TECHNISCHE
FAKULTÄT

Department Informatik

Technical Reports / ISSN 2191-5008

Dario Kresic, Kai-Steffen Hielscher, Reinhard German

Spezifikation und Implementation des CAN-Arbitrierungsverfahrens in UPPAAL

Technical Report CS-2010-05

December 2010

Please cite as:
Dario Kresic, Kai-Steffen Hielscher, Reinhard German, “Spezifikation und Implementation des CAN-Arbitrierungsverfahrens in UPPAAL,” University of Erlangen, Dept. of Computer Science, Technical Reports, CS-2010-05, December 2010.

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.informatik.uni-erlangen.de

Spezifikation und Implementation des CAN-Arbitrierungsverfahrens in UPPAAL

Dario Kresic, Kai-Steffen Jens Hielscher, Reinhard German

Kurzfassung: In dieser Arbeit stellen wir eine durch Zeitautomaten modellierte Spezifikation des Mediumzugriffs im CAN-Protokoll vor sowie ihre Implementierung in UPPAAL. Zeitliche Anforderungen wurden dabei durch entsprechende Uhren-Nebenbedingungen erfasst. Dieses Zeitautomaten-Modell wurde anschließend automatisch verifiziert (Model Checking), wobei mehrere Anforderungen identifiziert wurden, die das CAN-Protokoll erfüllen muß (wie Deadlock-Freiheit des Modells, Űbertragungsrecht für die höchstpriore Nachricht, exklusives Űbertragungsrecht für beliebigen CAN-Adapter nach der gewonnenen Arbitrage etc.). All diese Eigenschaften wurden in einer Variante der temporalen Logik CTL spezifiziert; die automatische Verifikation selbst wurde dabei mit UPPAAL durchgeführt, einem Model Checker für zeitautomaten-basierte Modelle.

1. Einleitung

Der CAN-Bus ist ein sehr verbreitetes Kommunikationsmedium in Fahrzeugen, wo zwischen der Teilnehmer Daten wie Drehzahl, Temperatur, Druck etc. ausgetauscht werden. Das CAN-Protokoll geht vom sog. Multimaster-Betrieb aus, wo der Inhalt der über den CAN-Bus übertragenen Rahmen durch Identifier gekennzeichnet ist. Jeder an den CAN-Bus angeschlossene Teilnehmer kann die Rahmen senden und empfangen. Dabei werden etwaige Kollisionen beim Buszugriff durch ein verlustfreies Arbitrierungsverfahren aufgelöst.
In diesem Bericht wird ein auf Zeitautomaten (s. [1]) basierendes Modell des CAN-Arbitrierungsverfahrens in UPPAAL modelliert und analysiert. Der Bericht ist wie folgt aufgebaut: zunächst wird im Abschnitt 2. das CAN-Buszugriffsverfahren kurz vorgestellt. Dieses wird im Abschnitt 3. durch ein UPPAAL-Zeitautomat modelliert und implementiert; zur Verifikation des Modells wurden mehrere Eigenschaften identifiziert und geprüft.

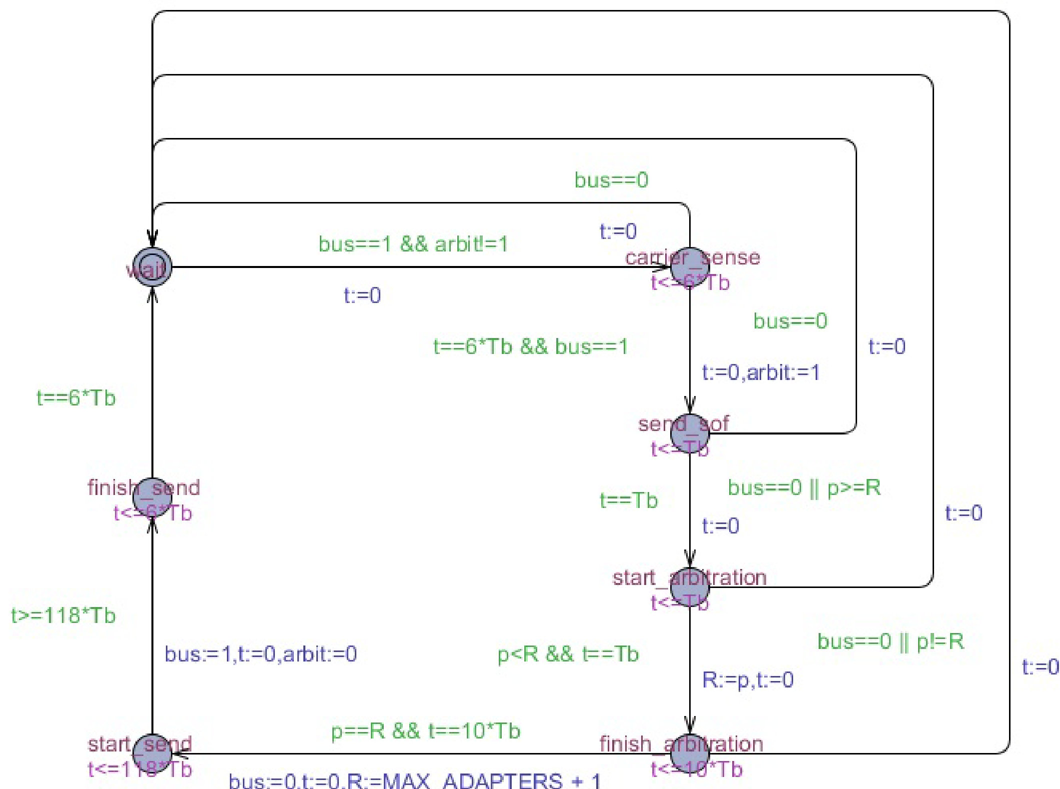
2. Das CAN-Arbitrierungsverfahren

Wesentlich für das Verständnis des CAN-Protokolls ist die Art und Weise des Buszugriffs seitens der Rahmen verschiedener Prioritäten. Dieser Zugriff basiert auf dem sog. CSMA/BA-Verfahren (*Carrier Sense Multiple Access with Bitwise Arbitration*) wo jede Station (Teilnehmer) den Bus abhört, während sie die Daten sendet. Etwaige Kollisionen beim Buszugriff werden durch sog. Bit-Arbitrierung auf Basis des Identifier-Feldes des zu sendenden Rahmens aufgelöst. Dabei überwacht der Sender den Bus, während er gerade den Identifier sendet. Versuchen mehrere Sender gleichzeitig ihre Rahmen senden, so überschreibt das erste dominante Bit (0) eines der Rahmen das korrespondierende rezessive Bit (1) der anderen Rahmen. Die Sender der Rahmen, deren Bit an der Vergleichsstelle rezessiv war, erkennen dies und beenden ihren Übertragungsversuch. Verwenden zwei Sender denselben Identifier, wird nicht sofort Fehler (genauer: ein Error-Rahmen) erzeugt, sondern erst bei einer Kollision innerhalb der restlichen Bits, was durch die Arbitrierung ausgeschlossen sein sollte. Mehr zum Verfahren findet sich in [6].

Das Bit-Arbitrierungsverfahren impliziert damit auch eine Rahmen-Hierarchie. Der Rahmen mit dem niedrigsten Identifier (00000000000) darf nämlich immer übertragen werden. Bei der zeitkritischen Kommunikation ist es dann sinnvoll, Identifier entsprechend hoher Priorität zu vergeben, um so Prioritäten bei der Übertragung zu gewährleisten. Dabei ist es zu berücksichtigen, daß der Zeitpunkt der Sendung nichtdeterministischer Natur ist, da evtl. aktuell übertragene Rahmen nicht unterbrochen werden dürfen. Der Zeitpunkt einer Sendung kann sich somit bis zur maximalen Länge des Rahmens verzögern. Für den höchstprioren Rahmen (und nur für ihn!) kann im Übrigen die maximale Übertragungsverzögerung berechnet werden.

3. UPPAAL-Spezifikation

Das CAN-Protokoll wurde in Form eines einzigen Zeitautomaten modelliert, wie in der folgenden Abbildung dargestellt ist:



Das Arbitrierungsverfahren in unserem Modell basiert auf der folgenden Beobachtung: nachdem jede Instanz des Automaten durch eine ganzzahlige Nummer p eindeutig identifiziert wird, liegt es nahe, diese Nummer mit dem dezimalen Wert des ID-Feldes eines Rahmens zu assoziieren. Dann kann man, nachdem (nichtdeterministisch ausgewählt) mehrere Adapter die Abhörphase (6 Bitzeiten) des Buses erfolgreich beendet haben (und das dominante SOF-Bit geschickt wurde), die p-Nummern vergleichen und dabei das minimale p suchen. Dieser Vergleich findet frühestens nach der Übertragung des ersten ID-Bits (beim Verlassen des Zustands *start_arbitration*) statt und endet (beim Verlassen des Zustands *finish_arbitration*), spätestens nach 11 Bitzeiten (so lang ist ja das ID-Feld eines jeden Rahmens). Verliert ein Adapter die Arbitrierung (d.h. ist die p-Nummer seines Rahmens größer als die aktuell ermittelte minimale p-Nummer), so kehrt er zurück in den Zustand *wait*. Derjenige Adapter, der alle Vergleiche gewinnt (der also den höchstprioren Rahmen senden will), wechselt in den Zustand *start_send,* erklärt den Bus für dominant und sendet die restlichen

Bits. Dieser Wechsel findet nach exakt 11 Bitzeiten statt – dies weicht etwas vom CAN-Standard ab, wo die Arbitrierung ggf. auch früher entschieden werden kann. Dies wurde im Modell aber nicht berücksichtigt, weil wir sicherstellen wollen, daß alle Adapter ihre *p*-Nummern verglichen haben bevor das minimale *p* gefunden wurde (dadurch wird gleichzeitig auch das Synchronisationsproblem gelöst, das in einem früheren Modell aufgetaucht war, wo der höchstpriore Rahmen u.U. kein Übetragungsrecht erhalten hat, weil die Arbitrierung entschieden werden konnte noch bevor das erste ID-Bit des höchstprioren Rahmens geschickt wurde). Da wir jedoch an Grenzwerten (nämlich oberen Schranken) für die Übertragungszeiten interessiert sind, ist die getroffene Lösung unerheblich. Die übrigen Teile der Spezifikation dürften selbsterklärend sein.

4. Verifikation

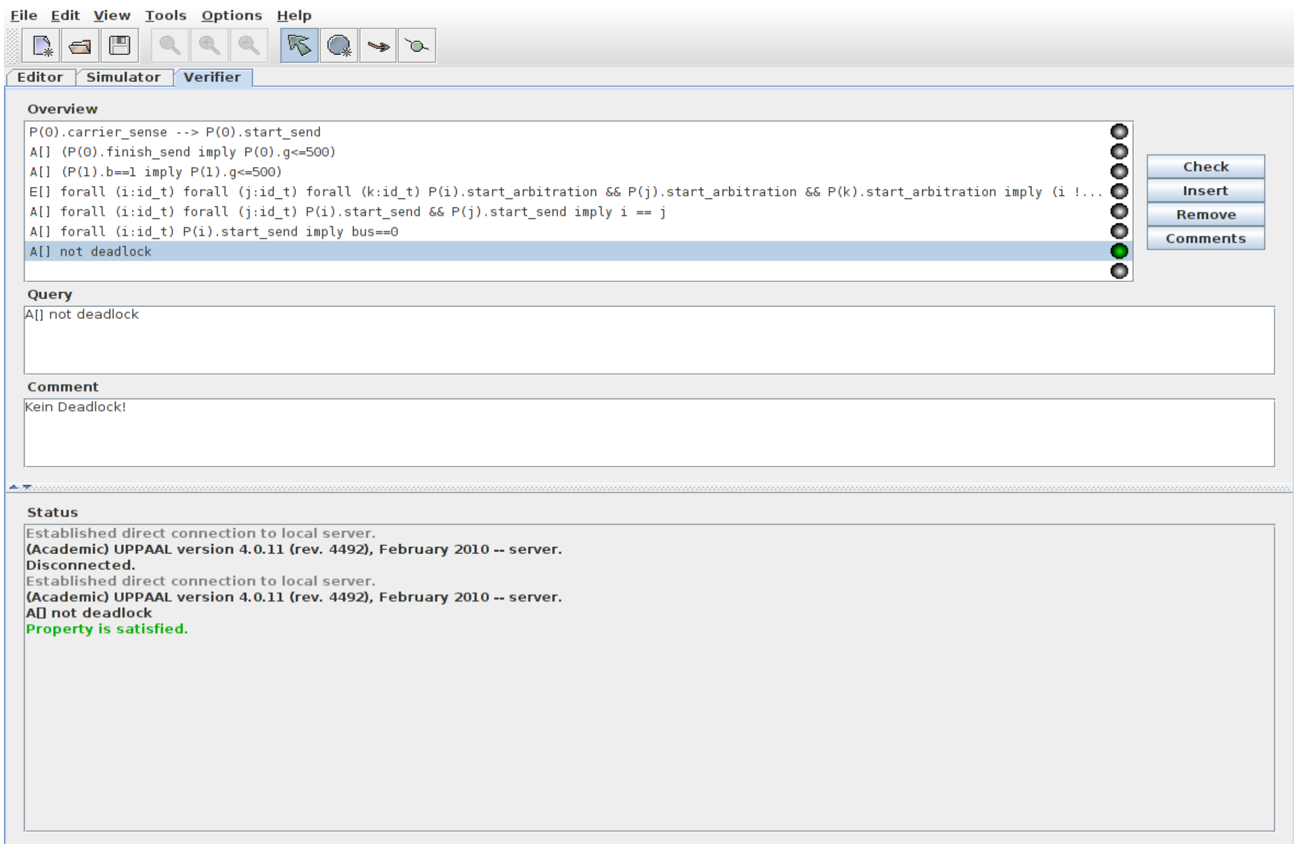
UPPAAL bietet zur Spezifikation von zu verifizierenden Eigenschaften eine vereinfachte Variante von CTL an. Dabei werden jeweils eine Pfad- und eine Zustandsformel definiert, die sich auf ganze Abläufe bzw. auf einzelne Zustände beziehen. UPPAAL unterstützt kein Einbetten von Formeln; ebenso werden nicht alle CTL-Operatoren implementiert (z.B. der *Until*-Operator).

a) Sicherheitseigenschaften

Unter Sicherheitseigenschaften versteht man „intuitiv" Eigenschaften der Form: etwas „Schlechtes" wird nie passieren! Die grundlegende Sicherheitseigenschaft eines jeden Protokolls ist sicherlich die der *Deadlock*-Freiheit. In UPPAAL wird hierfür das Schlüsselwort `deadlock` zur Spezifikation der entsprechenden Eigenschaft angeboten; die *Deadlock*-Freiheit des CAN-Modells kann damit durch den standardisierten Ausdruck

```
A[] not deadlock
```

beschrieben werden:



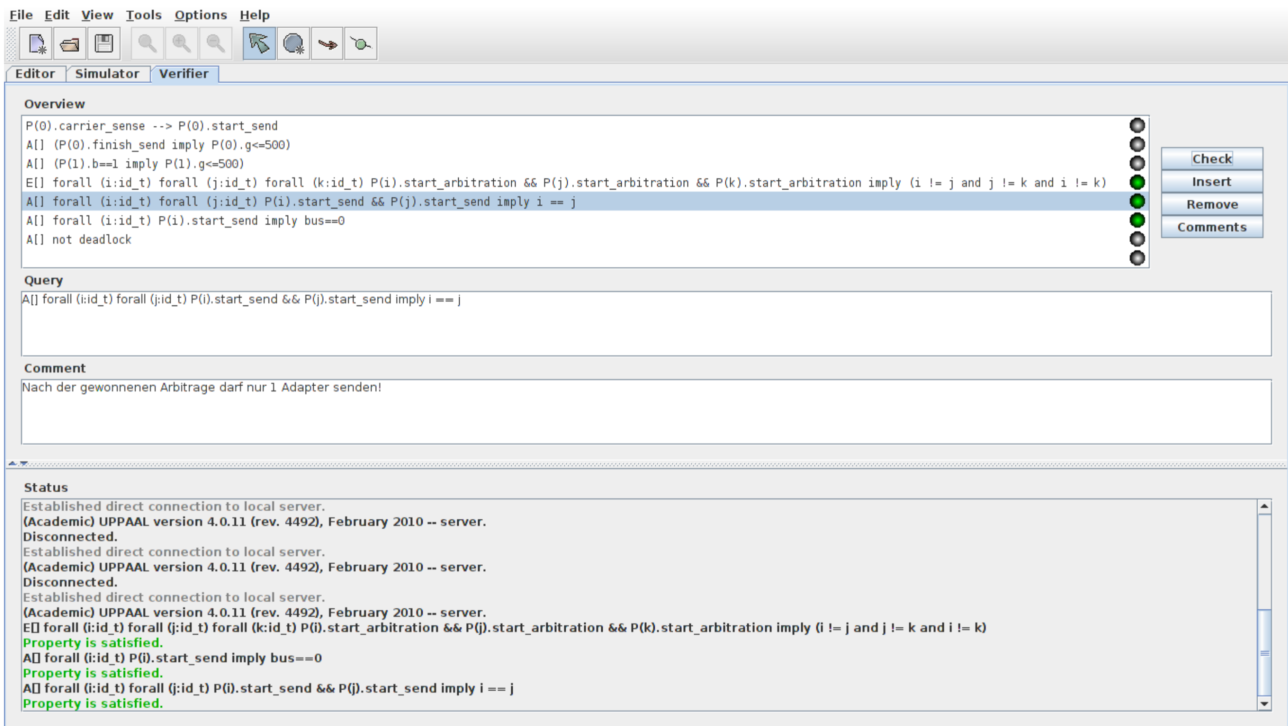
In UPPAAL wird *Deadlock* im Übrigen durch eine Zustandsformel spezifiziert; sie gilt für alle *Deadlock*-Zustände (das sind Zustände aus denen es selbst - oder irgendeinem ihrer Nachfolgezustände - keine Ausgänge geben kann).

Eine *Livelock*-Prüfung kann in UPPAAL dagegen nicht direkt vorgenommen werden. Hierfür könnten zunächst „ausreichend“ viele Simulationen am Modell durchgeführt werden. Denkbar wäre es auch, das *Livelock*-Problem als Erreichbarkeitsproblem aufzufassen und dann entsprechende Verifikation durchzuführen.

Eine weitere notwendige Sicherheitseigenschaft ist die folgende: zu jedem Zeitpunkt darf nur ein einziger Adapter senden (nachdem er die Arbitrierung gewonnen hat). Hier kann man – unter dem Einsatz des Universalquantors `forall` (verfügbar ab Uppaal 4.0) – folgende Anfrage aufstellen:

```
A[] forall (i:id_t) forall (j:id_t) P(i).start_send &&
                    P(j).start_send imply i == j
```

Der Uppaal-Verifikator meldet, wie man es auf der Abbildung unten sehen kann, daß diese Eigenschaft erfüllt ist.



Diese Anfrage ist zugleich ein Beispiel für die Nutzung der Pfadformel „A□ f“ in Uppaal (dargestellt durch den Ausdruck „A[] f“). Diese Pfadformel gilt dann, wenn in jedem (erreichbaren) Zustand die Formel f gilt (also auf allen Pfaden und in allen erreichbaren Zuständen).

b) Lebendigkeitseigenschaften

Unter Lebendigkeitseigenschaften versteht man „intuitiv“ Eigenschaften der Form: etwas „Gutes“ kann (in Zukunft) passieren! Eine solche Eigenschaft in unserem Arbitrierungsmodell ist die folgende: wir möchten nämlich sicherstellen, daß es u.U. auch solche Situationen gibt, wo sich zum „gleichen“ Zeitpunkt alle Adapter um das Zugriffsrecht auf den Bus bewerben können. Da dies natürlich nicht immer der Fall sein muß, können wir unsere Anforderung etwas „abschwächen“ und die Pfadformel „E□ f“ (in Uppaal dargestellt durch den Ausdruck „E[] f“) verwenden:

```
    E[] forall (i:id_t) forall (j:id_t) forall (k:id_t)
    P(i).start_arbitration && P(j).start_arbitration &&
P(k).start_arbitration imply (i != j and j != k and i != k)
```

Der Übersichtlichkeit der Spezifikation halber gehen wir von 3 (statt 4) Adaptern aus (die gewünschte Eigenschaft läßt sich jedoch auf beliebig viele Adapter ausdehnen).

Der Uppaal-Verifikator meldet, wie man es auf der Abbildung unten sehen kann, daß auch diese Eigenschaft erfüllt ist.

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

P(0).carrier_sense --> P(0).start_send
A[] (P(0).finish_send imply P(0).g<=500)
A[] (P(1).b==1 imply P(1).g<=500)
E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)
A[] forall (i:id_t) forall (j:id_t) P(i).start_send && P(j).start_send imply i == j
A[] forall (i:id_t) P(i).start_send imply bus==0
A[] not deadlock

Check
Insert
Remove
Comments

Query

E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)

Comment

In der Arbitrierung können sich mehrere (z.B. 3) Adapter befinden!

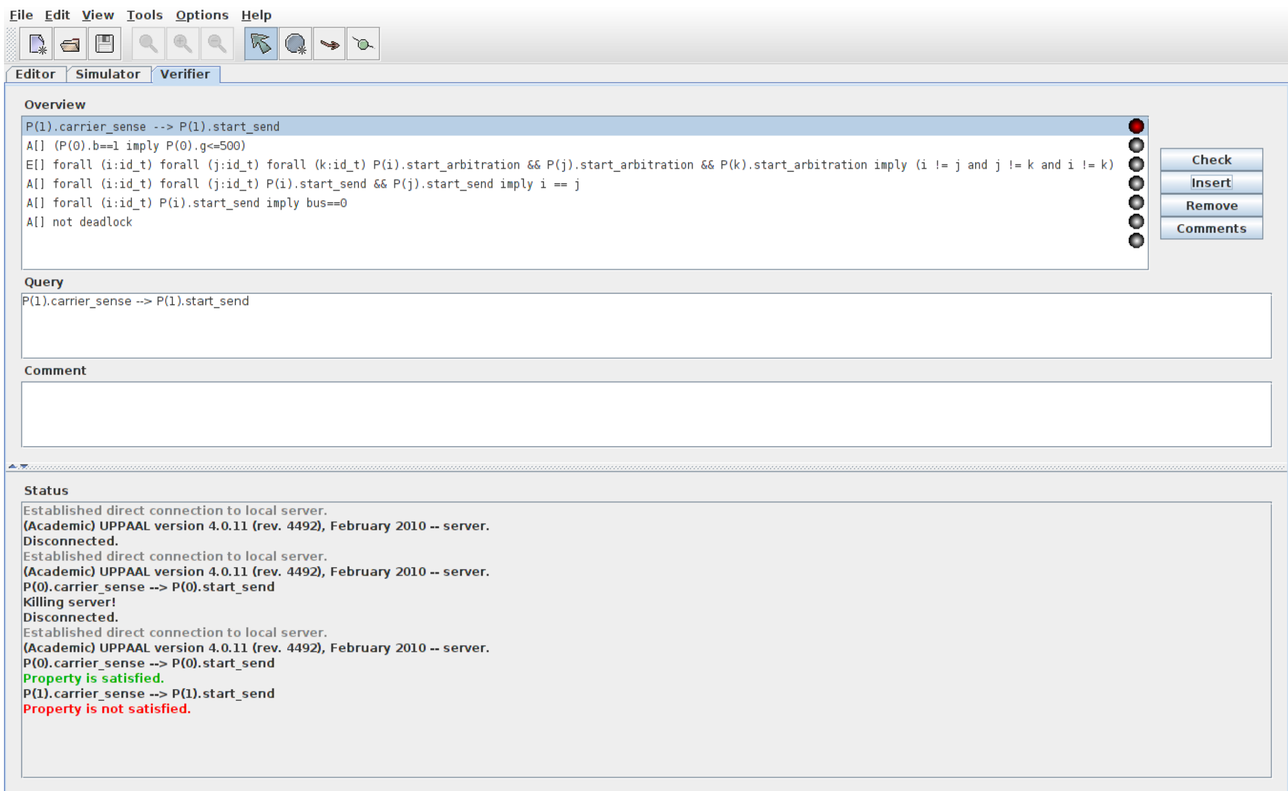
Status

Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.11 (rev. 4492), February 2010 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.11 (rev. 4492), February 2010 -- server.
E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)
Property is satisfied.
A[] forall (i:id_t) P(i).start_send imply bus==0
Property is satisfied.
A[] forall (i:id_t) forall (j:id_t) P(i).start_send && P(j).start_send imply i == j
Property is satisfied.
E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)
Property is satisfied.

Eine weitere Lebendigkeitseigenschaft im CAN-Modell ist diese: der Adapter mit der *p*-Nummer 1 soll nicht immer aus dem Zustand *carrier_sense* in den Zustand *start_send* wechseln (d.h. solcher Adapter kann die Arbitrierung u.U. mal verlieren). Dies wird nämlich immer dann der Fall sein, wenn sich der höchstpriore Adapter (also solcher mit der *p*-Nummer 0) um die Arbitrierung bewirbt! In UPPAAL steht für solche Lebendigkeitseigenschaften der Operator „-->“ zur Verfügung; die entsprechende Eigenschaft kann somit wie folgt ausgedrückt werden:

```
P(1).carrier_sense - - > P(1).start_send
```

Erwartungsgemäß meldet der Uppaal-Verifikator, wie man es auf der Abbildung auf der nächsten Seite sehen kann, daß diese Eigenschaft nicht erfüllt ist (also nicht immer gelten muß). Würde man jedoch dieselbe Anfrage für den höchstprioren Adapter (also `P(0)`) stellen, so wäre die Eigenschaft erfüllt.

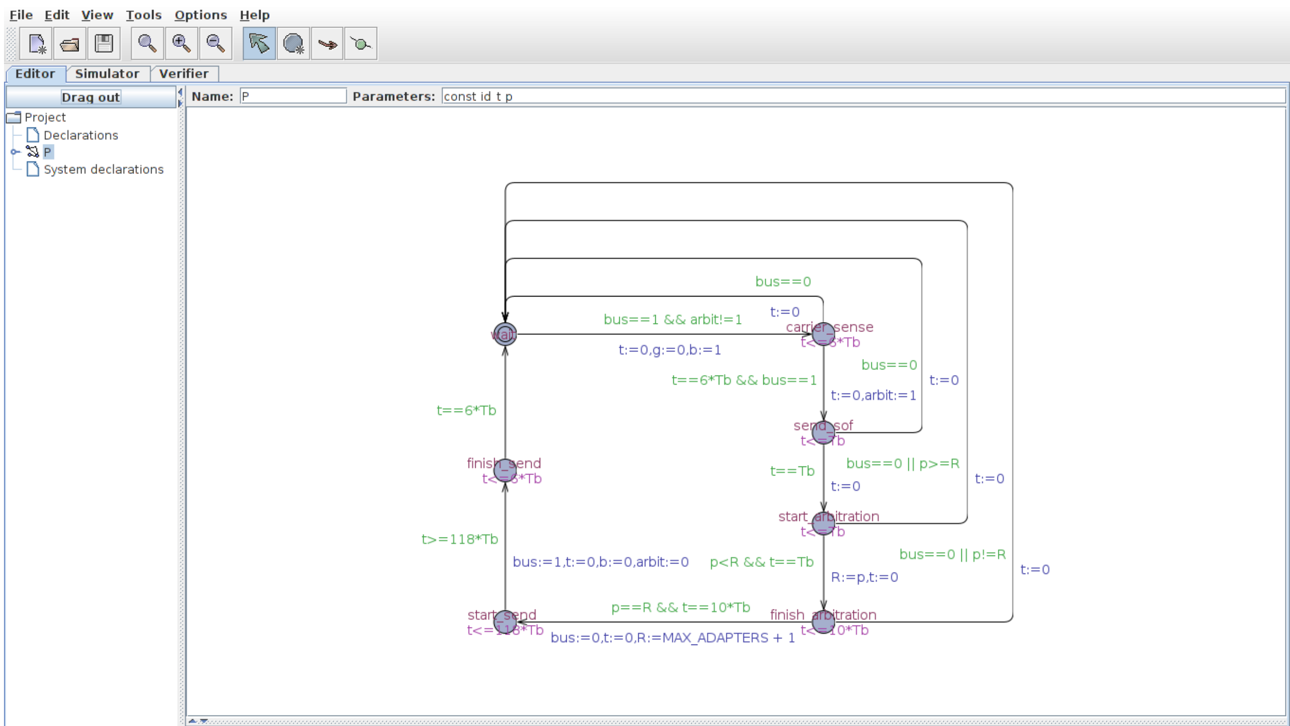


c) Echtzeitlebendigkeit

Die Klasse der sog. Echtzeitlebendigkeitseigenschaften bezieht sich auf solche (Lebendigkeits-) Eigenschaften, die innerhalb einer festgelegten (oben beschränkten) Zeitschranke gelten müssen. Dadurch kann zusätzliche (Zeit-)Information geliefert werden, was insbesondere dann von Bedeutung ist, wenn feste Zeitschranken erfüllt werden müssen.

Um solche Eigenschaften in UPPAAL verifizieren zu können, müssen UPPAAL-Modelle zunächst erweitert werden: um eine zusätzliche (lokale) Uhrenvariable *z* sowie um eine boolesche Variable *b*. Die boolesche Variable wird dabei initial auf *false* gesetzt; wenn eine Zustandseigenschaft gültig wird, so wird *b* auf *true* gesetzt und die Uhr *z* zurückgesetzt. Durch den Ablauf der Uhr *z* wird die Zeit gemessen, die bis zum Zurücksetzen von *b* auf *false* verstreicht. In UPPAAL werden solche Echtzeitlebendigkeitseigenschaften auf die „normale“ Lebendigkeitseigenschaften reduziert; ihre Form ist „`A[](b ==> z<=t)`“.

Auf dieser Basis wurde unser CAN-Modell entsprechend um eine boolesche Variable *b* und um eine lokale Uhr *g* erweitert (beide werden auf dem Übergang von *wait* nach *carrier_sense* initialisiert, s. die Abbildung unten). Nach dem Ende des Ablaufs (beim nochmaligen Eintritt in den Zustand *wait*) wird *b* auf *false* zurückgesetzt (im Deklarationsteil).



Damit kann die erste Echtzeitlebendigkeitseigenschaft, nämlich „*Ist es sichergestellt, daß der höchstpriore Adapter immer innerhalb von 136 Bitzeiten mit dem Senden eines Rahmens fertig wird?*“, wie folgt spezifiziert werden:

```
A[] (P(0).b==1 imply P(0).g<=272)
```

(da eine Bitzeit hier *2 ns* beträgt, wurde der Wert *272* als zeitliche Oberschranke genommen). Wie man auf der Abbildung unten sehen kann, ist diese Eigenschaft erfüllt.

File Edit View Tools Options Help
Editor Simulator Verifier

Overview

```
P(0).carrier_sense --> P(0).start_send
A[] (P(0).b==1 imply P(0).g<=272)
E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)
A[] forall (i:id_t) forall (j:id_t) P(i).start_send && P(j).start_send imply i == j
A[] forall (i:id_t) P(i).start_send imply bus==0
A[] not deadlock
```

Check
Insert
Remove
Comments

Query

A[] (P(0).b==1 imply P(0).g<=272)

Comment

Adapter X beendet das Senden des gesamten Rahmens in weniger als Y Zeiteinheiten.

Status

```
(Academic) UPPAAL version 4.0.11 (rev. 4492), February 2010 -- server.
E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)
Property is satisfied.
A[] forall (i:id_t) P(i).start_send imply bus==0
Property is satisfied.
A[] forall (i:id_t) forall (j:id_t) P(i).start_send && P(j).start_send imply i == j
Property is satisfied.
E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)
Property is satisfied.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.11 (rev. 4492), February 2010 -- server.
A[] (P(0).b==1 imply P(0).g<=272)
Property is satisfied.
```

Würde man jedoch dieselbe Eigenschaft auch für den Adapter mit der *p*-Nummer 1 verifizieren wollen (man setze dann anstelle von „0“ den Wert „1“ in die obige Formel), so wäre diese Eigenschaft nicht mehr erfüllt! Dies gilt auch dann, wenn man eine (beliebig) „große“ obere Schranke wählen würde (s. die Abbildung unten). Dies ist genau im Sinne des CAN-Standards, wo man die obere Schranke nur für den höchstprioren Adapter garantieren kann, nicht jedoch für alle anderen.

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

```
P(0).carrier_sense --> P(0).start_send
A[] (P(1).b==1 imply P(1).g<=150000)
E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)
A[] forall (i:id_t) forall (j:id_t) P(i).start_send && P(j).start_send imply i == j
A[] forall (i:id_t) P(i).start_send imply bus==0
A[] not deadlock
```

Check
Insert
Remove
Comments

Query

A[] (P(1).b==1 imply P(1).g<=150000)

Comment

Adapter X beendet das Senden des gesamten Rahmens in weniger als Y Zeiteinheiten.

Status

```
Property is satisfied.
A[] forall (i:id_t) forall (j:id_t) P(i).start_send && P(j).start_send imply i == j
Property is satisfied.
E[] forall (i:id_t) forall (j:id_t) forall (k:id_t) P(i).start_arbitration && P(j).start_arbitration && P(k).start_arbitration imply (i != j and j != k and i != k)
Property is satisfied.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.11 (rev. 4492), February 2010 -- server.
A[] (P(0).b==1 imply P(0).g<=272)
Property is satisfied.
A[] (P(1).b==1 imply P(1).g<=272)
Property is not satisfied.
A[] (P(1).b==1 imply P(1).g<=150000)
Property is not satisfied.
```

5. Zusammenfassung und Ausblick

In dieser Arbeit wurde das CAN-Arbitrierungsverfahren mit Hilfe eines Zeitautomaten modelliert und anschließend in UPPAAL implementiert und verifiziert. Dabei ist es gelungen, den Medienzugriff im CAN-Protokoll durch lediglich einen einzigen Zeitautomaten zu beschreiben, statt durch eine Komposition von mehreren Automaten wie in einigen früheren Arbeiten [4].
Um obere Schranken für Űbertragungszeiten von Rahmen beliebiger Prioritäten zu bestimmen, braucht man jedoch einiger Erweiterungen im Modell selbst. Dazu bieten sich sog. Zyklen an, die festlegen, in welchen Intervallen ein jeder Adapter einen Rahmen schicken darf [3]. Solche Zyklen sollen in UPPAAL später implementiert werden.
Das auf Model Checking basierende Verifikationsparadigma in UPPAAL erwies sich in unseren Szenarien leider als wenig effizient – wir konnten effektiv von nur 4 CAN-Adaptern ausgehen! Bereits bei 6 Adaptern dauerte etwa die Deadlock-Űberprüfung länger als 1 Stunde. Als eine Möglichkeit, diesem Problem zu begegnen, bietet sich u.a. an, unser Zeitautomaten-Modell mit expliziten Prioritäten, d.h. mit prioritisierten Zustandsübergängen zu erweitern. Zeitautomaten mit Prioritäten können nämlich noch kompakter dargestellt werden, was auch eine effizientere Verifikation verspricht. In UPPAAL gibt es bereits die Möglichkeit, solche Prioritäten explizit auszudrücken.

6. Literatur

[1] R. Alur, D.L. Dill: A Theory of Timed Automata. Theoretical Computer Science, 126:183-235, 1994.
[2] G. Behrmann, A. David, K. G. Larsen: A Tutorial on Uppaal, 2004.
[3] T. Herpel, K.S.J. Hielscher, U. Klehmet, R. German: Stochastic and Deterministic Performance Evaluation of Automotive CAN Communication. Computer Networks 53, 2009.
[4] J. Krakora, Z. Hanzalek: Verifying Real-Time Properties of CAN Bus by Timed Automata. FISITA 2004 - World Automotive Congress, Barcelona, 2004.
[5] D. Kresic: Verifying Real-Time Properties by Intelligent Parsing. Proc. of the IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS 2009), Shanghai, 2009.
[6] W. Lawrenz (Hrsg.): CAN Controller Area Network - Grundlagen und Praxis. Hüthig, ISBN 3-7785-2780-0, 2010.