab, muss gezeigt werden, dass ein Kommunikationsprotokoll frei von Deadlocks ist. Formell definiert wird eine solche Anforderung eine *Eigenschaft* (*property*) genannt.

Es existieren bereits eine ganze Reihe von Model Checkern und einige davon werden in dieser Arbeit kurz besprochen. Allerdings weicht der Model Checker, der als Thema der vorliegenden Dissertation vorgestellt wird, von diesen ab, indem er eng verknüpft ist mit einem bereits existierenden Entwicklungsrahmen, *CIP*. CIP (Communicating Interacting Processes) bietet bereits deutliche Verbesserungen bei der Entwicklung von eingebetteter Software durch seine Methodik, Struktur, Unterhaltsfreundlichkeit und bei der Erstellung von Dokumentationen. Der Model Checker, der hier vorgestellt wird, benützt diese Strukturen vorteilhaft, indem sie als Basis für seine Verifikationstechnik dienen.

Ein Model Checker interpretiert ein System als eine Menge von *Zuständen* (*states*), welche durch *Übergänge* (*transitions*) untereinander verbunden werden. Der Model Chekker analysiert die so dargestellte Struktur und sucht *Pfade* (*paths*) (durch Übergänge miteinander verbundene Reihen von Zuständen) welche die zu beweisende Eigenschaft widerlegen. Wenn ein derartiges Gegenbeispiel nicht existiert, ist die Eigenschaft bewiesen.

Der offensichtlichste Weg, die nicht-existenz solcher Gegenbeispiele zu beweisen, ist ein systematisches Durchprobieren aller Pfade. In der Praxis ist die Anzahl der möglichen Zustände allerdings derart groß, das der benötigte Speicherplatz und Zeitbedarf ein solches Vorgehen ausschließen. Dieses Problem wird *State Explosion* genannt. In echten Systemen, die mit CIP implementiert werden, wirken die CIP-typischen Interaktionen diesem Problem entgegen. CIP verhindert gewisse Übergänge und verhindert damit die Erreichbarkeit vieler Zustände. Weitere Einsparungen werden ermöglicht durch die Wahl einer geeigneten Darstellung im Speicher.

Zusätzlich kann mit dem sogenannten *Partial Order Reduction* das State Explosion Problem wesentlich vermindert werden. Partial Order Reduction nützt Symmetrieeigenschaften des Systems und reduziert dadurch die Anzahl der Pfade, die betrachtet werden müssen, um zu einer sinnvollen Aussage zu kommen. Die Anzahl der erreichten Zustän-

de und der Zeitbedarf nehmen dadurch ebenfalls ab. Diese Methode macht von dem Umstand Gebrauch, dass nur eine von allen möglichen Reihenfolgen betrachtet werden muß, wenn die Reihenfolge gewisser Übergänge beliebig ist.

Eine wichtige Quelle falscher Ergebnisse, die bei Model Checking auftreten können, sind Pfade, welche von den benötigten Übergängen zugelassen werden, jedoch kein tatsächlich mögliches Verhalten des zu modellierenden physikalischen Systems darstellen. Solche nicht zulässigen Pfade können verursacht werden von Übergangssequenzen, die sich unendlich wiederholen und dadurch die Ausführung anderer Übergänge verhindern, die im physikalischem System sicher zur Ausführung kommen würden. Ein solcher Pfad, welcher im Modell auftreten kann, aber im physikalischen System nicht, wird als *unfair* bezeichnet. Im Modell, fehlen die mechatronische Zusammenhänge die das Unterscheiden von zulässige und nicht-zulässige Pfade ermöglichen. Deshalb müssen vom Benutzer zusätzlich *Fairness Constraints* definiert werden, um in dieser Hinsicht das Modell zu vervollständigen. Dieses ist die einzige Ergänzung des Modells, das der Model Checker dem Benutzer abverlangt.

Der wichtigste Beitrag in diesem Projekt liegt in der Bildung eines Modells für den Model Checker aus dem CIP Modell, und die Behandlung dieses Modells innerhalb des CIP Rahmens. Dieses wird erreicht durch eine Erweiterung der CIP Funktionalität, damit er von einem Anwender, der CIP ausreichend kennt, aber nur sehr geringe Model Checking Kenntnisse besitzt, verstanden und verwendet werden kann.

Diese These führt eine Notation ein zur formellen Beschreibung von CIP Modellen und ihren Komponenten. Das CIP Tool selber wird erweitert durch die Möglichkeit, Fairness Constraints zu definieren, die es dem Benutzer erlauben, das zu modellierende System für den Model Checker besser zu beschreiben, damit keine falschen Gegenbeispiele oder Ergebnisse geliefert werden.

Ein weiterer Beitrag dieses Projektes ist das Hinzufügen eines *Execution Testers* zum CIP Tool. Dieses Werkzeug erlaubt es dem Entwickler, neu definierte oder veränderte Strukturen einfach und rasch zu testen. Dieser Schritt war eine notwendige Grundlage für den Model Checker, bildet aber ebenfalls ein selbständiges Test- und Simulationswerkzeug.

Bei der Umsetzung des CIP Model Checkers wurde auf eine enge Integration mit dem CIP Tool Wert gelegt. Mit diesem wurden Fehler korrekt identifizieren und industrielle Systeme erfolgreich traversiert. Der CIP Model Checker ist eine wertvolle Erweiterung des CIP Tools und hilft CIP Anwendern bessere und sicherere Software zu schreiben.

Abstract

Verification is a central component of the development process of any product. To be able to verify embedded software in a complete, and reproducible way, clearly defined techniques must be applied. *Model Checking* is such a technique. A model checker can analyse an appropriately presented system description and make statements about that system. Some examples are a railway signalling system or an air traffic control system, where it is vital to be able to show that conflicting paths can never be allocated simultaneously. Or on a smaller and more mundane scale, it must be shown that communications protocols are free of deadlocks. When such a requirement is formalised it is called a *property*.

Many model checkers already exist and some are briefly discussed in this thesis. However, the model checker which is developed in this project differs in being closely integrated with a pre-existing development tool, *CIP*. CIP (Communicating Interacting Processes) in itself already offers considerable advances in embedded software development through its methodology, structure and ease of maintenance and documentation. The model checker presented in this thesis takes advantage of these structures and uses them as a basis for integrating a verification technique.

A model checker basically views the system as a set of *states* which are connected by *transitions*. Looking at the structure so represented, it searches for *paths* (sequences of states connected by transitions) which disprove the property to be verified. Failure to find such a counter-example in an exhaustive search of the system proves that the property *holds*.

The most obvious way of executing such a proof is by brute force exhaustive testing of all possible paths. In practice, however, the sheer number of possible states of the system is so great that the physical memory of normal computer systems would be insufficient to hold them and the time required would be prohibitive. This problem is known as the *state explosion* problem. In real systems implemented with CIP Tool this problem is reduced to an extent by the restrictive and interactive nature of CIP. CIP disallows certain transitions and so intrinsically excludes the reacheability of many states. Further savings are made possible by the choice of a suitable coding in memory.

A method known as *partial order reduction* further strongly reduces the state explosion problem. Partial order reduction makes use of symmetries in the system to decrease the number of paths that must be followed and so the number of states found and also the time required. This method makes use of the fact that when transitions are *interleaving* (their order of execution is irrelevant), then only one of all possible orders must be looked at.

An important source of false results in Model Checking is caused by infinite paths which are legal from the point of view of all states being connected by transitions, but do not represent real behaviours of the physical system being modelled. Such illegal paths can be caused by certain sequences of transitions being infinitely repeated to the exclusion of other transitions which certainly would be executed in the real system. Such a path which can occur in the model but not in the physical system is an *unfair* path. *Fairness constraints* can be defined to tell the model checker which transition sequences are unfair.

The principle contribution of this project lies in the extraction of the Model Checking model from the CIP model and the treatment of that model within the CIP framework. This can be achieved through an extension of the CIP functionality in a way which is easily understandable for a user with very basic Model Checking knowledge but adequate understanding of CIP.

This thesis also presents a notation for formally describing CIP models and their components. The CIP Tool itself is also extended through the addition of fairness constraints which allow the user to tell the model checker more about the behaviour of the real system so that no physically impossible counter-examples or claims are produced.

Another addition to the CIP Tool made in this thesis is the *execution tester*. This greatly facilitates testing of the model by the developer. As soon as a component is defined, the user can already test how this component responds to inputs. This was a necessary step towards the creation of a model checker but also provides a useful testing and simulation tool in its own right.

The CIP model checker has been implemented in close integration with CIP Tool. It has been shown to correctly identify constructed errors and successfully traverse real industrial systems. It should prove to be a valuable enhancement to CIP Tool, helping CIP users to create better and safer software.