Information Societies Technology (IST) Programme

Future and Emerging Technologies - Open Domain

Project acronym: Advance

Project full title:

Advanced Validation Techniques
for Telecommunication Protocols

Contract no: IST-1999-29082

Deliverable D11

Integration of Tools for Parametric Reasoning

Year 2 : October 2001 - September 2002

Date of preparation: September 2002

Contributors: Paris, Grenoble, Uppsala, Weizmann Inst.

1 Introduction

The ADVANCE project is related to the development of several different tools for verification of systems with a parameterized structure. Namely, this concerns the Uppsala regular model checking tool (URMC) [1, 2, 15], the TLV[P/T] tools [11, 10, 14], and the PAX tool [3, 4, 13].

In order to achieve integration of the above mentioned tools, we have taken the following steps: (1) We have proposed a formal model suitable for describing systems with a somehow parameterized (process and/or data) structure. The model has been based on automata extended by generalized, parametric arrays [6]. This model naturally combines with the original formal basis of the IF language [7, 12], which is a common model description language being designed in the ADVANCE project. The proposed model is also close to the formalisms on which the considered parametric verification tools are built. (2) The IF language has been extended with syntactical constructs allowing automata with parametric arrays to be practically used. (3) The URMC and TLV[P/T] tools have been endowed by input modules working over (fragments of) the extended IF language. For the PAX tool, such an extension is planned.

In the following, we first introduce the proposed notion of automata with arrays. Next, we mention the corresponding extension of the IF language. Finally, we comment on the parametric verification tools URMC and TLV[P/T] and their interfaces from the extended IF language.

2 Automata with Parameterized Arrays

As a common conceptual basis for describing a large class of systems with a somehow parameterized (process and/or data) structure, we have proposed *automata working with generalized, parametric arrays* [6]. Indeed, entries of such arrays with parameterized bounds may hold control states and local data of processes whose parametric number appears in the modelled systems. Parameterized arrays may also directly correspond to parameterized memory structures (as, e.g., cache tables) present in the systems being investigated.

The considered arrays 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. In this way, we cover a wide class of systems based on structures like lines, grids, cubes, rings, trees, or their combinations. Clearly, processes organized, e.g., in a linear way may be numbered, and their control states and local data may be stored in parametric linear arrays and made accessible via the numbers of processes. On the other hand, processes organized in a tree may be identified by the corresponding paths from the root process, in which we always say via which son process of the just encountered process we go on.

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. An automaton with arrays can then be viewed as an extended automaton equipped with a number of generalized, parametric arrays and with transition actions that allow for a manipulation of such arrays. A key part of such actions is quantification over entries of arrays. The quantification allows one to say that the local data of *some* process are used, that *all* memory cells must contain a certain value, etc.

An advantage of the notion of automata with parametric arrays is that it naturally combines with the original formal basis of the IF language [7, 12] intended to be a common modelling language for different verification tools (not only) in the ADVANCE project. Moreover, this model is also close to the formalisms on which the considered parametric verification tools are built (i.e. regular automata and transducers, WS1S, etc.).

Finally, in [6], we have also proposed 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 this case, we explicitly specify the architecture of the system at hand in the form of a special generalized array in whose entries particular processes are supposed to work. The processes can then use specialized simple ways for addressing their neighbors, etc.

3 Extension of the IF Language

The common model description language IF [7, 12] provides actually several novel features allowing for a description of parameterized systems. The most important additions are listed below. With these extensions, the IF language provides a concrete and almost complete syntax for describing automata with arrays as proposed in [6]. For the time being, networks of automata that may be considered a higher-level language definable in terms of the basic model of automata with arrays have not yet been implemented.

3.1 Tree-Type Definitions

In addition to arrays and strings, tree types as defined in [6] have been introduced. Complete regular trees have only been taken into account. They may be characterized by two integer parameters: (1) the height H and (2) the branching factor F. The elements in the tree have all the same type.

The concrete syntax is the following:

```
<type-def> := tree [ <constant-body> , <constant-body> ]
                    of <type-name> |  ...
```

As an example, we may have the following definitions that define a binary tree `T` with height `H` and elements of type `A` which, in turn, is an array with length `L` of integers.

```
type A = array[L] of integer;
type T = tree[H,2] of A;
```

In order to allow for dealing with terms constructed over tree variables, the expressions syntax has been extended too. It is now possible to access tree elements using a concrete indexed notation

```
base-expression [ index-expression ]
```

where `base-expression` is a tree reference and `index-expression` is a *compatible tree path*. If the given tree has height H and branching factor F, compatible tree paths are strings of length at most H with integer elements ranging from 0 to $F-1$.

In the concrete example above, tree elements can be accessed using strings of length at most `H` with elements `0` and `1`.

3.2 Extended Transitions with Quantified Variables

The second main improvement of the IF language has been the extension of transitions with quantified variables. Both universal and existential quantifiers are allowed. The concrete syntax is as follows:

```
<transition> := [ <qvar-decl> {, <qvar-decl> }* ]
                [ deadline { eager | lazy | delayable } ; ]
                [ provided <expression> ; ]
                [ when <constraint> ; ]
                [ <input> ]
                [ <block> ]
                terminator

 <qvar-decl> := { exists | forall } <var-name> <type-name>
```

This extension increases significantly the expressive power of the language. Complex transformations of structures like arrays or trees can now be described easily in one transition.

To illustrate the use of quantified variables, we present below a simplified version of the Dijkstra's mutual exclusion protocol[1] written in the concrete syntax of the IF language. A graphical representation of the corresponding automaton with parametric arrays is depicted in Fig. 1. For a better understanding, the IF model is commented.

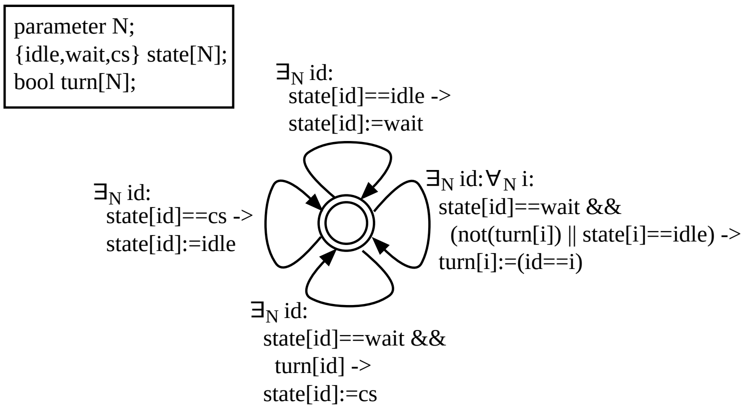

Figure 1: A simple parameterized mutual exclusion protocol

```
system mutual_exclusion;

process atm_with_arrays(1);

/* Parameter N represents the number of processes. */
/* The default value for finite-state evaluation is 10. */
const N = 10;

type index = range 0 .. N;
```

[1]The example is taken from [5].

```
/* TS is an array of control states of the processes. */
type S = enum idle, wait, cs endenum;
type TS = array[N] of S;

/* TB is an array of data specific to the processes. */
type TB = array[N] of boolean;

var st TS;    /* st stands for state. */
var turn TB;

state q0 #start ;

   /* Any process identified via id may go from 'cs' to
    * 'idle'. This is implemented by a change of its local
    * state stored in the id entry of st.
    */
   exists id index:
   provided st[id] = cs;
     task st[id] := idle;
     nextstate q0;

   /* Any process may go from 'idle' to 'wait'. */
   exists id index:
   provided st[id] = idle;
     task st[id] := wait;
     nextstate q0;

   /* Any waiting process identified via id whose turn[id] is
    * set to true may enter the critical section.
    */
   exists id index:
   provided st[id] = wait and turn[id];
     task st[id] := cs;
     nextstate q0;

   /* A waiting process given via id may enter the critical
    * section if every process is either idling or its turn is
    * not set. The value of turn[id] is set to true, for all
    * other processes it is set to false.
    */
   exists id index, forall j index:
   provided st[id] = wait and (not turn[j] or st[j]=idle);
     task turn[j] := (id = j);
     nextstate q0;

endstate;

endprocess;

endsystem;
```

4 The Uppsala Regular Model Checking Tool

The Uppsala regular model checking tool (URMC) [15] is a workbench to manipulate regular relations. In particular, it implements several algorithms to compute the transitive closure of a given relation. It has a modular conception, and relies heavily on BDD's for its internal representations and operations.

Within the URMC tool, a translator of a part of the IF language (with the parameterization extensions) to the URMC's own language for reasoning about regular relations has been implemented. The subset is large enough to be able to cover various classes of protocols such as mutual exclusion protocols or cache coherence protocols. The (modified) Dijkstra's mutual exclusion protocol presented above is a concrete example of a model that can be handled.

Linear arrays of finite types with assignment to individual elements as well as comparison between elements are supported. There is a support for quantification, either existential or universal. It is also possible to compare two indices for equality.

The implementation is considered a prototype implementation to see if the (extended) IF language is suitable for describing models common in the domain where URMC is supposed to be applied. For example, the experience obtained so far indicates that more general quantifications than the ones currently present in IF could be manageable and useful. (There could be a separate quantification section for transition actions in addition to the global one.)

The IF language is planned to be used as a basis for the next version of the URMC tool, rather than having just an input translator for it. Making the prototype translator has lead to experience in how to do this.

The tool does not currently handle tree structures, but their support is planned. Another thing that is not handled is signals between processes. This functionality does not directly fit into the framework of URMC. However, even such a feature could be added with some additional work.

5 The TLV[P/T] Tools

TLV[P/T] [11, 10] is a programmable model checker developed for exploring algorithmic and deductive techniques for the verification of parameterized reactive systems. It is an extension of the TLV [9, 14] programmable model checker which, in turn, is based on the SMV tool developed at CMU.

In its basic functionality, TLV[P] (like TLV) accepts two input files. A file `sys.smv` contains a description of the system to be verified, written in an extended version of the SMV input language. The second file `script.pf` contains a script, written in an extended version of the Tlv-Basic programming language, which instructs the TLV[P] system to proceed with appropriate operations.

In its simplest form, the script file will contain invocation of special commands such as model checking a specified LTL formula over the presented system under study, or application of an appropriate deductive rule for verifying an LTL property. However, the script file may contain complete Tlv-Basic programs for manipulating WS1S formulas (or their underlying representations as finite-state automata). For example, the techniques of acceleration of transitions, such as computation of transitive closure are programmed as Tlv-Basic procedures and are included as part of the script file.

The extended SMV input language allows specification of transition systems of parameterized arrays of processes. It generalizes the original SMV input language by allowing the construct `1..N`, where `N` is a special symbol denoting the system's parameter. Consequently, program variables may have the following types: finite-range (including the Booleans), natural numbers in the range `1..N`, and arrays of size `N` ranging over bounded types. Only one parameter `N` is allowed per system.

Presenting systems and parameterized protocols within the SMV input language is quite tedious and awkward. The user has the choice between specifying the dynamics of each module by either specifying initial and next value for each variable, or writing an explicit transition relation for each module. Therefore, a project for translating appropriately restricted SPL [8] programs into the TLV[P] input language has been launched. SPL is quite a rich programming language which allows communication by shared variables, synchronous, and asynchronous message-passing. So far, a communication by shared variables and synchronous message-passing has been planned for implementation.

The IF interface for TLV[P] is based on the ideas incorporated in the SPL to TLV[P] translator. In the first version, we do not allow asynchronous communication. However, even this first version is already able to cover rich classes of systems such as mutual exclusion protocols (including, e.g., the Dijkstra's and Szymanski's protocols).

TLV[T] is an extension of TLV[P] where the underlying parameterized structure are trees rather than strings. Here the work is still under progress and an interface to IF remains an issue for the future.

References

[1] P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Regular Model Checking Made Simple and Efficient. In *Proc. 13th Int. Conf. on Concurrency Theory (CONCUR 2002)*, volume 2421 of *LNCS*, pages 116–130. Springer-Verlag, 2002.

[2] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. 14th Intern. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 555–568. Springer-Verlag, 2002.

[3] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Prameterized Networks. In *Proceedings of TACAS 2000*, volume 1785 of *LNCS*. Springer-Verlag, 2000.

[4] K. Baukus, Y. Lakhnech, S. Bensalem, and K. Stahl. Networks of Processes with Parameterized State Space. In *Proc. ICALP 2001 Satellite Workshop VEPAS 2001*, volume 50 of *ENTCS*, pages 388–402. Elsevier, 2001.

[5] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. In *Proceedings of FTRTFT 2000*, 2000.

[6] A. Bouajjani and T. Vojnar. Automata with Parameterized Arrays and Parameterized Networks of Automata. A research report of the ADVANCE project (IST FET project No. IST-1999-29082). LIAFA, University Paris 7, France, 2002.

[7] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In *Proc. 14th Intern. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 343–348. Springer-Verlag, 2002.

[8] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.

[9] E. Shahar. The TLV System and Its Applications. Master's thesis, The Weizmann Institute of Science, 1996.

[10] E. Shahar. *TLV[P] User Manual*. The Weizmann Institute of Science, Rehovot, Israel, 2002.

[11] E. Shahar. *Tools and Techniques for Verifying Parameterized Systems*. PhD thesis, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2002.

[12] WWW presentation of the IF environment, 2002.
URL: `http://www-verimag.imag.fr/~async/IF/`.

[13] The PAX tool—Parameterized systems Abstracted and eXplored, 2002.
URL: `http://www.informatik.uni-kiel.de/~kba/pax/`.

[14] TLV—Temporal Logic Verifier, 2002.
URL: `http://www.wisdom.weizmann.ac.il/~verify/tlv/`.

[15] WWW presentation of the Uppsala regular model checking tool, 2002.
URL: `http://www.regularmodelchecking.com/`.

Automata with Parameterized Arrays and Parameterized Networks of Automata

Tomáš Vojnar, Ahmed Bouajjani
LIAFA, University Paris 7
{Ahmed.Bouajjani,Tomas.Vojnar}@liafa.jussieu.fr

January 2002

1 Introduction

The goal of this study is to propose a common basis for describing a 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. Indeed, entries of such arrays with parameterized bounds may hold control states and local data of processes whose parametric number appears in the modelled systems. Parameterized arrays may also directly correspond to parameterized memory structures (as, e.g., cache tables) present in the systems being investigated.

In addition to the above, we further 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 the following in such a way that it seems not difficult to include a *language support* for them into the IF language [7, 12] that is being developed as a common model description language within the ADVANCE project. Subsequently, there may be implemented *interfaces* to the different tools for verifying systems with somehow parameterized structure—the Uppsala regular model checking tool [1, 2, 15], the TLV[P/T] tools [11, 10, 14], and the PAX tool [3, 4, 13]—whose development is also related to the ADVANCE project. 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 conveniently 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, the IF toolset [7, 12] intended for dealing with finite-state models may be extended such that it would be at least possible to simulate and/or verify some chosen instances of the modelled families of systems (after fixing all the parameters used).

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

[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.

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. In this way, we cover a wide class of systems based on structures like lines, grids, cubes, rings, trees, or their combinations. Clearly, processes organized, e.g., in a linear way may be numbered, and their control states and local data may be stored in parametric linear arrays and made accessible via the numbers of processes. On the other hand, processes organized in a tree may be identified by the corresponding paths from the root process, in which we always say via which son process of the just encountered process we go on.

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. An automaton with arrays can then be viewed as an extended automaton equipped with a number of generalized, parametric arrays and with transition actions that allow for a manipulation of such arrays. A key part of such actions is quantification over entries of arrays. The quantification allows one to say that the local data of *some* process are used, that *all* memory cells must contain a certain value, etc.

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 $a \in A$, $S(a) = (D^a, (K_1^a, ..., K_{n^a}^a))$ where $D^a \in \mathcal{D}$ and $\forall_{n^a} i : K_i^a = b_i^a \in \mathbb{N}^+ \cup P$ or $K_i^a = (r_i^a, h_i^a) \in (\mathbb{N}^+ \cup P) \times (\mathbb{N}^+ \cup P)$.[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 τ ::= $[(\forall|\exists)_b x \mid (\forall|\exists)_{r,h} x]^*$: <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 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.

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

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]

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

[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.

[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.

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.

2.3 An example

In Figure 1, we present an example of an automaton with arrays. It is a model of the simplified version of the Dijkstra's mutual exclusion protocol taken from [5].

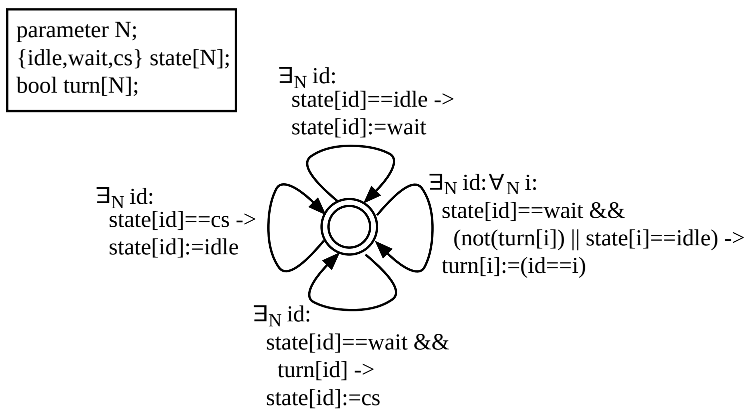

Figure 1: An example of an automaton with arrays

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]

[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 $((p), \exists_{2,h} q : 0 < |q| < h \wedge p = q)$.

- 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 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_{nf}`|`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]

[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`.

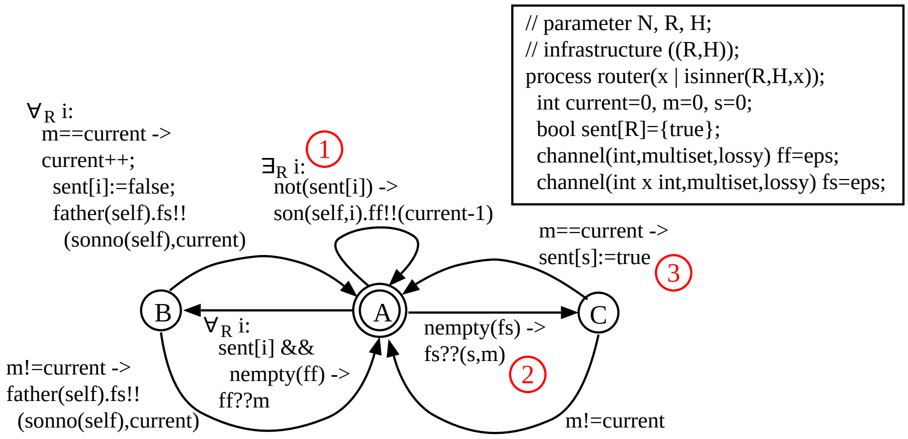

Figure 2: The automaton template for the inner nodes in the OPT network

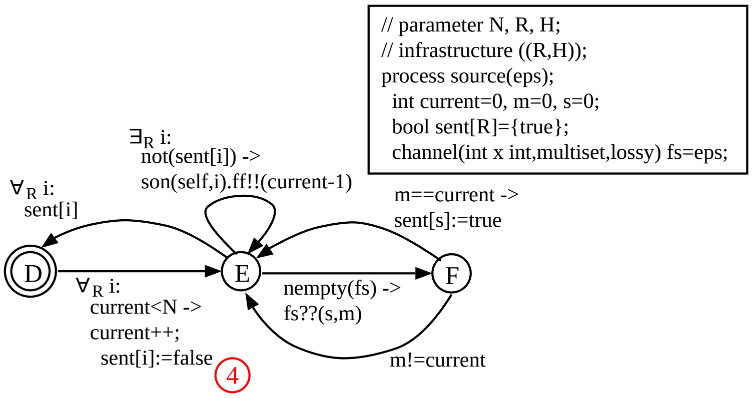

Figure 3: The automaton template for the root node in the OPT network

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.)

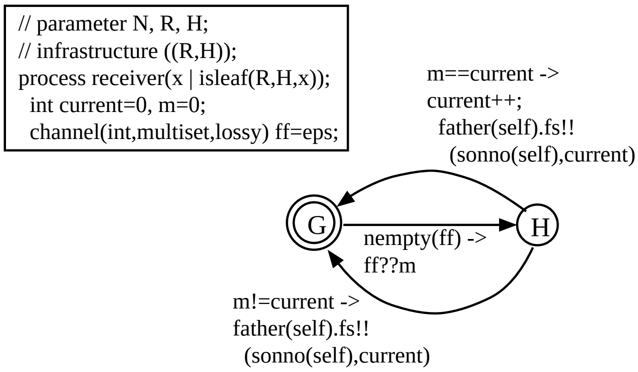

Figure 4: The automaton template for the leaf nodes in the OPT network

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 corre-

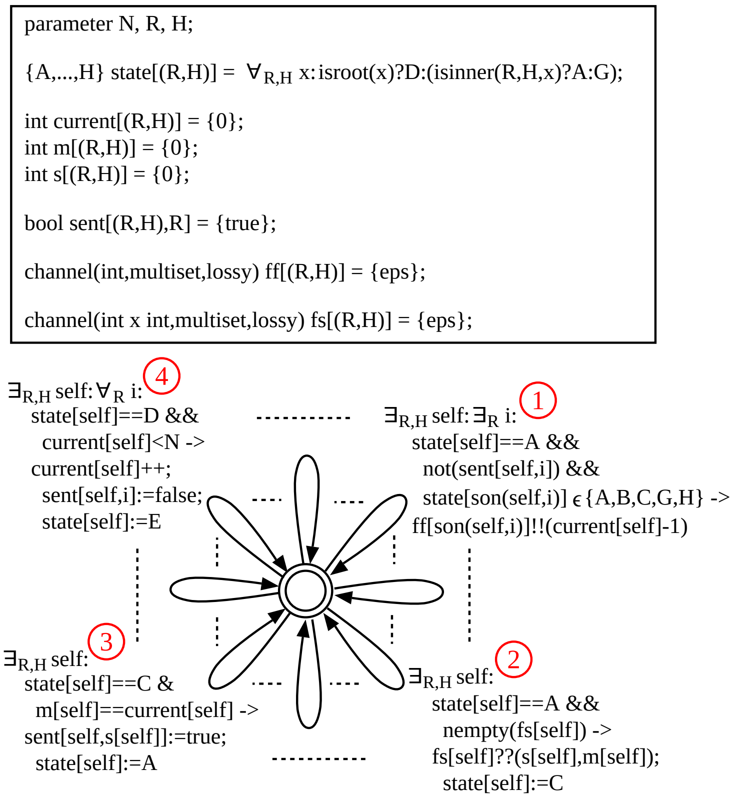

Figure 5: The OPT network folded into a single automaton (partially optimized w.r.t. the procedure described in Subsection 3.4)

sponding 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 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.

5 Conclusion

In this report, we have proposed a notion of automata working with structurally generalized, parametric arrays. Moreover, we have also proposed a higher-level notion of parameterized networks of automata. Both of these notions are designed to provide a common basis for describing a large class of systems with a somehow parameterized process and/or data structure. The first notion is easier to be included into the IF language [12] being developed in the ADVANCE project. On the other hand, the second notion can sometimes yield easier to write, read, and handle models.

For the future, it is necessary to enrich the IF language by (some of) the here proposed constructs. This will allow developers of the different tools for verification of systems with a parameterized structure [15, 14, 13] to implement interfaces from the IF language in their tools. Subsequently, it will be possible to apply different tools on the same models and to compare or complement the results.

References

[1] P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Regular Model Checking Made Simple and Efficient. In *Proc. 13th Int. Conf. on Concurrency Theory (CONCUR 2002)*, volume 2421 of *LNCS*, pages 116–130. Springer-Verlag, 2002.

[2] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. 14th Intern. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 555–568. Springer-Verlag, 2002.

[3] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Prameterized Networks. In *Proceedings of TACAS 2000*, volume 1785 of *LNCS*. Springer-Verlag, 2000.

[4] K. Baukus, Y. Lakhnech, S. Bensalem, and K. Stahl. Networks of Processes with Parameterized State Space. In *Proc. ICALP 2001 Satellite Workshop VEPAS 2001*, volume 50 of *ENTCS*, pages 388–402. Elsevier, 2001.

[5] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. In *Proceedings of FTRTFT 2000*, 2000.

[6] A. Bouajjani and T. Vojnar. Automata with Parameterized Arrays and Parameterized Networks of Automata. A research report of the ADVANCE project (IST FET project No. IST-1999-29082). LIAFA, University Paris 7, France, 2002.

[7] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In *Proc. 14th Intern. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 343–348. Springer-Verlag, 2002.

[8] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[9] E. Shahar. The TLV System and Its Applications. Master's thesis, The Weizmann Institute of Science, 1996.

[10] E. Shahar. *TLV[P] User Manual*. The Weizmann Institute of Science, Rehovot, Israel, 2002.

[11] E. Shahar. *Tools and Techniques for Verifying Parameterized Systems*. PhD thesis, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2002.

[12] WWW presentation of the IF environment, 2002.
URL: `http://www-verimag.imag.fr/~async/IF/`.

[13] The PAX tool—Parameterized systems Abstracted and eXplored, 2002.
URL: `http://www.informatik.uni-kiel.de/~kba/pax/`.

[14] TLV—Temporal Logic Verifier, 2002.
URL: `http://www.wisdom.weizmann.ac.il/~verify/tlv/`.

[15] WWW presentation of the Uppsala regular model checking tool, 2002.
URL: `http://www.regularmodelchecking.com/`.