

Entwurf und Implementierung paralleler Programme

Prof. Dr. Rolf Hennicker

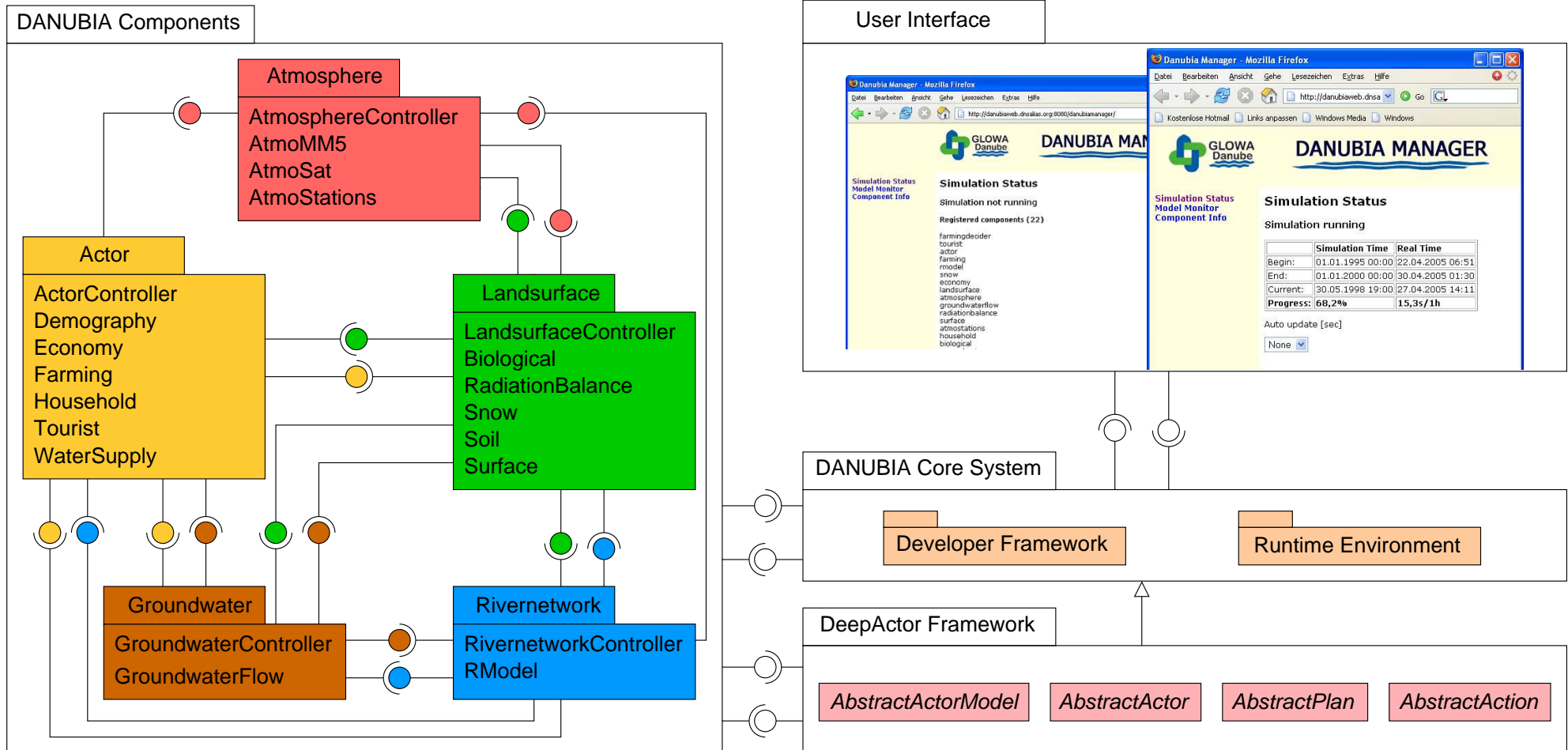
22.01.2007

Kapitel 9
Anwendung:
Koordination verteilter
Simulationen

9.1 Verteilte Umweltsimulationen in GLOWA-Danube

- GLOWA-Danube ist ein vom BMBF gefördertes Projekt innerhalb der Initiative GLOWA (Globaler Wandel des Wasserkreislaufs)
- Projekttitle:
“Integrative Techniques, Scenarios and Strategies for the Future of Water in the Upper Danube Basin“
- Projektpartner aus mehr als einem Dutzend verschiedener Fachgebiete aus Naturwissenschaften und Sozialwissenschaften
- Projektziel: Entwicklung des verteilten Systems DANUBIA
 - für integrative Simulationen gekoppelter Simulationsmodelle,
 - zur Analyse von transdisziplinären Effekten wechselseitig abhängiger Prozesse,
 - zur Entscheidungsunterstützung auf der Grundlage wasserbezogener Szenarien.

Grobarchitektur von DANUBIA



9.2 Simulationsmodelle und integrative Simulationen

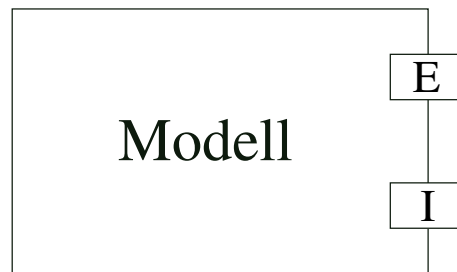
Simulationszeit

- Ein Simulationsmodell simuliert einen physikalischen oder sozialen Prozess über eine bestimmte Zeitspanne (*Modellzeit*).
- Die Simulationszeit ist endlich, d.h. es gibt einen Anfangs- und einen Endzeitpunkt einer Simulation.
- Die Simulationszeit wird durch eine diskrete Menge von Zeitpunkten repräsentiert, zu denen bestimmte durch das Simulationsmodell berechnete Daten zur Verfügung gestellt werden (z.B. stündliche Temperaturwerte, Wasserverbrauch pro Monat).
- Ein Simulationsmodell hat einen individuellen *Zeitschritt*, der den Abstand zweier aufeinanderfolgender Simulationszeitpunkte bestimmt.
- Wir nehmen an, dass der Zeitschritt eines Modells während der gesamten Simulation gleich bleibt.
- Simulationszeitpunkte werden durch natürliche Zahlen repräsentiert.

Export- und Importports

Ein Simulationsmodell

- stellt über Exportports Daten (für andere Modelle) zur Verfügung,
- holt über Importports Daten (von anderen Modellen), die es für die eigenen Berechnungen benötigt.



Lebenszyklus eines Simulationsmodells

- Nachdem ein Modell gestartet wurde, stellt es zunächst bestimmte Anfangswerte über seine Exportports zur Verfügung.
- Bis zum Simulationsende führt es dann zyklisch, entsprechend des Modellzeitschritts, folgende Aktivitäten durch:
 1. Holen der (für die eigenen Berechnungen) benötigten Daten über die Importports.
 2. Berechnung neuer Simulationsdaten, die zum nächsten Simulationszeitpunkt gültig sind.
 3. Bereitstellen der neu berechneten Werte über die Exportports.

Modellierung von Simulationsmodellen

```
//Simulationszeit
const SimStart = 0
const SimEnd = 6
range Time = SimStart..SimEnd

//Berechnungszyklus
MODEL(Step=1) = (start → INIT),
INIT = (prov[SimStart] → M[SimStart]),
M[t:Time] = if (t+Step <= SimEnd)
    then (get[t] → compute[t] → prov[t+Step] → M[t+Step])
    else STOP.
```

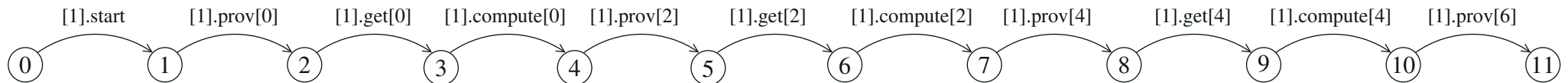
Beachte:

- prov[x] repräsentiert das Bereitstellen von Export-Daten, die zum Zeitpunkt x gültig sind.
- get[x] repräsentiert das Holen von Import-Daten, die zum Zeitpunkt x gültig sind.
- compute[x] repräsentiert die Berechnung von neuen Daten, auf der Grundlage der zum Zeitpunkt x gültigen Import-Daten.

Modellierung von Instanzen von Simulationsmodellen

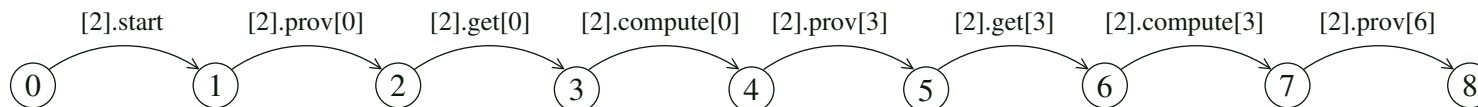
Simulationsmodell mit Nummer 1 und Zeitschritt 2:

[1]:MODEL(2)

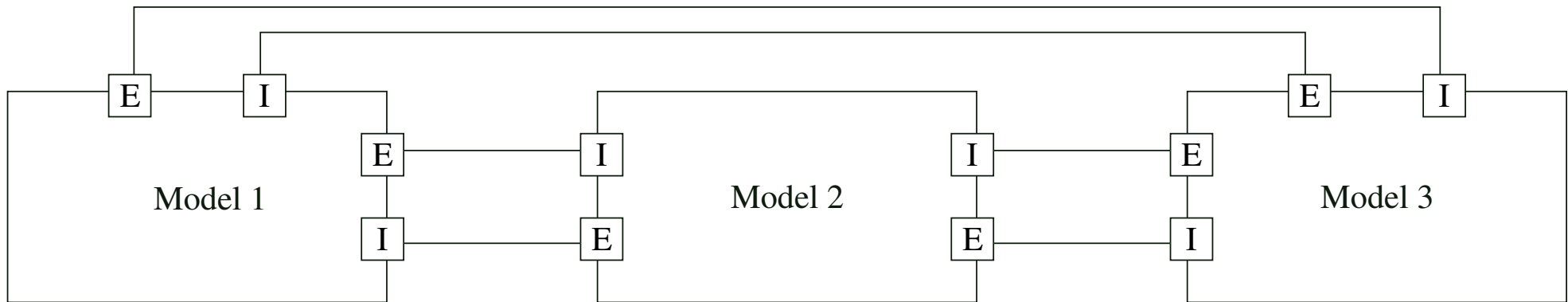


Simulationsmodell mit Nummer 2 und Zeitschritt 3:

[2]:MODEL(3)



Integrative Simulationen



- In einer integrativen Simulation arbeiten verschiedene Simulationsmodelle zusammen, indem sie (zur Laufzeit) gegenseitig Daten austauschen.
- Bei einer integrativen Simulation haben alle Modelle denselben Simulationszeitraum, jedoch (i.a.) verschiedene lokale Zeitschritte.

Parallele Komposition von zwei Simulationsmodellen

```
const nrModels = 2    range Models = 1..nrModels
||SYS = ([1]:MODEL(2) || [2]:MODEL(3)) / {start/[Models].start}.
```

Mögliche Abläufe

(a) *Fehlende Daten:*

start → [1].prov[0] → [1].get[0] →

Das Modell mit Nummer 1 holt Daten, während Modell 2 noch keine Daten bereitgestellt hat.

(b) *Zu alte Daten:*

start → [2].prov[0] → [1].prov[0] → [1].get[0] → [1].compute[0] → [1].prov[2] →
[1].get[2] → [1].compute[2] → [1].prov[4] → [1].get[4] →

Modell 1 holt Daten, während Modell 2 noch nicht die aktuellsten Daten bereitgestellt hat.

(c) *Überschriebene Daten:*

start \rightarrow [2].prov[0] \rightarrow [1].prov[0] \rightarrow [2].get[0] \rightarrow [2].compute[0] \rightarrow [2].prov[3] \rightarrow
[1].get[0] \rightarrow

Modell 2 liefert Daten, während Modell 1 die zum gewünschten Zeitpunkt gültigen Daten noch nicht geholt hat.

9.3 Formalisierung des Koordinatenproblems

Idee

- Betrachte ein Simulationsmodell unter verschiedenen Rollen, als Benutzer (“User“) oder als Lieferant (“Provider“) von Daten.
- Betrachte die Anforderungen paarweise für einen Benutzer und einen Lieferanten.

Anforderungen zur Gültigkeit von Daten

(1) *Für den Benutzer (U):*

U holt nur dann Daten, die zu einem Zeitpunkt t_U gültig sein sollen, wenn gilt:
Die nächsten Daten, die P liefert, sind gültig zu einem Zeitpunkt t_P mit $t_U < t_P$.

(2) *Für den Lieferanten (P):*

P liefert nur dann Daten, die zu einem Zeitpunkt t_P gültig sind, wenn gilt:
Die nächsten Daten, die U holt, sollen gültig sein zu einem Zeitpunkt t_U mit $t_U \geq t_P$.

Legale Abläufe

Ein Ablauf w einer aus beliebig vielen Simulationsmodellen

$[1]:\text{MODEL}(\text{Step}_1), \dots, [n]:\text{MODEL}(\text{Step}_n)$ zusammengesetzten verteilten Simulation ist *legal* bzgl. eines Benutzers U und eines Lieferanten P , wenn w die o.g. Eigenschaften (1) und (2) für U und P erfüllt.

Modellierung der legalen Abläufe durch Property-Prozesse

Ohne Berücksichtigung des Simulationsendes:

```
property VALIDDATA(User=1,StepUser=1,Prov=1,StepProv=1) =  
  VD[SimStart][SimStart],  
  
VD[nextGet:Time][nextProv:Time] =  
  (when (nextGet<nextProv)  
    [User].get[nextGet] -> VD[nextGet+StepUser][nextProv]  
  |when (nextGet>=nextProv)  
    [Prov].prov[nextProv] -> VD[nextGet][nextProv+StepProv]).
```

Mit Simulationseende:

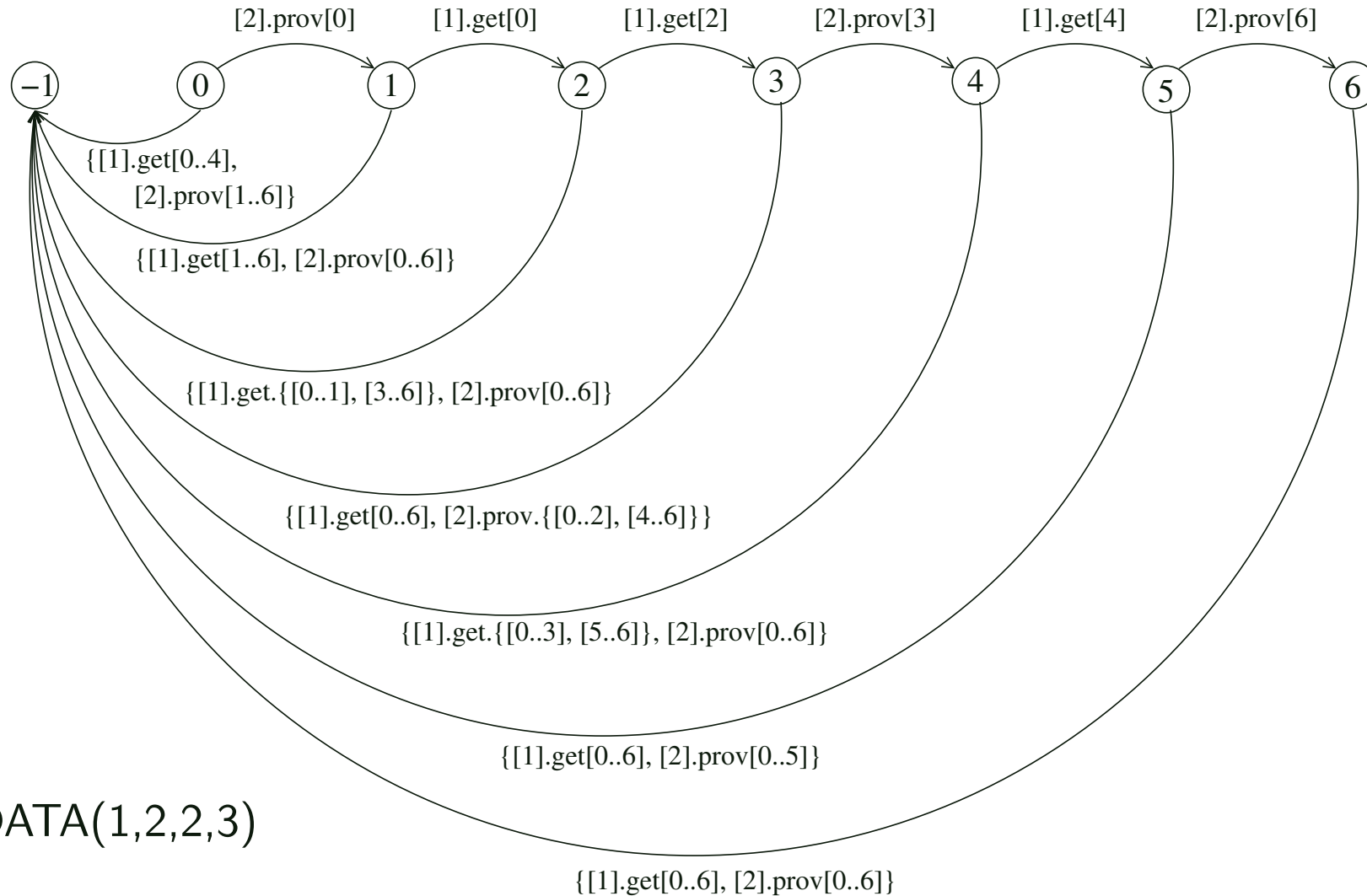
```

property VALIDDATA(User=1,StepUser=1,Prov=1,StepProv=1) =
  VD[SimStart][SimStart],

VD[nextGet:Time][nextProv:Time] =
  (when (nextGet<nextProv)
    [User].get[nextGet] ->
      if (nextGet+2*StepUser<=SimEnd)
        then VD[nextGet+StepUser][nextProv] else PROVFINISH[nextProv]
  |when (nextGet>=nextProv)
    [Prov].prov[nextProv] ->
      if (nextProv+StepProv<=SimEnd)
        then VD[nextGet][nextProv+StepProv] else USERFINISH[nextGet]),
PROVFINISH[nextProv:Time] =
  ([Prov].prov[nextProv] -> if (nextProv+StepProv<=SimEnd)
    then PROVFINISH[nextProv+StepProv]),
USERFINISH[nextGet:Time] =
  ([User].get[nextGet] -> if (nextGet+2*StepUser<=SimEnd)
    then USERFINISH[nextGet+StepUser]).

```

Sicherheitseigenschaft für den Fall User = Modell 1 mit Zeitschritt 2 und
 Provider = Modell 2 mit Zeitschritt 3



Bemerkungen

- Eine Sicherheitseigenschaft für dasselbe Modell als Benutzer und als Lieferant ist nicht notwendig, da in diesem Fall das gewünschte Verhalten schon durch den Lebenszyklus des Modells garantiert wird.
- Bei einer verteilten Simulation mit n Modellen sind $n*(n-1)$ Sicherheitseigenschaften zu berücksichtigen.
- Wichtige Testfälle sind die Sicherheitseigenschaften für Benutzer U und Lieferant P mit folgenden Zeitschritten:
 - $\text{Step}_U = \text{Step}_P$
 - $\text{Step}_U < \text{Step}_P$ und Step_P ist ein ganzzahliges Vielfaches von Step_U
 - wie oben, aber Step_P ist kein ganzzahliges Vielfaches von Step_U
 - $\text{Step}_U > \text{Step}_P$ und Step_U ist ein ganzzahliges Vielfaches von Step_P
 - wie oben, aber Step_U ist kein ganzzahliges Vielfaches von Step_P

Anforderungen für den Datenzugriff

- Ein Modell darf nur dann Daten holen, wenn kein anderes Modell zur selben Zeit Daten liefert.
- Umgekehrt darf ein Modell nur dann Daten liefern, wenn kein anderes Modell zur selben Zeit Daten holt.

Modellierung der kritischen Bereiche

set GetProvs = {{get,prov}[Time]}

set EnterExits = {{enterGet,exitGet,enterProv,exitProv}[Time]}

set Labels = {GetProvs,EnterExits}

MODEL(Step=1) = (start → INIT),

INIT = (enterProv[SimStart] → prov[SimStart] → exitProv[SimStart] → M[SimStart]),

M[t:Time] = if (t+Step ≤ SimEnd)

then (enterGet[t] → get[t] → exitGet[t] →

compute[t] →

enterProv[t+Step] → prov[t+Step] → exitProv[t+Step] → M[t+Step])

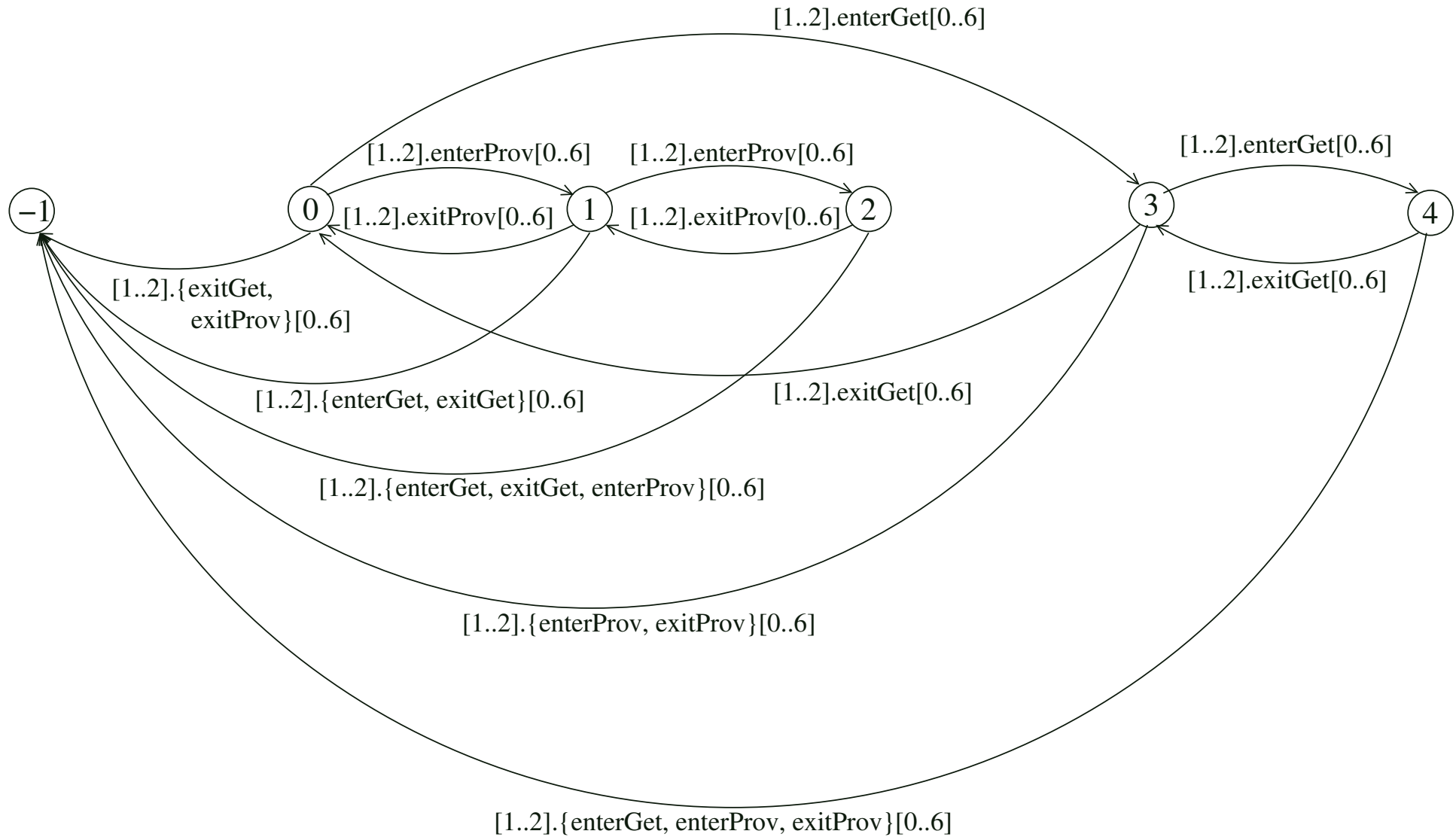
else STOP + Labels.

Modellierung des gegenseitigen Ausschlusses durch einen Property-Prozess

property EXCLUSION = ([Models].enterGet[Time] → GET[1]
| [Models].enterProv[Time] → PROV[1]),

GET[i:Models] = ([Models].enterGet[Time] → GET[i+1]
| when (i==1) [Models].exitGet[Time] → EXCLUSION
| when (i>1) [Models].exitGet[Time] → GET[i-1]),

PROV[i:Models] = ([Models].enterProv[Time] → PROV[i+1]
| when (i==1) [Models].exitProv[Time] → EXCLUSION
| when (i>1) [Models].exitProv[Time] → PROV[i-1]).



Lebendigkeitseigenschaften

Intuitiv

Jedes Modell liefert in einer verteilten Simulation in jedem Modellzeitschritt Daten.

Genauer

Für jeden Systemablauf w , für jedes Modell $m \in \text{Models}$ und für jeden Zeitpunkt $t \in \text{Time}$ mit $t \% \text{Step}_m = 0$ ist

$$[m].\text{prov}[t] \in w$$

9.4 Systementwurf

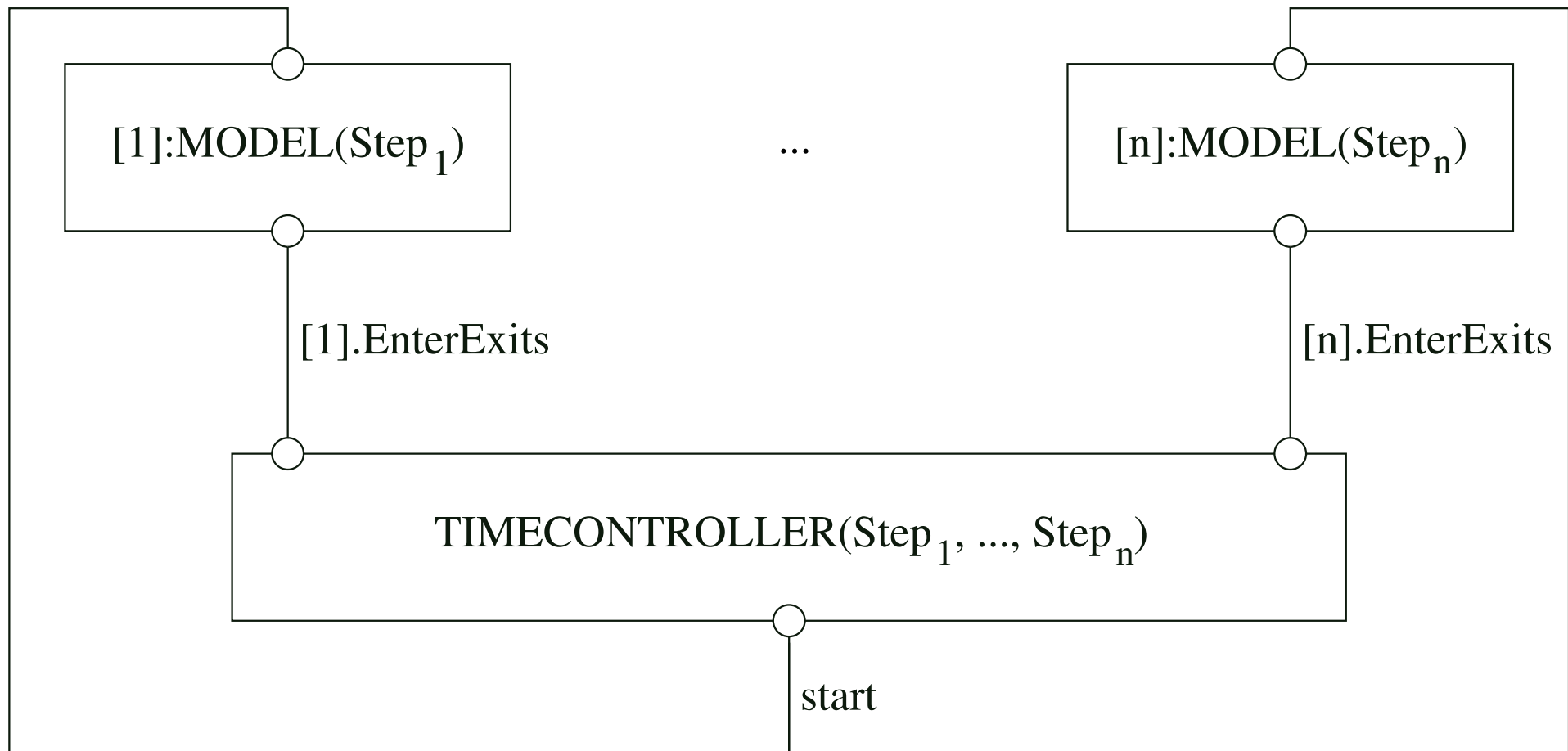
Idee

Verwende einen globalen Timecontroller zur Koordination aller an einer verteilten Simulation teilnehmenden Simulationsmodelle.

Ziel

Konstruiere einen FSP-Prozess TIMECONTROLLER, so dass für jeweils n Modelle das folgende System alle Sicherheits- und Lebendigkeitseigenschaften erfüllt:

$$\begin{aligned} \parallel \text{SYS} = & ([1]:\text{MODEL}(\text{Step}_1) \parallel \dots \parallel [n]:\text{MODEL}(\text{Step}_n)) \\ & \parallel \text{TIMECONTROLLER}(\text{Step}_1, \dots, \text{Step}_n)) \\ & / \{ \text{start} / [\text{Models}].\text{start} \} \end{aligned}$$



Konstruktion des Timecontroller-Modells

Grundlegende Ideen

- Der Timecontroller ist ein Monitor, der die Operationen enterGet, exitGet, enterProv und exitProv anbietet.
- Die Operationen werden durch die Mengen von Aktionen [Models].enterGet[Time], etc., modelliert. Eine einzelne Aktion hat die Form [m].enterGet[t], etc., wobei $m \in \text{Models}$ und $t \in \text{Time}$ als Parameter verstanden werden.
- enter-Aktionen werden bewacht durch eine Bedingung, die die Anforderungen bzgl. der Gültigkeit von Daten und des gegenseitigen Ausschlusses von get- und prov-Aktionen gewährleisten soll.
- Die Bedingungen sind abhängig vom Monitorzustand, der durch Indexvariablen lokaler Prozesse modelliert wird.

- Der Monitor merkt sich
 - für jedes an einer verteilten Simulation teilnehmende Modell $m \in \text{Models}$ den Zeitpunkt, zu dem das Modell zum nächsten Mal Daten holt (`nextGetm`) und zu dem es zum nächsten Mal Daten liefert (`nextProvm`)
 - die Anzahl der Modelle, die gerade Daten holen (`nrGet`) sowie die Anzahl der Modelle, die gerade Daten liefern (`nrProv`).
- Die Zeitschritte der an einer verteilten Simulation teilnehmenden Modelle werden dem Timecontroller über Prozessparameter bekannt gegeben, z.B. Timecontroller für zwei Modelle mit den Zeitschritten 2 und 3: `TIMECONTROLLER(2,3)`.

Modell des Timecontrollers

```
TIMECONTROLLER(Step1=1,Step2=1) =
  (start -> TC[SimStart][SimStart][SimStart][SimStart]),

TC[nextGet1:Time][nextProv1:Time][nextGet2:Time][nextProv2:Time] =
  (dummy[t:Time] ->
    //enterGet
    (when (t<nextProv1 && t<nextProv2)
      [Models].enterGet[t] ->
        TC[nextGet1][nextProv1][nextGet2][nextProv2]
    //exitGet
    | [1].exitGet[t] -> TC[t+Step1][nextProv1][nextGet2][nextProv2]
    | [2].exitGet[t] -> TC[nextGet1][nextProv1][t+Step2][nextProv2])
```

```
//enterProv
|when (nextGet1>=t && nextGet2>=t)
    [Models].enterProv[t] ->
        TC[nextGet1] [nextProv1] [nextGet2] [nextProv2]
//exitProv
|[1].exitProv[t] ->
    if (t+Step1<=SimEnd)
    then TC[nextGet1] [t+Step1] [nextGet2] [nextProv2]
    else TC[SimStart] [SimStart] [SimStart] [SimStart]
|[2].exitProv[t] ->
    if (t+Step2<=SimEnd)
    then TC[nextGet1] [nextProv1] [nextGet2] [t+Step2]
    else TC[SimStart] [SimStart] [SimStart] [SimStart]
|dummy[t] -> TC[nextGet1] [nextProv1] [nextGet2] [nextProv2])
)\{dummy[Time]}.
```

Frage: Ist nrGet und nrProv wirklich nötig?

Modell einer verteilten Simulation

```
const StepModel1 = 2
```

```
const StepModel2 = 3
```

```
||SYS = ([1]:MODEL(StepModel1) || [2]:MODEL(StepModel2)  
        || TIMECONTROLLER(StepModel1,StepModel2)) / {start/[Models].start}.
```

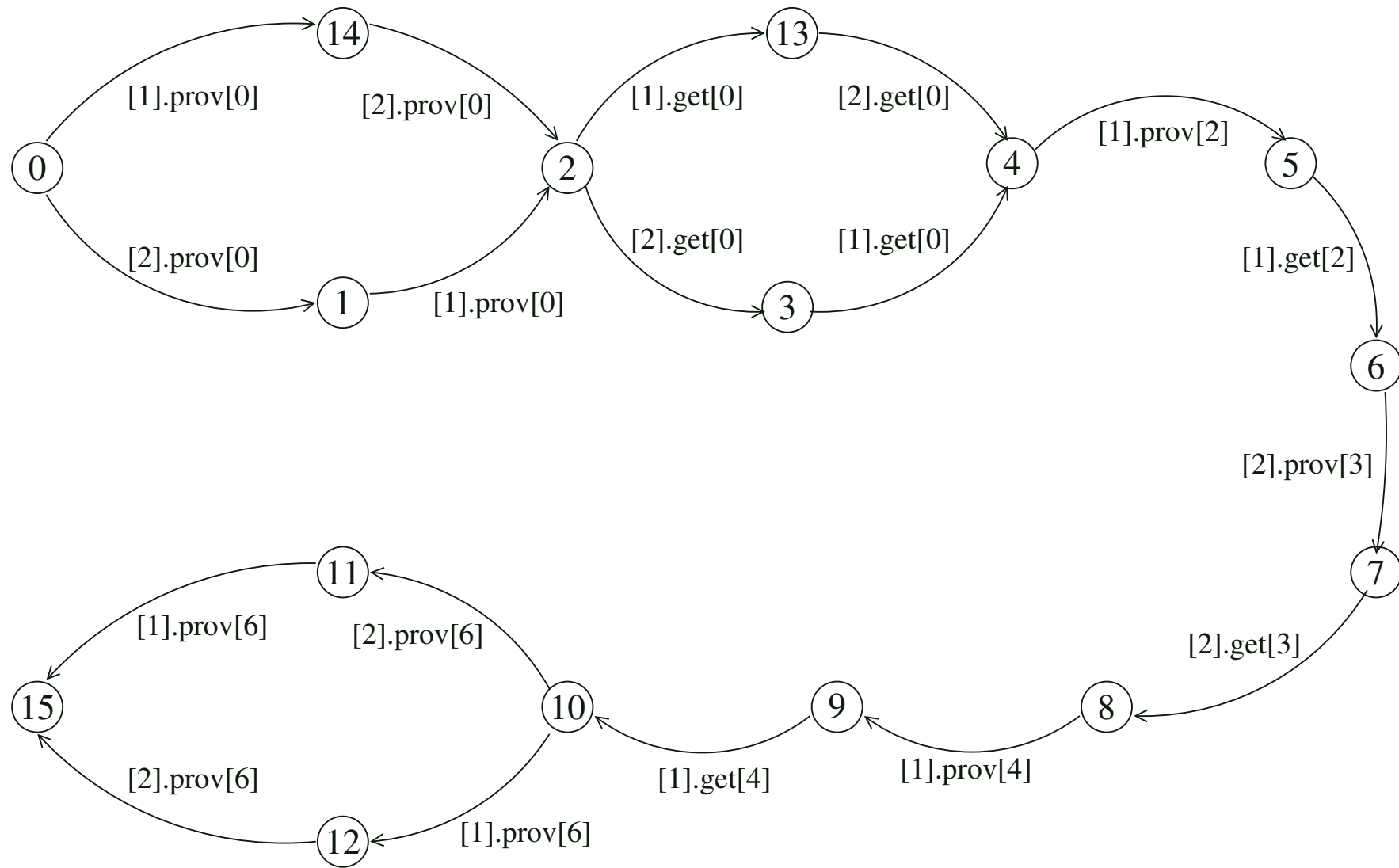
Sichten auf die verteilte Simulation

1. *Sicht auf das Holen und Liefern von Daten*

```
set GetProvs = {{get,prov}[Time]}
```

```
||VIEW_GetProvs = SYS@{[Models].GetProvs}.
```

LTS nach Minimalisierung bzgl. beobachtbarer Äquivalenz:

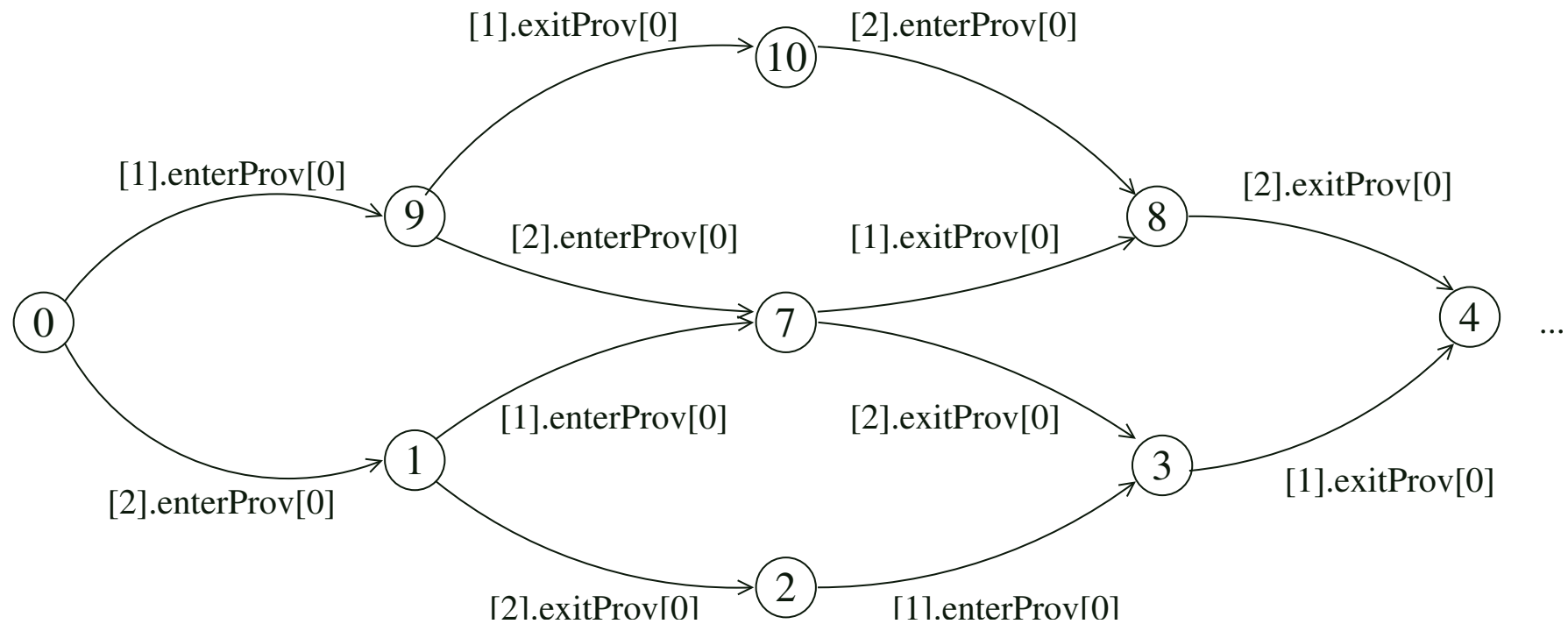


2. Sicht auf den gegenseitigen Ausschluss

set EnterExits = $\{\{\text{enterGet}, \text{exitGet}, \text{enterProv}, \text{exitProv}\}[\text{Time}]\}$

$\|\text{VIEW_EnterExits} = \text{SYS}@ \{\{\text{Models}\}.\text{EnterExits}\}.$

LTS nach Minimalisierung bzgl. beobachtbarer Äquivalenz (Anfangsteil):



Nachweis der Sicherheitseigenschaften

$\| \text{CHECK_VALIDDATA_USER1_PROV2} =$
 $(\text{SYS} \| \text{VALIDDATA}(1, \text{StepModel1}, 2, \text{StepModel2}))$.

Fehlerzustand nicht erreichbar.

$\| \text{CHECK_VALIDDATA_USER2_PROV1} =$
 $(\text{SYS} \| \text{VALIDDATA}(2, \text{StepModel2}, 1, \text{StepModel1}))$.

Fehlerzustand nicht erreichbar.

$\| \text{CHECK_EXCLUSION} = (\text{SYS} \| \text{EXCLUSION})$.

Fehlerzustand nicht erreichbar.

Nachweis der Lebendigkeitseigenschaften

Gewünscht:

Für alle Abläufe w , Modelle $m \in \text{Models}$ und Zeitpunkte $t \in \text{Time}$ mit $t \% \text{Step}_m = 0$ ist $[m].\text{prov}[t] \in w$.

Fortschrittseigenschaften in FSP

Für alle $m \in \text{Models}$ und $t \in \text{Time}$ mit $t \% \text{Step}_m = 0$ definiere
progress $\text{PROV_Model}_m_t = \{ [m].\text{prov}[t] \}$

Probleme

1. Es sind viele Fortschrittseigenschaften zu definieren.
2. Keine der Fortschrittseigenschaften ist erfüllt, wegen der Endlichkeit einer Simulation.

Beachte:

Die Anforderungen sind schwächer als die oben formulierten Fortschrittseigenschaften.

Lösungsmöglichkeit für Problem 1:

Verwende indizierte Fortschrittseigenschaften!

Für jedes Modell $m \in \text{Models}$ definiere

$$\begin{aligned} \text{progress PROV_Modelm}[i:0..(\text{SimEnd}-\text{SimStart})/\text{StepModelm}] \\ = \{ [m].\text{prov}[\text{SimStart}+i*\text{StepModelm}] \} \end{aligned}$$

Lösungsmöglichkeit für Problem 2:

Führe “künstliche“ Zyklen ein bei den Simulationsmodellen und beim Timecontroller und überprüfe dann die obigen Fortschrittseigenschaften.

Simulationsmodelle mit Zyklen

MODELProgress(Step=1) = (start \rightarrow INIT),

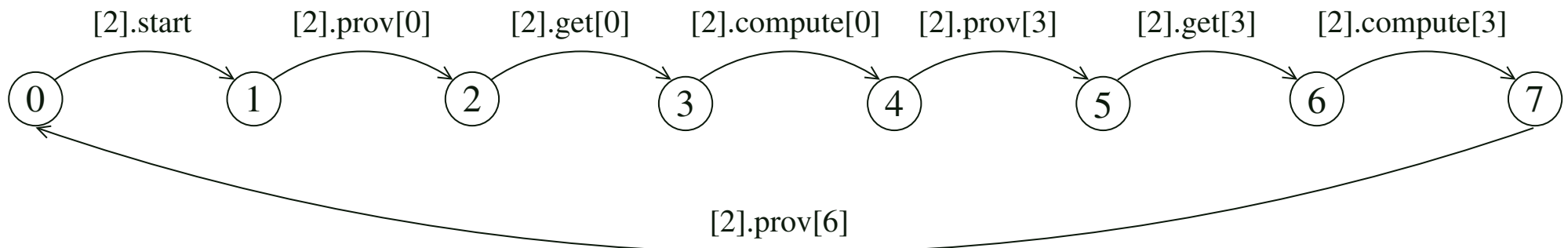
INIT = (enterProv[SimStart] \rightarrow prov[SimStart] \rightarrow exitProv[SimStart] \rightarrow M[SimStart]),

M[t:Time] = if (t+Step \leq SimEnd)

then (enterGet[t] \rightarrow get[t] \rightarrow exitGet[t] \rightarrow
 compute[t] \rightarrow

enterProv[t+Step] \rightarrow prov[t+Step] \rightarrow exitProv[t+Step] \rightarrow M[t+Step])

else (start \rightarrow INIT) + Labels.



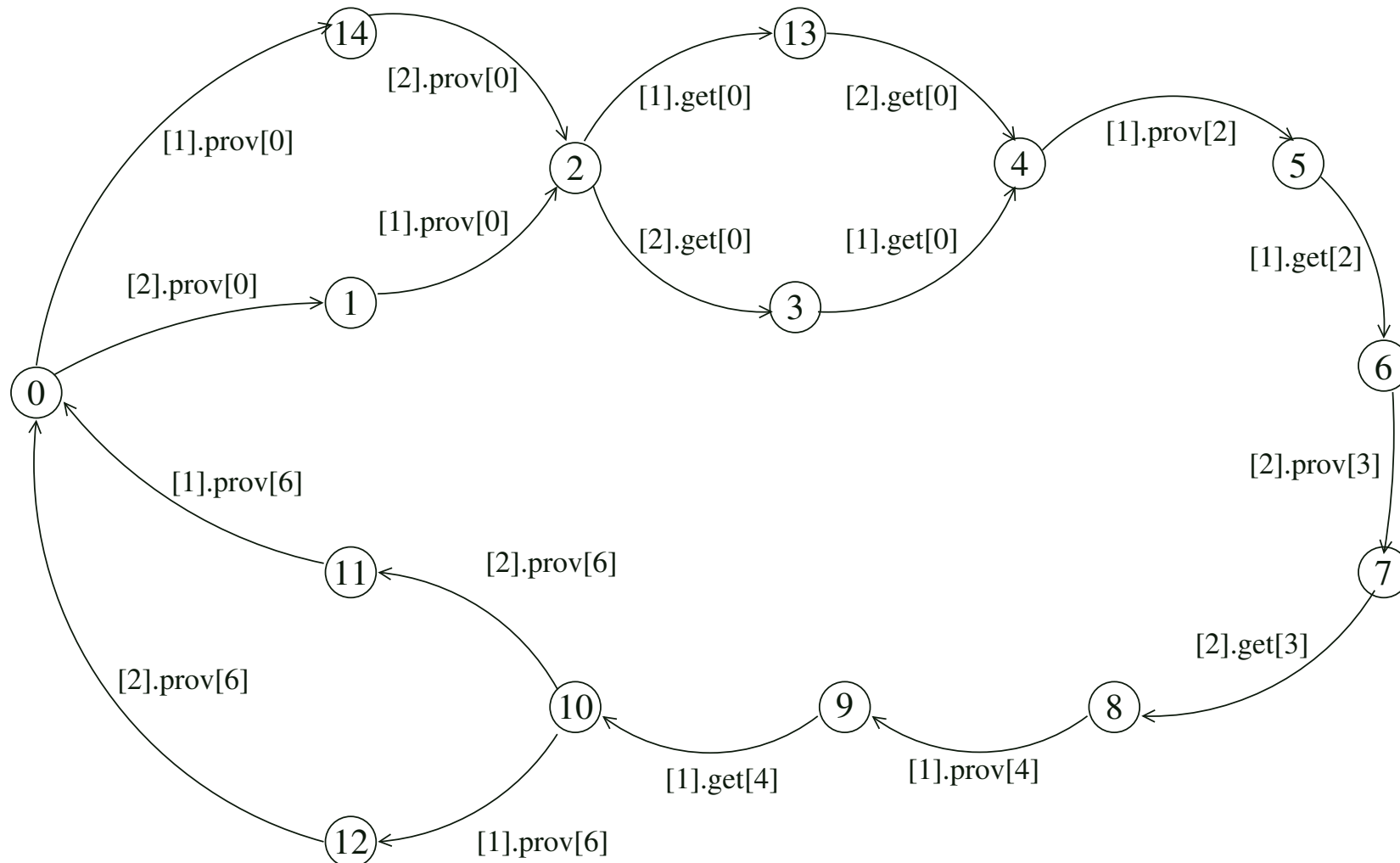
Timecontroller mit Zyklus

```

TIMECONTROLLERProgress(Step1=1,Step2=1) =
    (start → TC[SimStart][SimStart][SimStart][SimStart][0][0]),

TC[nextGet1:Time][nextProv1:Time][nextGet2:Time][nextProv2:Time] =
    (dummy[t:Time] →
        . . .
        //für Progress-Analyse
        | start → TC[SimStart][SimStart][SimStart][SimStart]
        )\{dummy[Time]}.
```

Sicht auf das Gesamtsystem mit Zyklus

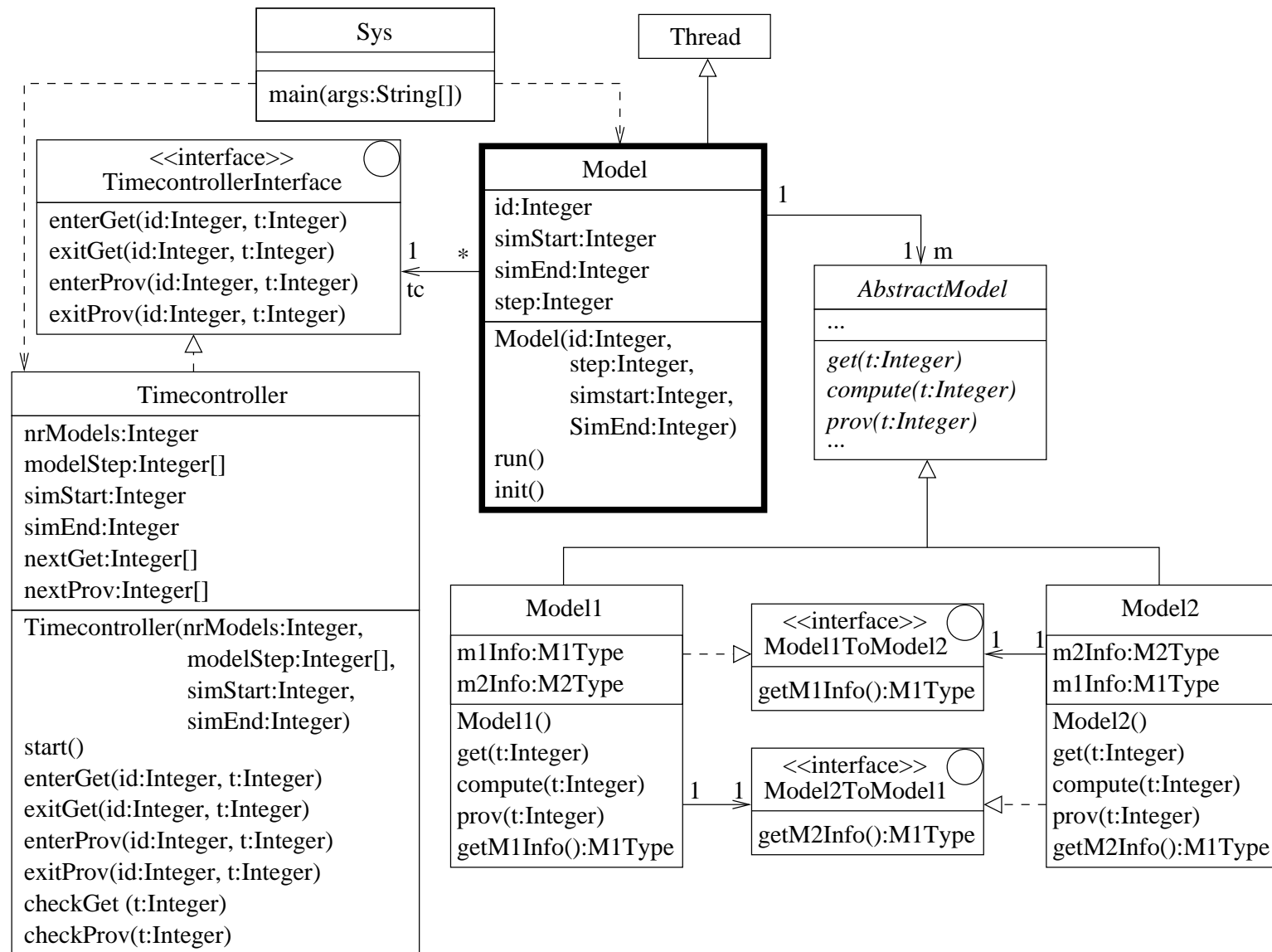


Das Gesamtsystem erfüllt alle Fortschrittseigenschaften!

9.5 Systemimplementierung

- Simulationsmodelle werden durch Threads implementiert (aktive Objekte).
- Der Timecontroller wird durch einen Monitor implementiert (passives Objekt), der beliebig viele Simulationsmodelle koordinieren kann.
- Für den Zugriff der Modelle auf den Timecontroller wird ein Interface eingeführt, das die enter- und exit-Operationen anbietet.
- Für die Ausführung der get, compute und prov Operationen wird eine abstrakte Klasse eingeführt (*AbstractModel*), das entsprechend konkreter, fachspezifischer Simulationen implementiert wird.
- Für den Datenaustausch zwischen fachspezifischen Simulationen werden entsprechende Interfaces verwendet.

Statische Struktur der Implementierung



Java-Code

1. *main-Methode der Systemklasse*

```
public static void main(String[] args) {
    int simStart = 0;
    int simEnd = 6;
    int nrModels = 2;
    int[] modelStep = new int[] { 2, 3 };
    Timecontroller tc =
        new Timecontroller(nrModels, modelStep, simStart, simEnd);
    tc.start();
    new Model(1, 2, simStart, simEnd, tc,
        new Model1()).start();
    new Model(2, 3, simStart, simEnd, tc,
        new Model2()).start();
}
```

2. *Methoden der Klasse Model*

```
private void init() {
    try {
        tc.enterProv(id, 0);
    } catch (InterruptedException {}) {}
    prov(0);
    tc.exitProv(id, 0);
}

public void run() {
    init();
    int t=0;
    while (t+step<=simEnd) {
        try {
            tc.enterGet(id, t);
        } catch (InterruptedException e) {}
        m.get(t);
        tc.exitGet(id, t);
    }
}
```

```
m.compute(t);
try {
    tc.enterProv(id, t+step);
} catch (InterruptedException e) {}
m.prov(t+step);
tc.exitProv(id, t+step);
t = t+step; }
}
```

3. *Methoden der Klasse Timecontroller*

```
public void start() {
    for (int i=0; i<nrModels; i++) {
        nextGet[i]=simStart;
        nextProv[i]=simStart; }
}
```

```
public synchronized void enterGet(int id, int t)
    throws InterruptedException {
    while (!checkProv(t)) wait();
}

private boolean checkProv(int t) {
    boolean b = true;
    for (int i = 0; i < nrModels; i++) {
        b = (b && (t < nextProv[i]));
    }
    return b;
}

public synchronized void exitGet(int id, int t) {
    // subtract 1 to match model id!
    nextGet[id-1] = nextGet[id-1] + modelStep[id-1];
    notifyAll();
}
```

```
public synchronized void enterProv(int id, int t)
    throws InterruptedException {
    while (!checkGet(t)) wait();
}

private boolean checkGet(int t) {
    boolean b = true;
    for (int i = 0; i < nrModels; i++) {
        b = (b && (nextGet[i] >= t));
    }
    return b;
}

public synchronized void exitProv(int id, int t) {
    nextProv[id-1] = nextProv[id-1] + modelStep[id-1];
    notifyAll();
}
```