Introduction to MARIA and High-Level Petri Nets

Marko Mäkelä

Laboratory for Theoretical Computer Science

Helsinki University of Technology

P.O.Box 9700

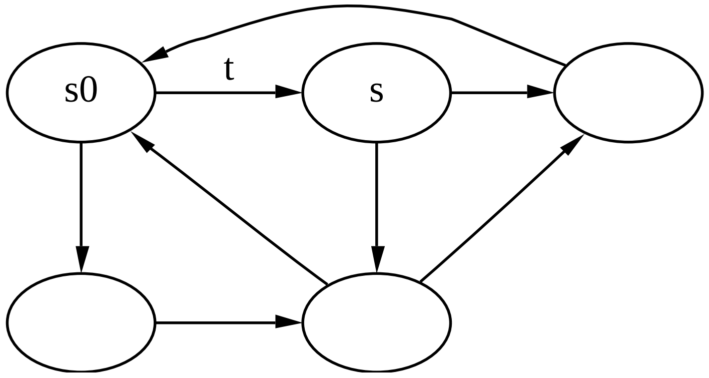
02015 HUT

Finland

October 9, 2001

Modelling Concurrent Systems (1/3)

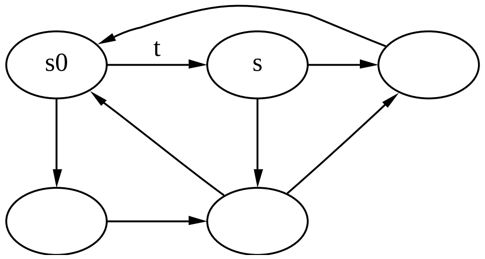
All computing systems can be described with some kind of state automata or transition systems. A system consists of a set of states S and a set of transformation rules $T \subseteq S \times S$ that describe how the system can evolve once it has started from an initial state $s_0 \in S$.

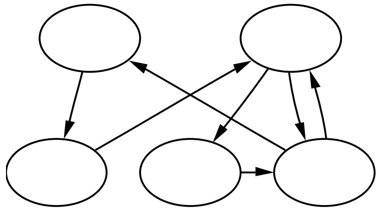


Above, states are depicted with ellipses and transformations with arcs. Formally, we could write $t = \langle s_0, s \rangle \in T$ or $t(s_0) = s$.

Modelling Concurrent Systems (2/3)

Complex systems are often divided into components to allow more compact notation. This is the natural way of describing distributed systems. The “big picture” $\langle S, T \rangle$ is constructed by putting the components $\langle S_i, T_i \rangle$ together in a parallel composition.





Each component has a local state S^i and some local actions $T^i_{\text{local}} \subseteq T^i \subseteq S^i \times S^i$, and a set of global actions that communicate with other components, capable of transforming the states of multiple components at a time. The state of the composed system is a product of the component states: $S = S^1 \times \cdots \times S^n$. The initial state is $s_0 = \langle s_0^1, \ldots, s_0^n \rangle$.

Modelling Concurrent Systems (3/3)

Each modelling formalism defines a state and a transformation rule in a different way. In finite state automata or labelled transition systems, the states and transformations are listed explicitly.

Higher-level formalisms define the transformation rules in the terms of some algebra. Usually, a transformation rule is divided into two parts:

- an enabling condition $c : S \to \mathbb{B} = \{\perp, \top\}$, for example $x > 3$, and
- a transformation $T \subseteq S \times S$, for example $x \leftarrow x + 1$.

Low-Level Petri Nets (1/3)

Low-level Petri nets are among the simplest modelling formalisms for concurrent systems. They consist of places, which may contain a number of tokens, and transitions, which may remove tokens from a set of input places and insert tokens to a set of output places.

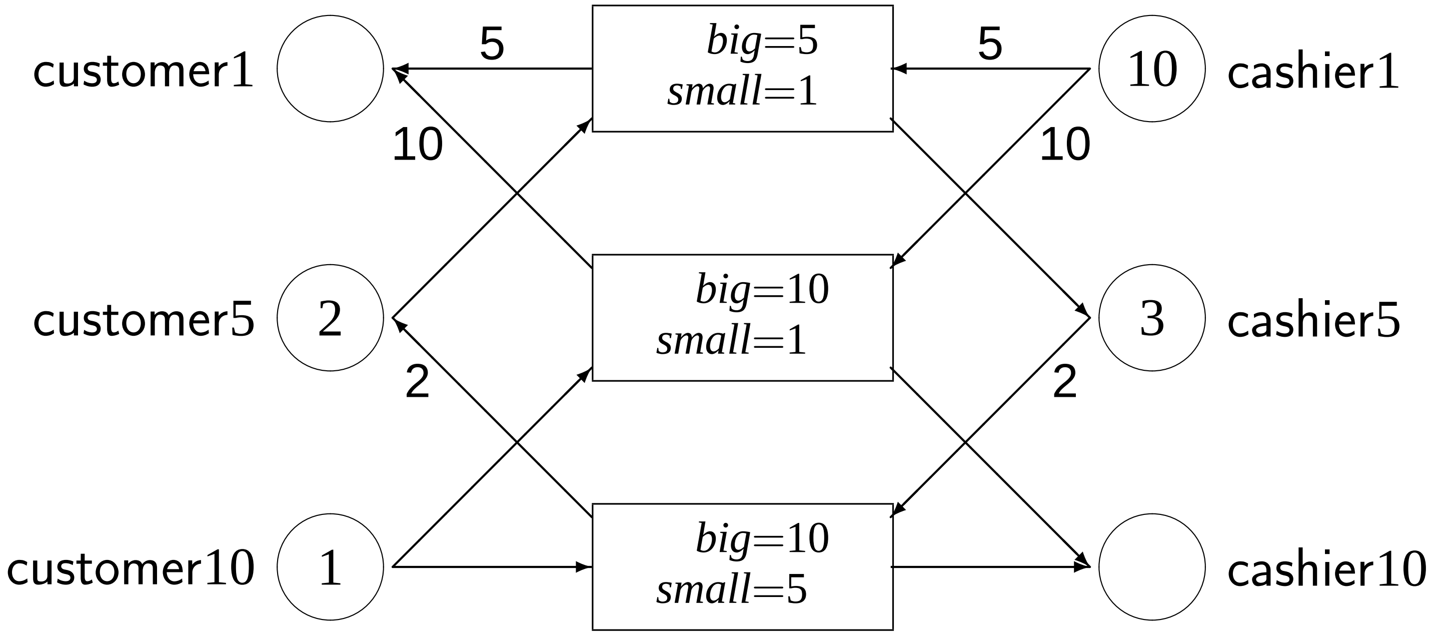
A state of a low-level Petri net is a mapping from the Petri net places to the numbers of tokens in the places: $\mathcal{P} \to \mathbb{N}$.

A Petri net transition is enabled if its each input place contains at least as many tokens as the weight of the arc indicates.

Firing an enabled transition transforms the state by subtracting the input arc weights from the token counts of the input places and by adding the output arc weights to the token counts of the output places.

Low-Level Petri Nets (2/3)

The following Petri net describes a system where a customer wants to change coins to smaller ones. In the depicted configuration, the customer has two ⑤ coins and one ⑩.



Low-Level Petri Nets (3/3)

The set of potentially reachable states of a low-level Petri net is a power-set of $\mathcal{P} \to \mathbb{N}$, meaning that each assignment $\mathcal{P} \to \mathbb{N}$ represents a potentially reachable state.

The initial configuration of our example system can be represented as

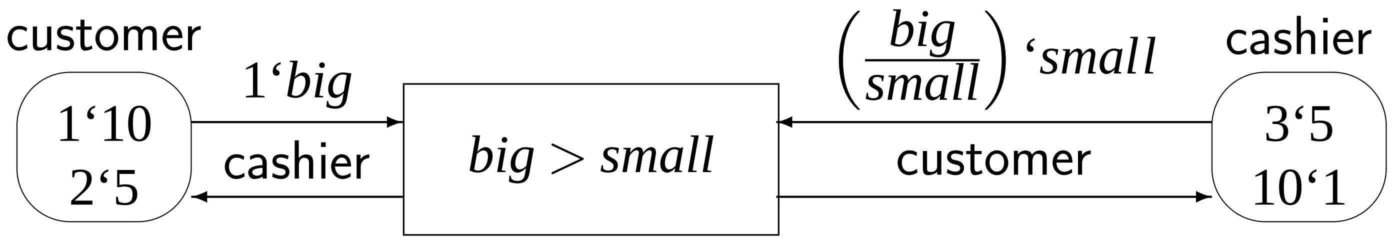
$$\begin{array}{ll} \{ & \mathsf{customer1} \mapsto 0, \mathsf{customer5} \mapsto 2, \mathsf{customer10} \mapsto 1, \\ & \quad \mathsf{cashier1} \mapsto 10, \mathsf{cashier5} \mapsto 3, \mathsf{cashier10} \mapsto 0 \qquad \} \end{array}$$

or shorter $s_0 = \langle 0, 2, 1, 10, 3, 0 \rangle$.

The configurations $\langle 5, 1, 1, 5, 4, 0 \rangle$, $\langle 10, 0, 1, 0, 5, 0 \rangle$, $\langle 0, 4, 0, 10, 1, 1 \rangle$, $\langle 5, 3, 0, 5, 2, 1 \rangle$, and $\langle 10, 2, 0, 0, 3, 1 \rangle$ are reachable, and there are 8 actions leading from one state to another. Note that whenever the initial configuration is changed, also the reachable state space changes.

High-Level Petri Nets

Plain Petri nets are a rather low-level modelling formalism. It is often more convenient to use a higher-level formalism. In high-level variants of Petri nets, places contain values, and the arc inscriptions of transitions contain expressions and variables. Our money-changing example can be represented in a much more compact way as a high-level net:



Note that we could introduce a ② coin without updating the transition definition.

Understanding High-Level Petri Nets (1/3)

The data model of high-level nets (e.g. algebraic nets, many-sorted nets or coloured nets) differs greatly from low-level nets. The major additions are:

- data types $\mathcal{D}$ (also known as colours or algebraic sorts)
- high-level places p that contain multi-sets of typed tokens
  - the marking (or state) of a high-level place p is a multi-set of the place's data type: $M(p) = (\mathcal{D} \to \mathbb{N})$
  - in the underlying low-level net, there is a place for each $\langle p, d \rangle$ pair, $d \in \mathcal{D}$

Understanding High-Level Petri Nets (2/3)

The transitions in a high-level net have input and output arcs, just like transitions in low-level nets. High-level arcs are labelled with expressions that must evaluate to multi-sets that match the data types of the attached places.

Usually, the arc expressions refer to variables. In theory, the value of a variable is selected nondeterministically from the domain of the variable. In practice, for efficiency reasons, high-level Petri net analysis tools usually derive the values for variables from the input arc expressions and the markings of the input places.

In the underlying low-level net, there is a transition for each valuation that can be assigned to the variables of a high-level transition. It is possible to restrict these valuations by specifying conditions (guards or gates) on them, e.g. $x \neq y \wedge z < y$.[*]

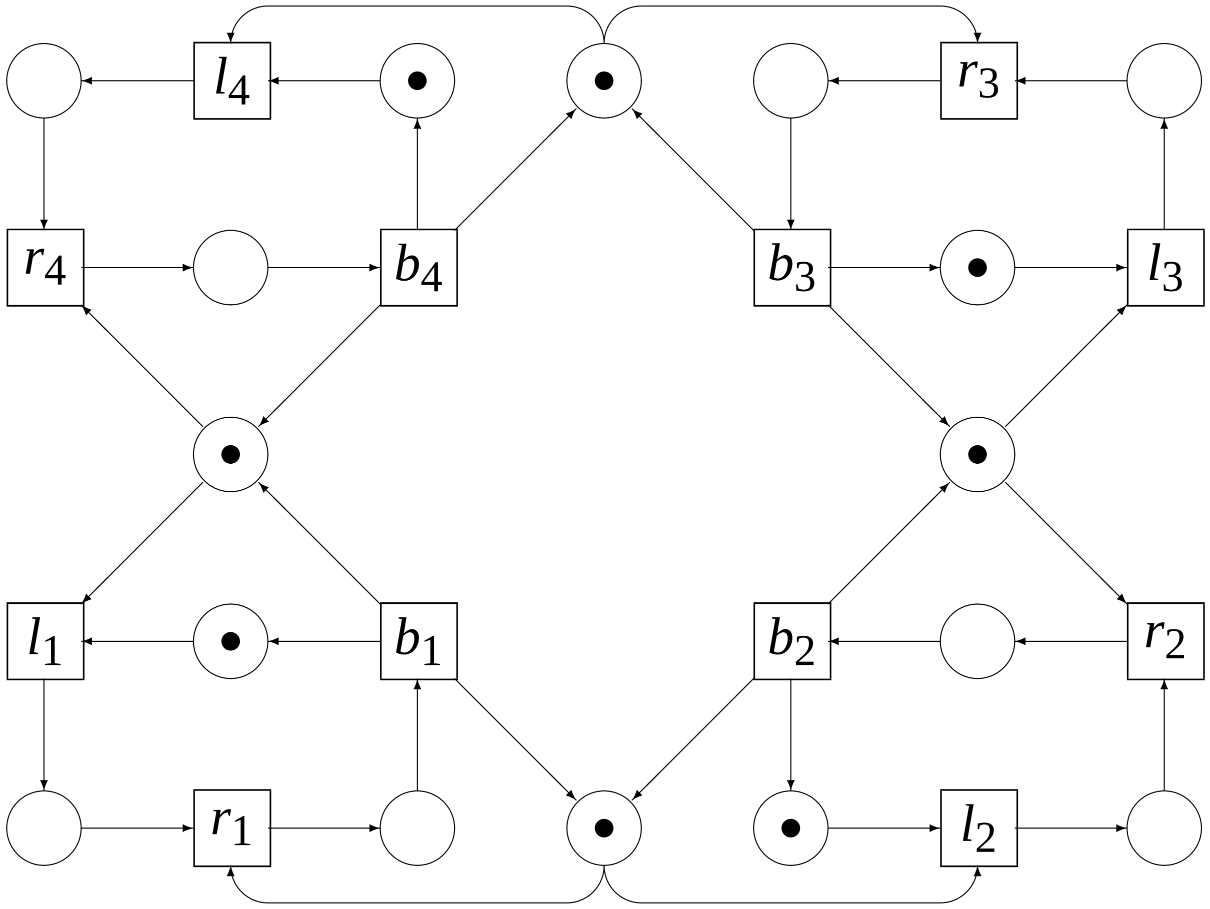
[*]Although equality conditions, such as $x = y + z$, are possible, there often is a more efficient way: just eliminate one of the variables, e.g. replace all occurrences of x with $y + z$.

Understanding High-Level Petri Nets (3/3)

High-level Petri nets can be viewed as computing systems that operate on shared data (the markings of the high-level places). In a sense, the high-level places can be understood as global variables and the high-level transitions as some kind of guarded assignment statements that operate on the data.

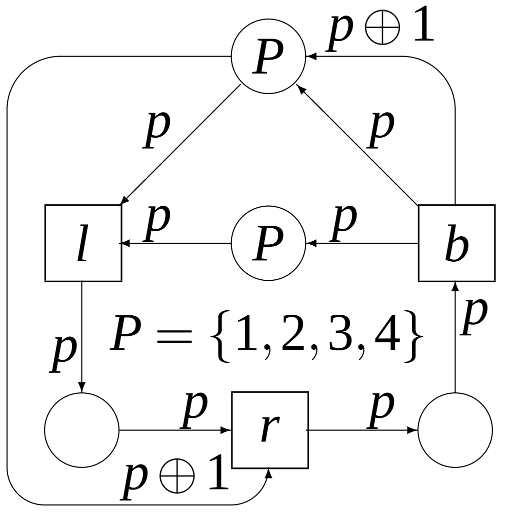
When studying or constructing Petri net models, it is often helpful to recognise different flows in the system being modelled: the control flows of the local entities, and various information flows. In many cases, these flows are clearly visible in the graphical representation of the net. Sometimes the places that represent a flow are folded.

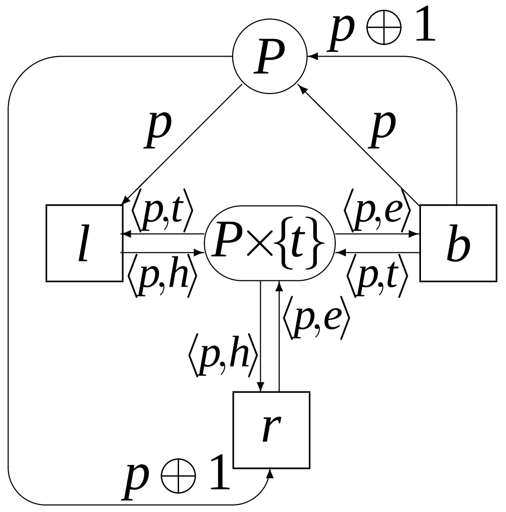
Folding Nets: An Example with Dining Philosophers (1/2)



Folding Nets: An Example with Dining Philosophers (2/2)

The model is much more compact when the fork and philosopher places are folded. The two flows are still visible after this transformation:





It is a matter of taste whether folding the philosopher status places improves the readability of the model. More folding allows transitions to be more flexible. At the extreme, any model can be folded to one place and one transition.

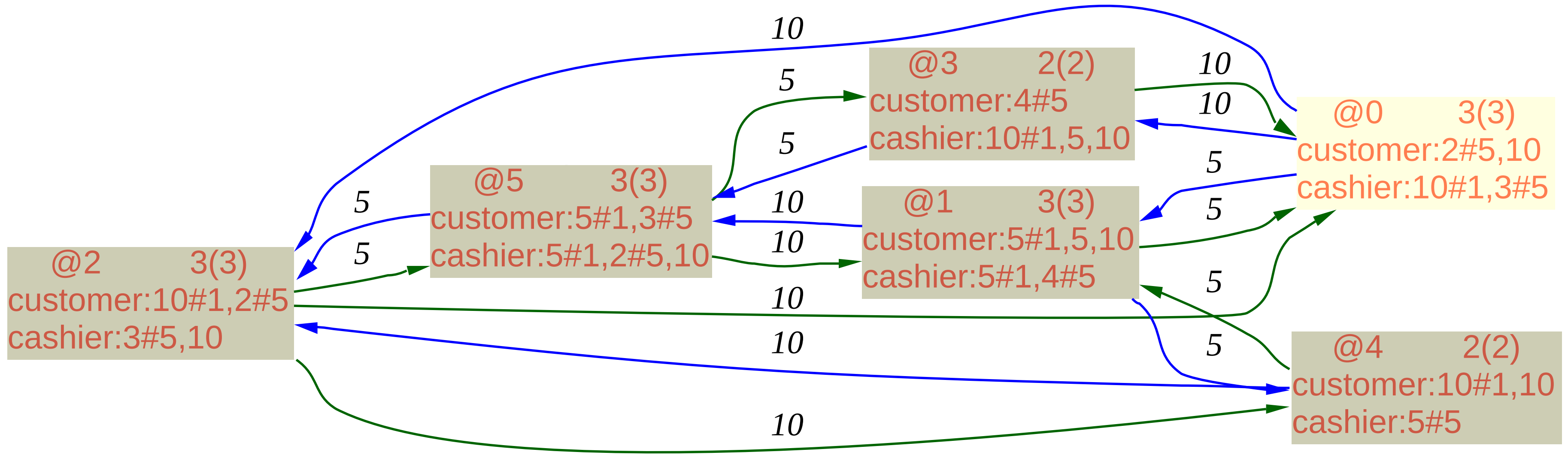
MARIA

MARIA (Modular Reachability Analyser for Algebraic System Nets) is a tool for analysing the logical consistency of concurrent systems, such as data communication protocols or parallel processes communicating via shared memory. The system is described in a textual language whose expressions, data types, operations, and grammar resemble common programming languages, such as C or Java.

Given a model system, MARIA can either perform interactive simulations of the system's execution or it can exhaustively search the reachable state space of the system. During exhaustive search, it can verify correctness properties expressed in linear temporal logic formulae. The search may be sped up by compiling the model to executable C functions.

MARIA Example (1/2)

This reachability graph of our money-changing example was computed by MARIA from the model represented on next slide. The output was slightly edited to add colours and to simplify the edge labels, which now denote the “big” money taking part in the transaction, and blue edges denote actions where the customer’s money is changed to smaller units.



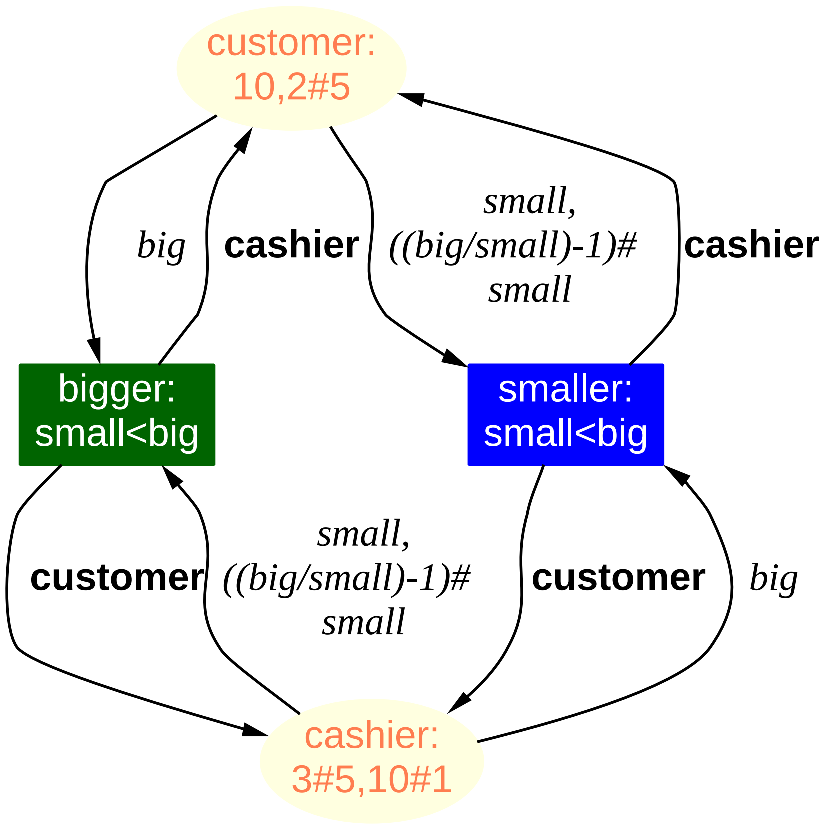
MARIA Example (2/2)

```
typedef unsigned (1,5,10,50,100,500) money_t;

place customer money_t: 10,2#5;
place cashier money_t: 3#5,10#1;

trans smaller
in {
  place customer: big;
  place cashier: small, (big/small-1)#small;
}
gate big > small
out {
  place cashier: place customer;
  place customer: place cashier;
};

trans bigger
in {
  place customer: small, (big/small-1)#small;
  place cashier: big;
}
gate big > small
out {
  place customer: place cashier;
  place cashier: place customer;
};
```



MARIA Data Types (1/2)

The data type system of MARIA was designed with high-level programming and specification languages in mind. The basic types are:

`bool`	`false`,`true`	truth values
`char`	`'\0'..'\377'`	8-bit characters
`unsigned`	`0`, `1`, ...	non-negative integers (usually 32 or 64 bits)
`int`	..., `-1`, `0`, `1`, ...	signed integers (usually 32 or 64 bits)
`enum`		enumerations

Composite data types can be defined by applying the following constructs:

`struct { int a; char b; }`	`{0, 'x'}`	structure, tuple or record
`union { int a; char b; }`	`b = 'x'`	union (one of the alternatives)
`int(0..2)[bool]`	`{0,1}`	array
`bool[queue 2]`	`{false,false}`	a queue of 0..2 truth values
`char[stack 3]`	`{'z'}`	a stack of 0..3 characters

MARIA Data Types (2/2)

All data types can be combined arbitrarily. If you need to define a queue of arrays of stacks indexed by records, go ahead.

The domains of data types can be restricted by applying constraints. We have already done so for integer types, as in the definition `int(0..2)[bool]` on the previous slide. Constraints do not need to be single ranges: it is possible to define multiple ranges: `unsigned(..3,10,15..20)`. Also composite data types can be constrained.

MARIA defines a total order for all its data types. For any value `x`, the successor and predecessor operations (`+x` and `|x`) are defined in such a way that the successor of the maximum value is the minimum value, and the predecessor of the minimum value is the maximum value. If `x` is an `unsigned(0..4)`, you can simply write `+x` instead of something like `(x+1)%5`.

Basic MARIA Expressions (1/2)

The expressions in the MARIA input language are heavily based on the C, C++ or Java programming languages. There are no pointers or references or method calls, or even assignments* (not even such as `x*=3` or `x++`).

There are a few new operators in addition to the predecessor and successor operators that were introduced in the previous slide. Some unary operators are just short-hand notation: `<int` yields the smallest value of the built-in `int` data type, `>bool` yields `true`, and `#char` yields `256`, the number of different `char` values.

The if-then-else operator `condition?true_expr:false_expr` of C has been generalised to a selection operator. For instance, if `x` is an `unsigned(0..4)`, the expression `x?3:2:1:0:4` is equivalent to `|x`.

*In Petri nets, expressions do not have side effects. All effects are modelled with transitions.

Basic MARIA Expressions (2/2)

The active component of a union value can be determined with the binary `is` operator. For instance, if `x` is a union whose components are named `a`, `b` and `c`, the condition `x is b` holds only when the second component of the union is active.

The unary `is` operator is used for data type conversions. For instance, a signed value `c` can be converted to unsigned with `is unsigned c`. An error is reported if `c` is negative.

For queues and stacks, the unary `/` and `%` operators report the used and the available capacity, respectively. The binary `+` operator inserts an item and the unary `-` removes an item. The unary `*` reads an item. The default access point can be overridden by specifying an integer index between `[` and `]` brackets: `-b[2]` is a copy of the queue or stack `b` without its third item. Indexing starts from zero, the default index.

Please refer to the user's guide for a detailed list of all operators.

Multi-Set Operations in MARIA

Multi-sets are just like sets, but an item can be contained several times in a multi-set. Formally, if A is a set, any multi-set μ of A can be represented as $\mu : A \to \mathbb{N}$. A “conventional” set $A' \subseteq A$ could be written as $A' : A \to \{0,1\}$.

Only a fraction of the built-in multi-set operations of MARIA is needed for basic modelling. When a multi-set is expected, singleton items are converted automatically. Multi-sets can be combined by separating them with commas, like in `10,2#5`, which is equivalent to `10,5,5`. The binary # operator multiplies the amount of items in a multi-set.

The multi-set summation operator may be useful when specifying an initial marking. The MARIA notation for $\bigcup_{d\in\mathcal{D}}\bigcup_{e\in\mathcal{D}\setminus\{d\}}\{\langle d,e\rangle\}$ is `D d: D e (e != d): {d,e}`.

Other multi-set operations are usually used in conjunction with more advanced features of MARIA, such as specifying the desired logical properties of the model.

Defining the Data Structures: `typedef` and `place`

Any non-trivial computing system operates on data. In MARIA, two constructs are related with data structures: `typedef` (data type definition) and `place` (storage definition).

```
/** a data type for philosophers or forks */
typedef unsigned (1..4) ph_t;
/** the available forks (0 to 4); initially all present */
place fork (0..#ph_t) ph_t: ph_t f: f;
/** the thinking philosophers (initially all of them) */
place thinking (0..#ph_t) ph_t: ph_t ph: ph;
/** the hungry philosophers (initially none) */
place hungry (0..#ph_t) ph_t;
/** the eating philosophers (initially none) */
place eating (0..#ph_t) ph_t;
```

Advanced `place` Features

Although it is not necessary to define the capacity (the allowed number of tokens) of a place, it is a good practice to do so, since the limits can be exploited in the analysis, and they can help to catch errors. The same applies to data type definitions: there is no need to assign a name to each data type:

```
/** unconstrained place that contains pairs */
place pairs struct { int a; char b; }: 2#{ 0, ’x’ };
```

On the other hand, it is possible to impose even tighter limits. In our model of dining philosophers, only those philosophers who are not hungry or eating can be thinking:

```
place thinking (0..#ph_t) ph_t: (ph_t f: f)
  minus place hungry minus place eating;
```

Transition Definitions (1/3)

Transformations on data are the most interesting aspect of any model of computation. In high-level Petri nets, there is only one type of actions, high-level transitions with:

- local variables (defined implicitly in MARIA)
- input and output arcs labelled with multiset-valued expressions
- guards or gates that restrict the valuations on local variables
- additional features, such as priorities, fairness conditions or nondeterminism

Transition Definitions (2/3)

The transitions of our dining philosophers model could be defined like this:

```
trans left
in { place thinking: ph; place fork: ph; }
out { place hungry: ph; };
trans back
in { place eating: ph; }
out { place thinking: ph; place fork: ph, +ph; };
```

Exercise: How would you define `trans right`?

Transition Definitions (3/3)

In the money-changing example, we wrote expressions like `place customer` on output arcs. The construct is short-hand notation for the total value of the transition's input arcs connected to the place. In that example, we also defined restrictions on the valuations by using the `gate` keyword:

```
trans smaller
in { customer: big; cashier: small,(big/small-1)#small; }
out { cashier: place customer; customer: place cashier; }
gate big > small;
```

Why don't we simply write `(big/small)#small` on the second input arc? MARIA takes a little performance shortcut and applies a unification algorithm on input arcs. The value of `small` could not be determined from such an expression, since the multiplicity `big/small` cannot be evaluated without knowing the value of `small`.

Nondeterminism (1/2)

In high-level Petri nets, nondeterminism has traditionally been modelled by defining a constant place that contains all the possible values, and by binding a variable from the place. In MARIA, such places can be identified with the `const` keyword:

```
/** all philosophers who can be picked randomly */
place random (#ph_t-1) ph_t const: ph_t ph (ph != <ph_t): ph;
/** pick a random philosopher */
trans random
in { place random: p; /* ... */ }
out { place random: p; /* ... */ };
```

Nondeterminism (2/2)

When a nondeterministic value is not needed in any input arc expressions, the more efficient nondeterminism operator can be used on an output arc:

```
/** change the identity of a thinking philosopher */
trans random_think
in { place thinking: ph; }
out { place thinking: ph_t p! (p != <ph_t && p != ph); };
```

The condition is optional. When the value of the variable is needed only once, also the variable name can be omitted, and the expression would simply be `ph_t!`.

Specifying State Properties (1/2)

A model of a computing system is not very useful, if nothing is stated about the desired properties of the system. Sure, an automated tool can explore all reachable states of the model and report any evaluation errors or deadlocks, but it cannot do much beyond that.

Let us assume that we want to verify if the forks can run out in our dining philosopher model. We could do this either by adjusting the capacity constraint of the `fork` place, or by writing a `reject` formula:

```
/** the available forks (always at least 1) */
place fork (1..#ph_t) ph_t: ph_t f: f;
/** report states where no forks are available */
reject place fork equals empty;
```

Specifying State Properties (2/2)

By default, MARIA does not report any deadlocks (states where no transitions are enabled) it encounters. To have all encountered deadlocks reported, specify `deadlock true`. Uninteresting deadlocks can be filtered out with a condition:

```
/** report all deadlocks where forks are available */
deadlock !(place fork equals empty);
```

The `reject` and `deadlock` formulae can be any conditions on the state. Please refer to the user's guide for advanced multi-set operations.

If a `reject` or `deadlock` formula evaluates to the special value `fatal`, the analysis is stopped. The short-circuit evaluation of `||`, && and `?:` is very useful here:

```
/** halt if an unexpected deadlock occurs */
deadlock !(place fork equals empty) && fatal;
```

Editing MARIA Models

MARIA models can be edited with any text editor. Both `CR` and `LF` are accepted as white space, so there should not be any problems with Apple or Microsoft systems.

If you are using GNU EMACS 20, you may find the file `pn-mode.el` useful. It is based on `cc-mode` (which is the default editing mode for C-like languages) and provides smart indentation and syntax highlighting, which makes it easier to read and write MARIA models.

The MARIA user's guide contains brief instructions for setting up the editing mode and syntax highlighting in EMACS.

If you want to use other types of identifiers than `[A-Za-z_][A-Za-z0-9_]*`, you can include the names in double quotes like this: `place "Møbius string"`. The only character that is not allowed in identifiers is the `NUL` character `"\0"`.

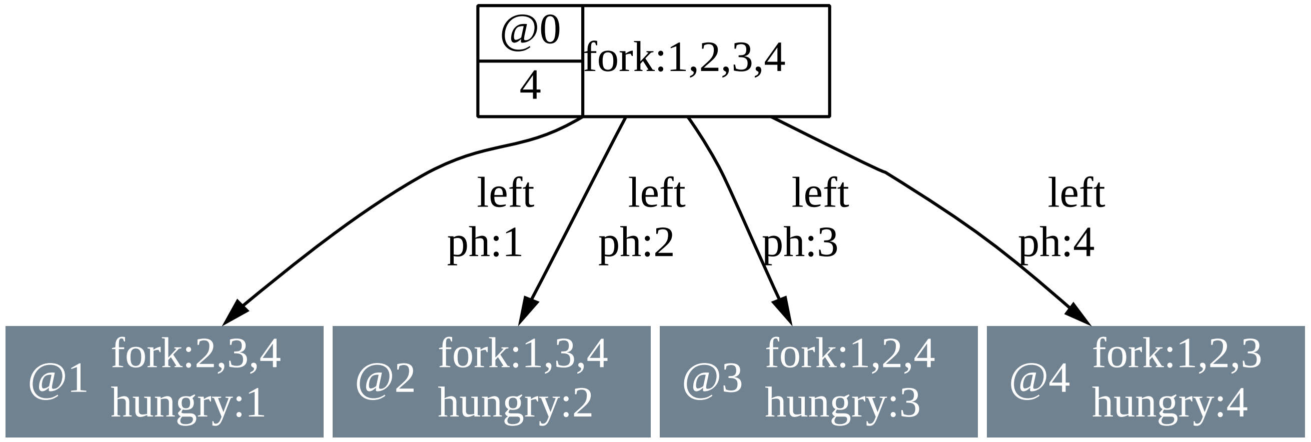
Simulating and Exploring a Model (1/3)

Many people prefer to construct a formal model of a system in small steps. The interactive simulator in MARIA supports this way of working. At any time, MARIA can be invoked to check the syntactical correctness of the model and to compute the initial marking. For example, let us examine our model of dining philosophers:

```
>maria -m dinner.pn
@0$show
unprocessed state (
 fork:
  1,2,3,4
 thinking:
  1,2,3,4
)
@0$hide thinking; visual show
```

Simulating and Exploring a Model (2/3)

We requested a graphical view of the state, without the contents of the `thinking` place. After clicking the left mouse button on the rectangle that represents the initial marking, the display looks like the following:



The state space of the system can be explored interactively by left-clicking on the states of interest. Clicking and holding down the right mouse button over a node, arc or the graph background reveals a menu with the applicable choices.

Simulating and Exploring a Model (3/3)

With the visualisation interface, it is possible to input almost all commands. If you prefer a textual interface or in case the graphical interface is unavailable for some reason, you can find the MARIA query language commands in the user's guide.

One thing that cannot easily be accessed from the graphical interface is the verification of temporal properties. Suppose that we want to know if the forks will eventually run out in all executions starting from the initial state:

```
>maria -m dinner.pn -p lbt
@0$<> place fork equals empty
(command line):2:constructing counterexample
(command line):2:counterexample path:
(command line):2: @0 @1 @8 @8 @0
@0$exit
```

Graphical display can be requested by prepending the formula with `visual`.

Exhaustive Reachability Analysis (1/2)

The best way to verify `reject` and `deadlock` properties and to detect all evaluation errors of a model is exhaustive breadth-first or depth-first analysis. In these modes, MARIA runs silently (unless it is requested to report the number of explored states and arcs with the `-E` option) until the whole state space has been explored or a fatal error occurs.

For optimal performance, the disk files that represent the reachability graph should be kept on a local disk. Create a subdirectory on the local disk and switch there:

```
>mkdir /tmp/maria-$USER; cd /tmp/maria-$USER
>maria -b ~/dinner.pn -e exit
deadlock state @27
"dinner.pn": 34 states, 88 arcs
```

You can later explore the generated reachability graph by invoking MARIA with the `-g` option. You could also verify temporal properties in this way: `maria -m ~/dinner.pn -e ’<> place fork equals empty’ -e exit`.

Exhaustive Reachability Analysis (2/2)

When you are analysing a small model that generates a big reachability graph,[*] you can try the `-C` option of MARIA to compile the model to executable C code. The option takes one parameter: the name of the directory where the code will be created.

For big models, you can try the experimental `-P` option together with `-e breadth` or `-e depth`. With that option, no reachability graph is generated, but instead the set of reachable states is approximated with a hash-based memory structure. If a hash collision occurs, some parts of the state space will not be explored. No liveness properties can be verified with this option.

[*] A big model can generate a small state space; there is no rule of thumb here.

Conclusion

MARIA aims to be an easy-to-use formal tool for analysing the behaviour of concurrent and distributed systems. The modelling formalism is powerful, since it was designed with high-level programming and specification languages in mind.

The analyser can be interfaced both to higher-level formalisms (such as the CCITT Specification and Description Language, or a subset of the Java programming language) and to lower-level formalisms, such as low-level Petri nets (LOLA, PEP, PROD) and labelled transition systems.

For more information, visit the MARIA home page at the address

`http://www.tcs.hut.fi/maria/`

and contact us.