Anwendung spezieller Modellprüfungstechniken bei der Verifikation einer mehrfädigen Prozessorarchitektur

Studienarbeit

Jochen Schröder-Babo

14. Dezember 1998

bei: Prof. Dr.-Ing. D. Schmid
Betreuerin: Dr.rer.nat. M. Huhn
Betreuer: Dr.rer.nat. K. Schneider

Erklärung

Hiermit versichere ich, daß ich die vorliegende Studienarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 14. Dezember 1998

Inhaltsverzeichnis

Kapitel 1

Motivation und Aufgabenstellung

1.1 Motivation

Bei dem Entwurf von elektronischen Schaltungen können Fehler auftreten. Diese Entwurfsfehler entstehen bei der Umsetzung der Entwurfsvorgaben (Spezifikation) in eine konkrete Realisierung (Implementierung). Da einerseits die nachträgliche Beseitigung von Entwurfsfehlern sehr teuer ist und andererseits elektronische Schaltungen oftmals in sicherheitskritischen Anwendungen (Steuerelektronik im Flugzeug oder Automobil etc.) eingesetzt werden, ist man sehr daran interessiert, Entwurfsfehler zu vermeiden.

Die Entwurfsfehlerfreiheit einer Schaltung läßt sich – theoretisch – mit einer vollständigen Simulation zeigen. Hierbei sind, für alle möglichen Kombinationen der verschiedenen internen Zustände der Schaltung mit den verschiedenen Eingabewerten, die korrekten Ausgabewerte mit den tatsächlichen Ausgabewerten zu vergleichen. Der Aufwand für diese Vorgehensweise wächst exponentiell mit dem Umfang der Eingabe- und Zustandsvariablen, weshalb eine vollständige Simulation schon bei sehr kleinen Schaltungen nicht mehr möglich ist.

Die Abwesenheit von Entwurfsfehlern läßt sich, aufgrund der im allgemeinen hohen Zustandszahl, nicht durch Simulation, sondern nur durch Verifikation garantieren. Für die Hardware-Verifikation ist ein formaler Beweis zu führen, daß Implementierung und Spezifikation äquivalent sind oder daß aus der Implementierung die Spezifikation folgt. Ein Nachweis der Äquivalenz ist im allgemeinen nicht möglich, da die Spezifikation Entwurfsfreiheiten enthält und somit nicht äquivalent zur Implementierung sein kann. Es reicht aber aus, zu zeigen, daß aus der Implementierung die Spezifikation folgt, da dies bedeutet, daß die Entwurfsvorgaben erfüllt werden.

Eine Methode, mit der man diesen Beweis führen kann, ist die Modellprüfung. Bei der Modellprüfung wird die Implementierung als eine endliche temporale Struktur dargestellt. Eine temporale Struktur besteht aus Zuständen und Zustandsübergängen. Die Zustände sind mit den Variablen markiert, die im entsprechenden Systemzustand den Wert *wahr* annehmen. Die Spezifikation wird als temporallogische Formel angegeben. Durch temporallogische Formeln lassen sich nicht nur logische Zusammenhänge, sondern auch zeitliche Abhängigkeiten zwischen den Systemvariablen beschreiben. Zu prüfen ist, ob die durch die Struktur gegebene Abfolge von Belegungen ein Modell für die Formel ist und somit die Formel erfüllt.

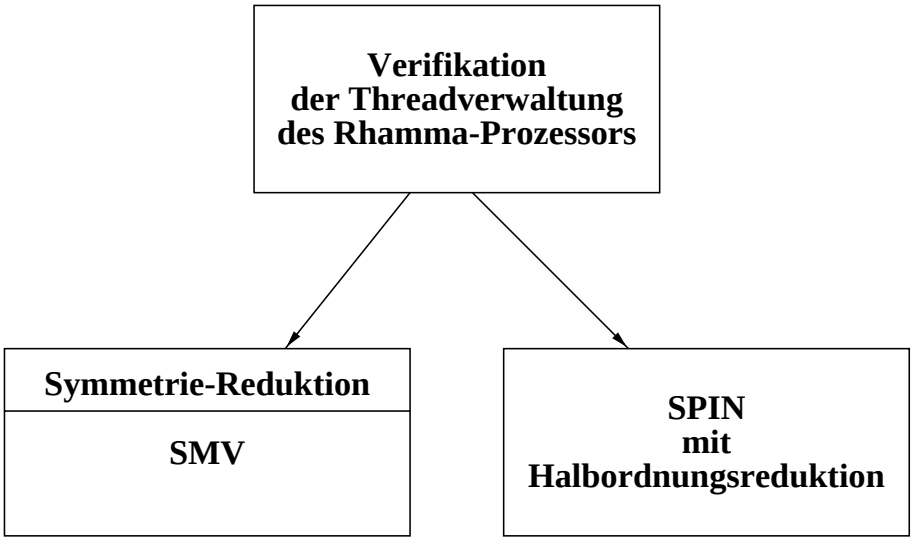

Abbildung 1.1: Aufgabenstellung

1.2 Aufgabenstellung

Die Aufgabe bei dieser Arbeit war, mit Modellprüfung die Korrektheit der Threadverwaltung bei einer mehrfädigen Prozessorarchitektur – dem Rhamma-Prozessor [GrUn96] – zu zeigen. Dabei waren auch Meßdaten über die Wirkung spezieller Modellprüfungstechniken zu sammeln. Diese speziellen Techniken sind die Halbordnungsreduktion [PePH96, Pele93, Pele96] und die Symmetrie-Reduktion [ClFJ93, ClGL92].

Für diese Arbeit vorgegeben waren die Spezifikation des Rhamma-Prozessors und eine Beschreibung des Prozessors, die von [GrSc98] für die Verifikation des Threadmanagements mit der alten Implementierung des Modellprüfers SMV ohne Symmetrie-Reduktion benutzt wurde. Für die Verifikation waren die beiden Modellprüfer SPIN [HoPe96, Holz92] und SMV [McMi93a, McMi92b, CMCH96] gegeben, die mit unterschiedlichen Ansätzen arbeiten und unterschiedliche Techniken unterstützen. Der wesentliche Unterschied ist, daß der SMV für CTL-Modellprüfung [BCLM94, BCMD90, CGHJ93a] ist und mit BDDs (symbolische Darstellung des Zustandsraums) [Brya86] arbeitet, während SPIN für LTL-Modellprüfung [Pnue77, MaPn92] und automatenbasiert ist. SPIN unterstützt die Halbordnungsreduktion, während der SMV in einer neuen Implementierung die Symmetrie-Reduktion unterstützt. Im Rahmen dieser Arbeit waren Beschreibungen des Rhamma-Prozessors für SPIN und für den neuen SMV zu erstellen und die Verifikationen durchzuführen.

In den nächsten Kapiteln werden zunächst der Rhamma-Prozessor und die beiden Modellprüfer SPIN und SMV vorgestellt. Anschließend werden die Beschreibung des Rhamma-Prozessors für den Modellprüfer SPIN erläutert und die Meßergebnisse der Verifikationen mit SPIN und dem SMV angegeben. Abschließend werden die Ergebnisse dieser Arbeit zusammengefaßt.

Kapitel 2

Der Rhamma-Prozessor

Der Rhamma-Prozessor ist eine mehrfädige Mikroprozessorarchitektur, die einen schnellen Kontextwechsel erlaubt. Dazu besitzt der Prozessor eine Ausführungseinheit und mehrere Registersätze, denen jeweils ein Thread dynamisch zugeordnet wird, sowie einen Befehlssatz, der Befehle für die Threaderzeugung enthält.

2.1 Das virtuelle Thread Modell (VTM)

Das Programmiermodell des Rhamma-Prozessors ist das virtuelle Thread-Modell. Im virtuellen Modell des Prozessors geht man vereinfachend von einer unbegrenzten Anzahl an Threads und Registersätzen aus. Eine weitere Vereinfachung im virtuellen Modell ist die Annahme, daß alle aktiven Threads parallel einen neuen Befehl ausführen können. Ein Thread befindet sich immer in einem der drei Zustände *active*, *stopped* oder *free*. Zustandsübergänge finden durch Signale statt, die durch die Ausführung spezieller Befehle des Befehlssatzes erzeugt werden. Dies sind die Befehle:

getfr: Ein neuer Thread wird erzeugt. Der Befehl bewirkt das Signal $getfr_j$ an einen Thread j im Zustand *free* und dadurch den Zustandsübergang des Thread j vom Zustand *free* in den Zustand *stopped*.

relfr(j): Der Thread j wird freigegeben. Der Befehl bewirkt das Signal $relfr_j$ an den Thread j und dadurch den Zustandsübergang in den Zustand *free*. Hierbei muß gelten: $i \neq j \Rightarrow$ *state(j)* = *stopped*, wenn i die Nummer des Threads ist, der den Befehl *relfr(j)* ausführt.

start(j): Der Thread j wird gestartet. Der Befehl bewirkt das Signal $start_j$ an den Thread j und somit den Übergang in den Zustand *active*. Hierbei muß gelten: *state(j)* = *stopped*.

stop: Der Thread beendet seine Ausführung und geht in den Zustand *stopped* über.

Alle weiteren Befehle des Befehlssatzes bewirken keine Änderungen des Zustandes der Threads und können deshalb für die Verifikation durch den Befehl **noop** ersetzt werden.

2.2 Verfeinerung: MPM und OSM

Da im virtuellen Modell, im Gegensatz zu real existierenden Prozessoren, unendlich viele Registersätze vorhanden sind, ist es notwendig, das Modell zu verfeinern. In

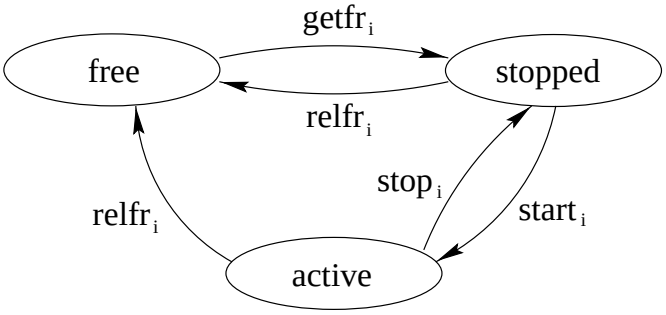

Abbildung 2.1: Zustandsübergänge für Thread i im VTM

einer Verfeinerung des Modells geht man davon aus, daß nur endlich viele Registersätze zur Verfügung stehen, die zu jedem Zeitpunkt eine Teilmenge der vorhandenen Threads enthalten. Threads, denen zum jeweiligen Zeitpunkt kein Registersatz zur Verfügung steht, werden in den Speicher ausgelagert. Dadurch ergibt sich eine Zweiteilung des Modells in die Modelle OSM (Betriebsystemmodell, Speichermodell) und MPM (Mikroprozessormodell), die gemeinsam das virtuelle Modell implementieren. Die Übergänge zwischen beiden Modellen werden mit Hilfe von Interrupt-Routinen realisiert. Da einerseits nicht immer allen aktiven Threads ein Registersatz zugeordnet sein kann und andererseits nur eine Ausführungseinheit zur Verfügung steht, muß die Forderung nach der gleichzeitigen Ausführung aller aktiven Threads aufgegeben werden. Hierzu führt man ein Signal *enable* pro Registersatz ein und verlangt, daß einerseits zu jedem Zeitpunkt eines der *enable*-Signale gesetzt ist und andererseits, daß jedes *enable*-Signal unendlich oft gesetzt ist.

2.2.1 Betriebssystemmodell (OSM)

Das Betriebssystemmodell verwaltet den Zustand aller Threads, dabei ist der genaue Zustand nur für die Threads bekannt, denen zum jeweiligen Zeitpunkt kein Registersatz zugeordnet ist. Im OSM befindet sich jeder Thread in einem der folgenden vier Zustände: *active*, *stopped*, *free* und *on processor*. Gegenüber dem VTM ist hier der Zustand *on processor* hinzugekommen, in dem sich ein Thread genau dann befindet, wenn ihm ein Registersatz zugeordnet ist. In den restlichen Zuständen befindet sich der Thread genau dann, wenn er sich im VTM in dem korrespondierenden Zustand befindet und ihm kein Registersatz zugeordnet ist. Der Thread ist demnach in den Speicher ausgelagert.

Die entsprechenden Zustandsübergänge geschehen durch die Ausführung einer der folgenden Interrupt-Routinen:

freeTag: Die Routine *freeTag* lagert einen freien Thread von dem ihm zugeordneten Registersatz in den Speicher aus. Im OSM erfolgt dadurch der Übergang vom Zustand *on processor* in den Zustand *free*.

newTag: Dies ist die zu *freeTag* inverse Interrupt-Routine. Einem freien Thread im Speicher wird ein Registersatz zugeordnet. Der Thread geht im OSM vom Zustand *free* in den Zustand *on processor* über.

writeOut: Ein gestoppter Thread wird in den Speicher ausgelagert.

readIn: Ein gestoppter Thread wird in einen Registersatz geladen.

o_load: Ein aktiver Thread wird in den Speicher zurückgeschrieben.

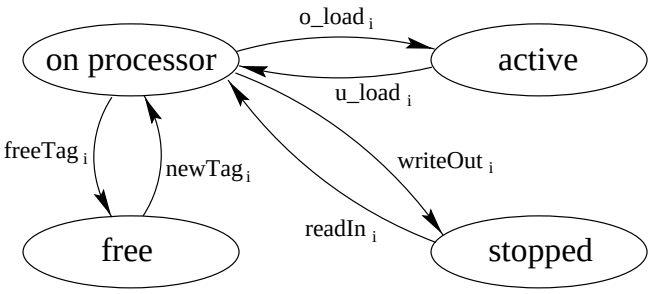

Abbildung 2.2: Zustandsübergänge für Thread i im OSM

u_load: Einem aktiven Thread wird ein Registersatz zugeordnet. Voraussetzung für ein faires Threadmanagement ist eine faire Interrupt-Routine *u_load*. Wenn die Routine *u_load* unendlich oft aufgerufen wird, dann muß jeder Thread, der unendlich oft aktiv ist, auch unendlich oft ausgewählt werden.

2.2.2 Mikroprozessormodell (MPM)

Das Mikroprozessormodell verwaltet die Zustände der Registersätze und somit den Zustand der Threads, denen zum jeweiligen Zeitpunkt ein Registersatz zugeordnet ist. Dabei befindet sich ein Registersatz in einem der Zustände *active0*, *stopped0* oder *free0*, wenn sich der diesem Registersatz zugeordnete Thread im VTM im entsprechenden Zustand befindet. Zusätzlich zu diesen Zuständen existiert ein Zustand *rsfree0*, den die Registersätze annehmen, denen kein Thread zugeordnet ist. Neben diesen Grundzuständen unterscheidet man eine Reihe von Zwischenzuständen, die bei der Ausführung einer Interrupt-Routine durchlaufen werden. Die Zustandsübergänge geschehen durch die vom VTM bekannten Signale sowie durch die zusätzlichen Signale in Tabelle 2.1. Falls die Ausführung eines Befehls eine Interrupt-Sequenz auslöst, wird das entsprechende VTM-Signal – $getfr_j$, $relfr_j$ oder $start_j$ – aufgrund der Zwischenzustände ein- oder zweimal wiederholt, bis der Zielzustand erreicht ist.

Die Ausführung einer Interrupt-Routine ist unter anderem dann notwendig, wenn ein Thread einen Befehl ausführt, der sich auf einen anderen Thread bezieht, dem zur Zeit kein Registersatz zugeordnet ist. Dies ist zum Beispiel in der folgenden Situation der Fall: Thread j ist in den Speicher ausgelagert und Thread i, dem ein Registersatz zugeordnet ist, führt den Befehl *start(j)* aus. Je nach Zustand der übrigen Registersätze werden dann ein oder zwei Interrupt-Routinen ausgeführt. Falls ein Registersatz im Zustand *rsfree0* vorhanden ist, wird die Interrupt-Routine *readIn* ausgeführt, um den Thread j vom Speicher in diesen Registersatz zu laden. Wenn sich kein Registersatz im Zustand *rsfree0* befindet, aber ein Registersatz im Zustand *free0* oder *stopped0* existiert, wird zunächst die Interrupt-Routine *freeTag* oder *writeOut* ausgeführt, bevor die Routine *readIn* ausgeführt werden kann. Wenn keiner der Registersätze im Zustand *rsfree0*, *free0* oder *stopped0* ist, kann der Befehl zunächst nicht ausgeführt werden, da keine Registersätze, denen ein aktiver Thread zugeordnet ist oder die schon an einem Interrupt beteiligt sind, als Interrupt-Partner ausgewählt werden. Für diesen Fall existiert der Wartezustand *active0_waiting*, in dem sich zu jedem Zeitpunkt höchstens einer der Registersätze befinden darf. Wenn sich schon einer der anderen Registersätze im Zustand *active0_waiting* befindet, wird der Thread i ausgelagert.

Die restlichen Befehle werden, ähnlich wie der Befehl *start(j)* in diesem Beispiel, bearbeitet. Ein wesentlicher Unterschied ist dabei nur, daß beim Befehl *getfr* der Re-

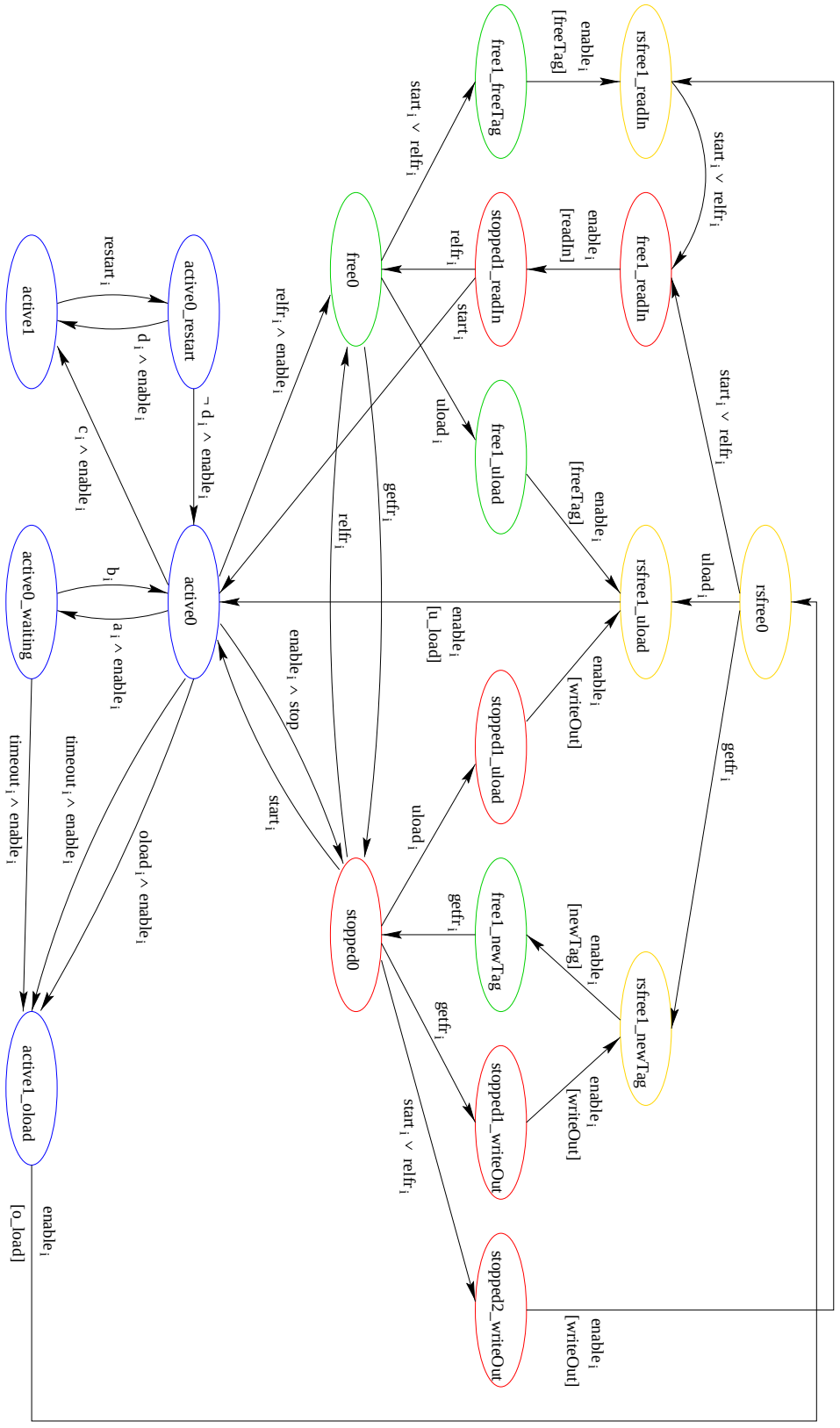

Abbildung 2.3: Zustandsübergänge für Registersatz i

Signal	Wird für Registersatz i gesetzt, wenn ...
a_i	der Befehl zur Zeit nicht ausführbar ist und sich kein Registersatz im Zustand *active0_waiting* befindet.
b_i	sich der Registersatz im Zustand *active0_waiting* befindet und der Befehl wieder ausführbar wird.
c_i	zur Ausführung des Befehls ein Interrupt nötig ist.
$\neg d_i$	die durch den Befehl des Registersatzes ausgelöste Interrupt-Sequenz beendet ist.
$oload_i$	der Befehl zur Zeit nicht ausführbar ist und sich ein anderer Registersatz im Zustand *active0_waiting* befindet.
$restart_i$	eine Interrupt-Routine, die durch den Befehl des Registersatzes ausgelöst wurde, terminiert.
$enable_i$	dem Registersatz die Ausführungseinheit zur Verfügung steht.
$uload_i$	in den Registersatz ein aktiver Thread aus dem Speicher geladen werden soll, da kein Registersatz einen aktiven Thread enthält.
$timeout_i$	der Thread des Registersatzes in den Speicher ausgelagert werden soll.

Tabelle 2.1: Zusätzliche Signale im MPM

gistersatz des Partners zustandsabhängig in der Reihenfolge *free0*, *rsfree0*, *stopped0* ausgewählt wird, während bei den Befehlen *start(j)* und *relfr(j)* in der Reihenfolge *rsfree0*, *free0*, *stopped0* vorgegangen wird.

Da nur im oben genannten Spezialfall ein aktiver Thread ausgelagert wird, besitzt der Rhamma-Prozessor einen Timeout-Mechanismus, mit dem verhindert wird, daß ein Registersatz für immer von einem aktiven Thread belegt wird. Für jeden Registersatz ist ein Signal *timeout* vorhanden, welches unendlich oft gesetzt wird. Wenn der Registersatz in einem der Zustände *active0* oder *active0_waiting* ist und das Signal *timeout* gesetzt wird, lagert die Interrupt-Routine *o_load* den Thread in den Speicher aus.

Eine weitere unerwünschte Situation ist der Stillstand des Prozessors, wenn kein Registersatz einen aktiven Thread enthält, obwohl gleichzeitig im virtuellen Modell mindestens ein aktiver Thread vorhanden ist. Demnach sind zu diesem Zeitpunkt alle aktiven Threads in den Speicher ausgelagert. Der Rhamma-Prozessor enthält einen Mechanismus, um auf diese Situation zu reagieren. Sobald jeder Registersatz in einem der Zustände *rsfree0*, *free0* oder *stopped0* ist und ein Thread im OSM im Zustand *active* ist, wird für einen Registersatz das Signal *uload* gesetzt, wodurch die Interrupt-Routine *u_load* ausgeführt wird. Der Registersatz wird dabei zustandsabhängig in der Reihenfolge *rsfree0*, *free0*, *stopped0* ausgewählt.

Für eine detailliertere Darstellung des Mikroprozessormodells sei auf Abbildung 2.3 und auf den Artikel [GrSc98] verwiesen.

Kapitel 3

Die Modellprüfer SMV und SPIN

Für die im Rahmen dieser Arbeit durchgeführten Verifikationen des Threadmanagements des Rhamma-Prozessors wurden die beiden Modellprüfer SMV und SPIN verwendet. In den folgenden Abschnitten werden die beiden Modellprüfer mit ihren unterschiedlichen Ansätzen und Techniken vorgestellt.

3.1 Der Modellprüfer SMV

Der Modellprüfer SMV verwendet als Spezifikationssprache die Temporallogik CTL (*Computation Tree Logic*, siehe Abbildung 3.1). Es wird eine symbolische Darstellung des Zustandsraumes mit BDDs (*Binary Decision Diagram*) verwendet. Der Zustandsraum wird mit Breitensuche traversiert.

Ein SMV-Programm besteht aus einer Liste von Variablenzuweisungen. Zu jedem Zeitpunkt werden alle Zuweisungen parallel ausgeführt; es existiert demnach kein Programmzähler. Deshalb darf ein SMV-Programm keine zyklischen Abhängigkeiten der Variablen und keine Mehrfachzuweisungen an Variablen enthalten. Zur Gliederung eines SMV-Programms können Programmteile zu Modulen zusammengefaßt werden. In einem Programm können mehrere Instanzen eines Moduls verwendet werden.

In einer neuen Implementierung unterstützt der SMV die spezielle Modellprüfungstechnik der Symmetrie-Reduktion. Die grundlegende Idee der Symmetrie-Reduktion ist die Ausnutzung von Isomorphien im Zustandsraum. Zustände, die sich nur durch die Permutation einer symmetrischen Variablen unterscheiden, sind äquivalent. Bei der Verifikation muß deshalb nur jeweils einer dieser Zustände stellvertretend für eine Klasse von Zuständen betrachtet werden. Hierdurch kann der Zustandsraum verkleinert werden.

Voraussetzung für eine Verifikation mit Symmetrie-Reduktion ist, daß zunächst erkannt wird, welche Variablen symmetrisch sind. Für diese Variablen muß in der Beschreibung des zu verifizierenden Systems der spezielle Datentyp `scalarset` verwendet werden. Anschließend wird die Symmetrie-Reduktion bei der Verifikation automatisch angewendet.

3.2 Der Modellprüfer SPIN

Der Modellprüfer SPIN untersucht den Zustandsraum mit Tiefensuche. Das zu verifizierende System wird in der Sprache PROMELA beschrieben. Ein PROMELA-

Syntax von CTL

p	boolesche Variablen
$\varphi \wedge \psi$, $\neg\varphi$, ...	boolesche Operatoren
X, U, F, G	zusätzliche Operatoren
A, E	Pfadquantoren

Die Pfadquantoren A und E und die Operatoren X, U, F und G dürfen nur paarweise verwendet werden. Auf einen Pfadquantor muß unmittelbar einer der Operatoren X, U, F oder G folgen. Daraus ergeben sich die acht temporalen Operatoren: AX, AU, AF, AG, EX, EU, EF und EG.

Semantik von CTL

AX φ	Auf allen Pfaden muß im nächsten Zustand φ gelten.
A (φ U ψ)	Auf allen Pfaden muß irgendwann einmal ψ gelten. Bis dahin muß φ gelten.
AF φ	Auf allen Pfaden muß irgendwann einmal φ gelten.
AG φ	Auf allen Pfaden muß immer φ gelten.
EX φ	Auf mindestens einem Pfad muß im nächsten Zustand φ gelten.
E (φ U ψ)	Auf mindestens einem Pfad muß irgendwann einmal ψ gelten. Bis dahin muß φ gelten.
EF φ	Auf mindestens einem Pfad muß irgendwann einmal φ gelten.
EG φ	Auf mindestens einem Pfad muß immer φ gelten.

Abbildung 3.1: Syntax und Semantik von CTL

Programm besteht aus parallelen Prozessen, die zur Ausführung sequentialisiert werden, indem zu jedem Zeitpunkt die Menge aller ausführbaren Befehle aller Prozesse gebildet wird und anschließend einer dieser Befehle ausgewählt wird. Neben Variablen bietet die Sprache PROMELA synchrone und asynchrone Nachrichtenkanäle für die Kommunikation der Prozesse. Es gibt die folgenden Möglichkeiten, das Verifikationsziel zu spezifizieren:

accept-labels: In der Beschreibung des Systems können Zustände mit `accept`-labels markiert werden. In allen möglichen Systemabläufen darf dann kein Zyklus einen so gekennzeichneten Zustand enthalten.

progress-labels: Zustände können mit `progress`-labels versehen werden. In allen möglichen Abfolgen von Zuständen muß ein `progress`-Zustand unendlich oft angenommen werden.

LTL-Formeln: Die wichtigste Variante, das Verifikationsziel zu spezifizieren, sind die LTL-Formeln, deren Syntax und Semantik in 3.2 zusammengefaßt ist. Für die Verifikation werden die LTL-Formeln auf `accept`-labels abgebildet, indem ein zusätzlicher Prozeß einen Zyklus mit `accept`-Zustand durchläuft, wenn die Formel nicht erfüllt ist.

Zur Effizienzsteigerung bei der automatenbasierten Modellprüfung wird im SPIN die Halbordnungsreduktion verwendet [HoPe94]. Man kann feststellen, daß die Reihenfolge, in der die Befehle verschiedener paralleler Prozesse ausgeführt werden, oft keinen Einfluß auf die zu beweisende Aussage hat. Es existieren somit Äquivalenzklassen von Zuständen. Deshalb ist das Ziel bei der Halbordnungsreduktion, den Zustandsraum zu verkleinern, indem für jede Klasse von Zuständen nur jeweils ein

Syntax von LTL

p	boolesche Variablen
$\varphi \wedge \psi$, $\neg\varphi$, ...	boolesche Operatoren
X,U	temporale Operatoren
F,G (auch $\Diamond$,$\Box$)	abgeleitete temporale Operatoren

Semantik von LTL

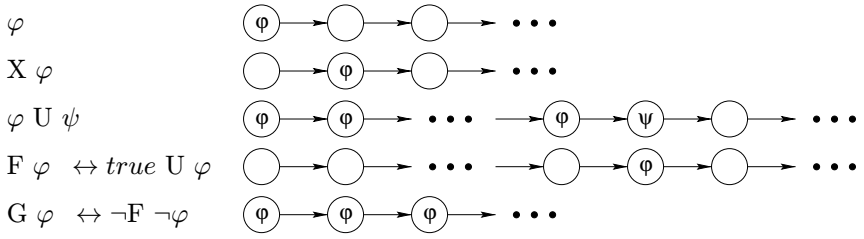

Abbildung 3.2: Syntax und Semantik von LTL (*Linear Time Logic*)

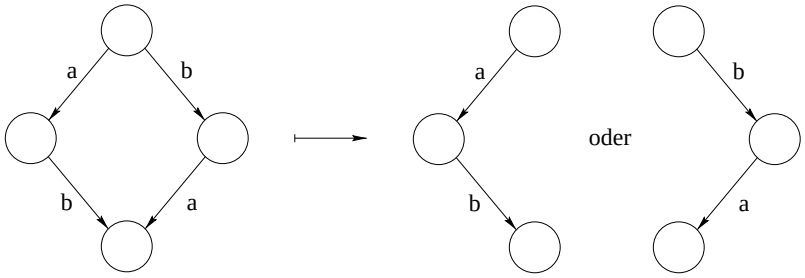

Abbildung 3.3: Halbordnungsreduktion

Stellvertreter betrachtet wird. Hierfür wird die Tiefensuche so modifiziert, daß in jedem Zustand nur eine hinreichend große Teilmenge der möglichen Folgezustände betrachtet wird. Es werden in jedem Zustand die möglichen Zustandsübergänge untersucht, um festzustellen, welche Folgezustände auch weiterhin betrachtet werden müssen.

Die Auswirkung der Halbordnungsreduktion ist, wie in Abbildung 3.3 dargestellt, das Streichen irrelevanter Zustandsabfolgen. Wenn in einem Zustand zwei verschiedene Prozesse zwei unabhängige Befehle a und b ausführen können und in den sich daraus ergebenden Folgezuständen der jeweils andere Befehl ausführbar ist, reicht es aus, stellvertretend eine feste Reihenfolge der Befehle zu betrachten. Der Modellprüfer SPIN führt die Halbordnungsreduktion automatisch durch. Hierfür wird vor der Verifikation untersucht, welche Zustandsabfolgen nicht betrachtet werden müssen. Diese Analyse läßt sich effizient durchführen.

Kapitel 4

Verifikation mit SPIN

4.1 Ein erster Entwurf des Prozessors

In einem ersten Entwurf des Rhamma-Prozessors wurden die einzelnen Bestandteile des Prozessors direkt mit PROMELA-Prozessen modelliert. Dies hat zu folgenden Prozessen geführt:

register-frame: Prozeß, der das Verhalten eines Registersatzes modelliert. Dieser Prozeß enthält sämtliche Zustände und Zustandsübergänge, die im MPM spezifiziert sind. In jedem Zustand wird auf die passenden Signale gewartet und anschließend mit den entsprechenden Signalen und Zustandsübergängen reagiert.

processor: Prozeß, der die Ausführungseinheit modelliert und dazu den aktuellen Befehl von einem Registersatz empfängt. Es werden, wenn nötig, ein Partner-Registersatz ausgewählt und die entsprechenden Signale gesendet.

program: Dieser Prozeß erzeugt einen neuen Befehl für den jeweils aktiven Thread, indem nichtdeterministisch einer der möglichen Befehle ausgewählt wird. Welche Befehle erlaubt sind, wird mit Hilfe der Prozesse *vtm_state* und *frame_counter* festgestellt.

frame_counter: Dieser Prozeß zählt die im System vorhandenen Threads. Die Zahl der vorhandenen Threads wird durch den Befehl *getfr* erhöht und durch den Befehl *relfr(j)* reduziert. Wenn ein festgelegtes Maximum erreicht ist, darf der Befehl *getfr* vorerst nicht mehr erzeugt werden.

vtm_state: Für jeden Thread existiert eine Instanz dieses Prozesses, der den Zustand des Threads im virtuellen Modell verwaltet. Sobald ein Befehl zugeordnet wird, werden die sich daraus ergebenden Zustandsänderungen mit diesen Prozessen vorgemerkt. Dadurch wird verhindert, daß Befehle, die nicht zum Zustand passen, erzeugt werden.

Die restlichen Prozesse wurden für den zweiten Entwurf nicht oder nur unwesentlich verändert und werden in Abschnitt 4.2.1 näher beschrieben. Insgesamt ergab sich für den ersten Entwurf eine Struktur nach Abbildung 4.1. Hierbei wurden die Signale durch synchrone Nachrichtenkanäle realisiert.

Eine Verifikation der Fairness des Threadmanagements war mit diesem Entwurf aufgrund der hohen Zustandszahl nur für zwei Registersätze und zwei Threads möglich. Dadurch wurde ein zweiter Entwurf notwendig.

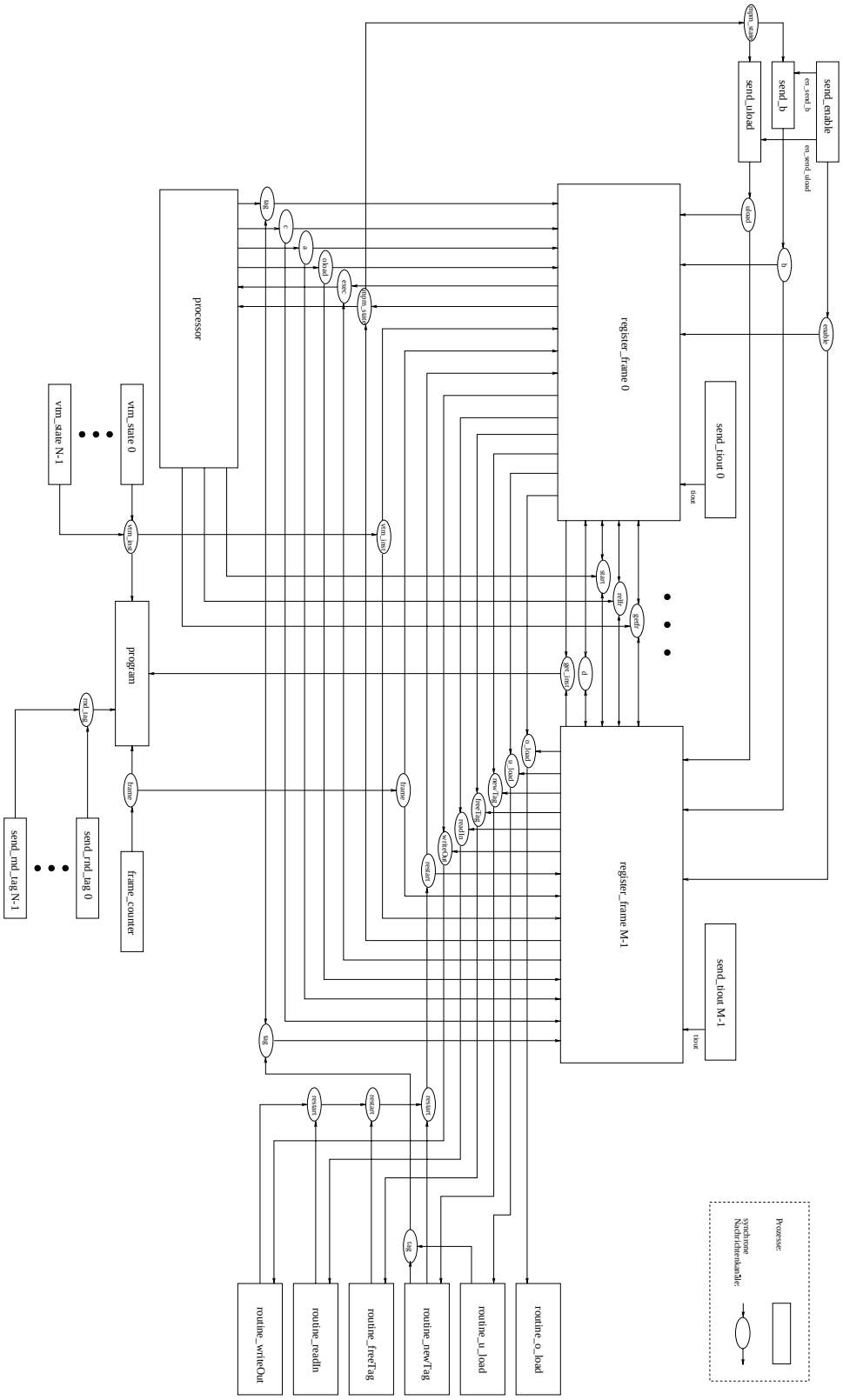

Abbildung 4.1: Struktur des ersten Entwurfs

4.2 Der zweite Entwurf

Dem zweiten Entwurf gingen folgende Beobachtungen und Überlegungen voraus.

Die Befehlserzeugung durch den Prozeß *program* geschieht mit Hilfe der Prozesse *frame_counter* und *vtm_state*. Diese Hilfsprozesse implementieren – durch ihre synchrone Kommunikation – eine *test and set* Operation für den Zustand im virtuellen Modell. Diese Konstruktion wurde gewählt, da im MPM lediglich verlangt wird, daß zu jedem Zeitpunkt *mindestens* eines der *enable*-Signale gesetzt ist. Andererseits ist aber das *enable*-Signal derart realisiert, daß zu jedem Zeitpunkt nur eines der *enable*-Signale gesetzt und somit nur ein Registersatz aktiviert ist. Dadurch ist es möglich, auf die Prozesse *frame_counter* und *vtm_state* zu verzichten und den VTM-Zustand mit einfachen Variablen zu verwalten.

Im ersten Entwurf wurde für jeden Registersatz eine Instanz des Prozesses *register_frame* verwendet. Hierbei wird der Zustand jedes Registersatzes doppelt gespeichert, einmal in Form des Programmzählers des PROMELA-Prozesses und zusätzlich in einem Array, damit der Zustand der Registersätze für die anderen Prozesse abfragbar ist. Die Tatsache, daß zu jedem Zeitpunkt nur einer der Registersätze durch das *enable*-Signal aktiviert ist, führte zu dem Ziel, im zweiten Entwurf nur noch eine Instanz eines Prozesses, der den jeweils aktivierten Registersatz simuliert, zu verwenden. Ein Problem hierbei waren jedoch die Signale, die zwischen zwei Registersätzen gesendet werden. Aufgrund der Realisierung der Signale mittels synchroner Kommunikationsoperationen müssen Sender und Empfänger zwei verschiedene Prozesse sein. Dieses Problem konnte durch folgende Beobachtung gelöst werden.

Die Zustände eines Registersatzes lassen sich in zwei Klassen von Zuständen unterteilen. Einerseits die *aktiven Zustände*, in denen dem Registersatz ein aktiver Thread zugeordnet ist und andererseits die *passiven Zustände*. Die aktiven Zustände sind die Zustände *active0*, *active1*, *active0_restart*, *active0_waiting* und *active1_oload*. Die übrigen Zustände gehören zu den passiven Zuständen. Unter Verwendung dieser Unterteilung läßt sich feststellen, daß bei jeder möglichen Sendeoperation eines Signals zwischen zwei Registersätzen immer der eine Registersatz in einem aktiven Zustand und der andere in einem passiven Zustand ist. Diese Feststellung führte dazu, den Prozeß für einen Registersatz in zwei Prozesse aufzuteilen. Einer der Prozesse beschreibt das Verhalten eines Registersatzes, der sich in einem aktiven Zustand befindet. Entsprechend modelliert der andere Prozeß einen Registersatz in den passiven Zuständen. Unabhängig von der Anzahl der Registersätze wird von beiden Prozessen jeweils nur eine Instanz benötigt.

Aus diesen Überlegungen ergab sich für den zweiten Entwurf eine Struktur nach Abbildung 4.2. Die einzelnen Prozesse werden in den nächsten Abschnitten näher beschrieben.

4.2.1 Beschreibung der einzelnen Prozesse

register_frame_a

Der Prozeß *register_frame_a* modelliert das Verhalten eines aktiven Registersatzes. Im Grundzustand wird auf Signale, die über die Kanäle *enable_a* und *in_a* empfangen werden, gewartet. Eine über diese Kanäle empfangene Nachricht beinhaltet neben dem Signal immer auch die Nummer des Registersatzes, für den das Signal bestimmt ist. Anhand der Nummer des Registersatzes wird – mit Hilfe eines globalen Arrays – festgestellt, in welchem Zustand sich der betreffende Registersatz befindet. Anschließend wird auf das empfangene Signal reagiert. Diese Reaktionen können, je nach Zustand des Registersatzes, aus Senden und Empfangen weiterer Signale sowie einem Zustandsübergang des Registersatzes bestehen. Im Zustand *active0* etwa umfaßt die Reaktion auf das Signal *enable_a* die Ausführung eines Befehls. Hierfür

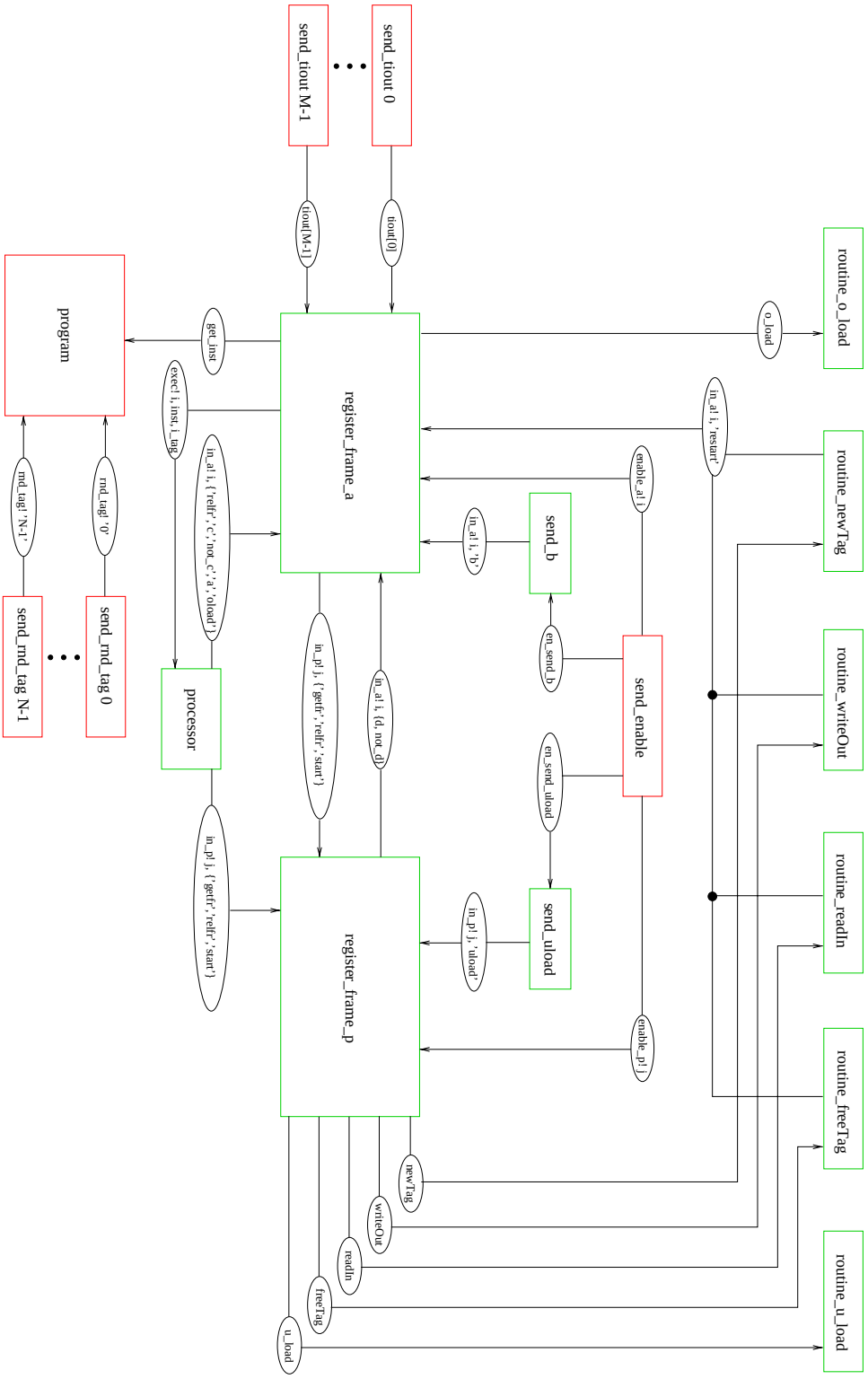

Abbildung 4.2: Struktur des zweiten Entwurfs

wird ein Signal zur Befehlserzeugung an den Prozeß *program* und anschließend ein Signal zur Befehlsausführung an den Prozeß *processor* gesendet. Danach wird auf ein Signal vom *processor* gewartet und der entsprechende Zustandsübergang durchgeführt. Im Anschluß an die Reaktionen kehrt der Prozeß in den Grundzustand zurück und wartet erneut auf Signale.

register_frame_p

Die Grundstruktur des Prozesses *register_frame_p* gleicht der des Prozesses *register_frame_a*. Die Signale für einen passiven Registersatz werden, zusammen mit der Nummer des betreffenden Registersatzes, über die Kanäle *enable_p* und *in_p* empfangen.

processor

Die Ausführungseinheit des Rhamma-Prozessors wird durch den Prozeß *processor* modelliert. Dieser Prozeß wartet auf eine Nachricht über den Kanal *exec*. Die Nachrichten bestehen aus der Nummer des den Befehl ausführenden Registersatzes und dem Befehl. Anschließend wird überprüft, ob ein Interrupt nötig und möglich ist und gegebenenfalls der beteiligte passive Registersatz festgelegt. Daraufhin werden die entsprechenden Signale an den Prozeß *register_frame_a* und eventuell an den Prozeß *register_frame_p* gesendet. Nachdem auf diese Weise eine Interrupt-Sequenz gestartet wurde, kehrt der Prozeß in den Grundzustand zurück und wartet auf den nächsten Befehl. Die verbleibenden Signale und Zustandsübergänge einer solchen Interrupt-Sequenz geschehen ohne Beteiligung des Prozesses *processor* durch die Prozesse *register_frame_a* und *register_frame_p* als Reaktion auf das *enable*-Signal.

routine_newTag, routine_freeTag, routine_writeOut, routine_readIn, routine_u_load und routine_o_load

Diese Prozesse modellieren die Interrupt-Routinen. Hierfür wird auf ein Signal über den entsprechenden Nachrichtenkanal gewartet. Die Nachrichten bestehen aus den von der Interrupt-Routine benötigten Informationen. Dies sind je nach Interrupt-Routine ein bis drei der folgenden Werte: die Nummer t des Threads und die Nummern i und j des aktiven und des passiven Registersatzes. Mit Hilfe dieser Daten führen die Prozesse die passenden Änderungen an den globalen Variablen durch. Diese Variablen sind die Arrays für die OSM-Zustände sowie für die Zuordnung von Threads zu Registersätzen. Gegebenenfalls wird vor der Rückkehr in den Grundzustand das Signal *restart* über den Kanal *in_a* an den aktiven Registersatz gesendet.

send_tiout

Für jeden Registersatz i existiert eine Instanz dieses Prozesses, der den Timeout-Mechanismus realisiert. Hierfür ist das Signal $timeout_i$ für eine feste Anzahl von Sendeoperationen – über den jeweiligen Kanal *tiout[i]* – nicht gesetzt, bevor es einmal gesetzt wird.

send_uload

Dieser Prozeß realisiert den Mechanismus, der den Stillstand des Prozessors verhindert. Sobald der Prozeß durch ein Signal über den Kanal *en_send_uload* aktiviert wird, werden die MPM- und OSM-Zustände überprüft und gegebenenfalls über den Kanal *in_p* das Signal *uload* gesendet. Der Registersatz, für den das Signal *uload* gesendet wird, wird hierbei zustandsabhängig in der Reihenfolge *rsfree0*, *free0* und *stopped0* ausgewählt.

send_b

Der Prozeß *send_b* stellt sicher, daß ein Registersatz, der in den Wartezustand *active0_waiting* eintritt, den Zustand wieder verläßt, sobald sein Befehl wieder ausführbar ist. Wenn der Prozeß durch ein Signal über den Kanal *en_send_b* aktiviert wird, werden die Zustände der Registersätze überprüft. Solange ein Registersatz im Wartezustand ist, wird nach einem Registersatz gesucht, der als Interrupt-Partner dienen könnte. Sobald ein entsprechender Registersatz gefunden wird, wird über den Kanal *in_a* das Signal *b* gesendet. Zur Suche nach einem passenden Registersatz werden, bei jeder Aktivierung durch *en_send_b*, alle Registersätze einmal überprüft.

send_enable

Der Prozeß *send_enable* schickt das *enable*-Signal der Reihe nach an alle Registersätze. Hierfür muß anhand des Zustands des Registersatzes unterschieden werden, ob das Signal über den Kanal *enable_a* an den Prozeß *register_frame_a*, oder über *enable_p* an *register_frame_p* gesendet werden muß. Sobald für jeden Registersatz ein *enable*-Signal gesendet wurde, werden nacheinander über die Kanäle *en_send_uload* und *en_send_b* die Prozesse *send_uload* und *send_b* aktiviert.

send_rnd_tag

Es existiert pro Thread eine Instanz dieses Prozesses, der dazu dient, eine nichtdeterministische Auswahl unter den Threads zu ermöglichen. Hierfür sendet jede Instanz für immer die Nummer des ihr zugeordneten Threads über den gemeinsamen Kanal *rnd_tag*. Zu der korrespondierenden Empfangsoperation im Prozeß *program* wird somit nichtdeterministisch die Sendeoperation eines *send_rnd_tag*-Prozesses ausgewählt.

program

Der Prozeß erzeugt auf Anforderung eines Registersatzes einen neuen Befehl, indem nichtdeterministisch einer der legalen Befehle ausgewählt wird. Im Grundzustand wartet der Prozeß auf einen Auftrag zur Befehlsgenerierung, der in Form der Thread-Nummer *i* über den Kanal *get_inst* empfangen wird. Anschließend erhält der Prozeß eine zweite Thread-Nummer *j* über den Kanal *rnd_tag*, die eventuell für die Befehle *start(j)* oder *relfr(j)* verwendet wird. Daraufhin findet die eigentliche Auswahl des Befehls statt, indem nichtdeterministisch einer der erlaubten Befehle ausgewählt wird. Welche Befehle erlaubt sind, wird anhand des VTM-Zustands der Threads sowie der Anzahl bereits vorhandener Threads festgestellt. Die Auswahl der Befehle *noop*, *stop* und *relfr(i)* ist immer möglich, während der Befehl *getfr* nur zulässig ist, wenn die vorgegebene maximale Anzahl an Threads noch nicht erreicht ist. Die Befehle *start(j)* und *relfr(j)* können nur ausgewählt werden, wenn der VTM-Zustand des Threads *j* der Zustand *stopped* ist. Die VTM-Zustände der Threads werden gemäß des gewählten Befehls aktualisiert, und anschließend wird auf einen neuen Auftrag gewartet.

4.3 Verifikation der Fairness des Threadmanagements

Im Gegensatz zur Spezifikation – dem VTM – kann in der Implementierung – den Modellen MPM und OSM – die Ausführung eines Threads unterbrochen werden. Deshalb ist es notwendig, zu beweisen, daß die Ausführung der Threads nicht für immer unterbrochen wird. Dieser Nachweis der Fairness des Threadmanagements

		mit Halbordnungsreduktion			ohne Halbordnungsreduktion		
m	n	Suchtiefe	MByte	Sekunden	Suchtiefe	MByte	Sekunden
2	1	670	2.302	0.0	9212	6.416	94.0
2	2	4420	2.820	3.0	363145	16.801	121048.0
2	3	51115	8.568	41.0	–	–	–
2	4	330781	52.025	369.0	–	–	–
2	5	1715150	341.572	3056.0	–	–	–
3	1	840	2.302	0.0	11542	17.438	433.0
3	2	5450	2.946	4.0	448897	21.790	548373.0
3	3	30096	12.650	85.0	–	–	–
3	4	1501024	308.174	2624.0	–	–	–

Tabelle 4.1: Verifikation der Fairness gemäß (4.1) für m Registersätze und maximal n Threads mit dem Modellprüfer SPIN. Die Messungen wurden auf einer Sun Ultra 1 mit 512 MByte Arbeitsspeicher durchgeführt.

m	n	BDD	MByte	Sekunden
2	1	12286	18.55	4.38
2	2	51624	19.13	26.80
2	3	125255	20.32	154.00
2	4	315499	23.39	774.09
2	5	845920	31.85	3329.78
3	1	39020	18.94	17.34
3	2	205809	21.62	112.01

Tabelle 4.2: Verifikation der Fairness gemäß (4.1) für m Registersätze und maximal n Threads mit dem Modellprüfer SMV nach [GrSc98]. Die Messungen wurden auf einer Sun Ultra 1 mit 196 MByte Arbeitsspeicher durchgeführt.

wurde mit zwei verschiedenen Ansätzen geführt, die in den folgenden Abschnitten vorgestellt werden.

4.3.1 Ein erster Ansatz für ein faires Threadmanagement

Nach dem ersten Ansatz ist die Fairness des Threadmanagements gewährleistet, wenn kein aktiver Thread für immer in den Speicher verdrängt wird. Diese Auffassung der Fairness des Threadmanagements führte zu der folgenden Spezifikation.

$$Fairness := \bigwedge_{i=1}^{m} \Box(osm\_active_i \rightarrow \Diamond osm\_on\_processor_i) \tag{4.1}$$

Nach (4.1) ist das Threadmanagement fair, wenn für jeden Thread i gilt, daß immer wenn osm_active_i erfüllt ist, auch irgendwann wieder $osm\_on\_processor_i$ erfüllt sein wird. Hierbei ist die boolesche Variable osm_active_i genau dann *wahr*, wenn sich der Thread i im OSM-Zustand *active* befindet. Der Thread ist demnach aktiv und in den Speicher ausgelagert. Entsprechend ist die boolesche Variable $osm\_on\_processor_i$ genau dann *wahr*, wenn sich der Thread i im OSM-Zustand *on processor* befindet. Dem Thread ist demnach ein Registersatz zugeordnet.

Die Ergebnisse der Verifikation von (4.1) mit dem Modellprüfer SPIN sind in Tabelle 4.1 dargestellt. Demnach ist eine Verifikation nur mit Hilfe der Halbordnungsreduktion möglich. Tabelle 4.2 enthält die entsprechenden Ergebnisse der Verifikation von (4.1) mit dem SMV nach [GrSc98]. Diese Werte wurden mit einer

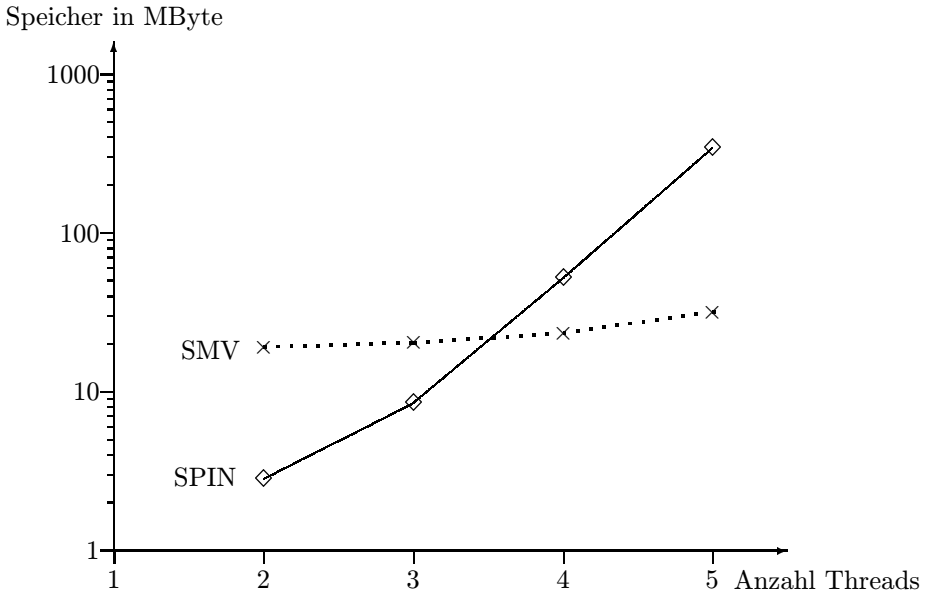

Abbildung 4.3: Speicherbedarf bei der Verifikation von (4.1) für zwei Registersätze

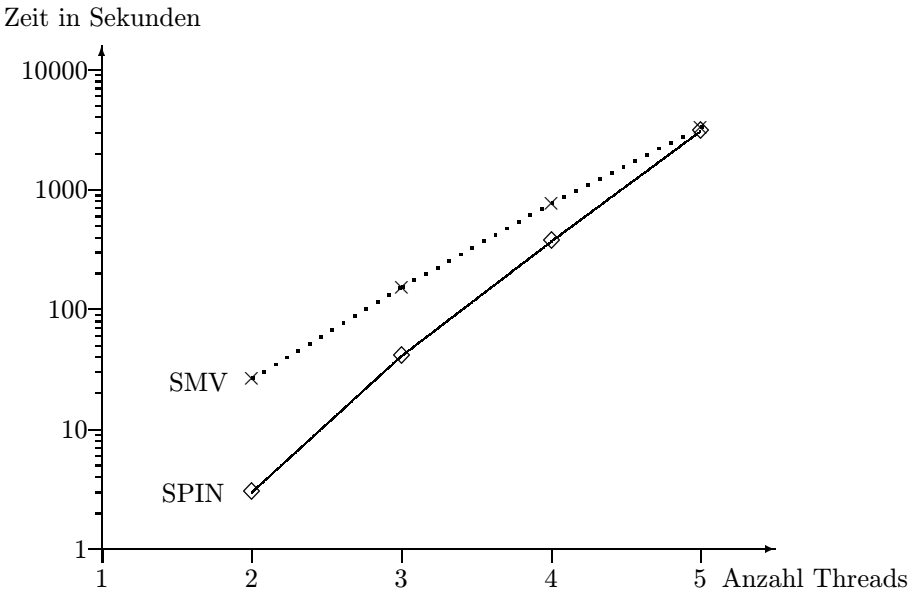

Abbildung 4.4: Ausführungszeit bei der Verifikation von (4.1) für zwei Registersätze

		mit Halbordnungsreduktion			ohne Halbordnungsreduktion		
m	n	Suchtiefe	MByte	Sekunden	Suchtiefe	MByte	Sekunden
2	2	4521	2.934	6.0	363260	17.518	217036.0
2	3	51115	10.585	75.0	–	–	–
2	4	330395	64.313	606.0	–	–	–
3	2	5583	3.163	8.0	–	–	–
3	3	30096	16.768	167.0	–	–	–
3	4	1501104	413.748	4897.0	–	–	–

Tabelle 4.3: Verifikation der Fairness gemäß (4.2) für m Registersätze und maximal n Threads mit dem Modellprüfer SPIN. Die Messungen wurden auf einer Sun Ultra 1 mit 512 MByte Arbeitsspeicher durchgeführt.

optimierten Variablenordnung erzielt. Diese vorbereitende Arbeit ist bei der Verifikation mit SPIN nicht notwendig. Die Meßwerte gemäß der Tabellen 4.1 und 4.2 sind für zwei Registersätze in den Abbildungen 4.3 und 4.4 graphisch dargestellt. Bei der Verifikation mit SPIN ist, im Vergleich zur Verifikation mit dem SMV, der Speicherbedarf zunächst niedriger, steigt aber wesentlich stärker an. Die Ausführungszeit mit SPIN liegt bei allen Meßwerten unter der des SMV, allerdings läßt sich für SPIN ein leicht stärkeres Wachstum feststellen.

Die Definition der Fairness gemäß (4.1) gewährleistet lediglich, daß ein aktiver Thread, der in den Speicher verdrängt wird, irgendwann wieder auf den Prozessor geladen wird. Es wird dabei nicht garantiert, daß der Thread auch ausgeführt wird. Deshalb wurde die Verifikation zusätzlich mit einem zweiten Ansatz durchgeführt.

4.3.2 Ein strikterer Ansatz für ein faires Threadmanagement

Ein Threadmanagement ist fair, wenn es gewährleistet, daß jeder aktive Thread irgendwann einen Befehl ausführt. Diese Forderung läßt sich wie folgt formalisieren:

$$Fairness := \bigwedge_{i=1}^{m} \Box(vtm\_active_i \rightarrow \Diamond(Befehl_i \wedge \Diamond\neg Befehl_i)) \tag{4.2}$$

Die boolesche Variable vtm_active_i ist genau dann *wahr*, wenn sich der Thread i nach dem virtuellen Modell im Zustand *active* befindet. Genau dann, wenn dem Thread i ein noch nicht ausgeführter Befehl zugeordnet ist, hat die boolesche Variable $Befehl_i$ den Wert *wahr*. Somit ist nach (4.2) die Fairness erfüllt, wenn für jeden Thread gilt, daß er, immer wenn er aktiv ist, irgendwann einen Befehl erhält und irgendwann diesen Befehl ausführt.

Die Resultate der Verifikation der Fairness gemäß (4.2) sind für den Modellprüfer SPIN in Tabelle 4.3 eingetragen. Im Vergleich zu den mit (4.1) erzielten Werten ist hier der Speicherverbrauch höher. Zur Ausführung wurde die doppelte Zeit benötigt.

Kapitel 5

Verifikation mit dem SMV

Im zweiten Teil der Arbeit wurde die Verifikation mit der neuen Implementierung des Modellprüfers SMV unter Ausnutzung der Symmetrien wiederholt. Hierbei sind die symmetrischen Variablen des Rhamma-Prozessors die Nummern der Registersätze und die Nummern der Threads.

Für die SMV-Beschreibung des Rhamma-Prozessors wurden die folgenden vier Module verwendet:

frames: Dieses Modul beschreibt das Verhalten der Registersätze.

mpm: Das Modul *mpm* realisiert, mit Hilfe einer Instanz des Moduls *frames*, das Mikroprozessormodell.

osm: Zur Beschreibung des Betriebssystemmodells wurde das Modul *osm* verwendet.

main: Mit diesem Modul werden die Teile der Beschreibung zusammengefügt. Es wird hierfür jeweils eine Instanz der Module *mpm* und *osm* verwendet. Die Spezifikation wird ebenfalls in diesem Modul beschrieben.

5.1 Verifikation der Fairness

Für die Verifikation der Fairness des Threadmanagements wurde die Definition (4.1) verwendet.

Die Ergebnisse der Verifikationen mit und ohne Symmetrie sind in Tabelle 5.1 dargestellt. Hierbei läßt sich eine deutliche Verbesserung durch die Symmetrie-Reduktion feststellen.

In den Abbildungen 5.1 und 5.2 sind die Meßwerte der Verifikation mit Symmetrie-Reduktion zusammen mit den Werten von SPIN und denen des alten SMV ohne Symmetrie-Reduktion graphisch dargestellt. Die mit dem neuen SMV mit Symmetrie-Reduktion erzielten Werte sind schlechter als die mit SPIN und dem alten SMV ohne Symmetrie-Reduktion erzielten Meßwerte.

Eine Ursache hierfür ist aber auch, daß für die Verifikation mit dem alten SMV die Variablenordnung optimiert wurde. Dagegen war es, aufgrund eines Fehlers in der Implementierung des neuen SMV, nicht möglich, für die Verifikation mit dem neuen SMV ebenfalls eine optimierte Variablenordnung zu verwenden.

		mit Symmetrie-Reduktion			ohne Symmetrie-Reduktion		
m	n	BDD	MByte	Sekunden	BDD	MByte	Sekunden
1	1	9244	3.63	0.68	6019	3.50	0.55
1	2	13614	3.76	2.06	37858	4.43	5.92
1	3	87622	5.09	5.59	490862	11.91	76.76
1	4	157111	6.54	28.21	2653574	49.43	4284.6
1	5	955378	22.54	233.73	–	–	–
1	6	2979510	62.40	754.81	–	–	–
1	7	9361035	196.34	4466.35	–	–	–
2	1	28135	4.29	6.24	18448	4.01	4.6
2	2	64349	5.00	12.56	271639	8.55	55.71
2	3	1300317	25.82	128.57	21213948	363.36	33412.4
2	4	12567805	227.11	1528.2	–	–	–
3	1	57603	5.32	32.34	57937	5.24	30.7
4	1	99049	7.86	86.86	–	–	–
5	1	212024	10.43	159.11	–	–	–
6	1	277885	14.73	254.56	–	–	–

Tabelle 5.1: Verifikation der Fairness gemäß (4.1) für m Registersätze und maximal n Threads mit der neuen Implementierung des Modellprüfers SMV. Die Messungen wurden auf einer Sun Ultra 1 mit 512 MByte Arbeitsspeicher durchgeführt.

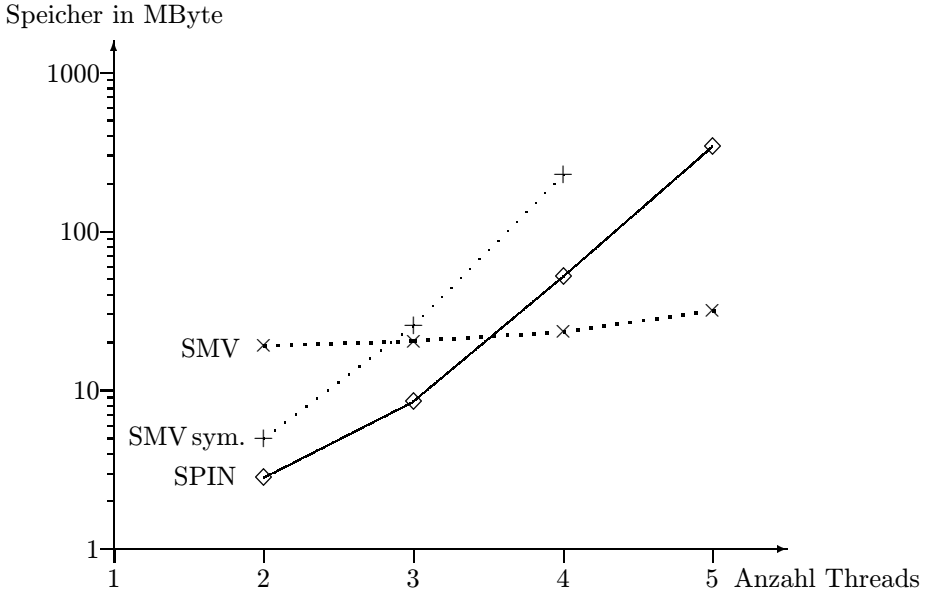

Abbildung 5.1: Speicherbedarf bei der Verifikation von (4.1) für zwei Registersätze

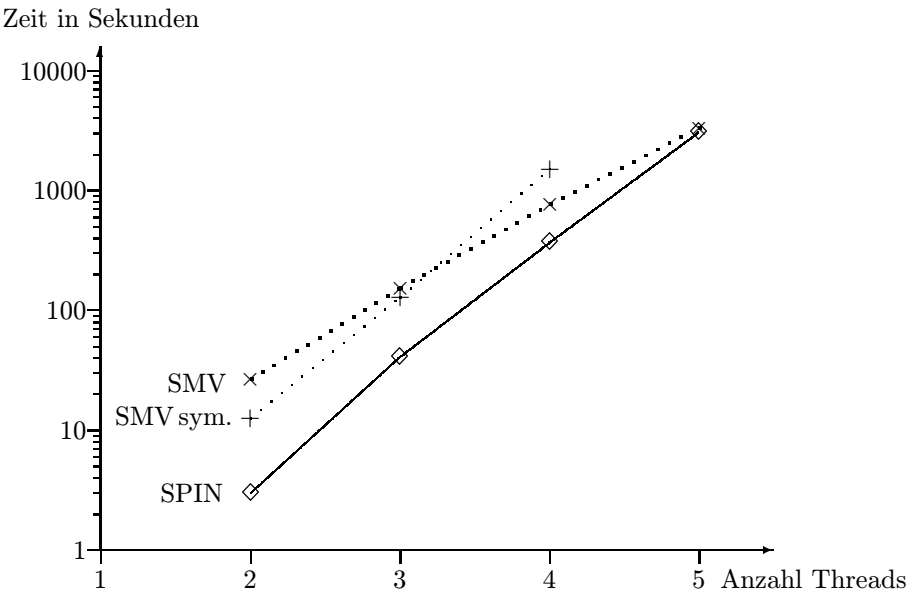

Abbildung 5.2: Ausführungszeit bei der Verifikation von (4.1) für zwei Registersätze

5.2 Verifikation der Korrektheit der Implementierung

Zusätzlich zur Fairness des Threadmanagements wurde mit dem neuen SMV mit Symmetrie-Reduktion verifiziert, daß die Modelle MPM und OSM das VTM implementieren.

Hierfür wurde zunächst das VTM über das MPM und über das OSM definiert und anschließend gezeigt, daß das so definierte Modell wirklich das VTM ist:

Alle Zustände und Signale des VTM wurden durch bestimmte Teilmengen der Zustände des MPM und OSM definiert. So wurde zum Beispiel der VTM-Zustand eines Threads als Zustand *free* definiert, wenn der OSM-Zustand dieses Threads der Zustand *free* ist oder der Thread einem Registersatz zugeordnet ist, der sich in einem der Zustände *free0*, *free1_freeTag*, *free1_uload* oder *free1_newTag* befindet. Die restlichen Zustände wurden entsprechend definiert. Aufgrund der möglichen Interrupt-Sequenzen werden die Signale $getfr_i$, $relfr_i$ und $start_i$ teilweise mehrfach gesendet, während das Signal im VTM nur einmal gesendet wird. Für die Definition der VTM-Signale wurde in allen Fällen die letzte Wiederholung des MPM-Signals vor der vollständigen Abarbeitung des Befehls verwendet.

Es mußte bewiesen werden, daß das so definierte Modell das VTM ist. Hierfür waren im einzelnen folgende Punkte zu zeigen: Die Definitionen der VTM-Zustände decken alle erreichbaren Zustände im MPM und OSM ab und sind eindeutig; jeder Thread nimmt zu jedem Zeitpunkt genau einen VTM-Zustand an. Zu jedem Zeitpunkt wird mit diesen Definitionen mindestens ein VTM-Signal gesendet und diese Signale werden nur erzeugt, wenn der entsprechende Thread aufgrund seines Zustands darauf reagieren kann. Es finden genau und ausschließlich die Zustandsübergänge des VTM statt. [GrSc98]

Die Werte, die bei der Verifikation der Korrektheit der Implementierung mit dem neuen SMV mit und ohne Symmetrie-Reduktion gemessen wurden, sind in Ta-

		mit Symmetrie-Reduktion			ohne Symmetrie-Reduktion		
m	n	BDD	MByte	Sekunden	BDD	MByte	Sekunden
1	1	10049	3.71	2.03	35744	4.11	3.98
1	2	26386	4.02	4.03	136011	6.08	18.64
1	3	192213	6.83	10.47	1685342	32.40	171.42
1	4	262717	8.29	27.05	9891195	171.50	4193.37
1	5	1320995	26.41	286.75	–	–	–
1	6	6866953	127.79	1499.38	–	–	–
2	1	28163	4.12	5.98	100417	5.49	21.98
2	2	117839	5.85	18.53	1401908	27.47	162.58
2	3	1046220	21.64	107.74	–	–	–
2	4	6232357	115.36	1917.63	–	–	–
3	1	114392	6.07	35.14	374230	11.01	124.42
4	1	450387	13.24	95.05	–	–	–
5	1	611131	17.14	218.94	–	–	–
6	1	1454027	33.89	488.41	–	–	–

Tabelle 5.2: Verifikation der Korrektheit der Implementierung für m Registersätze und maximal n Threads mit der neuen Implementierung des Modellprüfers SMV. Die Messungen wurden auf einer Sun Ultra 1 mit 512 MByte Arbeitsspeicher durchgeführt.

m	n	BDD	MByte	Sekunden
1	1	10106	18.42	0.52
1	2	10176	18.42	0.92
1	3	10370	18.42	1.88
1	4	14959	18.48	4.96
1	5	34496	18.87	10.31
1	6	151961	20.77	24.81
2	1	12286	18.55	4.84
2	2	47101	19.07	15.29
2	3	110532	20.12	33.15
2	4	311280	23.33	91.52
2	5	789233	30.93	298.87
2	6	1801003	47.12	1120.28
3	1	39020	18.94	19.27
3	2	199716	21.50	58.91

Tabelle 5.3: Verifikation der Korrektheit der Implementierung für m Registersätze und maximal n Threads mit dem Modellprüfer SMV nach [GrSc98]. Die Messungen wurden auf einer Sun Ultra 1 mit 196 MByte Arbeitsspeicher durchgeführt.

belle 5.2 zusammengefaßt. Die Tabelle 5.3 enthält die entsprechenden Daten nach [GrSc98] für den alten SMV. Ebenso wie in Abschnitt 5.1 ist eine deutliche Verbesserung der Meßwerte durch die Symmetrie-Reduktion erkennbar, obwohl die besten Werte mit dem alten SMV ohne Symmetrie-Reduktion erreicht wurden. Ein Grund hierfür ist auch hier, daß – im Gegensatz zur Verifikation mit dem neuen SMV – bei der Verifikation mit dem alten SMV eine optimierte Variablenordnung verwendet werden konnte.

5.3 Fehler in der Implementierung des neuen SMV

Neben dem bereits erwähnten Fehler wurde bei der Durchführung der Verifikationen der Abschnitte 5.1 und 5.2 ein weiterer gravierender Fehler in der Implementierung des neuen SMV gefunden.

Mehrere Fehler in der SMV-Beschreibung des Rhamma-Prozessors wurden bei der Verifikation mit dem neuen SMV unter Verwendung der Symmetrie-Reduktion nicht gefunden. Erst bei der anschließenden Verifikation mit dem neuen SMV ohne Verwendung der Symmetrie-Reduktion wurden alle Fehler aufgedeckt und konnten daraufhin beseitigt werden.

Die im neuen SMV verwendete Implementierung der Symmetrie-Reduktion ist demnach nicht fehlerfrei, weshalb eine Verwendung des neuen SMV mit Symmetrie-Reduktion nicht ratsam ist.

Kapitel 6

Zusammenfassung

Im Rahmen dieser Arbeit wurde das Threadmanagement des Rhamma-Prozessors mit LTL- und mit CTL-Modellprüfung verifiziert. Die Resultate der unterschiedlichen Ansätze wurden verglichen.

Dabei hat sich gezeigt, daß die LTL-Modellprüfung mit Halbordnungsreduktion eine effiziente Verifikationsmethode ist. Ein Vorteil gegenüber der CTL-Modellprüfung ist dabei, daß keine Variablenordnung optimiert werden muß, um ein gutes Ergebnis zu erhalten.

Trotz guter Variablenordnung war der SMV mit CTL-Modellprüfung ohne weitere Abstraktionstechnik nicht besser. Im Gegenteil, wenn man ausschließlich die Ausführungszeit betrachtet, war der SMV hier immer schlechter.

Eine Möglichkeit für zukünftige Arbeit besteht in der Kombination von Symmetrie-Reduktion und Halbordnungsreduktion, um das starke Wachstum des Speicherverbrauchs und der Ausführungszeit besser in den Griff zu bekommen [EmJP97].

Literaturverzeichnis

[BCLM94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society Press.

[Brya86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CGHJ93a] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 5–20, Ottawa, Canada, April 1993. IFIP WG10.2, CHDL'93, IEEE COMPSOC, Elsevier Science Publishers B.V., Amsterdam, Netherland.

[ClFJ93] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis, editor, *Workshop on Computer Aided Verification (CAV)*, pages 450–462, June/July 1993.

[ClGL92] E. Clarke, O. Grumberg, and D. Long. Model Checking and Abstraction. In *ACM Symposium on Principles of Programming Languages (POPL)*, New York, January 1992. ACM.

[CMCH96] E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[EmJP97] E. Allen Emerson, Somesh Jha, and Doron Peled. Combining partial order and symmetry reductions. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1217 of *Lecture Notes in Computer Science*, pages 19–34. Springer Verlag, 1997.

[GrSc98] W. Grünewald and K. Schneider. Modeling and verifying abstract multithreaded systems. In *Gemeinsamer Workshop der*

GI/ITG/GME Fachgruppen Methoden des Entwurfs und der Verifikation digitaler Schaltungen und Systeme und Beschreibungssprachen und Modellierung von Schaltungen und Systemen. HNI-Verlagsschriften, ISBN 3-931466-35-3, 1998.

[GrUn96] W. Grünewald and T. Ungerer. Towards extremely fast context switching in a blockmultithreaded processor. In *Proceedings of the 22nd Euromicro Conference*, pages 592–599, September 1996.

[Holz92] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1992.

[HoPe94] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *7th International Conference on Formal Description Techniques (FORTE '94)*, pages 177–194, 1994.

[HoPe96] G. J. Holzmann and D. Peled. The state of SPIN. In Rajeev Alur and Thomas A. Henzinger, editors, *Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 385–389, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[Krop97] T. Kropf. *Hardware-Verifikation – Verfahren und Werkzeuge zum Entwurf korrekter Schaltungen und Systeme.* Skriptum zur Vorlesung im WS 1997/98. Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1997.

[MaPn92] Z. Manna and A. Pnueli. *The temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, Berlin, Heidelberg, 1992.

[McMi92b] K.L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[McMi93a] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[Pele93] Doron Peled. All from one, one for all: On model checking using representatives. In *Conference on Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer Verlag, 1993.

[Pele96] Doron Peled. Partial order reduction: Model-checking using representatives. In *Mathematical Foundations of Computer Science*, volume 1113 of *Lecture Notes in Computer Science*, pages 93–112. Springer Verlag, 1996.

[PePH96] Doran Peled, Vaughan Pratt, and Gerard Holzmann, editors. *Partial Order Methods for Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1996.

[Pnue77] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighth Annual Symposium on Foundations of Computer Science*, volume 18, pages 46–57, New York, 1977. IEEE.

Anhang A

PROMELA-Beschreibung des Rhamma-Prozessors

Dieser Anhang enthält die PROMELA-Beschreibung des Rhamma-Prozessors, die für die Verifikation mit dem Modellprüfer SPIN verwendet wurde. Da die Halbordnungsreduktion mit einem Schalter abgeschaltet werden kann, sind bei SPIN keine unterschiedlichen Beschreibungen für die Verifikationen mit und ohne Halbordnungsreduktion notwendig.

In der folgenden Beschreibung sind vor der Verifikation für die Anzahl `M` der Registersätze und die Obergrenze `N` der Threads die gewünschten Werte einzusetzen.

```
#define N 2             /* # thread frames */
#define M 2             /* # register frames */
#define max_ti 1        /* tiout max_ti mal 0 dann 1 */


#define rsfree0               0   /* mpm_states */
#define rsfree1_readIn        1
#define rsfree1_uload         2
#define rsfree1_newTag        3
#define free1_newTag          4
#define free1_uload           5
#define free1_freeTag         6
#define free0                 7
#define free1_readIn          8
#define stopped1_readIn       9
#define stopped1_uload       10
#define stopped1_writeOut    11
#define stopped2_writeOut    12
#define stopped0             13
#define act1                 14
#define act0_restart         15
#define act1_oload           16
#define act0_waiting         17
#define act0                 18

#define free                 19   /* osm_state, vtm_state */
#define stopped              20   /* osm_state, vtm_state */
#define act                  21   /* osm_state, vtm_state */
#define on_processor         22   /* osm_state */
```

```
#define getfr                28   /* Signale der register_frames */
#define start                29
#define relfr                30
#define uload                31
#define oload                32
#define restart              33
#define a                    34
#define b                    35
#define c                    36
#define not_c                37
#define d                    38
#define not_d                39
#define no_sig               40

mtype = {inst_none, inst_getfr, inst_relfr, inst_start, inst_stop,
         inst_noop};

                               /* pro thread ein Eintrag:     */
byte   osm_state[N] = free;    /* OSM-Zustaende siehe #define */

                               /* pro register_frame ein Eintrag: */
byte   mpm_state[M] = rsfree0; /* MPM-Zustaende siehe #define     */

byte   vtm_state[N] = free;    /* VTM-Zustand */

byte   n = 0;                  /* aktuelle Anzahl von Threads */

                               /* Zuordnungstabellen:         */
byte   reg_thread_tags[M] = N; /* register-id  -> thread-id   */
byte   thread_reg_tags[N] = M; /* thread-id    -> register-id */

byte   partner[M] = M;
byte   thread_id[M] = N;

mtype  inst[N] = inst_none;     /* pro thread ein Befehl         */
byte   i_tag[N] = N;            /* pro thread ein Zielthread des */
                                /*  naechsten Befehls            */

chan   enable_a =[0] of {byte};  /* Kanaele der register_frames  */
chan   enable_p =[0] of {byte};
chan   tiout[M] =[0] of {bit};
chan   in_a     =[0] of {byte,byte};   /* Kanaele fuer Signale   */
chan   in_p     =[0] of {byte,byte};   /*  siehe #define    */

chan   newTag   =[0] of {byte,byte};   /* Auftraege IRQ-Routinen */
chan   freeTag  =[0] of {byte,byte};
chan   readIn   =[0] of {byte,byte,byte};
chan   writeOut =[0] of {byte,byte};
chan   o_load   =[0] of {byte};
chan   u_load   =[0] of {byte};

chan   en_send_b     =[0] of {bit};
chan   en_send_uload =[0] of {bit};
```

```
chan   exec =[0] of {byte,mtype,byte}; /* Auftrag Prozessor */

chan   get_inst =[0] of {byte};        /* Kanaele zur Befehls- */
chan   rnd_tag  =[0] of {byte};        /*  generierung         */


/*-------------------------------------------------------------*/
/*                  PROCTYPE REGISTER_FRAME_P                  */
/*-------------------------------------------------------------*/

proctype register_frame_p()
     {
     byte reg_i;
     byte signal=no_sig;

end: if
     ::enable_p?reg_i;
     ::in_p?reg_i,signal;
     fi;

     if
     ::(mpm_state[reg_i] == rsfree0) ->
         if
         ::(signal == getfr) ->
             signal = no_sig;
             mpm_state[reg_i] = rsfree1_newTag;
         ::(signal == uload) ->
             signal = no_sig;
             mpm_state[reg_i] = rsfree1_uload;
         ::(signal == start) ->
             signal = no_sig;
             mpm_state[reg_i] = free1_readIn;
         ::(signal == relfr) ->
             signal = no_sig;
             mpm_state[reg_i] = free1_readIn;
         ::else ->
             enable_p?eval(reg_i);
         fi;

     ::(mpm_state[reg_i] == rsfree1_readIn) ->
         if
         ::(signal == start) ->
             signal = no_sig;
             mpm_state[reg_i] = free1_readIn;
             in_a!partner[reg_i],d;
         ::(signal == relfr) ->
             signal = no_sig;
             mpm_state[reg_i] = free1_readIn;
             in_a!partner[reg_i],d;
         fi;

     ::(mpm_state[reg_i] == rsfree1_uload) ->
         u_load!reg_i;
```

```
        (reg_thread_tags[reg_i] != N);   /* IRQ Ende */
        mpm_state[reg_i] = act0;
        enable_p?eval(reg_i);

    ::(mpm_state[reg_i] == rsfree1_newTag) ->
        newTag!reg_i,partner[reg_i];
        (reg_thread_tags[reg_i] != N);   /* IRQ Ende */
        mpm_state[reg_i] = free1_newTag;
        enable_p?eval(reg_i);

    ::(mpm_state[reg_i] == free1_newTag) ->
        signal = no_sig;
        vtm_state[reg_thread_tags[reg_i]] = stopped;
        in_a!partner[reg_i],not_d;
        partner[reg_i] = M;
        mpm_state[reg_i] = stopped0;

    ::(mpm_state[reg_i] ==free1_uload) ->
        freeTag!reg_i,M;
        (reg_thread_tags[reg_i] == N);   /* IRQ Ende */
        mpm_state[reg_i] = rsfree1_uload;
        enable_p?eval(reg_i);

    ::(mpm_state[reg_i] == free1_freeTag) ->
        freeTag!reg_i,partner[reg_i];
        (reg_thread_tags[reg_i] == N);   /* IRQ Ende */
        mpm_state[reg_i] = rsfree1_readIn;
        enable_p?eval(reg_i);

    ::(mpm_state[reg_i] == free0) ->
        if
        ::(signal == start) ->
            signal = no_sig;
            mpm_state[reg_i] = free1_freeTag;
        ::(signal == relfr) ->
            signal = no_sig;
            mpm_state[reg_i] = free1_freeTag;
        ::(signal == uload) ->
            signal = no_sig;
            mpm_state[reg_i] = free1_uload;
        ::(signal == getfr) ->
            signal = no_sig;
            vtm_state[reg_thread_tags[reg_i]] = stopped;
            mpm_state[reg_i] = stopped0;
        ::else ->
            enable_p?eval(reg_i);
        fi;

    ::(mpm_state[reg_i] == free1_readIn) ->
        readIn!reg_i,partner[reg_i],thread_id[reg_i];
        (reg_thread_tags[reg_i] == thread_id[reg_i]); /*IRQ Ende*/
        thread_id[reg_i] = N;
        mpm_state[reg_i] = stopped1_readIn;
        enable_p?eval(reg_i);
```

```
::(mpm_state[reg_i] == stopped1_readIn) ->
    if
    ::(signal == relfr) ->
        signal = no_sig;
        in_a!partner[reg_i],not_d;
        partner[reg_i] = M;
        mpm_state[reg_i] = free0;
    ::(signal == start) ->
        signal = no_sig;
        in_a!partner[reg_i],not_d;
        partner[reg_i] = M;
        mpm_state[reg_i] = act0;
    fi;

::(mpm_state[reg_i] == stopped1_uload) ->
    writeOut!reg_i,M;
    (reg_thread_tags[reg_i] == N);   /* IRQ Ende */
    mpm_state[reg_i] = rsfree1_uload;
    enable_p?eval(reg_i);

::(mpm_state[reg_i] == stopped1_writeOut) ->
    writeOut!reg_i,M;
    (reg_thread_tags[reg_i] == N);   /* IRQ Ende */
    mpm_state[reg_i] = rsfree1_newTag;
    enable_p?eval(reg_i);

::(mpm_state[reg_i] == stopped2_writeOut) ->
    writeOut!reg_i,partner[reg_i];
    (reg_thread_tags[reg_i] == N);   /* IRQ Ende */
    mpm_state[reg_i] = rsfree1_readIn;
    enable_p?eval(reg_i);

::(mpm_state[reg_i] == stopped0) ->
    if
    ::(signal == uload) ->
        signal = no_sig;
        mpm_state[reg_i] = stopped1_uload;
    ::(signal == getfr) ->
        signal = no_sig;
        mpm_state[reg_i] = stopped1_writeOut;
    ::(signal == start) ->
        signal = no_sig;
        if
        ::(thread_id[reg_i] == reg_thread_tags[reg_i]) ->
            thread_id[reg_i] = N;
            mpm_state[reg_i] = act0;
        ::else ->
            mpm_state[reg_i] = stopped2_writeOut;
        fi;
    ::(signal == relfr) ->
        signal = no_sig;
        if
        ::(thread_id[reg_i] == reg_thread_tags[reg_i]) ->
```

```
                thread_id[reg_i] = N;
                mpm_state[reg_i] = free0;
            ::else ->
                mpm_state[reg_i] = stopped2_writeOut;
            fi;
        ::else ->
            enable_p?eval(reg_i);
        fi;
    fi;

    goto end;
    }


/*-------------------------------------------------------------*/
/*                  PROCTYPE REGISTER_FRAME_A                  */
/*-------------------------------------------------------------*/

proctype register_frame_a()
    {
    byte reg_i;
    byte state = rsfree0;

end: if
    ::enable_a?reg_i;
    ::in_a?reg_i,restart ->
        mpm_state[reg_i] = act0_restart;
        goto end;
    ::in_a?reg_i,b ->
        mpm_state[reg_i] = act0;
        goto end;
    fi;

    if
    ::(mpm_state[reg_i] == act1) ->
        enable_a?eval(reg_i);

    ::(mpm_state[reg_i] == act0_restart) ->
        state = mpm_state[partner[reg_i]];
        if
        ::(inst[reg_thread_tags[reg_i]] == inst_getfr) ->
            in_p!partner[reg_i],getfr;
        ::(inst[reg_thread_tags[reg_i]] == inst_relfr) ->
            in_p!partner[reg_i],relfr;
        ::(inst[reg_thread_tags[reg_i]] == inst_start) ->
            in_p!partner[reg_i],start;
        fi;
        if
        ::in_a?eval(reg_i),d ->
            (mpm_state[partner[reg_i]] != state);
            state = rsfree0;
            mpm_state[reg_i] = act1;
            enable_a?eval(reg_i);
        ::in_a?eval(reg_i),not_d ->
```

```
        (mpm_state[partner[reg_i]] != state);
        state = rsfree0;
        partner[reg_i] = M;
        if
        ::(inst[reg_thread_tags[reg_i]] == inst_relfr) ->
            n = n - 1;
        ::else ->
            skip;
        fi;
        inst[reg_thread_tags[reg_i]] = inst_none;
        i_tag[reg_thread_tags[reg_i]] = N;
        mpm_state[reg_i] = act0;
        enable_a?eval(reg_i);
    fi;

::(mpm_state[reg_i] == act1_oload) ->
    o_load!reg_i;
    (reg_thread_tags[reg_i] == N);   /* IRQ Ende */
    mpm_state[reg_i] = rsfree0;
    enable_a?eval(reg_i);

::(mpm_state[reg_i] == act0_waiting) ->
    if
    ::tiout[reg_i]?1 ->
        mpm_state[reg_i] = act1_oload;
    ::tiout[reg_i]?0 ->
        skip;
    fi;
    enable_a?eval(reg_i);

::(mpm_state[reg_i] == act0) ->
    if
    ::tiout[reg_i]?1 ->
        mpm_state[reg_i] = act1_oload;
        enable_a?eval(reg_i);
    ::tiout[reg_i]?0 ->
        if
        ::(inst[reg_thread_tags[reg_i]] == inst_none) ->
            get_inst!reg_thread_tags[reg_i];
            (inst[reg_thread_tags[reg_i]] != inst_none);
        ::else ->
            skip;
        fi;
        exec!reg_i,inst[reg_thread_tags[reg_i]],
                i_tag[reg_thread_tags[reg_i]];
        if
        ::in_a?eval(reg_i),relfr ->
            n = n - 1;
            inst[reg_thread_tags[reg_i]] = inst_none;
            i_tag[reg_thread_tags[reg_i]] = N;
            mpm_state[reg_i] = free0;
            enable_a?eval(reg_i);
        ::(inst[reg_thread_tags[reg_i]] == inst_stop) ->
            inst[reg_thread_tags[reg_i]] = inst_none;
```

```
                i_tag[reg_thread_tags[reg_i]] = N;
                mpm_state[reg_i] = stopped0;
                enable_a?eval(reg_i);
            ::in_a?eval(reg_i),c ->
                mpm_state[reg_i] = act1;
                enable_a?eval(reg_i);
            ::in_a?eval(reg_i),a ->
                mpm_state[reg_i] = act0_waiting;
                enable_a?eval(reg_i);
            ::in_a?eval(reg_i),oload ->
                mpm_state[reg_i] = act1_oload;
                enable_a?eval(reg_i);
            ::in_a?eval(reg_i),not_c ->      /* kein IRQ */
                if
                ::(inst[reg_thread_tags[reg_i]] == inst_relfr) ->
                    n = n - 1;
                ::else ->
                    skip;
                fi;
                inst[reg_thread_tags[reg_i]] = inst_none;
                i_tag[reg_thread_tags[reg_i]] = N;
                enable_a?eval(reg_i);
            fi;
        fi;
    fi;
    goto end;
    }


/*-----------------------------------------------------------------*/
/*                     PROCTYPE PROCESSOR                          */
/*-----------------------------------------------------------------*/

proctype processor()
    {
    byte    reg_i,reg_j,inst_tag = 0;
    byte    state = rsfree0;
    mtype   instruction = inst_none;

end: exec?reg_i,instruction,inst_tag;
    if
    ::(instruction == inst_getfr) ->
        instruction = inst_none;
        inst_tag = 0;
        do
        ::(reg_j == M) -> break;
        ::else ->
            if
            ::(mpm_state[reg_j] == free0) ->
                in_p!reg_j,getfr;
                (mpm_state[reg_j] != free0);
                reg_j=0;
                in_a!reg_i,not_c;
                reg_i=0;
```

```
            goto end;
        ::else ->
            reg_j = reg_j + 1;
        fi;
    od;
    reg_j = 0;
    do
    ::(reg_j == M) -> break;
    ::else ->
        if
        ::(mpm_state[reg_j] == rsfree0) ->
            goto irq_getfr;
        ::else ->
            reg_j = reg_j + 1;
        fi;
    od;
    reg_j = 0;
    do
    ::(reg_j == M) -> break;
    ::else ->
        if
        ::(mpm_state[reg_j] == stopped0) ->
            state = stopped0;
            goto irq_getfr;
        ::else ->
            reg_j = reg_j + 1;
        fi;
    od;
    reg_j=0;
    goto otherwise;

::(instruction == inst_relfr) ->
    instruction = inst_none;
    if
    ::(inst_tag == reg_thread_tags[reg_i]) ->
        inst_tag = 0;
        in_a!reg_i,relfr;
        reg_i=0;
        goto end;
    ::else -> skip;
    fi;
    reg_j = thread_reg_tags[inst_tag];
    if
    ::(reg_j < M) ->
        if
        ::(mpm_state[reg_j] == stopped0) ->
            thread_id[reg_j] = inst_tag;
            inst_tag = 0;
            in_p!reg_j,relfr;
            (mpm_state[reg_j] != stopped0);
            reg_j = 0;
            in_a!reg_i,not_c;
            reg_i = 0;
            goto end;
```

```
            ::else ->
                reg_j = 0;
                inst_tag = 0;
                goto otherwise;
            fi;
        ::else ->
            skip;
        fi;
        reg_j = 0;
        do
        ::(reg_j == M) -> break;
        ::else ->
            if
            ::(mpm_state[reg_j] == rsfree0) ->
                goto irq_relfr;
            ::else ->
                reg_j = reg_j + 1;
            fi;
        od;
        reg_j = 0;
        do
        ::(reg_j == M) -> break;
        ::else ->
            if
            ::(mpm_state[reg_j] == free0) ->
                state = free0;
                goto irq_relfr;
            ::else ->
                reg_j = reg_j + 1;
            fi;
        od;
        reg_j = 0;
        do
        ::(reg_j == M) -> break;
        ::else ->
            if
            ::(mpm_state[reg_j] == stopped0) ->
                state = stopped0;
                goto irq_relfr;
            ::else ->
                reg_j = reg_j + 1;
            fi;
        od;
        reg_j = 0;
        inst_tag = 0;
        goto otherwise;

    ::(instruction == inst_start) ->
        instruction = inst_none;
        reg_j = thread_reg_tags[inst_tag];
        if
        ::(reg_j < M) ->
            if
            ::(mpm_state[reg_j] == stopped0) ->
```

```
        thread_id[reg_j] = inst_tag;
        inst_tag = 0;
        in_p!reg_j,start;
        (mpm_state[reg_j] != stopped0);
        reg_j = 0;
        in_a!reg_i,not_c;
        reg_i = 0;
        goto end;
    ::else ->
        inst_tag = 0;
        reg_j = 0;
        goto otherwise;
    fi;
::else ->
    skip;
fi;
reg_j = 0;
do
::(reg_j == M) -> break;
::else ->
    if
    ::(mpm_state[reg_j] == rsfree0) ->
        goto irq_start;
    ::else ->
        reg_j = reg_j + 1;
    fi;
od;
reg_j = 0;
do
::(reg_j == M) -> break;
::else ->
    if
    ::(mpm_state[reg_j] == free0) ->
        state = free0;
        goto irq_start;
    ::else ->
        reg_j = reg_j + 1;
    fi;
od;
reg_j = 0;
do
::(reg_j == M) -> break;
::else ->
    if
    ::(mpm_state[reg_j] == stopped0) ->
        state = stopped0;
        goto irq_start;
    ::else ->
        reg_j = reg_j + 1;
    fi;
od;
reg_j = 0;
inst_tag = 0;
goto otherwise;
```

```
    ::(instruction == inst_stop) ->
        instruction = inst_none;
        inst_tag = 0;
        reg_i = 0;
        goto end;

    ::(instruction == inst_noop) ->
        instruction = inst_none;
        inst_tag = 0;
        in_a!reg_i,not_c;
        reg_i = 0;
        goto end;
    fi;

irq_getfr:  partner[reg_j] = reg_i;
            partner[reg_i] = reg_j;
            in_p!reg_j,getfr;
            (mpm_state[reg_j] != state);
            state = rsfree0;
            reg_j = 0;
            in_a!reg_i,c;
            reg_i = 0;
            goto end;

irq_relfr:  thread_id[reg_j] = inst_tag;
            inst_tag = 0;
            partner[reg_j] = reg_i;
            partner[reg_i] = reg_j;
            in_p!reg_j,relfr;
            (mpm_state[reg_j] != state);
            state = rsfree0;
            reg_j = 0;
            in_a!reg_i,c;
            reg_i = 0;
            goto end;

irq_start:  thread_id[reg_j] = inst_tag;
            inst_tag = 0;
            partner[reg_j] = reg_i;
            partner[reg_i] = reg_j;
            in_p!reg_j,start;
            (mpm_state[reg_j] != state);
            state = rsfree0;
            reg_j = 0;
            in_a!reg_i,c;
            reg_i = 0;
            goto end;

otherwise:  do
            ::(reg_j == M) ->
                reg_j = 0;
                in_a!reg_i,a;
                reg_i = 0;
```

```
                goto end;
            ::else ->
                if
                ::(mpm_state[reg_j] == act0_waiting) ->
                    reg_j = 0;
                    in_a!reg_i,oload;
                    reg_i = 0;
                    goto end;
                ::else ->
                    reg_j = reg_j + 1;
                fi;
            od;
    }


/*-------------------------------------------------------------*/
/*                  PROCTYPE ROUTINE_NEWTAG                    */
/*-------------------------------------------------------------*/

proctype routine_newTag()
    {
    byte i,j,t = 0;

    do
    ::newTag?j,i ->
        do
        ::(osm_state[t] == free) ->
            break;
        ::else ->
            t = (t+1) % N;
        od;
        osm_state[t] = on_processor;
        thread_reg_tags[t] = j;
        in_a!i,restart;
        (mpm_state[i] != act1);
        reg_thread_tags[j] = t;
        i = 0;
        j = 0;
        t = 0;
    od
    }


/*-------------------------------------------------------------*/
/*                 PROCTYPE ROUTINE_WRITEOUT                   */
/*-------------------------------------------------------------*/

proctype routine_writeOut()
    {
    byte i,j = 0;

    do
    ::writeOut?j,i ->
        osm_state[reg_thread_tags[j]] = stopped;
```

```
        thread_reg_tags[reg_thread_tags[j]] = M;
        if
        ::(i < M) ->
            in_a!i,restart;
            (mpm_state[i] != act1);
        ::else ->
            skip;
        fi;
        reg_thread_tags[j] = N;
        i = 0;
        j = 0;
    od
    }


/*-----------------------------------------------------------------*/
/*                   PROCTYPE ROUTINE_READIN                       */
/*-----------------------------------------------------------------*/

proctype routine_readIn()
    {
    byte i,j,t = 0;

    do
    ::readIn?j,i,t ->
        osm_state[t] = on_processor;
        thread_reg_tags[t] = j;
        in_a!i,restart;
        (mpm_state[i] != act1);
        reg_thread_tags[j] = t;
        i = 0;
        j = 0;
        t = 0;
    od
    }


/*-----------------------------------------------------------------*/
/*                   PROCTYPE ROUTINE_FREETAG                      */
/*-----------------------------------------------------------------*/

proctype routine_freeTag()
    {
    byte i,j = 0;

    do
    ::freeTag?j,i ->
        osm_state[reg_thread_tags[j]] = free;
        thread_reg_tags[reg_thread_tags[j]] = M;
        if
        ::(i < M) ->
            in_a!i,restart;
            (mpm_state[i] != act1);
        ::else ->
```

```
            skip;
        fi;
        reg_thread_tags[j] = N;
        i = 0;
        j = 0;
    od
    }


/*-------------------------------------------------------------*/
/*                  PROCTYPE ROUTINE_U_LOAD                    */
/*-------------------------------------------------------------*/

proctype routine_u_load()
    {
    byte j,t = 0;

    do
    ::u_load?j ->
        do
        ::(osm_state[t] == act) ->
            break;
        ::else ->
            t = (t+1) % N;
        od;
        osm_state[t] = on_processor;
        thread_reg_tags[t] = j;
        reg_thread_tags[j] = t;
        j = 0;
        t = (t+1) % N;
    od
    }


/*-------------------------------------------------------------*/
/*                  PROCTYPE ROUTINE_O_LOAD                    */
/*-------------------------------------------------------------*/

proctype routine_o_load()
    {
    byte j = 0;

    do
    ::o_load?j ->
        osm_state[reg_thread_tags[j]] = act;
        thread_reg_tags[reg_thread_tags[j]] = M;
        reg_thread_tags[j] = N;
        j = 0;
    od
    }
```

```
/*-------------------------------------------------------------*/
/*                    PROCTYPE SEND_ENABLE                     */
/*-------------------------------------------------------------*/

proctype send_enable()
     {
     byte i = 0;

end: if
     ::(i == 0) ->
         en_send_uload!1;
         en_send_uload!0;
         en_send_b!1;
         en_send_b!0;
     ::else ->
         skip;
     fi;
     if
     ::(mpm_state[i] >= act1) ->
         enable_a!i;
         enable_a!i;
     ::(mpm_state[i] < act1) ->
         enable_p!i;
         enable_p!i;
     fi;
     i = (i+1) % M;
     goto end;
     }


/*-------------------------------------------------------------*/
/*                     PROCTYPE SEND_TIOUT                     */
/*-------------------------------------------------------------*/

proctype send_tiout(byte reg_i)
     {
     byte ti = 0;

end: do
     ::(ti < max_ti) ->
         ti = ti + 1;
         tiout[reg_i]!0;
     ::(ti == max_ti) ->
         ti = 0;
         tiout[reg_i]!1;
     od
     }



/*-------------------------------------------------------------*/
/*                       PROCTYPE SEND_B                       */
/*-------------------------------------------------------------*/
```

```
proctype send_b()
     {
     byte i,j = 0;

end: en_send_b?1;
     do
     ::(i == M) ->
         i = 0;
         en_send_b?0;
         goto end;
     ::(i < M) ->
         if
         ::(mpm_state[i] == act0_waiting) ->
             break;
         ::else ->
             i = i + 1;
         fi;
     od;
     do
     ::(j == M) ->
         j = 0;
         i = 0;
         en_send_b?0;
         goto end;
     ::(j < M) ->
         if
         ::(mpm_state[j] == free0) ->
             j = 0;
             in_a!i,b;
             (mpm_state[i] != act0_waiting);
             i = 0;
             en_send_b?0;
             goto end;
         ::(mpm_state[j] == rsfree0) ->
             j = 0;
             in_a!i,b;
             (mpm_state[i] != act0_waiting);
             i = 0;
             en_send_b?0;
             goto end;
         ::(mpm_state[j] == stopped0) ->
             j = 0;
             in_a!i,b;
             (mpm_state[i] != act0_waiting);
             i = 0;
             en_send_b?0;
             goto end;
         ::else ->
             j = j+1;
         fi;
     od;
     }
```

```
/*-------------------------------------------------------------*/
/*                    PROCTYPE SEND_ULOAD                      */
/*-------------------------------------------------------------*/

proctype send_uload()
     {
     byte i,j = 0;
     byte r,f,s = M;

end: en_send_uload?1;
     do
     ::(i == N) ->
         i = 0;
         en_send_uload?0;
         goto end;
     ::else ->
         if
         ::(osm_state[i] != act) ->
             i = i + 1;
         ::(osm_state[i] == act) ->
             break;
         fi;
     od;
     do
     ::(j == M) ->
         j = 0;
         break;
     ::(j < M) ->
         if
         ::(mpm_state[j] == rsfree0) ->
             r = j;
             j = j+1;
         ::(mpm_state[j] == free0) ->
             f = j;
             j = j+1;
         ::(mpm_state[j] == stopped0) ->
             s = j;
             j = j+1;
         ::else ->
             i = 0;
             j = 0;
             r = M;
             f = M;
             s = M;
             en_send_uload?0;
             goto end;
         fi;
     od;
     if
     ::(osm_state[i] == act) ->
         i = 0;
         if
         ::(r < M) ->
             f = M;
```

```
            s = M;
            in_p!r,uload;
            (mpm_state[r] != rsfree0);
            r = M;
        ::else ->
            if
            ::(f < M) ->
                s = M;
                in_p!f,uload;
                (mpm_state[f] != free0);
                f = M;
            ::else ->
                in_p!s,uload;
                (mpm_state[s] != stopped0);
                s = M;
            fi;
        fi;
    ::else ->
        i = 0;
        r = M;
        f = M;
        s = M;
    fi;
    en_send_uload?0;
    goto end;
    }


/*-------------------------------------------------------------*/
/*                    PROCTYPE SEND_RND_TAG                    */
/*-------------------------------------------------------------*/

proctype send_rnd_tag(byte t)
    {
    do
    ::rnd_tag!t;
    od
    }


/*-------------------------------------------------------------*/
/*                      PROCTYPE PROGRAM                       */
/*-------------------------------------------------------------*/

proctype program()
    {
    byte i,j;

    do
    ::get_inst?i ->
        rnd_tag?j;
        if
        ::(vtm_state[j] == stopped) ->
            vtm_state[j] = free;
```

```
            i_tag[i] = j;
            j = 0;
            inst[i] = inst_relfr;
            i = 0;
        ::(vtm_state[i] == act) ->
            j = 0;
            vtm_state[i] = free;
            i_tag[i] = i;
            inst[i] = inst_relfr;
            i = 0;
        ::(vtm_state[i] == act) ->
            j = 0;
            vtm_state[i] = stopped;
            i_tag[i] = N;
            inst[i] = inst_stop;
            i = 0;
        ::(vtm_state[j] == stopped) ->
            vtm_state[j] = act;
            i_tag[i] = j;
            j = 0;
            inst[i] = inst_start;
            i = 0;
        ::(n < N) ->
            n = n + 1;
            j = 0;
            i_tag[i] = N;
            inst[i] = inst_getfr;
            i = 0;
        ::(vtm_state[i] == act) ->
            j = 0;
            i_tag[i] = N;
            inst[i] = inst_noop;
            i = 0;
        fi;
    od
    }


/*-----------------------------------------------------------------*/
/*                            INIT                                 */
/*-----------------------------------------------------------------*/

init
    {
    byte i = 0;

    run register_frame_a();
    run register_frame_p();
    run routine_newTag();
    run routine_writeOut();
    run routine_readIn();
    run routine_freeTag();
    run routine_o_load();
    run routine_u_load();
```

```
     run processor();
     run send_b();
     run send_uload();
end: run send_tiout(i);
     i = i+1;
     if
     ::(i < M) -> goto end;
     ::else -> i=0;
     fi;

     run program();
     do
     ::(i < N) ->
         run send_rnd_tag(i);
         i = i + 1;
     ::(i == N) ->
         i = 0;
         break;
     od;

     n = 1;               /* einen thread aktivieren */
     rnd_tag?i;
     vtm_state[i] = act;
     osm_state[i] = act;
     i = 0;

     run send_enable();   /* erst jetzt duerfen   */
                          /* die Prozesse starten */
     }



#define osm_act (osm_state[0] == act)
#define osm_on_processor (osm_state[0] == on_processor)

/*
 * Formula As Typed: []  (osm_act  -> <>  osm_on_processor)
 * The Never Claim Below Corresponds
 * To The Negated Formula !([]  (osm_act  -> <>  osm_on_processor))
 * (formalizing violations of the original)
 */

never {    /* !([]  (osm_act  -> <>  osm_on_processor)) */
T0_init:
        if
        :: (1) -> goto T0_init
        :: (! ((osm_on_processor)) && (osm_act)) -> goto accept_S4
        fi;
accept_S4:
        if
        :: (! ((osm_on_processor))) -> goto T0_S4
        fi;
T0_S4:
        if
```

```
        :: (! ((osm_on_processor))) -> goto accept_S4
        fi;
accept_all:
        skip
      }
```

Anhang B

SMV-Beschreibung des Rhamma-Prozessors

In diesem Anhang ist die SMV-Beschreibung des Rhamma-Prozessors, die für die Verifikation unter Ausnutzung der Symmetrien mit der neuen Implementierung des SMV verwendet wurde, aufgeführt. Um eine Verifikation ohne Symmetrie-Reduktion durchzuführen, ist die Beschreibung wie folgt zu ändern:

- Die Variablen für die Nummern der Registersätze und die Nummern der Threads dürfen nicht als symmetrisch erklärt werden. Deshalb müssen die Zeilen
  `scalarset THREAD_INDEX 1..3;`
  `scalarset FRM_INDEX 1..2;`
  durch die Zeilen
  `#define N 3`
  `#define M 2`
  `#define THREAD_INDEX 1..N`
  `#define FRM_INDEX 1..M`
  ersetzt werden.

- Einige Formulierungen, die bei der Verifikation mit Symmetrie-Reduktion verwendet werden müssen, dürfen bei der Verifikation ohne Symmetrie-Reduktion nicht verwendet werden. Deshalb sind zusätzlich folgende Arten von Ersetzungen notwendig:
  `forall(i in THREAD_INDEX)` $\mapsto$ `for(i=1; i<=N; i=i+1)`
  `{i :i in THREAD_INDEX}` $\mapsto$ `{i :i = THREAD_INDEX}`
  `&[cond[j] :j in FRM_INDEX]` $\mapsto$ `&[cond[j] :j = FRM_INDEX]`

Auch in der SMV-Beschreibung sind vor der Verifikation die gewünschten Werte für die Anzahl der Registersätze und die Obergrenze der Threads einzusetzen.

```
scalarset THREAD_INDEX 1..3;
scalarset FRM_INDEX 1..2;

typedef OSM_STATES {m_non_exist, m_stopped, m_running, m_processor};
typedef FRM_STATES
        {rsfree0, rsfree1_readIn, rsfree1_u_load, rsfree1_newTag,
         free0, free1_readIn, free1_u_load, free1_freeTag,
         free1_newTag, stopped0, stopped1_writeOut,
         stopped2_writeOut, stopped1_readIn, stopped1_u_load,
```

```
         active0, active1, active0_restart, waiting,
         active1_o_load};
typedef INSTRUCTION
        {inst_getfr, inst_relfr, inst_start, inst_stop, inst_noop};


module osm(error,freeTag,newTag,writeOut,readIn,o_load,u_load)
{
  input error    : boolean;
  input freeTag  : array THREAD_INDEX of boolean;
  input newTag   : array THREAD_INDEX of boolean;
  input writeOut : array THREAD_INDEX of boolean;
  input readIn   : array THREAD_INDEX of boolean;
  input o_load   : array THREAD_INDEX of boolean;
  input u_load   : array THREAD_INDEX of boolean;

  state   : array THREAD_INDEX of OSM_STATES;

  forall(i in THREAD_INDEX)
    init(state[i]) := {m_non_exist, m_stopped, m_running};

  forall(i in THREAD_INDEX)
    next(state[i]) :=
      case
        {
        error                                 : m_non_exist;
        (state[i]=m_non_exist) & newTag[i]    : m_processor;
        (state[i]=m_stopped) & readIn[i]      : m_processor;
        (state[i]=m_running) & u_load[i]      : m_processor;
        (state[i]=m_processor) & freeTag[i]   : m_non_exist;
        (state[i]=m_processor) & writeOut[i] : m_stopped;
        (state[i]=m_processor) & o_load[i]    : m_running;
        default                               : state[i];
        };
}


module frames(error,enable,instruction,thread_id,getfr,relfr,start,
              restart,oload,uload,a,b,c,d,timeout,passive,
              thread4irq,freeTag,newTag,writeOut,readIn,o_load,
              u_load)
{
  input error         : boolean;
  input enable        : FRM_INDEX;
  input instruction   : INSTRUCTION;
  input thread_id     : THREAD_INDEX;
  input getfr         : array FRM_INDEX of boolean;
  input relfr         : array FRM_INDEX of boolean;
  input start         : array FRM_INDEX of boolean;
  input restart       : array FRM_INDEX of boolean;
  input oload         : array FRM_INDEX of boolean;
  input uload         : array FRM_INDEX of boolean;
  input a             : array FRM_INDEX of boolean;
  input b             : array FRM_INDEX of boolean;
```

```
input c             : array FRM_INDEX of boolean;
input d             : array FRM_INDEX of boolean;
input timeout       : array FRM_INDEX of boolean;
input passive       : FRM_INDEX;
input thread4irq    : THREAD_INDEX;

output freeTag      : array THREAD_INDEX of boolean;
output newTag       : array THREAD_INDEX of boolean;
output writeOut     : array THREAD_INDEX of boolean;
output readIn       : array THREAD_INDEX of boolean;
output o_load       : array THREAD_INDEX of boolean;
output u_load       : array THREAD_INDEX of boolean;

state         : array FRM_INDEX of FRM_STATES;
tag           : array FRM_INDEX of THREAD_INDEX;
tag_valid     : array FRM_INDEX of boolean;
partner       : array FRM_INDEX of FRM_INDEX;
last_instr    : array THREAD_INDEX of INSTRUCTION;
last_thread_id: array THREAD_INDEX of THREAD_INDEX;
timeouted     : array FRM_INDEX of boolean;


forall (j in FRM_INDEX)
  init(state[j]) := rsfree0;

forall (j in FRM_INDEX)
  init(tag_valid[j]) := 0;

forall (j in FRM_INDEX)
  init(tag[j]) := {i :i in THREAD_INDEX};

forall (j in FRM_INDEX)
  init(partner[j]) := {k :k in FRM_INDEX};

forall (i in THREAD_INDEX)
  init(last_instr[i]) := inst_noop;

forall (i in THREAD_INDEX)
  init(last_thread_id[i]) := {l :l in THREAD_INDEX};

forall (j in FRM_INDEX)
  init(timeouted[j]) := 0;

forall (j in FRM_INDEX)
  next(state[j]) :=
    case
      {
      error:                                    rsfree0;
      (state[j]=rsfree0) & getfr[j]:            rsfree1_newTag;
      (state[j]=rsfree0) & start[j]:            free1_readIn;
      (state[j]=rsfree0) & relfr[j]:            free1_readIn;
      (state[j]=rsfree0) & uload[j]:            rsfree1_u_load;
      (state[j]=free1_readIn) & enable=j:       stopped1_readIn;
      (state[j]=stopped1_readIn) & relfr[j      free0;
```

```
      (state[j]=stopped1_readIn) & start[j]:    active0;
      (state[j]=free0) & getfr[j]:              stopped0;
      (state[j]=free0) & start[j] & ~(thread_id=tag[j]):
                                                free1_freeTag;
      (state[j]=free0) & uload[j]:              free1_u_load;
      (state[j]=free0) & relfr[j] & ~(thread_id=tag[j]):
                                                free1_freeTag;
      (state[j]=free1_freeTag) & enable=j:      rsfree1_readIn;
      (state[j]=rsfree1_readIn) & start[j]:     free1_readIn;
      (state[j]=rsfree1_readIn) & relfr[j]:     free1_readIn;
      (state[j]=free1_u_load) & enable=j:       rsfree1_u_load;
      (state[j]=rsfree1_newTag) & enable=j:     free1_newTag;
      (state[j]=rsfree1_u_load) & enable=j:     active0;
      (state[j]=free1_newTag) & getfr[j]:       stopped0;
      (state[j]=stopped1_writeOut) & enable=j:  rsfree1_newTag;
      (state[j]=stopped2_writeOut) & enable=j   rsfree1_readIn;
      (state[j]=stopped1_u_load) & enable=j:    rsfree1_u_load;
      (state[j]=stopped0) & start[j] & (tag[j]=thread_id):
                                                active0;
      (state[j]=stopped0)&start[j]& ~(tag[j]=thread_id):
                                                stopped2_writeOut;
      (state[j]=stopped0) & relfr[j] & (tag[j]=thread_id):
                                                free0;
      (state[j]=stopped0)&relfr[j]& ~(tag[j]=thread_id):
                                                stopped2_writeOut;
      (state[j]=stopped0) & getfr[j]:           stopped1_writeOut;
      (state[j]=stopped0) & uload[j]:           stopped1_u_load;
      (state[j]=active0) & timeouted[j] & enable=j:
                                                active1_o_load;
      (state[j]=active0) & oload[j] & enable=j: active1_o_load;
      (state[j]=active0) & c[j] & enable=j:     active1;
      (state[j]=active0) & (instruction=inst_stop) & enable=j:
                                                stopped0;
      (state[j]=active0) & relfr[j] & enable=j: free0;
      (state[j]=active0) & a[j] & enable=j:     waiting;
      (state[j]=active1) & restart[j]:          active0_restart;
      (state[j]=active0_restart) & ~d[j] & enable=j:
                                                active0;
      (state[j]=active0_restart) &  d[j] & enable=j:
                                                active1;
      (state[j]=active1_o_load) & enable=j:     rsfree0;
      (state[j]=waiting) & timeout[j]:          active1_o_load;
      (state[j]=waiting) & b[j]:                active0;
      default:                                  state[j];
      };

forall (j in FRM_INDEX)
  next(tag[j]) :=
    case
      {
      error:                              {i: i in THREAD_INDEX};
      (state[j]=rsfree1_newTag) & enable=j:       thread4irq;
      (state[j]=rsfree1_u_load) & enable=j:       thread4irq;
      (state[j]=rsfree1_readIn) & start[j]:       thread_id;
```

```
      (state[j]=rsfree1_readIn) & relfr[j]:       thread_id;
      (state[j]=rsfree0) & start[j]:              thread_id;
      (state[j]=rsfree0) & relfr[j]:              thread_id;
      default:                                    tag[j];
      };

forall (j in FRM_INDEX)
  next(tag_valid[j]) :=
    case
      {
      error:                                                0;
      (state[j]=free1_freeTag) & enable=j:                  0;
      (state[j]=free1_u_load) & enable=j:                   0;
      (state[j]=stopped1_writeOut) & enable=j:              0;
      (state[j]=stopped2_writeOut) & enable=j:              0;
      (state[j]=stopped1_u_load) & enable=j:                0;
      (state[j]=active1_o_load) & enable=j:                 0;
      enable=j & (state[j] in {rsfree1_newTag, rsfree1_u_load}):
                                                            1;
      (state[j]=rsfree1_readIn) & start[j]:                 1;
      (state[j]=rsfree1_readIn) & relfr[j]:                 1;
      (state[j]=rsfree0) & start[j]:                        1;
      (state[j]=rsfree0) & relfr[j]:                        1;
      default:                                          tag_valid[j];
      };


forall (j in FRM_INDEX)
  next(partner[j]) :=
    case
      {
      error:                                    {k: k in FRM_INDEX};
      (state[j]=active0) & c[j] & enable=j:                 passive;
      (state[j]=active0_restart) & ~d[j] & enable=j:
                                                {k: k in FRM_INDEX};
      (state[j]=free0) & start[j] & ~(thread_id=tag[j]):    enable;
      (state[j]=free0) & relfr[j] & ~(thread_id=tag[j]):    enable;
      (state[j]=stopped0) & start[j] & ~(tag[j]=thread_id):
                                                            enable;
      (state[j]=stopped0) & relfr[j] & ~(tag[j]=thread_id):
                                                            enable;
      (state[j]=stopped0) & getfr[j]:                       enable;
      (state[j]=rsfree0) & getfr[j]:                        enable;
      (state[j]=rsfree0) & start[j]:                        enable;
      (state[j]=rsfree0) & relfr[j]:                        enable;
      (state[j]=stopped1_readIn) & relfr[j]:    {k: k in FRM_INDEX};
      (state[j]=stopped1_readIn) & start[j]:    {k: k in FRM_INDEX};
      (state[j]=free1_newTag) & getfr[j]:       {k: k in FRM_INDEX};
      default:                                              partner[j];
      };

forall (i in THREAD_INDEX)
  next(last_instr[i]) :=
    case
```

```
      {
      error:                                          inst_noop;
      |[tag[j]=i & tag_valid[j] & state[j]=active0 & c[j]
        & enable=j :j in FRM_INDEX]:                  instruction;
      |[tag[j]=i & tag_valid[j] & state[j]=active0 & a[j]
        & enable=j :j in FRM_INDEX]:                  instruction;
      |[tag[j]=i & tag_valid[j] & state[j]=active0 & oload[j]
        & enable=j :j in FRM_INDEX]:                  instruction;
      |[tag[j]=i & tag_valid[j] & state[j]=active0 & ~timeouted[j]
        & ~c[j] & enable=j :j in FRM_INDEX]:          inst_noop;
      |[tag[j]=i & tag_valid[j] & state[j]=active0_restart & ~d[j]
        & enable=j :j in FRM_INDEX]:                  inst_noop;
      default:                                       last_instr[i];
      };

forall (i in THREAD_INDEX)
  next(last_thread_id[i]) :=
    case
      {
      error:                                {l:l in THREAD_INDEX};
      |[tag[j]=i & tag_valid[j] & state[j]=active0 & c[j]
        & enable=j :j in FRM_INDEX]:                    thread_id;
      |[tag[j]=i & tag_valid[j] & state[j]=active0 & a[j]
        & enable=j :j in FRM_INDEX]:                    thread_id;
      |[tag[j]=i & tag_valid[j] & state[j]=active0 & oload[j]
        & enable=j :j in FRM_INDEX]:                    thread_id;
      |[tag[j]=i & tag_valid[j] & state[j]=active0 & ~timeouted[j]
        & ~c[j]& enable=j :j in FRM_INDEX]: {l:l in THREAD_INDEX};
      |[tag[j]=i & tag_valid[j] & state[j]=active0_restart & ~d[j]
        & enable=j :j in FRM_INDEX]:        {l:l in THREAD_INDEX};
      default:                                    last_thread_id[i];
      };

forall (j in FRM_INDEX)
  next(timeouted[j]) :=
    case
      {
      timeout[j]                        : 1;
      (state[j]=rsfree1_u_load)         : 0;
      default                           : timeouted[j];
      };

forall (i in THREAD_INDEX)
  readIn[i] := |[enable=j & tag_valid[j] & tag[j]=i
                & state[j]=free1_readIn : j in FRM_INDEX];

forall (i in THREAD_INDEX)
  freeTag[i] := |[enable=j & tag_valid[j] & tag[j]=i
                  & state[j] in {free1_freeTag, free1_u_load}
                    : j in FRM_INDEX];

forall (i in THREAD_INDEX)
  o_load[i] := |[enable=j & tag_valid[j] & tag[j]=i
                & state[j]=active1_o_load : j in FRM_INDEX];
```

```
  forall (i in THREAD_INDEX)
    writeOut[i] := |[enable=j & tag_valid[j] & tag[j]=i
                     & state[j] in {stopped1_u_load,
                       stopped1_writeOut, stopped2_writeOut}
                       : j in FRM_INDEX];

  forall (i in THREAD_INDEX)
    u_load[i] := i=thread4irq & state[enable]=rsfree1_u_load;

  forall (i in THREAD_INDEX)
    newTag[i] := i=thread4irq & state[enable]=rsfree1_newTag;

}



module mpm(error,enable,instruction,thread_id,timeout,send_uload,
           thread4irq,freeTag,newTag,writeOut,readIn,o_load,u_load)
{
  input error : boolean;
  input enable : FRM_INDEX;
  input instruction : INSTRUCTION;
  input thread_id : THREAD_INDEX;
  input timeout       : array FRM_INDEX of boolean;
  input send_uload : boolean;
  input thread4irq :  THREAD_INDEX;
  output freeTag : array THREAD_INDEX of boolean;
  output newTag  : array THREAD_INDEX of boolean;
  output writeOut: array THREAD_INDEX of boolean;
  output readIn  : array THREAD_INDEX of boolean;
  output o_load  : array THREAD_INDEX of boolean;
  output u_load  : array THREAD_INDEX of boolean;

  getfr         : array FRM_INDEX of boolean;
  relfr         : array FRM_INDEX of boolean;
  start         : array FRM_INDEX of boolean;
  restart       : array FRM_INDEX of boolean;
  oload         : array FRM_INDEX of boolean;
  uload         : array FRM_INDEX of boolean;
  a             : array FRM_INDEX of boolean;
  b             : array FRM_INDEX of boolean;
  c             : array FRM_INDEX of boolean;
  d             : array FRM_INDEX of boolean;

  next_instr    : INSTRUCTION;
  next_thread_id: THREAD_INDEX;
  passive       : FRM_INDEX;
  passive_valid : boolean;
  frame4uload   : FRM_INDEX;

  regs : frames(error,enable,next_instr,next_thread_id,getfr,relfr,
                start,restart,oload,uload,a,b,c,d,timeout,passive,
                thread4irq,freeTag,newTag,writeOut,readIn,o_load,
                u_load);
```

```
next_instr :=
  case
    {
    regs.state[enable]=active0
      & regs.last_instr[regs.tag[enable]]=inst_noop  :instruction;
                                    /* neuen Befehl beginnen */
    regs.state[enable]=active0
      & ~regs.last_instr[regs.tag[enable]]=inst_noop
                              :regs.last_instr[regs.tag[enable]];
             /* alten Befehl forts., nach uload oder waiting */
    regs.state[enable]=active0_restart
             :regs.last_instr[regs.tag[enable]];
                        /* alten Befehl forts., waehrend irq */
    regs.state[enable]=free1_readIn
             :regs.last_instr[regs.tag[regs.partner[enable]]];
                        /* alten Befehl forts., waehrend irq */
    default : inst_noop;
    };

next_thread_id :=
  case
    {
    regs.state[enable]=active0
      & regs.last_instr[regs.tag[enable]]=inst_noop    :thread_id;
    regs.state[enable]=active0
      & ~regs.last_instr[regs.tag[enable]]=inst_noop
                           :regs.last_thread_id[regs.tag[enable]];
    regs.state[enable]=active0_restart
                           :regs.last_thread_id[regs.tag[enable]];
    regs.state[enable]=free1_readIn
            :regs.last_thread_id[regs.tag[regs.partner[enable]]];
    default :                               {i : i in THREAD_INDEX};
    };

passive :=       /* Zielframe fuer Signale start,relfr,getfr */
  case
    {
    next_instr=inst_relfr & next_thread_id=regs.tag[enable]
      & regs.tag_valid[enable] & ~regs.timeouted[enable] : enable;
    next_instr in {inst_relfr, inst_start, inst_getfr}
      & regs.state[enable]=active0_restart : regs.partner[enable];
    next_instr in {inst_relfr, inst_start}
      & ~regs.timeouted[enable]
      & |[next_thread_id=regs.tag[j] & regs.tag_valid[j]
          & regs.state[j]=stopped0:j in FRM_INDEX]
               : {j : j in FRM_INDEX,
                    next_thread_id=regs.tag[j]&regs.tag_valid[j]};
    next_instr = inst_getfr & ~regs.timeouted[enable]
      & |[regs.state[j]=free0:j in FRM_INDEX]
                     : {j : j in FRM_INDEX, regs.state[j]=free0};
    next_instr in {inst_relfr, inst_start, inst_getfr}
      & ~regs.timeouted[enable]
      & |[regs.state[j]=rsfree0:j in FRM_INDEX]
```

```
                    : {j : j in FRM_INDEX, regs.state[j]=rsfree0};
    next_instr in {inst_relfr, inst_start}
      & ~regs.timeouted[enable]
      & |[regs.state[j]=free0:j in FRM_INDEX]
                     : {j : j in FRM_INDEX, regs.state[j]=free0};
    next_instr in {inst_relfr, inst_start, inst_getfr}
      & ~regs.timeouted[enable]
      & |[regs.state[j]=stopped0:j in FRM_INDEX]
                  : {j : j in FRM_INDEX, regs.state[j]=stopped0};
    default                                : {j : j in FRM_INDEX};
    };

passive_valid :=       /* Zielframe gueltig ? */
  0
  |(next_instr=inst_relfr & next_thread_id=regs.tag[enable]
    & regs.tag_valid[enable] & ~regs.timeouted[enable])
  |(next_instr in {inst_relfr, inst_start, inst_getfr}
    & regs.state[enable]=active0_restart)
  |(next_instr in {inst_relfr, inst_start}
    & ~regs.timeouted[enable]
    & |[next_thread_id=regs.tag[j] & regs.tag_valid[j]
        & regs.state[j]=stopped0:j in FRM_INDEX])
  |(next_instr = inst_getfr & ~regs.timeouted[enable]
    & |[regs.state[j]=free0:j in FRM_INDEX])
  |(next_instr in {inst_relfr, inst_start, inst_getfr}
    & ~regs.timeouted[enable]
    & |[regs.state[j]=rsfree0:j in FRM_INDEX])
  |(next_instr in {inst_relfr, inst_start}
    & ~regs.timeouted[enable]
    & |[regs.state[j]=free0:j in FRM_INDEX])
  |(next_instr in {inst_relfr, inst_start, inst_getfr}
    & ~regs.timeouted[enable]
    & |[regs.state[j]=stopped0:j in FRM_INDEX]);

forall (j in FRM_INDEX)
  getfr[j] := next_instr=inst_getfr & passive=j & passive_valid;

forall (j in FRM_INDEX)
  relfr[j] := next_instr=inst_relfr & passive=j & passive_valid;

forall (j in FRM_INDEX)
  start[j] := next_instr=inst_start & passive=j & passive_valid;

forall (j in FRM_INDEX)
  restart[j] := regs.partner[enable]=j
      & regs.state[enable] in {free1_freeTag, free1_readIn,
                               rsfree1_newTag, stopped2_writeOut};

forall (j in FRM_INDEX)
  oload[j] := j=enable
      & next_instr in {inst_getfr, inst_relfr, inst_start}
      & ~passive_valid & ~regs.timeouted[enable]
      & regs.state[enable]=active0
      & |[regs.state[k]=waiting :k in FRM_INDEX];
```

```
  frame4uload :=
    case
      {
      send_uload & |[regs.state[j]=rsfree0 :j in FRM_INDEX]
        : {j :j in FRM_INDEX, regs.state[j]=rsfree0};
      send_uload & |[regs.state[j]=free0 :j in FRM_INDEX]
        : {j :j in FRM_INDEX, regs.state[j]=free0};
      send_uload & |[regs.state[j]=stopped0 :j in FRM_INDEX]
        : {j :j in FRM_INDEX, regs.state[j]=stopped0};
      default : {j :j in FRM_INDEX};
      };

  forall (j in FRM_INDEX)
    uload[j] := send_uload & j=frame4uload;

  forall (j in FRM_INDEX)
    a[j] := j=enable
      & next_instr in {inst_getfr, inst_relfr, inst_start}
      & ~passive_valid & ~regs.timeouted[enable]
      & regs.state[enable]=active0
      & ~|[regs.state[k]=waiting :k in FRM_INDEX];

  forall (j in FRM_INDEX)
    b[j] := |[regs.state[k] in {rsfree0, free0, stopped0}
                :k in FRM_INDEX];


  forall (j in FRM_INDEX)
    c[j] := j=enable & passive_valid & regs.state[enable]=active0
        & ((next_instr=inst_getfr & ~regs.state[passive]=free0)
           |(next_instr in {inst_relfr, inst_start}
             & ~(regs.tag[passive]=next_thread_id
                  & regs.tag_valid[passive] & regs.state[passive] in
                       {stopped0, stopped1_readIn}
               | passive=enable)));

  forall (j in FRM_INDEX)
    d[j] := regs.state[j]=active0_restart
              & regs.state[regs.partner[j]]=rsfree1_readIn;

}

module main(enable, instruction, thread_id, timeout)
{
  error         : boolean;
  input enable        : FRM_INDEX;
  input instruction   : INSTRUCTION;
  input thread_id     : THREAD_INDEX;
  input timeout       : array FRM_INDEX of boolean;
  send_uload    : boolean;
  thread4irq    : THREAD_INDEX;
  freeTag       : array THREAD_INDEX of boolean;
  newTag        : array THREAD_INDEX of boolean;
```

```
writeOut      : array THREAD_INDEX of boolean;
readIn        : array THREAD_INDEX of boolean;
o_load        : array THREAD_INDEX of boolean;
u_load        : array THREAD_INDEX of boolean;

MPM : mpm(error,enable,instruction,thread_id,timeout,send_uload,
         thread4irq,freeTag,newTag,writeOut,readIn,o_load,u_load);
OSM : osm(error,freeTag,newTag,writeOut,readIn,o_load,u_load);


send_uload := (|[OSM.state[i]=m_running :i in THREAD_INDEX])
      & (&[MPM.regs.state[j] in {rsfree0, free0, stopped0}
              :j in FRM_INDEX]);

thread4irq :=
  case
    {
    |[MPM.regs.state[j]=rsfree1_newTag :j in FRM_INDEX]
              : {i :i in THREAD_INDEX, OSM.state[i]=m_non_exist};
    |[MPM.regs.state[j]=rsfree1_u_load :j in FRM_INDEX]
              : {i :i in THREAD_INDEX, OSM.state[i]=m_running};
    default   : {i :i in THREAD_INDEX};
    };

init(error) := 0;

next(error) := 0
      |
      ~(MPM.next_instr=inst_start ->
              (OSM.state[MPM.next_thread_id]=m_stopped
              | |[MPM.regs.tag[j]=MPM.next_thread_id
                  & MPM.regs.tag_valid[j] & MPM.regs.state[j] in
                    {stopped0, stopped1_readIn} :j in FRM_INDEX]))
      |
      ~(MPM.next_instr=inst_relfr ->
              (OSM.state[MPM.next_thread_id]=m_stopped
              | |[MPM.regs.tag[j]=MPM.next_thread_id
                  & MPM.regs.tag_valid[j] & MPM.regs.state[j] in
                     {stopped0, stopped1_readIn} :j in FRM_INDEX]
              | (MPM.regs.tag[enable]=MPM.next_thread_id
                  & MPM.regs.state[enable]=active0)))
      |
      ~(MPM.next_instr=inst_getfr ->
              (|[OSM.state[i]=m_non_exist :i in THREAD_INDEX]
               | |[MPM.regs.state[j] in {free0, free1_newTag}
                  :j in FRM_INDEX]))
      |
      ~(|[newTag[i] :i in THREAD_INDEX] ->
              |[OSM.state[i]=m_non_exist :i in THREAD_INDEX])
      |
      ~(send_uload ->
              |[OSM.state[i]=m_running :i in THREAD_INDEX]);
```

```
  abstract fair_enable        : array FRM_INDEX of boolean;
  abstract fair_timeout       : array FRM_INDEX of boolean;

  forall (j in FRM_INDEX)
    fair_enable[j]: assert G F enable=j;

  forall (j in FRM_INDEX)
    fair_timeout[j]: assert G F timeout[j];

  not_prog_error: assert G ~error;

  abstract Tags_osm2mpm_injective : array FRM_INDEX of boolean;

  forall (j in FRM_INDEX)
    Tags_osm2mpm_injective[j]: assert
        G &[(MPM.regs.tag[k]=MPM.regs.tag[j] & MPM.regs.tag_valid[k]
            & MPM.regs.tag_valid[j]) -> k=j :k in FRM_INDEX];


/* ---------------------------------------------------- */
/*                      Fairness                        */
/* ---------------------------------------------------- */

  abstract u_load_fair   : array THREAD_INDEX of boolean;
  abstract Spec_fairness : array THREAD_INDEX of boolean;

  forall (i in THREAD_INDEX)
    u_load_fair[i]: assert (G F send_uload) ->
        G (OSM.state[i]=m_running -> F OSM.state[i]=m_processor);

  forall (i in THREAD_INDEX)
    Spec_fairness[i]: assert
        G (OSM.state[i]=m_running -> F OSM.state[i]=m_processor);


/* ---------------------------------------------------- */
/*                        VTM                           */
/* ---------------------------------------------------- */

  abstract vp_non_exist : array THREAD_INDEX of boolean;
  abstract vp_stopped   : array THREAD_INDEX of boolean;
  abstract vp_running   : array THREAD_INDEX of boolean;
  abstract vp_getfr     : array THREAD_INDEX of boolean;
  abstract vp_relfr     : array THREAD_INDEX of boolean;
  abstract vp_start     : array THREAD_INDEX of boolean;
  abstract vp_stop      : array THREAD_INDEX of boolean;
  abstract Spec_vtm     : array THREAD_INDEX of boolean;

  forall (i in THREAD_INDEX)
    vp_non_exist[i] := OSM.state[i]=m_non_exist
        | |[MPM.regs.tag[j]=i & MPM.regs.tag_valid[j]
            & MPM.regs.state[j] in {free0, free1_freeTag,
                    free1_u_load, free1_newTag}:j in FRM_INDEX];
```

```
forall (i in THREAD_INDEX)
  vp_stopped[i] := OSM.state[i]=m_stopped
      | |[MPM.regs.tag[j]=i & MPM.regs.tag_valid[j]
          & MPM.regs.state[j] in {stopped0, free1_readIn,
              stopped1_readIn, stopped1_u_load, stopped1_writeOut,
              stopped2_writeOut} :j in FRM_INDEX];

forall (i in THREAD_INDEX)
  vp_running[i] := OSM.state[i]=m_running
      | |[MPM.regs.tag[j]=i & MPM.regs.tag_valid[j]
          & MPM.regs.state[j] in {active0, active0_restart,
              active1, active1_o_load, waiting} :j in FRM_INDEX];


forall (i in THREAD_INDEX)
  vp_getfr[i] :=
      |[MPM.regs.tag[j]=i & MPM.regs.tag_valid[j]
          & MPM.regs.getfr[j] & MPM.regs.state[j] in {free0,
              free1_newTag} :j in FRM_INDEX];

forall (i in THREAD_INDEX)
  vp_relfr[i] :=
      |[MPM.regs.tag[j]=i & MPM.regs.tag_valid[j]
          & MPM.next_thread_id=i
          & (MPM.regs.state[j] in {stopped0, stopped1_readIn}
                & MPM.regs.relfr[j]
             | enable=j & ~MPM.regs.timeouted[j]
                & MPM.regs.state[j] in {active0, active0_restart}
                & MPM.next_instr=inst_relfr) :j in FRM_INDEX];

forall (i in THREAD_INDEX)
  vp_start[i] :=
      |[MPM.regs.tag[j]=i & MPM.regs.tag_valid[j]
          & MPM.next_thread_id=i & MPM.regs.state[j] in {stopped0,
            stopped1_readIn} & MPM.regs.start[j] :j in FRM_INDEX];

forall (i in THREAD_INDEX)
  vp_stop[i] :=
      |[MPM.regs.tag[j]=i & MPM.regs.tag_valid[j] & enable=j
        & MPM.next_instr=inst_stop & ~MPM.regs.timeouted[j]
        & MPM.regs.state[j]=active0 :j in FRM_INDEX];

forall (i in THREAD_INDEX)
  Spec_vtm[i]: assert
        (G (vp_non_exist[i] | vp_stopped[i] | vp_running[i]))
      & (G (vp_non_exist[i] -> ~vp_stopped[i] & ~vp_running[i]))
      & (G (vp_stopped[i] -> ~vp_non_exist[i] & ~vp_running[i]))
      & (G (vp_running[i] -> ~vp_non_exist[i] & ~vp_stopped[i]))

      & (G (vp_getfr[i] -> ~vp_relfr[i] & ~vp_start[i]
              & ~vp_stop[i]))
      & (G (vp_relfr[i] -> ~vp_getfr[i] & ~vp_start[i]
              & ~vp_stop[i]))
      & (G (vp_start[i] -> ~vp_relfr[i] & ~vp_getfr[i]
```

```
            & ~vp_stop[i]))
        & (G (vp_stop[i]  -> ~vp_relfr[i] & ~vp_start[i]
            & ~vp_getfr[i]))

        & (G ~(vp_non_exist[i] & vp_relfr[i]))
        & (G ~(vp_non_exist[i] & vp_start[i]))
        & (G ~(vp_non_exist[i] & vp_stop[i]))
        & (G ~(vp_stopped[i] & vp_getfr[i]))
        & (G ~(vp_stopped[i] & vp_stop[i]))
        & (G ~(vp_running[i] & vp_getfr[i]))
        & (G ~(vp_running[i] & vp_start[i]))

        & (G ((X vp_non_exist[i]) <->
            vp_stopped[i] & vp_relfr[i]
          | vp_running[i] & vp_relfr[i]
          | vp_non_exist[i] & ~vp_getfr[i] ))

        & (G ((X vp_stopped[i]) <->
            vp_non_exist[i] & vp_getfr[i]
          | vp_running[i] & vp_stop[i]
          | vp_stopped[i] & ~vp_relfr[i] & ~vp_start[i]))

        & (G ((X vp_running[i]) <->
            vp_stopped[i] & vp_start[i]
          | vp_running[i] & ~vp_relfr[i] & ~vp_stop[i]));


  using
    fair_enable, fair_timeout, not_prog_error, u_load_fair
  prove
    Spec_vtm, Spec_fairness, Tags_osm2mpm_injective;

}
```