

# Bounded Analysis and Decomposition for Behavioural Description of Components

Jean-Claude Royer

Ecole des Mines de Nantes, OBASCO INRIA, LINA

Jean-Claude.Royer@emn.fr

Collaboration with Pascal Poizat and Gwen Salaün

FMOODS 2006, Bologna, Italy

I am JC Royer from EMN

I will present an approach to analyse software components with protocols and data values.

This work has been done conjointly with PP and GS.

- ▶ Software component architectures

- ▶ Software component architectures
- ▶ Specifications and verifications

- ▶ Software component architectures
- ▶ Specifications and verifications
- ▶ Introducing complex protocols for expressiveness and readability

- ▶ Software component architectures
- ▶ Specifications and verifications
- ▶ Introducing complex protocols for expressiveness and readability
- ▶ Resource and service availability properties

- ▶ Software component architectures
- ▶ Specifications and verifications
- ▶ Introducing complex protocols for expressiveness and readability
- ▶ Resource and service availability properties
- ▶ Boundedness of dynamic systems with data

- ▶ Software component architectures
- ▶ Specifications and verifications
- ▶ Introducing complex protocols for expressiveness and readability
- ▶ Resource and service availability properties
- ▶ Boundedness of dynamic systems with data
- ▶ Reusing classic model-checking

 Context

- Software component architectures
- Specifications and verifications
- Introducing complex protocols for expressiveness and readability
- Resource and service availability properties
- Boundedness of dynamic systems with data
- Raising classic model-checking

Our context is software component architectures.

Here we focus on the specification and verification side of our component model but we are also implementing it in Java.

One major feature of our model is the use of complex protocol descriptions.

One goal of this model is to express and to verify resource and service availability properties.

One example is for instance : If I request a service How long I am waiting the response ?

We show here that the notion of protocol boundedness is an important and useful criterion for this.

We also expect to use classic model-checking to check properties.

- ▶ Related work

- ▶ Related work
- ▶ Symbolic Transition System (STS)

- ▶ Related work
- ▶ Symbolic Transition System (STS)
- ▶ Configuration graph and interpretations

- ▶ Related work
- ▶ Symbolic Transition System (STS)
- ▶ Configuration graph and interpretations
- ▶ Boundedness of counter systems

- ▶ Related work
- ▶ Symbolic Transition System (STS)
- ▶ Configuration graph and interpretations
- ▶ Boundedness of counter systems
- ▶ Decomposition

- ▶ Related work
- ▶ Symbolic Transition System (STS)
- ▶ Configuration graph and interpretations
- ▶ Boundedness of counter systems
- ▶ Decomposition
- ▶ Examples :

- ▶ Related work
- ▶ Symbolic Transition System (STS)
- ▶ Configuration graph and interpretations
- ▶ Boundedness of counter systems
- ▶ Decomposition
- ▶ Examples :
  - ▶ The ticket protocol example

- ▶ Related work
- ▶ Symbolic Transition System (STS)
- ▶ Configuration graph and interpretations
- ▶ Boundedness of counter systems
- ▶ Decomposition
- ▶ Examples :
  - ▶ The ticket protocol example
  - ▶ The resource allocator example

- ▶ Related work
- ▶ Symbolic Transition System (STS)
- ▶ Configuration graph and interpretations
- ▶ Boundedness of counter systems
- ▶ Decomposition
- ▶ Examples :
  - ▶ The ticket protocol example
  - ▶ The resource allocator example
- ▶ Conclusion and future work

 Plan

- Related work
- Symbolic Transition System (STS)
- Configuration graph and interpretations
- Boundedness of counter systems
- Decomposition
- Examples :
  - The ticket protocol example
  - The resource allocator example
- Conclusion and future work

The outline of my talk is the following ...

- ▶ To complement model-checking

- ▶ To complement model-checking
- ▶ Boundedness of generalized Petri nets (Finkel, Schnoebelen, ...)

- ▶ To complement model-checking
- ▶ Boundedness of generalized Petri nets (Finkel, Schnoebelen, ...)
- ▶ Bounded decomposition is an abstraction method (Bensalem, Clarke, Dams, ...)

- ▶ To complement model-checking
- ▶ Boundedness of generalized Petri nets (Finkel, Schnoebelen, ...)
- ▶ Bounded decomposition is an abstraction method (Bensalem, Clarke, Dams, ...)
- ▶ Acceleration technique (Finkel and al.)

- ▶ To complement model-checking
- ▶ Boundedness of generalized Petri nets (Finkel, Schnoebelen, ...)
- ▶ Bounded decomposition is an abstraction method (Bensalem, Clarke, Dams, ...)
- ▶ Acceleration technique (Finkel and al.)
- ▶ Theorem prover and model-checker (Rushby, ...)

- ▶ To complement model-checking
- ▶ Boundedness of generalized Petri nets (Finkel, Schnoebelen, ...)
- ▶ Bounded decomposition is an abstraction method (Bensalem, Clarke, Dams, ...)
- ▶ Acceleration technique (Finkel and al.)
- ▶ Theorem prover and model-checker (Rushby, ...)
- ▶ Constraint programming (Delzanno and Podelsky)

## └ Related Work

- ▶ To complement model-checking
- ▶ Boundedness of generalized Petri nets (Finkel, Schnoebelen, ...)
- ▶ Bounded decomposition is an abstraction method (Bensalem, Clarke, Dams, ...)
- ▶ Acceleration technique (Finkel and al.)
- ▶ Theorem prover and model-checker (Rushby, ...)
- ▶ Constraint programming (Delzanno and Podolsky)

One goal of this work was to complement classic model-checking to offer a simple and intuitive way to abstract dynamic system with data computation. We use a boundedness procedure which is close to the one defined for generalized Petri nets by Finkel and Schnoebelen. Our approach may be viewed as a specific abstraction technique mixing Cartesian projection and lifting values.

Other abstraction techniques are possible in our context.

This is also true with optimisation techniques like on-the-fly model-checking.

But we did not yet consider these features.

There are other approaches for infinite state systems, the acceleration technique seems a really promising one.

Theorem provers are also possible, however since they are not automatic coupling them with model-checkers is a successful technique experimented with PVS for example.

One may also mention the use of constraint logic programming which

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values
- ▶ STS : a finite state and transition formalism

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values
- ▶ STS : a finite state and transition formalism
- ▶ STS rather than automata, LTS or Petri net

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values
- ▶ STS : a finite state and transition formalism
- ▶ STS rather than automata, LTS or Petri net
- ▶ Process algebra with values (LOTOS)  
models : STG, I/O-STG, STS ...

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values
- ▶ STS : a finite state and transition formalism
- ▶ STS rather than automata, LTS or Petri net
- ▶ Process algebra with values (LOTOS)  
models : STG, I/O-STG, STS ...
- ▶ A transition label : [guard] event / action

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values
- ▶ STS : a finite state and transition formalism
- ▶ STS rather than automata, LTS or Petri net
- ▶ Process algebra with values (LOTOS)  
models : STG, I/O-STG, STS ...
- ▶ A transition label : [guard] event / action
- ▶ Input ( ? x ) and output ( ! v ) event parameters

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values
- ▶ STS : a finite state and transition formalism
- ▶ STS rather than automata, LTS or Petri net
- ▶ Process algebra with values (LOTOS)  
models : STG, I/O-STG, STS ...
- ▶ A transition label : [guard] event / action
- ▶ Input ( ? x ) and output ( ! v ) event parameters
- ▶ Guards : a condition to trigger the transition

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values
- ▶ STS : a finite state and transition formalism
- ▶ STS rather than automata, LTS or Petri net
- ▶ Process algebra with values (LOTOS)  
models : STG, I/O-STG, STS ...
- ▶ A transition label : [guard] event / action
- ▶ Input ( ? x ) and output ( ! v ) event parameters
- ▶ Guards : a condition to trigger the transition
- ▶ Action notation (imperative style in examples)

# Symbolic Transition System (STS)

- ▶ Component needs complex protocols with data values
- ▶ STS : a finite state and transition formalism
- ▶ STS rather than automata, LTS or Petri net
- ▶ Process algebra with values (LOTOS)  
models : STG, I/O-STG, STS ...
- ▶ A transition label : [guard] event / action
- ▶ Input ( ? x ) and output ( ! v ) event parameters
- ▶ Guards : a condition to trigger the transition
- ▶ Action notation (imperative style in examples)
- ▶ Formal notations are provided

## Symbolic Transition System (STS)

- Component needs complex protocols with data values
- STS : a finite state and transition formalism
- STS rather than automata, LTS or Petri net
- Process algebra with values (LOTOS)
- models : STG, I/O-STG, STS ...
- A transition label : [guard] event / action
- Input (? x) and output (! v) event parameters
- Guards : a condition to trigger the transition
- Action notation (imperative style in examples)
- Formal notations are provided

Components may exchange data with service requests, or may internally compute data values on which behaviours depend.

Therefore, there is a need for component models integrating data types within behaviours.

We choose the notion of STS rather than automata, LTS or Petri Nets for expressiveness and readability reasons. STS comes from the study of Process Algebra with Values for which several variants exist in the literature.

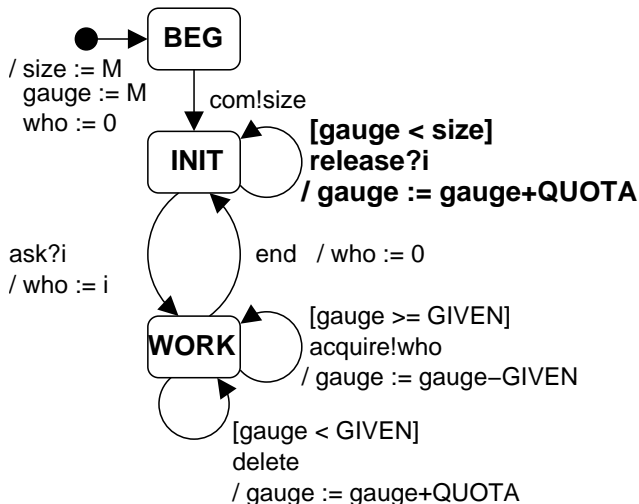
The general syntax of a transition is like that.

We consider that a transition may emit or receive values and a guard is a condition to trigger a transition.

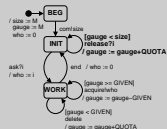
We have also an algebraic notation for actions but here we use a simple imperative style in the examples.

In the paper we provide formal notations, I will focus in my talk on concepts and examples.

# A Resource Allocator



## └ A Resource Allocator



Here is an STS example with three states which represents the dynamic part of a resource allocator.

The initial state is BEG and a transition has an event label with input or output values.

Inputs are prefixed by a question mark and outputs by an exclamation mark!

A guard is noted between square brackets and actions are prefixed by a slash.

Thus a transition like `[ ] release?i / ...` means that `i`: if gauge is lesser than size and if the release event occurs then `i` receives a value and the gauge value is increased of quota resources.

- ▶ Structured synchronous product for component composition

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :
  - ▶ Acceleration

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :
  - ▶ Acceleration
  - ▶ Theorem prover

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :
  - ▶ Acceleration
  - ▶ Theorem prover
  - ▶ Constraint programming

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :
  - ▶ Acceleration
  - ▶ Theorem prover
  - ▶ Constraint programming
- ▶ Model-checking : efficient and automatic

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :
  - ▶ Acceleration
  - ▶ Theorem prover
  - ▶ Constraint programming
- ▶ Model-checking : efficient and automatic
- ▶ Abstraction technique :

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :
  - ▶ Acceleration
  - ▶ Theorem prover
  - ▶ Constraint programming
- ▶ Model-checking : efficient and automatic
- ▶ Abstraction technique :
  - ▶ Component context

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :
  - ▶ Acceleration
  - ▶ Theorem prover
  - ▶ Constraint programming
- ▶ Model-checking : efficient and automatic
- ▶ Abstraction technique :
  - ▶ Component context
  - ▶ Related to boundedness

## └ Verifications Issues

- ▶ Structured synchronous product for component composition
- ▶ Model-checking : state explosion problems
- ▶ Several solutions for infinite state system :
  - Acceleration
  - Theorem prover
  - Constraint programming
- ▶ Model-checking : efficient and automatic
- ▶ Abstraction technique :
  - Component context
  - Related to boundedness

To analyse a composite system we extend the synchronous product of LTS to our STS.

One question is how to verify properties with this formalism.

Classic model-checking is generally not sufficient since we have state explosion problems.

There are techniques for infinite state systems, but they require advanced concepts and often they are not automatic.

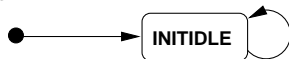
In this first attempt, we choose to target classic model-checking since it is well-known, efficient and automatic.

Thus we propose an abstraction technique which is simple to understand, adequate to our component context and related with the boundedness notion.

# Configuration Graph

Configuration graph : unfolding receipts and data evaluation

/ gauge:=0 ; size:=3 ; total:=3



[gauge<size]

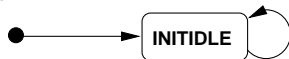
release

/ gauge:= gauge+1 ; total:= total-1

# Configuration Graph

Configuration graph : unfolding receipts and data evaluation

/ gauge:=0 ; size:=3 ; total=3



[gauge<size]

release

/ gauge:= gauge+1 ; total:= total-1

INITIDLE gauge=0 ; size=3 ; total=3



[gauge<size]

release

/ gauge:= gauge+1 ; total:= total-1

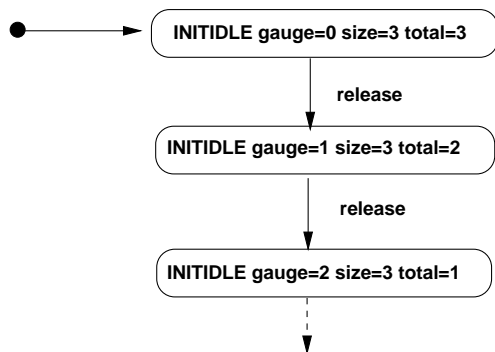
# Configuration Graph

Configuration graph : unfolding receipts and data evaluation



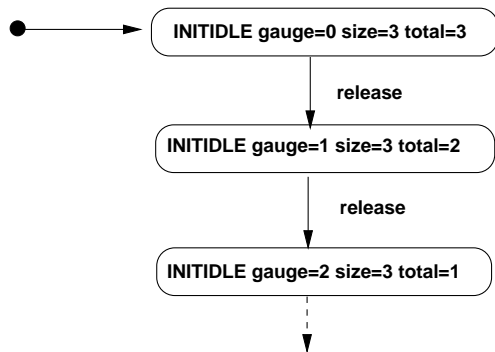
# Configuration Graph

Configuration graph : unfolding receipts and data evaluation



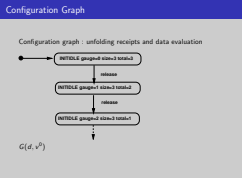
# Configuration Graph

Configuration graph : unfolding receipts and data evaluation



$G(d, v^0)$

## Configuration Graph



The semantics of STS are formalised using configuration graphs. They are obtained applying jointly the unfolding of receipts and the reduction of ground terms to their normal forms.

Let be this simple STS example, with one state and one transition.

Our semantics simulate transition triggering and action effects.

In the example actions and guards are evaluated relatively to the current state and the variables gauge, size and total.

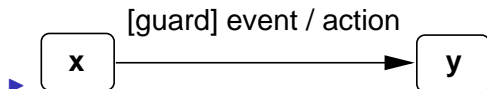
This leads to a new system with control state and data values as state labels.

In the sequel  $G$  denotes the configuration graph of the  $d$  STS with the  $v^0$  initial value.

- ▶ The configuration graph is not necessarily a finite machine (infinite state set or infinite event set)

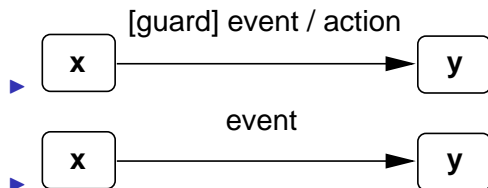
- ▶ The configuration graph is not necessarily a finite machine (infinite state set or infinite event set)
- ▶ Various LTS ( $I_{LTS}$ ) interpretations of STS

- ▶ The configuration graph is not necessarily a finite machine (infinite state set or infinite event set)
- ▶ Various LTS ( $I_{LTS}$ ) interpretations of STS

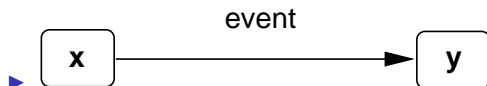
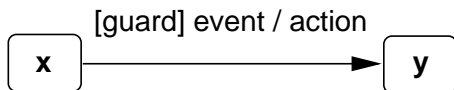


# Interpretation

- ▶ The configuration graph is not necessarily a finite machine (infinite state set or infinite event set)
- ▶ Various LTS ( $I_{LTS}$ ) interpretations of STS

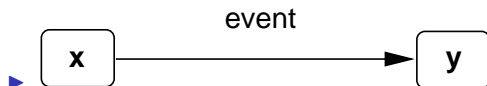
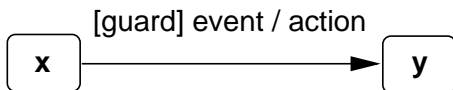


- ▶ The configuration graph is not necessarily a finite machine (infinite state set or infinite event set)
- ▶ Various LTS ( $I_{LTS}$ ) interpretations of STS




- ▶  $I_{LTS}(d) \succeq I_{LTS}(G(d, v^0))$

- ▶ The configuration graph is not necessarily a finite machine (infinite state set or infinite event set)
- ▶ Various LTS ( $I_{LTS}$ ) interpretations of STS



- ▶  $I_{LTS}(d) \succeq I_{LTS}(G(d, v^0))$
- ▶ Decomposition and boundedness

## └ Interpretation

- The configuration graph is not necessarily a finite machine (infinite state set or infinite event set)
  - Various LTS ( $\mathcal{L}_{STS}$ ) interpretations of STS
- 
- ```

graph LR
    x1[x] -- "guard event / action" --> y1[y]
    x2[x] -- "event" --> y2[y]
  
```
- $\mathcal{L}_{STS}(d) \doteq \mathcal{L}_{STS}(G(d, v^d))$
  - Decomposition and boundedness

This computation process may not terminate and may lead to an infinite system.

We may get an infinite set of states, or an infinite set of transitions.

STS can be interpreted as LTS in various ways.

One simple interpretation is to drop guards and actions in the STS transitions.

With this interpretation an *STS* viewed as a LTS defines a simulation of its configuration graph.

This is a bit coarse abstraction but we found it useful since we can refine more or less the unfolding process.

This refinement may be done by our decomposition notion and the termination of the CG computation can be ensured by boundedness.

I will present these two concepts later.

- ▶ We extend the LTS synchronous product to STS :  $\otimes_V$

- ▶ We extend the LTS synchronous product to STS :  $\otimes_V$
- ▶  $G(d_1 \otimes_V d_2, (v_1, v_2)) \equiv G(G(d_1, v_1) \otimes_V d_2, v_2) \equiv (G(d_1, v_1) \otimes_V G(d_2, v_2))$

- ▶ We extend the LTS synchronous product to STS :  $\otimes_V$
- ▶  $G(d_1 \otimes_V d_2, (v_1, v_2)) \equiv G(G(d_1, v_1) \otimes_V d_2, v_2) \equiv (G(d_1, v_1) \otimes_V G(d_2, v_2))$
- ▶ More computation implies more information

- ▶ We extend the LTS synchronous product to STS :  $\otimes_V$
- ▶  $G(d_1 \otimes_V d_2, (v_1, v_2)) \equiv G(G(d_1, v_1) \otimes_V d_2, v_2) \equiv (G(d_1, v_1) \otimes_V G(d_2, v_2))$
- ▶ More computation implies more information
- ▶ ...  $\succeq I_{LTS}(G(d_1, v_1) \otimes_V d_2) \succeq I_{LTS}(G(d_1 \otimes_V d_2, (v_1, v_2)))$

- ▶ We extend the LTS synchronous product to STS :  $\otimes_V$
- ▶  $G(d_1 \otimes_V d_2, (v_1, v_2)) \equiv G(G(d_1, v_1) \otimes_V d_2, v_2) \equiv (G(d_1, v_1) \otimes_V G(d_2, v_2))$
- ▶ More computation implies more information
- ▶  $\dots \succeq_{LTS} (G(d_1, v_1) \otimes_V d_2) \succeq_{LTS} (G(d_1 \otimes_V d_2, (v_1, v_2)))$
- ▶  $I_{LTS}(d_1 \otimes_V d_2) \succeq_{LTS} (G(d_1, v_1) \otimes_V d_2) \succeq \dots$

- ▶ We extend the LTS synchronous product to STS :  $\otimes_V$
- ▶  $G(d_1 \otimes_V d_2, (v_1, v_2)) \equiv G(G(d_1, v_1) \otimes_V d_2, v_2) \equiv (G(d_1, v_1) \otimes_V G(d_2, v_2))$
- ▶ More computation implies more information
- ▶  $\dots \succeq_{LTS} (G(d_1, v_1) \otimes_V d_2) \succeq_{LTS} (G(d_1 \otimes_V d_2, (v_1, v_2)))$
- ▶  $I_{LTS}(d_1 \otimes_V d_2) \succeq_{LTS} (G(d_1, v_1) \otimes_V d_2) \succeq \dots$
- ▶ Can be used to abstract some infinite and compound systems

## └ Composition

- We extend the LTS synchronous product to STS :  $\otimes_V$
- $G[d_1 \otimes_V d_2, (v_1, v_2)] = G[G(d_1, v_1) \otimes_V d_2, v_2] = [G(d_1, v_1) \otimes_V G(d_2, v_2)]$
- More computation implies more information
- $\dots \succeq \text{LTS}(G[d_1, v_1] \otimes_V d_2) \succeq \text{LTS}(G[d_1 \otimes_V d_2, (v_1, v_2)])$
- $\text{LTS}(d_1 \otimes_V d_2) \succeq \text{LTS}(G[d_1, v_1] \otimes_V d_2) \succeq \dots$
- Can be used to abstract some infinite and compound systems

We Extend the synchronous Product of automata to STS and we prove some related properties.

This first proposition gives three equivalent ways to compute the configuration graph of an STS product.

We may also refine more or less the interpretation as explained by the second property.

It shows that computing the CG of a component, then its product with the other component, gives a finer simulation for the configuration graph of the STS product than the LTS interpretation of this product.

These are simple results but they may be used to abstract some infinite state systems then to find errors or to prove properties.

- ▶ Finite resource allocation

- ▶ Finite resource allocation
- ▶ Finite configuration graph

- ▶ Finite resource allocation
- ▶ Finite configuration graph
- ▶ Semi-decidable, but decidable with Petri nets and some extensions (Finkel and al.)

- ▶ Finite resource allocation
- ▶ Finite configuration graph
- ▶ Semi-decidable, but decidable with Petri nets and some extensions (Finkel and al.)
- ▶ STS restrictions (counter machine) :

- ▶ Finite resource allocation
- ▶ Finite configuration graph
- ▶ Semi-decidable, but decidable with Petri nets and some extensions (Finkel and al.)
- ▶ STS restrictions (counter machine) :
  - ▶ Variables  $C_i$  are natural numbers

- ▶ Finite resource allocation
- ▶ Finite configuration graph
- ▶ Semi-decidable, but decidable with Petri nets and some extensions (Finkel and al.)
- ▶ STS restrictions (counter machine) :
  - ▶ Variables  $C_i$  are natural numbers
  - ▶ Guards are  $C_i \geq M_i$

- ▶ Finite resource allocation
- ▶ Finite configuration graph
- ▶ Semi-decidable, but decidable with Petri nets and some extensions (Finkel and al.)
- ▶ STS restrictions (counter machine) :
  - ▶ Variables  $C_i$  are natural numbers
  - ▶ Guards are  $C_i \geq M_i$
  - ▶  $C_i := \sum_{j=1}^m a_j * C_j \pm p_i$ ,  $a_j, p_i$  : Natural and at least one  $a_j$  is greater than 0

- ▶ Finite resource allocation
- ▶ Finite configuration graph
- ▶ Semi-decidable, but decidable with Petri nets and some extensions (Finkel and al.)
- ▶ STS restrictions (counter machine) :
  - ▶ Variables  $C_i$  are natural numbers
  - ▶ Guards are  $C_i \geq M_i$
  - ▶  $C_i := \sum_{j=1}^m a_j * C_j \pm p_i$ ,  $a_j, p_i$  : Natural and at least one  $a_j$  is greater than 0
- ▶ Look for an accumulating cycle in the configuration graph

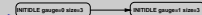
- ▶ Finite resource allocation
- ▶ Finite configuration graph
- ▶ Semi-decidable, but decidable with Petri nets and some extensions (Finkel and al.)
- ▶ STS restrictions (counter machine) :
  - ▶ Variables  $C_i$  are natural numbers
  - ▶ Guards are  $C_i \geq M_i$
  - ▶  $C_i := \sum_{j=1}^m a_j * C_j \pm p_i$ ,  $a_j, p_i$  : Natural and at least one  $a_j$  is greater than 0
- ▶ Look for an accumulating cycle in the configuration graph

▶ **INITIDLE gauge=0 size=3**

**INITIDLE gauge=1 size=3**

## Boundedness

- Finite resource allocation
- Finite configuration graph
- Semi-decidable, but decidable with Petri nets and some extensions (Finkel and al.)
- STS restrictions (counter machine) :
  - Variables  $C_i$  are natural numbers
  - Guards are  $C_i \geq M_i$
  - $C_i := \sum_{j \in A_i} a_j \cdot p_j$  : Natural and at least one  $a_j$  is greater than 0
- Look for an accumulating cycle in the configuration graph



We use the boundedness notion since it is intuitively related to finite resource allocation.

The boundedness property states that the reachability graph of a state machine is finite.

This is a semi-decidable property since the configuration graph computation may not terminate.

However this property is decidable for Petri Nets.

Furthermore it is decidable for two strict extensions of Petri Nets, see Finkel and others for a general survey about that.

In our case restricting STS variables to Natural numbers, guards to greater than comparisons and actions to counter assignment of this form, we get a decidable procedure to check boundedness.

The principle is close to the configuration graph computation but we look for an accumulating cycle. That is a control state cycle with a data value at the end strictly greater than the one at the beginning of the cycle.

# Decomposition Principle

- ▶ Unfolding is split in two steps from a partition of variables

# Decomposition Principle

- ▶ Unfolding is split in two steps from a partition of variables
- ▶ Data type guards and actions are decomposable in two parts

# Decomposition Principle

- ▶ Unfolding is split in two steps from a partition of variables
- ▶ Data type guards and actions are decomposable in two parts

/ gauge:=0 ; size:=3 ; total=3



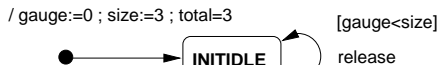
/ gauge:= gauge+1 ; total:= total-1

- ▶

A partition of the variables :  $\{ \text{gauge}, \text{size} \} + \{ \text{total} \}$

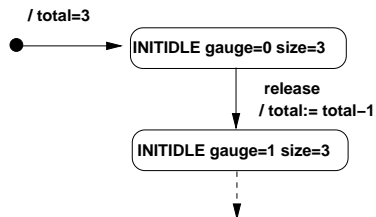
# Decomposition Principle

- ▶ Unfolding is split in two steps from a partition of variables
- ▶ Data type guards and actions are decomposable in two parts



- ▶ / gauge:= gauge+1 ; total:= total-1

A partition of the variables : {gauge, size } + {total}



# Decomposition Principle

- ▶ Unfolding is split in two steps from a partition of variables
- ▶ Data type guards and actions are decomposable in two parts

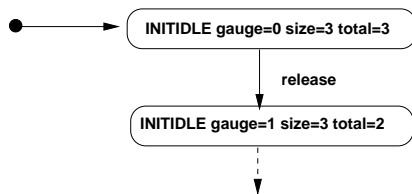
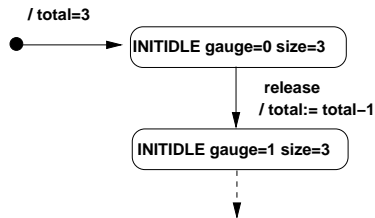
/ gauge:=0 ; size:=3 ; total=3



/ gauge:= gauge+1 ; total:= total-1

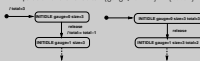


A partition of the variables : {gauge, size } + {total}



## └ Decomposition Principle

- Unfolding is split in two steps from a partition of variables
- Data type guards and actions are decomposable in two parts



The second concept is decomposition.

It means that there is a partition of the set of variables which allows to split the CG computation in two parts.

The data type, the guards and the actions are decomposed in two parts related to each subset of the variables.

In the example a partition is : {gauge, size} on one hand and {total} on the other hand.

The configuration graph computation may be done in two steps.

The first evaluates guards and actions related to the gauge and size variables. It does not consider the second subset of variables.

The second step will use guards and actions related to the total variable and leads to the expected configuration graph.

# Decomposition Property

- ▶  $G_1(d, v_1^0)$  a decomposition

# Decomposition Property

- ▶  $G_1(d, v_1^0)$  a decomposition
- ▶ Synchronous product is decomposable

# Decomposition Property

- ▶  $G_1(d, v_1^0)$  a decomposition
- ▶ Synchronous product is decomposable
- ▶ If  $d, d'$  are decomposable STS then  $d \otimes_V d'$  is decomposable

# Decomposition Property

- ▶  $G_1(d, v_1^0)$  a decomposition
- ▶ Synchronous product is decomposable
- ▶ If  $d, d'$  are decomposable STS then  $d \otimes_V d'$  is decomposable
- ▶  $d$  decomposable  $I_{LTS}(G_1(d, v_1^0)) \succeq I_{LTS}(G(d, v^0))$

# Decomposition Property

- ▶  $G_1(d, v_1^0)$  a decomposition
- ▶ Synchronous product is decomposable
- ▶ If  $d, d'$  are decomposable STS then  $d \otimes_V d'$  is decomposable
- ▶  $d$  decomposable  $I_{LTS}(G_1(d, v_1^0)) \succeq I_{LTS}(G(d, v^0))$
- ▶ Safety properties can be proved by simulation (Dams, Loiseaux, ...)

# Decomposition Property

- ▶  $G_1(d, v_1^0)$  a decomposition
- ▶ Synchronous product is decomposable
- ▶ If  $d, d'$  are decomposable STS then  $d \otimes_V d'$  is decomposable
- ▶  $d$  decomposable  $I_{LTS}(G_1(d, v_1^0)) \succeq I_{LTS}(G(d, v^0))$
- ▶ Safety properties can be proved by simulation (Dams, Loiseaux, ...)
- ▶ Bounded decomposition

# Decomposition Property

- ▶  $G_1(d, v_1^0)$  a decomposition
- ▶ Synchronous product is decomposable
- ▶ If  $d, d'$  are decomposable STS then  $d \otimes_V d'$  is decomposable
- ▶  $d$  decomposable  $I_{LTS}(G_1(d, v_1^0)) \succeq I_{LTS}(G(d, v^0))$
- ▶ Safety properties can be proved by simulation (Dams, Loiseaux, ...)
- ▶ Bounded decomposition
- ▶ They can be proved by model-checking on the bounded decomposition

## └ Decomposition Property

- $G_1(d, v_1^0)$  a decomposition
- Synchronous product is decomposable
- If  $d, d'$  are decomposable STS then  $d \odot_V d'$  is decomposable
- $d$  decomposable  $\models_{LTS} (G_1(d, v_1^0)) \succeq \models_{LTS} (G(d, v_1^0))$
- Safety properties can be proved by simulation (Dams, Liseaux, ...)
- Bounded decomposition
- They can be proved by model-checking on the bounded decomposition

This first computation step is noted G1.

Note that a synchronous product is naturally decomposable on each component, following the properties shown earlier.

In addition if the components are themselves decomposable then this offers several ways to decompose the product.

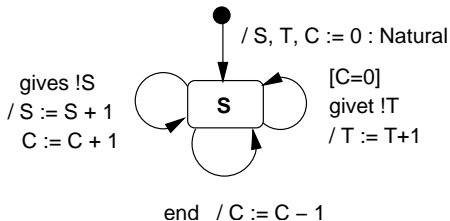
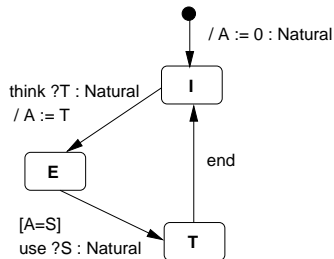
Lastly, If an STS is decomposable then the interpretation of the first decomposition step is a simulation of the configuration graph.

Recall that safety properties can be proved using simulation.

Thus if the decomposition is bounded safety properties may be proved using classic MC and then these properties are satisfied by the original system.

Now I will describe two application examples.

# Example 1 : Ticket Protocol



**Process**

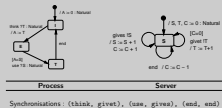
**Server**

Synchronisations : (think, givet), (use, gives), (end, end)

## Bounded Analysis and Decomposition

2006-06-11

# Example 1 : Ticket Protocol



We illustrate this decomposition principle on a mutual exclusion protocol inspired by the ticket protocol but in a distributed context with messages. We have a server which delivers tickets which are used by the processes to enter in critical section.

The server has three counters,  $T$  and  $S$  are ticket counters and  $C$  counts the number of processes in critical section.

The process gets a local copy of the  $T$  ticket and whenever this local counter is equal to the  $S$  ticket it enters in critical section.

The event `end` means leaving the critical section.

# A Bounded Decomposition

- ▶ With a finite number of processes

# A Bounded Decomposition

- ▶ With a finite number of processes
- ▶ Synchronous product is not bounded

# A Bounded Decomposition

- ▶ With a finite number of processes
- ▶ Synchronous product is not bounded
- ▶ Experiments with SPIN and CADP up to 6 processes

# A Bounded Decomposition

- ▶ With a finite number of processes
- ▶ Synchronous product is not bounded
- ▶ Experiments with SPIN and CADP up to 6 processes
- ▶ A bounded decomposition :

$$(\{A\}, \{C, T, S\}) = \boxed{(\{\}, \{C\})} + (\{A\}, \{T, S\})$$

# A Bounded Decomposition

- ▶ With a finite number of processes
- ▶ Synchronous product is not bounded
- ▶ Experiments with SPIN and CADP up to 6 processes
- ▶ A bounded decomposition :  
$$(\{A\}, \{C, T, S\}) = (\{\}, \{C\}) + (\{A\}, \{T, S\})$$
- ▶ The counter choice can be assisted using communication analysis

# A Bounded Decomposition

- ▶ With a finite number of processes
- ▶ Synchronous product is not bounded
- ▶ Experiments with SPIN and CADP up to 6 processes
- ▶ A bounded decomposition :  
$$(\{A\}, \{C, T, S\}) = \boxed{(\{\}, \{C\})} + (\{A\}, \{T, S\})$$
- ▶ The counter choice can be assisted using communication analysis
- ▶ One counter : boundedness is decidable!

## └ A Bounded Decomposition

- With a finite number of processes
- Synchronous product is not bounded
- Experiments with SPIN and CADP up to 6 processes
- A bounded decomposition :  
 $\langle\langle A \rangle\rangle, \langle\langle C, T, S \rangle\rangle = \langle\langle \langle\langle I \rangle\rangle, \langle\langle C \rangle\rangle \rangle + \langle\langle A \rangle\rangle, \langle\langle T, S \rangle\rangle$
- The counter choice can be assisted using communication analysis
- One counter : boundedness is decidable!

We consider a system with a finite number of processes.

Note that the counters are not bounded, at least A, T and S.

Thus classic model-checking bounds the variables and checks for mutual exclusion.

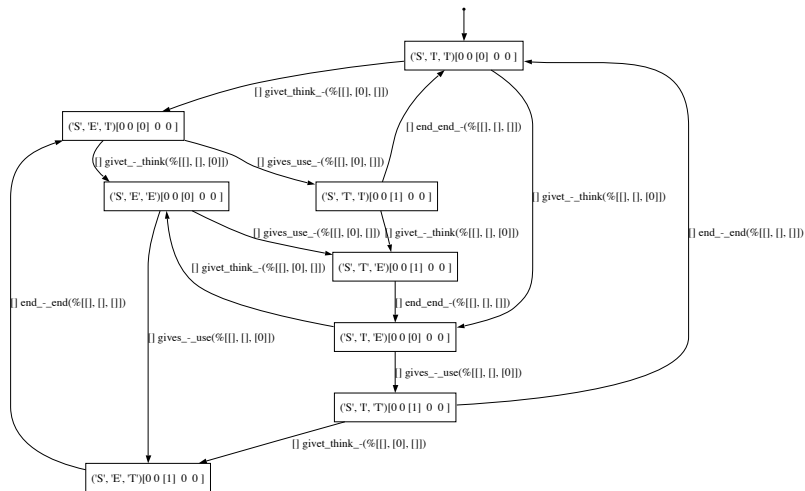
Experiments with SPIN and CADP show that they stop with 6 processes.

A decomposition is possible with this partition of the variables.

The decomposition choice can be assisted using an analysis of the input and output variables.

The decomposition is bounded and its configuration graph for two processes is the following.

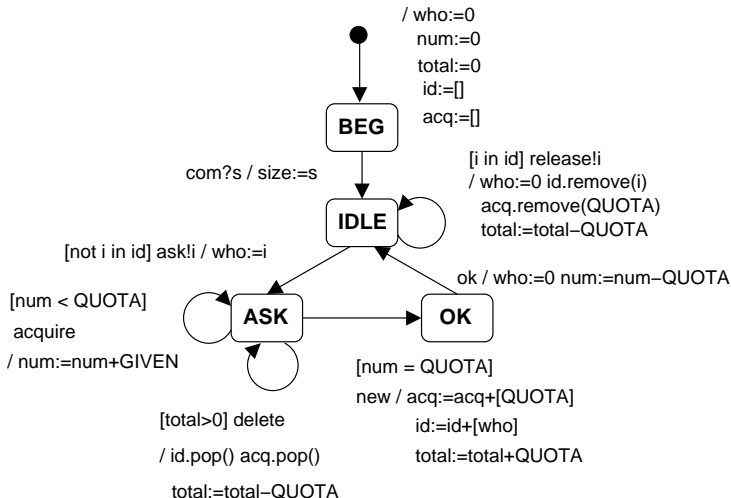
# The Bounded Analysis



Server x Process x Process



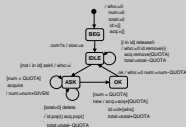
# Example 2 : A Resource Allocator : the Client System



## Bounded Analysis and Decomposition

2006-06-11

## Example 2 : A Resource Allocator : the Client System



We have also applied this technique to a resource allocator.

We have previously seen the allocator, the client system is like that :

We consider a finite but unknown number of clients.

who and  $i$  are used to denote client identities ,  $num$  and  $total$  are resource quantities,  $id$  is a list of clients and  $acq$  a list of their resources.

We have two constants :  $quota$  is the quantity requested by a client and given the quantity given by the allocator.

A client ask for a  $QUOTA$  quantity to the allocator, then the allocator gives  $GIVEN$  resources to the client may be with several interactions.

Event  $new$  and  $ok$  are used to close the allocation process for a client.

The  $delete$  and  $release$  events are used to release quota resources.

- ▶ A partition :  $(\{\text{size, gauge}\}, \{\text{size, num, total}\})$   
+  $(\{\text{who}\}, \{\text{who, acq, id}\})$

- ▶ A partition :  $(\{\text{size, gauge}\}, \{\text{size, num, total}\})$   
+  $(\{\text{who}\}, \{\text{who, acq, id}\})$
- ▶ Bounded decomposition (finite allocated quantities!)

- ▶ A partition :  $\boxed{(\{\text{size, gauge}\}, \{\text{size, num, total}\})}$   
+  $(\{\text{who}\}, \{\text{who, acq, id}\})$
- ▶ Bounded decomposition (finite allocated quantities!)
- ▶ The sequence delete ; delete cannot occur

- ▶ A partition :  $(\{\text{size, gauge}\}, \{\text{size, num, total}\})$   
+  $(\{\text{who}\}, \{\text{who, acq, id}\})$
- ▶ Bounded decomposition (finite allocated quantities!)
- ▶ The sequence delete ; delete cannot occur
- ▶ Deadlocks iff GIVEN does not divide QUOTA

- ▶ A partition :  $(\{\text{size, gauge}\}, \{\text{size, num, total}\})$   
+  $(\{\text{who}\}, \{\text{who, acq, id}\})$
- ▶ Bounded decomposition (finite allocated quantities!)
- ▶ The sequence delete ; delete cannot occur
- ▶ Deadlocks iff GIVEN does not divide QUOTA
- ▶ Deadlock freeness needs an additional property

- ▶ A partition :  $(\{\text{size, gauge}\}, \{\text{size, num, total}\})$   
+  $(\{\text{who}\}, \{\text{who, acq, id}\})$
- ▶ Bounded decomposition (finite allocated quantities!)
- ▶ The sequence delete ; delete cannot occur
- ▶ Deadlocks iff GIVEN does not divide QUOTA
- ▶ Deadlock freeness needs an additional property
- ▶ ask ; acquire<sup>p</sup> ; delete ; acquire<sup>r</sup> ; new ; ok  
where  $p + r = (\text{QUOTA \% GIVEN})$

- ▶ A partition :  $(\{\text{size, gauge}\}, \{\text{size, num, total}\})$   
+  $(\{\text{who}\}, \{\text{who, acq, id}\})$
- ▶ Bounded decomposition (finite allocated quantities!)
- ▶ The sequence delete ; delete cannot occur
- ▶ Deadlocks iff GIVEN does not divide QUOTA
- ▶ Deadlock freeness needs an additional property
- ▶ ask ; acquire<sup>p</sup> ; delete ; acquire<sup>r</sup> ; new ; ok  
where  $p + r = (\text{QUOTA \% GIVEN})$
- ▶ Bounded waiting time availability is a safety

- ▶ A partition :  $(\{\text{size, gauge}\}, \{\text{size, num, total}\})$   
+  $(\{\text{who}\}, \{\text{who, acq, id}\})$
- ▶ Bounded decomposition (finite allocated quantities!)
- ▶ The sequence `delete ; delete` cannot occur
- ▶ Deadlocks iff GIVEN does not divide QUOTA
- ▶ Deadlock freeness needs an additional property
- ▶ `ask ; acquirep ; delete ; acquirer ; new ; ok`  
where  $p + r = (\text{QUOTA \% GIVEN})$
- ▶ Bounded waiting time availability is a safety
- ▶ All possibly verified on the bounded decomposition

## └ Verifications

- A partition :  $\{([size, gsize], [size, num, total]) + ([sh], [sh, acq, is])\}$
- Bounded decomposition (finite allocated quantities!)
- The sequence delete ; delete cannot occur
- Deadlocks iff GIVEN does not divide QUOTA
- Deadlock freeness needs an additional property
- ask ; acquire<sup>r</sup> ; delete ; acquire<sup>r</sup> ; new ; ok where  $p + r = (QUOTA \% GIVEN)$
- Bounded waiting time availability is a safety
- All possibly verified on the bounded decomposition

It exists an interesting bounded decomposition which separate quantities from client identities. This system is bounded a strong reason is : we have a finite amount of resources.

We check some safety properties and the deadlock freeness on it.

Deadlock freeness on the bounded decomposition alone is not sufficient to ensure that the global system is deadlock free.

Assuming that each action has a maximum duration, we may be interested in the longest logical time sequence between a client request (ask) and its end (ok).

The longest sequence has the following form, it is an availability and a safety property.

It is also satisfied for any clients in the system but it requires an additional analysis observing that the client system freezes the client identity during the allocation and the system is not blocking.

# Conclusion and Future Work

- ▶ A formal component model with STS

# Conclusion and Future Work

- ▶ A formal component model with STS
- ▶ Configuration graphs, boundedness, decomposition

# Conclusion and Future Work

- ▶ A formal component model with STS
- ▶ Configuration graphs, boundedness, decomposition
- ▶ Other experiments : slip, bakery protocols, cash point system, ticket reservation

# Conclusion and Future Work

- ▶ A formal component model with STS
- ▶ Configuration graphs, boundedness, decomposition
- ▶ Other experiments : slip, bakery protocols, cash point system, ticket reservation
- ▶ A Python prototype : STS definition, synchronous product, boundedness checking

# Conclusion and Future Work

- ▶ A formal component model with STS
- ▶ Configuration graphs, boundedness, decomposition
- ▶ Other experiments : slip, bakery protocols, cash point system, ticket reservation
- ▶ A Python prototype : STS definition, synchronous product, boundedness checking
- ▶ Future work

# Conclusion and Future Work

- ▶ A formal component model with STS
- ▶ Configuration graphs, boundedness, decomposition
- ▶ Other experiments : slip, bakery protocols, cash point system, ticket reservation
- ▶ A Python prototype : STS definition, synchronous product, boundedness checking
- ▶ Future work
  - ▶ Integrate other abstractions

# Conclusion and Future Work

- ▶ A formal component model with STS
- ▶ Configuration graphs, boundedness, decomposition
- ▶ Other experiments : slip, bakery protocols, cash point system, ticket reservation
- ▶ A Python prototype : STS definition, synchronous product, boundedness checking
- ▶ Future work
  - ▶ Integrate other abstractions
  - ▶ Automate using communication analysis, data isomorphism, ...

# Conclusion and Future Work

- ▶ A formal component model with STS
- ▶ Configuration graphs, boundedness, decomposition
- ▶ Other experiments : slip, bakery protocols, cash point system, ticket reservation
- ▶ A Python prototype : STS definition, synchronous product, boundedness checking
- ▶ Future work
  - ▶ Integrate other abstractions
  - ▶ Automate using communication analysis, data isomorphism, ...
  - ▶ Optimise the prototype

## Conclusion and Future Work

- A formal component model with STS
- Configuration graphs, boundedness, decomposition
- Other experiments : slip, bakery protocols, cash point system, ticket reservation
- A Python prototype : STS definition, synchronous product, boundedness checking
- Future work
  - Integrate other abstractions
  - Automate using communication analysis, data isomorphism, ...
  - Optimise the prototype

To conclude : Behavioural interfaces are required in component based software engineering to perform analysis and relate efficiently models and implementations.

Expressive models such as STS are needed to take data encapsulation and value passing into account.

In this paper we proposed a framework for STS based on configuration graphs and LTS interpretations.

We have also presented specific analysis techniques, namely bounded analysis and bounded decomposition

Some other experiments have been done to demonstrate how these techniques may complement model-checking.

We have implemented a Python prototype which support these ideas.

Future work aims at extending our techniques :

We plan to integrate other abstraction techniques in our prototype. It is sometimes possible to automate the decomposition, we expect to use communication analysis and data isomorphism for this. We are also