Automata with Parameterized Arrays and Parameterized Networks of Automata

Tomáš Vojnar, Ahmed Bouajjani
LIAFA, University Paris 7

!!! a working document !!!

1 Introduction

The goal of this short report is to propose—or to at least start discussions and work leading to a proposal of—a common basis for describing (a relatively large class of) systems with a somehow parameterized (process and/or data) structure. We propose *automata working with generalized, parametric arrays* to be used for this purpose. We also propose a concept of *parametric networks of automata with arrays* which can be viewed as a higher-level and in some situations easier to use modelling tool that may, however, be defined in the terms of the basic automata with arrays. In models based on the parametric networks of automata, we allow global as well as local parametric arrays to be used.[1]

The above concepts are introduced in such a way that we believe it should not be extremely difficult to include a *language support* for them into the IF language [IF] that is being developed as a common model description language within the ADVANCE project. Subsequently, it is possible to think about *interfaces* to the different tools for verifying systems with a somehow parameterized structure [PAX,TLV[P],TLV[T],...]. Of course, the different tools will commonly be able to handle only different subclasses of the systems that can be modelled by automata with arrays. These *subclasses* may be defined with respect to the applied structure of the arrays, the number of parameters, the way of handling indexes of arrays, the data to be stored in these arrays, etc. Apart from that, after fixing all the parameters used, it should be at least possible to simulate and/or verify the appropriate instances of the modelled families of systems within the IF tool.

[1] Let us note that we do not explicitly deal with non-array variables here—they can be viewed as a shortcut for a 1-dimensional array with the bound fixed to 1.

2 Automata with Arrays

Let us start with the lower-level concept of automata with arrays which is simpler and more fundamental. The arrays used in these automata are *generalized* in the sense that they combine classical indexing (by subsets of integers) in some dimensions with path-based, tree-like addressing (by strings on subsets of integers) in other dimensions.

The proposed concept considers *dimensions* of arrays to be fixed while *bounds* for the classical indexing in the dimensions where this is used as well as *arities* and *heights* of the "embedded" trees may be parametric.

2.1 The concept of automata with arrays

We introduce the concept of automata with arrays such that it depends on two notions: (1) the *trivial data*, i.e. the data unstructured from our point of view, and (2) *indexes* used directly within classical indexing and indirectly as symbols in paths of the tree-like addressing. Let $\mathcal{D}$ denote a set of trivial data types, each of them with some set of values and some set of operators and predicates with a meaning in some first order theory. Let I denote integers associated with some first order theory to be used for dealing with indexes.

Now we can define an *automaton with arrays* w.r.t. some $\mathcal{D}$ and I as a tuple $M_{\mathcal{D},I} = (P, A, S, Q, T, C_0)$ where:

- P, A, and Q are finite, pairwise disjoint sets of *parameters* (ranging on positive integers), *arrays*, and *control states*, respectively.
- S is a function assigning *signatures* to arrays. For each $a \in A$, $S(a) = (D^a, (K^a_1, ..., K^a_{n^a}))$ where $D^a \in \mathcal{D}$ and $\forall_{n^a} i : K^a_i = b^a_i \in \mathbb{N}^+ \cup P$ or $K^a_i = (r^a_i, h^a_i) \in (\mathbb{N}^+ \cup P) \times (\mathbb{N}^+ \cup P)$.[2]
- T is a finite set of *transitions* of the form $t = (q_1, q_2, \tau)$ where $q_1, q_2 \in Q$ and $\tau ::= [(\forall|\exists)_b x \mid (\forall|\exists)_{r,h} x]^* : $ <a guard> $\rightarrow$ <an action> with $b, r, h \in \mathbb{N}^+ \cup P$, X being the set of quantified variables (disjunct with $P \cup A$), the guard being a quantifier-free formula, and the action an assignment to entries of arrays from A. The only variables allowed in the guard and the action are those from $P \cup A \cup X$.

  Guards and actions are built on data terms w.r.t. $\mathcal{D}$, index terms t_I w.r.t. I, path terms $t_P ::= \varepsilon \mid x \mid t_I \mid t_P.t_P$, and array terms $t_A ::= a[t_{IP}, ..., t_{IP}]$ where $a \in A$, $t_{IP} ::= t_I \mid t_P$, and the number and types of t_{IP} correspond to $S(a)$. The elements of A may only be used as allowed in t_A, but t_A may be used in the place of variables in terms. Guards are built on atomic formulae w.r.t. $\mathcal{D}$, I, and atomic path

[2] We use the notation $\forall_n i$ to refer to all $i \in \{0, ..., n-1\}$ for some $n \in \mathbb{N}^+$.

formulae $af_P ::= t_P \leq t_P \mid |t_P| \leq t_I \mid |t_P| \leq |t_P|$ using the common propositional connectives.

- C_0 is a finite set of *initial configurations* $c_0 = (q, \iota)$ where $q \in Q$ and ι is a quantified formula corresponding to τ from above without the assignment part.

Let us note that if there is used $K_i^a = b_i^a \in \mathbb{N}^+ \cup P$ in the dimension i of some array a, it means that we will use the classical, hypercube-like indexing in that dimension. The b_i^a is then referred to as the *bound* of a in the dimension i. Subsequently, integers from the set $\{0, ..., b_i^a - 1\}$ can be used for indexing in the given dimension of the given array.

If there is used $K_i^a = (r_i^a, h_i^a) \in (\mathbb{N}^+ \cup P) \times (\mathbb{N}^+ \cup P)$ in the dimension i of some array a, a path-based, tree-like addressing is to be used in that dimension. We refer to r_i^a and h_i^a as to the arity and the height of the tree embedded into the i-th dimension of a. Then, the array a is to be addressed in the dimension i by strings over the set $\{0, ..., r_i^a - 1\}$ with their maximum length being h_i^a.

The *path terms* $t_P ::= \varepsilon \mid x \mid t_I \mid t_P.t_P$ allow us to say that the empty path (referring to the root of a tree) is a path, a variable may represent a path, an index term represents a path with the length 1, and a concatenation of two paths is a path. *Atomic path formulae* $af_P ::= t_P \leq t_P \mid |t_P| \leq t_I \mid |t_P| \leq |t_P|$ allow us to compare paths lexicographically and to reason about their lengths.

The transitions may be viewed to work purely with formulae by replacing the assignments by equalities on array terms with primed array variables. The *evaluation* of variables in transition and initial constraint formulae can be defined in the usual way to map parameters to positive integers and arrays to functions from address sets to the appropriate data domains. The address set of an array $a \in A$ may be defined for some $ev : (P \to \mathbb{N}^+) \cup Id(\mathbb{N})$ as $addr(a, ev) = \{(x_1, ..., x_{n^a}) \mid \forall_{n^a} i : (K_i^a \in \mathbb{N}^+ \cup P \wedge x_i \in \{0, ..., ev(b_i^a) - 1\}) \vee (K_i^a \in (\mathbb{N}^+ \cup P) \times (\mathbb{N}^+ \cup P) \wedge x_i \in \{0, ..., ev(r_i^a) - 1\}^* \wedge |x_i| \leq ev(h_i^a))\}$. Within the scope of $(\forall|\exists)_b x$, the values from $\{0, ..., ev(b) - 1\}$ should be considered for x. The strings from $\{0, ..., ev(r) - 1\}^*$ with the length up to $ev(h)$ should be considered for x within the scope of $(\forall|\exists)_{r,h} x$.[3] The *interpretation* of the considered formulae is then as usual with the addition that we consider false all the formulae that do not respect the domains on which the applied operations are defined and all the formulae that try to access an entry of an array using an address outside the appropriate address set.[4]

[3] Note that quantified variables may be used not only for addressing, but also in data manipulation when $\mathcal{D}$ is respected.

[4] Alternatively, this could yield a transition to some error state.

2.2 The associated modelling language constructs

To be able to practically deal with automata with generalized arrays, it is desirable to have some support from the modelling language to be used.

In the case of the IF language, we—among all—need a *tree data type* whose declaration in the style of the IF syntax definition may look as follows:

```
type ::= tree [r,h] of type-id
```

The signatures in automata with generalized arrays may then be viewed as an unfolding of the nested definitions of types of arrays and trees in IF.[5]

Next we need some support for specifying the *quantification in transitions* for which we can have a special section in the declaration of transitions that can look, e.g., as follows:

```
[ forall(b,x);|exists(b,x);|forall(r,h,x);|exists(r,h,x); ]*
```

This declaration (if used) should always precede the declaration of guards as it is introduced in IF.[6] The quantified variables should not be declared as normal variables and the validity of their assignment should be limited to a single evaluation of the appropriate expression.

We should also have some way how to express the *initial constraints* on the contents of arrays. For this, some specialized construct could be introduced. Here, a limited support for quantification can be provided, e.g., only universal quantification, and the rest of the initialization may be put off into the transitions.

Finally, we need a support of the *path terms* and *path atomic formulae* to be (if needed) used in transitions and initial constraints. Their syntax (to be embedded into the expressions of IF) may be as follows:

```
tp  ::= eps|x|tp.tp|ti
afp ::= tp<=tp|len(tp)<=ti|len(tp)<=len(tp)
```

It could as well be helpful to have some predefined higher-level *address predicates* simplifying tests which will be probably often used when working with trees. They could include, for instance, `isroot(x)`, `isleaf(r,h,x)`, `isinner(r,h,x)`, or `isson(x,y)`. We could also have some predefined *address functions*, as, e.g., `son(x,i)` for computing the address of the `i`th son of the node `x` or `sonno(x)` for deriving the index of the node `x` within sons of its father. Furthermore, if we want to be able to *store paths* in states, we should introduce a data type `path[r,h]` for declaring regular (non-quantified) variables intended to store the paths.

[5] The notion of automata with arrays presented here does not take into account the possibility of combining arrays and trees with records in any way different from using records of trivial data.

[6] To increase readability, we could also allow quantification over variables used only in the action part of a transition to be put just before the action.

2.3 An example

In Figure 1, we present an example of an automaton with arrays. It is a model of the very simple mutual exclusion protocol from [PAX].

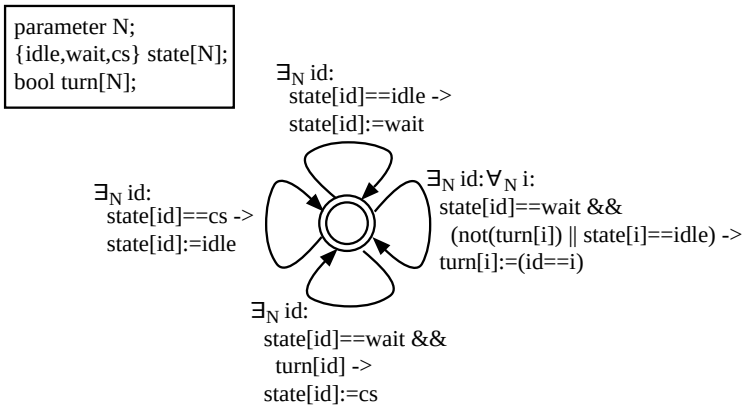

Figure 1: An example of an automaton with arrays

2.4 Limitations and Possible Extensions

The notion of automata with arrays as presented above does not provide an explicit support for *ring structures*. These can be dealt with by always evaluating values of indexes in a certain dimension modulo the appropriate bound. However, it is also possible to extend the definition of signatures to tuples involving a classical indexing (denoted, e.g., by $c(b_i^a)$), ring indexing ($r(b_i^a)$), and tree-based indexing ($t(r_i^a, h_i^a)$). From the point of view of the language support, this could lead to the addition of some construct like

`type ::= ring [b] of type-id`.

Addressing in ring-based dimensions could then be automatically augmented by the modulo operation.

The here proposed notion of automata with arrays does further not explicitly support *"irregular" structures* as, e.g., trees in which some subtrees are missing. The fact that some parts of the structures introduced will not be used is to be encoded into the behaviour of the automata.

We allow for parametric bounds, arities, and heights, but the dimensions of the generalized arrays considered here are fixed. Dealing with *parametric dimensions* would require some additional support for addressing (quantification over parts of addresses spanning through several dimensions).

We have not introduced any special kind of *quantification over data elements*. As an extension of $\forall_b x$ and $\forall_{r,h} x$ applicable both to indexes and (some) data, we could also allow using any $\forall_D x$ allowed by $D \in \mathcal{D}$.

Finally, *DAG structures* and *general graph structures* are not taken into account here.

3 Networks of Automata with Arrays

The further proposed notion of networks of automata with arrays can be considered to constitute a *higher-level modelling language* whose constructions can be folded into single automata with arrays. When describing the networks of automata, we will reuse many of the already introduced concepts.

3.1 The notion of networks of automata with arrays

As in the case of automata with arrays, we suppose networks of automata to be built w.r.t. some $\mathcal{D}$ and I. A *network of automata* $N_{\mathcal{D},I}$ may then be defined to consist of a finite set of *parameters* P, a finite set of *global arrays* A^g ($P \cap A^g = \emptyset$), a *signature function* S^g for the global arrays, a *process infrastructure* $f = (K_1^f, ..., K_{n^f}^f)$ with K_i^f as in the signatures of arrays, a finite set of *automata templates* $\mathcal{M}$, and a finite set C_0^g of *initial constraints* on the global arrays.

Perhaps the most interesting concept in the above is the *process infrastructure*. It is a structure whose nodes are to be associated with particular automata (i.e. instances of the automata templates). As described below, each particular automaton is aware of its position in this structure (accessible via the value of a pseudo-parameter $self$) and is able to use it for a communication with other automata.[7]

Each *automaton template* $M \in \mathcal{M}$ in $N_{\mathcal{D},I}$ consists of the following:

- A specification $((x_1^M, ..., x_{n^f}^M), \varphi^M)$ of a *set of addresses* of nodes in the process infrastructure f saying where instances of M may be situated (for a given evaluation of parameters). We require x_i^M to be a t_I term if $K_i^f \in \mathbb{N}^+ \cup P$ and to be a t_P term otherwise. Moreover, φ^M has to be a formula built on the quantifiers, atomic formulae, and t_{IP} terms applicable in transitions of automata with arrays (not using t_A). It may use parameters as free variables and its set of free variables must cover the variables used in $(x_1^M, ..., x_{n^f}^M)$.[8]

- A set of *local arrays* A^M, a *signature function* S^M for the local arrays, a set of *states* Q^M, and a set of *local initial configurations* C_0^M consisting of an element of Q^M and a constraint over the local arrays. All the mentioned sets should be finite, pairwise disjoint, and disjoint

[7] Here, we only deal with a *shared variable communication* implemented by allowing an automaton to access local arrays of other automata.

[8] An example of such a specification could be $((p), \exists_{2,h} q : 0 < |q| < h \wedge p = q)$.

with the sets of global components and components of other automata templates.

- A finite set of *transitions* $t = (q_1, q_2, \tau)$ where $q_1, q_2 \in Q^M$. The τ part of the transitions is as in automata with arrays where the set of arrays is $A^g \cup A^M$ and the set of parameters $P \cup \{self\}$ ($self \notin P$). Moreover, we allow local arrays of other processes to be accessed by a new type of terms $t_F ::= f[t_{IP}, ..., t_{IP}].t_A$ in which t_A may refer to local arrays of all the different automata templates. The t_F terms must be formed in conformance with the definition of the particular dimensions of f and may be used in the same way as t_A terms.

Let us note that an *initial state* of a network of automata consists of a certain set of instances of the particular automata templates distinguished by the addresses assigned to them. The address assigned to each of the instances must fulfill the appropriate φ^M w.r.t. the given values of parameters, and no two instances can have the same address. Moreover, the state must be maximal in the sense that no instance can be added into it without breaking the above conditions. The particular automata must be in their initial local states and the values stored in the global and local arrays must fulfill the appropriate initial constraints.

In a run of a network of automata, transitions in the involved automata are performed in an *interleaving* way. The τ part of the transitions may be treated as in the case of automata with arrays. Additionally, if there appears an attempt to use a t_F term to access an array which is not present in the referenced automaton, the appropriate transition is to be blocked (or should lead to some error state). The same action is to be taken in the case of referencing a node of the infrastructure which has not been associated with any automaton.

3.2 The associated modelling language constructs

For dealing with networks of automata with arrays, we need some more language support when compared to the basic automata with arrays.

Firstly, we need a way how to specify the *process infrastructure*. For this, we can introduce some new keyword followed by a declaration of some (nested) array type. The data type associated with the entries of this array can be ignored, or we can introduce a specialized data type generically denoting automata instances.

Secondly, we need some support for specifying the *set of addresses* of nodes in the process infrastructure where instances of a certain automaton template may be situated. In IF, this specification may be a part of the header of processes—for example, we may extend the headers of processes to the following:

`process process-id(`x_1`,...,`x_{n^f}`|`φ`);`

As a simplification, we may allow for using predefined address predicates, such as `isbetween(l,h,x)`, `isroot(x)`, `isleaf(r,h,x)`, etc., directly in the place of the particular x_i-s.[9]

Finally, the *remote referencing* of local arrays via the t_F terms should be allowed. When constructing addresses of remote processes, we can use predefined address functions, such as `left(i)`, `right(i)`, `father(x)`, `son(x,i)`, etc.

3.3 An example

As an example of a network of automata with arrays, we present a model of a simple protocol that should ensure that a sequence of messages is transmitted from a source node being the root of a tree downwards to a group of receivers represented by the leaves of the tree. The protocol is based on dealing with positive acknowledgments being sent upwards in the tree. We restrict ourselves to the case when the source node and the intermediary nodes cannot proceed to a new message before they are sure that all their successors have received the previous message (i.e. the size of the window of the protocol is just one).

The automaton template for the intermediary nodes in the mentioned system (let us call it the OPT network) is shown in Figure 2. It makes use of channels that do not guarantee the FIFO ordering and that are lossy. The automata templates for the source node and for the receiver nodes are shown in Figures 3 and 4, respectively. (The red numbers will be later used to refer to the corresponding transitions.)

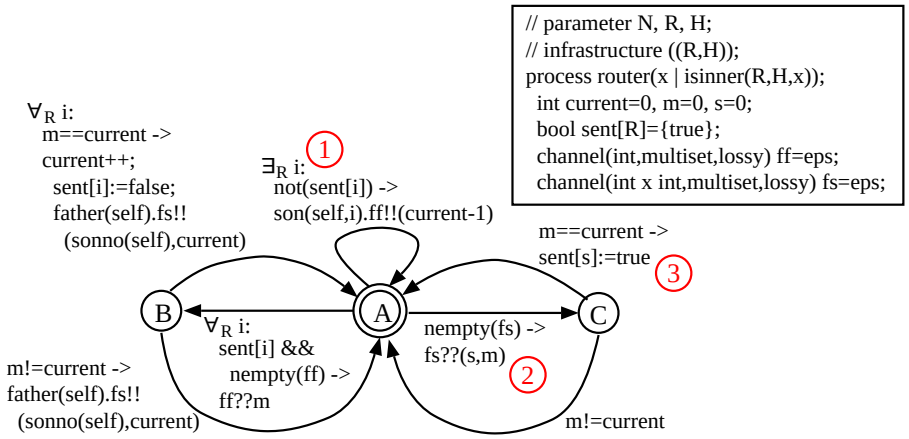

Figure 2: The automaton template for the inner nodes in the OPT network

[9] For the sooner presented example of the specification of a possible set of placements, we could then use `process router(isinner(2,h,x))` provided that the concerned automaton template is denoted as `router`.

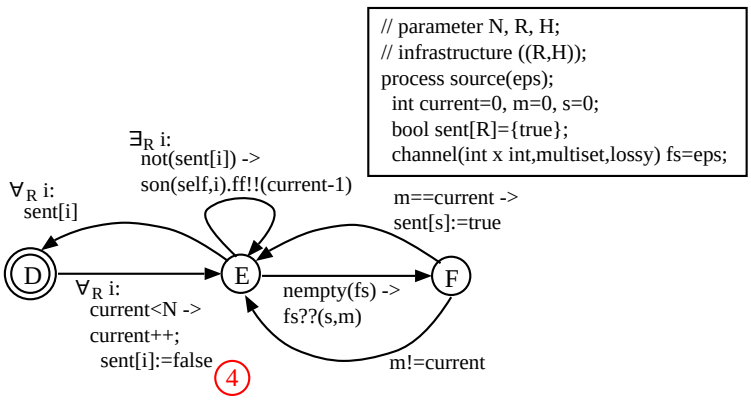

Figure 3: The automaton template for the root node in the OPT network

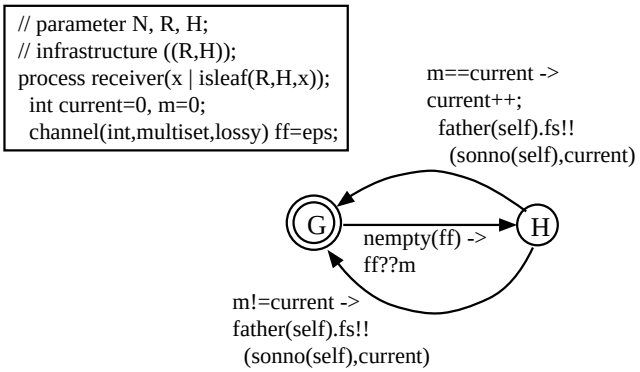

Figure 4: The automaton template for the leaf nodes in the OPT network

3.4 Folding networks of automata to automata with arrays

Let us now discuss the main principles of a possible translation of networks of automata with arrays to single automata with arrays.

First of all, we should create an array which will store the *local control states* of the different instances of the automata templates involved in runs of networks of automata. This array should have the union of all the sets of local states as its data type and the same structure as the infrastructure. Note that we required the sets of local control states to be disjunct. If this had not been the case, we would have had to distinguish the states by adding the information to what automaton template they belong. Into the domain of the array storing the local control states, we additionally add a special element to represent unoccupied elements of the infrastructure.

For each *local array* of each automaton template, we create a counterpart in the corresponding automaton with arrays. The structure of the new array should be the "concatenation" of the process infrastructure and the original, local structure. The data type remains the same. We select one "default" value from the date type to be invariantly stored in the entries of the resulting array that do not correspond to entries of a local array used by some instance of the appropriate automaton template.

Finally, *global arrays* of networks of automata may be directly used in automata with arrays.

When *initializing* the resulting automaton, we try to store an initial state of some automaton template M into each entry of the array of control states such that the address of the entry belongs to the set of the possible placements of instances of M in the process infrastructure. When this is impossible, we mark the appropriate node by the specially chosen state symbol. The contents of the originally local arrays should be initialized according to the local initialization constraints for the nodes corresponding to instances of the appropriate automata templates. The remaining entries of the originally local arrays should be set to the appropriate "default" values. The originally global arrays may be initialized directly according to the original global initial constraints.

Translated *transitions* should be existentially quantified by an address within the array of control states corresponding to the original pseudo-parameter `self` and denoting the currently advancing automaton. They should have the form of a self-loop on a *single global state* of the new automaton. The appropriate changes in the local control states of the original automata should be implemented by testing and manipulating the contents of the array of control states via the `self` address. Moreover, all references to the originally local arrays should be prefixed by the `self` address. Finally, for all "remote" references to the originally local arrays, we should add a predicate that tests whether in the corresponding entry of the array of control states there is a state belonging to an instance of the automaton template expected.

An example. In Figure 5, we present a part of the automaton with arrays obtained by folding the OPT network of automata. The folding has been partially optimized manually (e.g., we have matched some of the arrays that should encode the particular local variables of the different automata templates, etc.). (In the figure, the red numbers establish a relation with some of the transitions from the original network.)

A short discussion. When we translate a network of automata with arrays to a single automaton with arrays using the method sketched above, we of course encounter some *redundancy* in the sense that many entries of

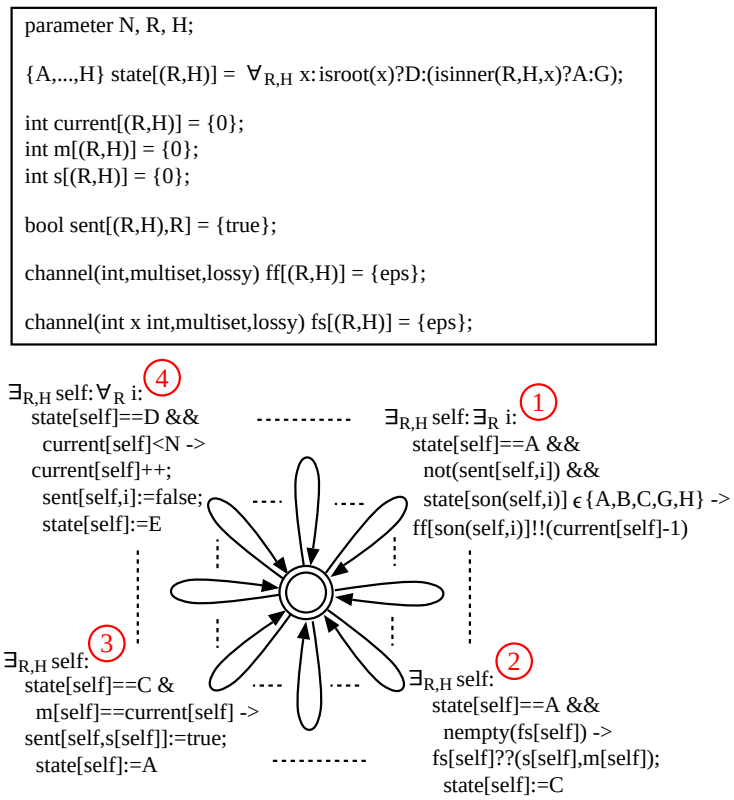

Figure 5: The OPT network folded into a single automaton (partially optimized w.r.t. the procedure described in Subsection 3.4)

the generated arrays need not be used (even if there are no unused nodes of the process infrastructure). This is a result of the fact that networks of automata allow us to define a process infrastructure whose nodes may be occupied by different kinds of processes with different local memories. If we use a single automaton template, this does not happen.

To avoid the redundancy in automata with arrays, we could augment them with some notion of *heterogeneity*, e.g., by allowing variant records of arrays as data types or by introducing "irregular" arrays not necessarily containing all addressable entries. However, without joining this notion of heterogeneity with automata templates as in the networks of automata, the resulting formalism could turn out to be even more complex and more difficult to handle than the networks of automata. Thus, the networks of automata with arrays seem to introduce some degree of heterogeneity which appears to be relatively reasonable.

3.5 Limitations and Possible Extensions

Apart from some of the limitations mentioned already in the context of automata with arrays, a limitation of the networks of automata which seems to be of interest is the missing support for some kind of *dynamic instantiation*. For the future, it could be interesting to allow new automata instances to join the process infrastructure at least in some of their originally border nodes.

Next, we can also think of a possible further generalization of the notion of networks of automata with arrays that could allow us to define *multiple independent infrastructures* and to somehow interconnect some of their nodes. This would introduce a certain additional tool making it easier to model (some) "asymmetric" networks.

Finally, from the point of view of an integration with the IF language, the relation of the here introduced constructs to using the so-called *signalroutes* of IF is not very clear. In the above examples, we have not used signalroutes at all and we have used a shared-memory communication or a communication via channels (queues) local to particular processes (cf. the OPT network). Unfortunately, the IF language does not directly support such (Promela-like) channels (they can only be captured as the so-called abstract data type here).

4 A Brief Discussion of Dealing with Quantifiers

Transitions with actions of the form $\forall_n i : \forall_m j :\ a[i] := 1;\ a[j] := 2$ should not be enabled when we want to stick strictly to the (logical-framework-like) semantics of the form $\forall_n i : \forall_m j :\ a'[i] == 1\ \wedge\ a'[j] == 2$. However, this could be relaxed to get closer to the common praxis of IF where the action $a := 1;\ a := 2$ leads to $a == 2$.

It is a question whether a similar evaluation principle should be introduced for the case of actions like $\forall_n i :\ a[1] := i$ or $\forall_n i :\ a[i\ mod\ 3] := i$ or whether they should lead to disabled transitions or should be syntactically prohibited. (For example, we could allow only a certain simple class of assignments with the right-hand side depending on some universally quantified variable, such as $\forall_n i :\ a[i] := i$.)

In the case of modifying the state in a way depending on an existentially quantified variable (as, e.g., in $\forall_n i : \exists_n j :\ a[i, j] := 1$), all the different evaluations of this variable should be taken into account leading possibly to different target states.