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

Deliverables 16 and 17

Integration of Tools for the Analysis of Parameterized Systems and Dynamic Networks

Finalized Common Validation Framework

Year 3 : October 2001 – December 2003

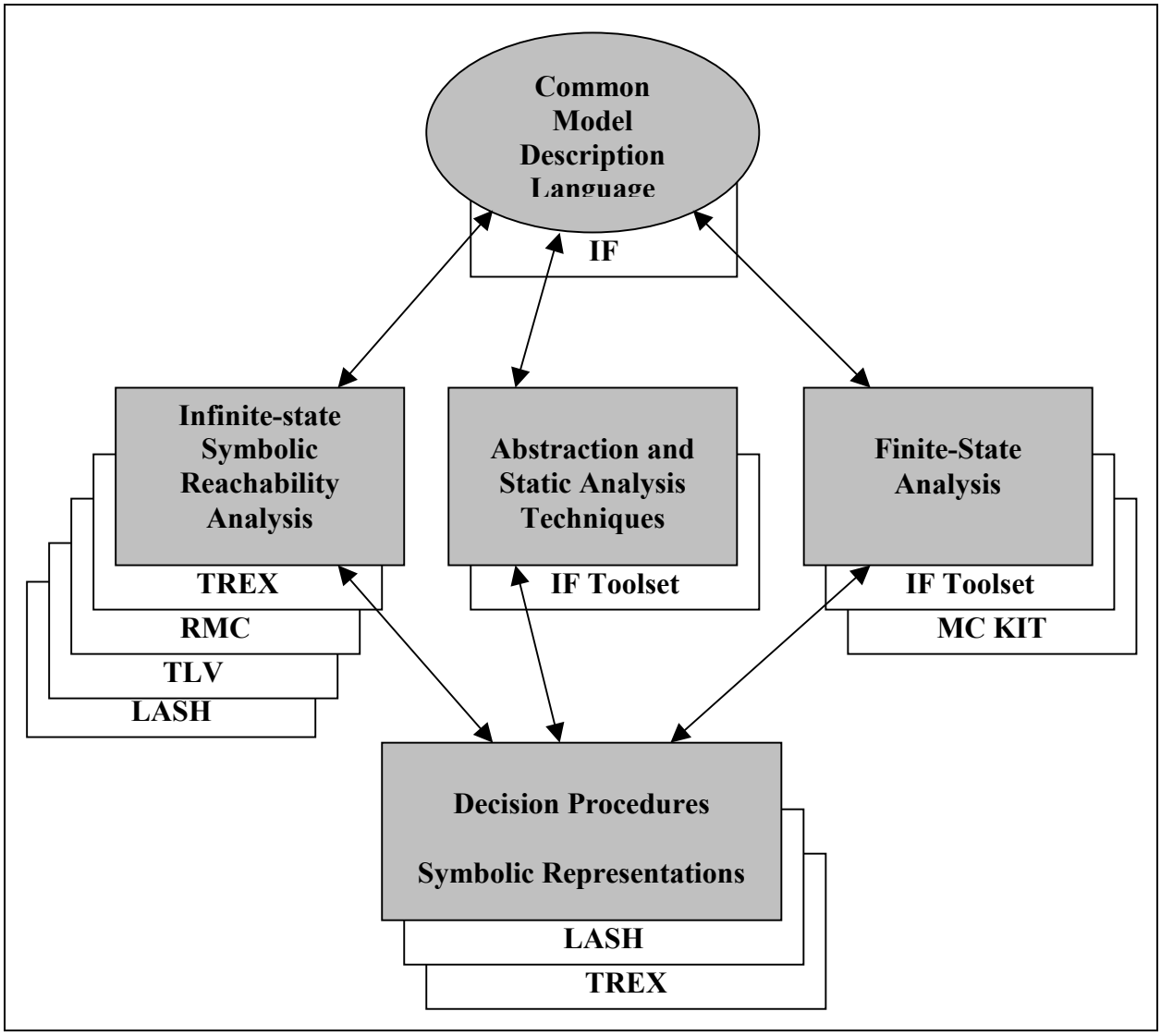
Date of preparation: December 2003

Contributors: Grenoble, Paris, Liège, Uppsala, Edinburgh, Weizmann Inst.

Introduction 3
Common Language 4
Extended Automata 4
Automata with Generalized, Parametric Arrays 5
Describing Parameterized Networks in IF 6
Parameterized Arrays and Trees 6
Index Quantification in Transitions of IF Processes 7
A Sample model: the Szymanski's Algorithm 7
Integrated Tools 9
Static Analysis Tools 9
Live variables analysis 9
Dead-code elimination 9
Variable abstraction 10
Finite State Tools 10
IF Toolset [Grenoble] 10
McKIT [Edinburgh] 10
Symbolic Reachability Tools 11
TREX [Paris] 11
LASH [Liege] 11
Parameterized Tools 11
RMC [Uppsala] 11
TLV [Weizmann] 12
Graphical Interface 12
References 14

Introduction

This report describes the validation framework developed in the ADVANCE project. The overall architecture is depicted in the figure below. The central point of the framework is the common model description language which is an extension of the IF language [2] with constructs for describing parameterized systems. All the framework tools are connected through this common language. They are classified according to their main functionality into *symbolic reachability and parameterised tools* (TREX[3], LASH[4], RMC[5], TLV[8]), *abstraction and static analysis* (IF Toolset [2]) and *finite state analysis tools* (IF Toolset[2], MC KIT [1]). Finally, the tools also share a layer containing symbolic representations and decision procedures allowing the manipulation of basic structures (automata, constraints, etc).



During the last period, the work done concerns essentially the integration of parameterized tools and experimentation. First of all, the syntactical extensions proposed to the common language have been effectively implemented. These extensions are derived from the formal model of automata with generalized, parametric arrays proposed last year [6]. Second, a base of common examples have been created using the common language and includes typical benchmark protocols such as Szymanski's protocol, Burns protocol, Peterson protocol, etc. Finally, the parameterized tools, RMC and TLV, have been upgraded and now, both of them

provides operational front-ends to the common language. They are currently intensively tested on common benchmarks, and the first experiments have been successful.

This report is organized as follows. First, we recall the features of the common modelling language. We will focus on the extensions defined for the modelling of parameterised systems. Subsequently, we will briefly present the tools connected in this framework and how the connections are effectively made. Finally, we will give an overview of the graphical interface developed on top of the framework.

Common Language

The common language proposed is an extension of the IF language developed at Verimag in order to model asynchronous communicating real-time systems. Since its definition in 1998, the IF language has been continuously improved and becomes the interchange format between a set of validation tools dedicated to real-time systems i.e, the so-called IF validation environment. In this project, we have extended the IF language to be able to model and analyse parametrized systems as well as systems that rely upon dynamic process creation and dynamic data types.

Extended Automata

The formal basis for the common language is a dynamic version of *extended automata*.

We focus on systems composed from several components, hereafter called processes, running in parallel and interacting through point-to-point message-passing. The number of processes may change over time: they may be created and destroyed dynamically using specific actions, during the system execution.

Each process is an extended timed automaton. They have unique process identifier (pid) values and local memory consisting of variables (including clocks), control state and a queue of pending messages (received and not yet consumed). As usual, processes move from one control state to another by executing transitions. In a control state, transitions are triggered by the presence of some message in the input queue and/or some (possibly timed) condition on variables. In addition, transition bodies are sequential programs consisting of elementary actions such as variable assignments, clock settings, message sending, process creation/destruction, etc.

The semantics of the extended automata model is given by the graph of its executions. This graph is obtained by the interleaved execution of processes, that is, one process is executing a transition at a time, while the others are waiting. In other words, we consider process transitions to define atomic, non-interruptive, execution steps.

The semantics of time is as for timed automata: time may progress only in states (i.e, all running processes wait in some state before selecting and executing some transition) and transitions take 0 time to be executed. In order to control the time progress, or equivalently, the waiting time in states, we rely on transition urgencies -- explicit deadlines attached to them defining when these transitions must be executed.

The common language inherits from IF some of its main features, in particular:

- **structuring concepts**: systems consist of processes, running in parallel and communicating through message passing via communication buffers; communication primitives: more exactly, processes may communicate either through signal exchange (directly or via signalroutes) or through shared variables;

- **real-time primitives**: each process may use several clocks to measure time during the execution and, in addition, transitions may be guarded with time constraints (depending on clocks) and decorated with explicit (eager, delayable, lazy) deadlines;
- **open systems**: the language offers the concept of open communication channel, connected to the environment, and transporting messages between it and the system;
- **non-determinism**: processes may be non-deterministic i.e, more than one transition may be enabled at some control state, and all situations have to be considered at execution;
- **complex data types**: the language provide several type constructors such as enumeration, range, array, record, abstract as well as predefined basic types in order to simplify complex data description and manipulation;

In order to fit the new requirements, the common language improves the IF language in several directions, the most important being listed below:

- **parameterization**: detailed in the next section
- **dynamic features**: the language include dynamic creation and destruction of process and signalroute (channel) instances. This makes system configuration to be dynamic, that is, the number of components running (and in turn, the number of clocks) may change during execution;
- **structured control**: the common language integrates *hierarchical states* (to structure automata) and, *composed transitions*: basic control statements such as if-then-else and while-do are provided to structure automata transitions;
- **external code integration**: the common language provide a simple an elegant way to abstract complex transformations on data through the integration of external code within procedures. The external code to be provided depend on tools used i.e, an executable implementation in order to simulate and model check.

Automata with Generalized, Parametric Arrays

The notion of automata working with generalized, parametric arrays that serves as a formal basis of the further described extension of the IF language for dealing with parametric systems was proposed in [6] where its detailed description can be found. 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. In this way, a wide class of systems based on structures like lines, grids, cubes, rings, trees, or their combinations is covered. 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. Unlike the original proposal, the automata being described by the below IF syntax can contain both global quantification whose scope covers the guard and action of a transition as well as local quantification whose scope is restricted to either the guard or the action.

Describing Parameterized Networks in IF

In the remaining of this section we focus on the new extensions of the IF language for modelling parameterized networks of processes. It is based on the above described notion of automata with generalized, parametric arrays. The generalized arrays are practically supported via using parameterized linear arrays and parameterized trees that can be arbitrarily nested as described in the preceeding section. The way these arrays and trees can be manipulated using index quantifiers is then described below. Compared to the first version of the IF extension for modelling parameterised networks, the possibility of local index quantification has been introduced. This distinction allows one to create models easier to read and easier to be manipulated in the tools for parametric verification.

Parameterized Arrays and Trees

We now proceed to the concrete syntax that can be used for for describing parameterized arrays and trees in IF. Complete regular trees are only taken into account---irregular trees may be simulated by storing some special values into some of their nodes. Let us note that the syntactic category <*constant-body*> may contain a named constant. Named constants are a way how parameters are introduced to IF---the value they are assigned in a model is taken into account by simulators and finite-state verification tools and ignored by parametric verification tools. The concrete syntax of the arrays and trees in IF is the following:

<*type-def*> :=

array [<*constant-body*> **] of** <*type-name*> |

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***;**

The syntactical constructions used for accessing elements of arrays are as usual. In order to allow for dealing with terms constructed over tree variables, the expressions syntax uses an 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.

Index Quantification in Transitions of IF Processes

The IF syntax allows both global and local quantification over variables used for indexing arrays and trees. Global quantification is restricted to transitions whereas local quantification is restricted to expressions. The concrete syntax for global quantification 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>*

A Sample model: the Szymanski's Algorithm

In order to illustrate the use of the new IF syntax for describing parameterized networks of processes, we now present a concrete, non-trivial example. It is a slightly idealized version of the Szymanski's algorithm for mutual exclusion of a parametric number of processes. A pseudo-code description of the algorithm is given below. The array *pc* may be viewed to record the current control locations of the particular processes.

1: **await** ∀j : pc[j] ∈ {1,2,4}

2: **skip**

3: **if** ∃j. pc[j] ∈ {2,5, 6, 7}

then goto 4

else goto 5

4: **await** ∃j: pc[j] ∈ 5, 6, 7}

5: **await** ∀j: pc[j] ∉ 3,4}

6: **await** ∀j: j < i: pc[j] ∈ 1,2,4}

7: the critical section; **goto** 1

Below, we give an encoding of the Szymanski's algorithm in the IF syntax. The model is highly commented to allow an easy understanding.

```
system szymanski;

/* The system is described as one automaton with arrays. */

process atm_with_arrays(1);

/* We declare parameter N. The value 10 is intended for
 * enumerative tools, it may be ignored by parameterised tools. */

const N = 10;

/* We declare a type range based on N, which will be used to index
```

```
 * the processes in the system and to access their local data stored
 * in parameterized arrays. */

type index = range 0 .. N;

/* We declare the control locations array of length N. */

type L = enum l1, l2, l3, l4, l5, l6, l7 endenum;
type TL = array[N] of L;

/* Here is the declaration of the state variable, above
 * there are just type declarations. */

var tl TL;

/* Here are the transitions of the automaton. In this case,
 * they are all self loops of some q0 control state. */

state q0  start ;

  /* Each transition is globally quantified by (possibly alternating) quantifiers
   * on variables. Here, we only choose the process to proceed. */

  exists id index:

  /* We now specify a condition under which the transition may be fired. It may
   * contain local quantifiers. Here, we ask whether all processes are in
   * certain control locations. */

  provided  tl[id] = l1 and forall j index: (tl[j]=l1 or tl[j]=l2 or tl[j]=l4);

    /* We specify the action that take place when the transition is fired.*/

    task tl[id] := l2;
    nextstate q0;

  /* Here ends the first transition. */

  exists id index:
  provided tl[id] = l2;
    task tl[id] := l3;
    nextstate q0;

  exists id index :
  provided  tl[id] = l3 and exists j index:
            (tl[j]=l2 or tl[j]=l5 or tl[j]=l6 or tl[j]=l7);
    task tl[id] := l4;
    nextstate q0;

  exists id index :
  provided  tl[id] = l3 and forall j index:
            not (tl[j]=l2 or tl[j]=l5 or tl[j]=l6 or tl[j]=l7);
    task tl[id] := l5;
    nextstate q0;

  exists id index :
  provided  tl[id] = l4 and exists j index: (tl[j]=l5 or tl[j]=l6 or tl[j]=l7);
    task tl[id] := l5;
    nextstate q0;

  exists id index :
  provided  tl[id] = l5 and forall j index: not (tl[j]=l3 or tl[j]=l4);
    task tl[id] := l6;
    nextstate q0;

  exists id index :
  provided  tl[id] = l6 and forall j index:
            (j>=id or tl[j]=l1 or tl[j]=l2 or tl[j]=l4);
```

```
    task tl[id] := l7;
    nextstate q0;

  exists id index:
  provided tl[id] = l7;
    task tl[id] := l1;
    nextstate q0;
endstate;
endprocess;
endsystem;
```

Integrated Tools

Static Analysis Tools

Our experience with IF has shown that simplification by means of static analysis is crucial for dealing successfully with real specifications. Moreover, even simple analysis such as live variables analysis or dead-code elimination can significantly reduce the size of the state-space of the model.

By moving up to the extended common language we must review all these static analysis. Several novel features of the language such as dynamic creation and destruction of processes and communication channels, plus the private/public scope of variables make them much more difficult to apply than in the previous version. In IF, specifications were completely static, the set of processes, queues, and variables is fixed and apriori known.

The following analysis have been adapted to work on the common language and are part of the integrated environment:

Live variables analysis

This analysis transforms an IF specification into an equivalent smaller one by removing globally dead variables and/or signal parameters and by adding systematic resets of locally dead variables.

Initially, all the private variables and signal parameters are supposed to be dead, unless otherwise specified by the user. Public variables are supposed to be always live. The analysis alternates local (standard) live variables computation on each process and inter-process live propagation through input/output signal parameters until a global fixpoint is reached.

Dead-code elimination

This analysis transforms an IF specification by removing unreachable control states and control transitions under some user-given assumptions about the environment. More precisely, the user may indicate the set (by default empty) of signals the environment may sent to the system at execution.

This analysis solves a simple static reachability problem : it computes, for each process separately, the set of control states (transitions) which may be statically reached (executed) starting from the initial control state. Meantime, it computes the set of processes that may have running instances: either they have instances in the initial configuration or some corresponding fork action may be reached and therefore executed. When done, processes without instances are removed from the specification. The other processes are restricted to the set of reachable states and transitions.

Variable abstraction

This analysis provides a simple and efficient way of computing abstractions by eliminating variables and their dependencies from a specification. Some initial set of *abstract variables* have to be specified by the user. Then, using data and input/output dependencies, the largest set of variables which *may* be influenced by these variables is computed, at each control state. The computation proceeds as for live analysis: processes are analyzed separately, and the results obtained are propagated between them using the input/output dependencies. Finally, actions and guards involving abstract variables are removed from the specifications

Contrarily to previous analysis which are exact, variable abstraction may introduce more behaviors. Nevertheless, it always simplifies the state vector, and therefore, we can extract automatically abstracted systems allowing for symbolic verification with tools like TREX[2] or LASH[3] which make strong assumptions about the input (systems with counters, clocks, queues only). Moreover, if we are interested in explicit enumerative finite-state model checking, this technique allows to reduce the overall number of states in the semantic model.

Finite State Tools

IF Toolset [Grenoble]

IF is an open validation platform for asynchronous timed systems such as telecommunication protocols or distributed applications developed at Verimag during the last 5 years.

The toolset is built upon an specification language based on timed automata extended with discrete data variables, various communication primitives, dynamic process creation and destruction. This language is expressive enough to represent most useful concepts of modeling and programming languages for distributed systems (like SDL, UML, Java, ...).

The core of the toolset consists of a set of finite-state model-based validation components including exhaustive/interactive simulation, on-the-fly temporal logic model-checking, test case generation and optimal pathextraction. In order to control state explosion, the IF toolset provides several static analysis tools operating at the source level such as live variable analysis, dead-code elimination and slicing. Finally, the toolset is connected to commercial environments (such as Rational Rose, Rhapsody, Objecteering, Object Geode) and may be used for validating SDL and UML specifications.

McKIT [Edinburgh]

The Model-Checking Kit [1] allows to model finite-state systems using a variety of input languages, translate them for use with a variety of model-checkers (including PEP, SPIN, PROD, SMV), and translate the result back into the terms of the original model. Previously, Edinburgh had added support for IF as an input language (see deliverable D4).

Since most model-checkers featured in the Kit work on finite-state systems, the IF interface of the Kit only supports a finite-state fragment of the language, excluding clocks and other infinite data types. Therefore, the interface has now been extended by a filter. When an IF model uses data types not supported by the Kit (such as a clocks), the Kit can use the Slice tool (developed by Grenoble) to obtain a projection which does not use variables of these types. In such a case, the Kit informs the user about the necessary changes and checks the abstracted model instead.

Symbolic Reachability Tools

TREX [Paris]

TREX [3] (*Tool for Reachability Analysis of Complex Systems*) is a tool for reachability analysis of extended automata. More precisely, the model supported is a (finite) set of (finite control) automata with parameters (i.e. uninstantiated constants), clocks, counters, and lossy fifo queues. The release 1.3 of TREX includes an IF language compiler.

The compiler acts in two phases: filtering and translation. The filtering phase consists in checking that the IF description can be handled by TREX. For instance, complex data types (i.e., records, arrays, trees), procedures, and communication buffers other than lossy fifo queues are not supported. If the check fails, it outputs a report indicating which features are not supported and the variables that should be sliced in order to obtain an abstract description which can be handled by TREX. If the check is successful, the IF description is translated into the input language of TREX. During the translation, a semantic analysis detects automatically the parameters of the model (i.e., variables which are not assigned).

LASH [Liege]

The Liege Automata-based Symbolic Handler (LASH) [4] is a tool for building and manipulating finite-state representations sets, and exploring the state space of infinite-state systems. The release 0.9 of LASH include an IF language compiler.

The IF language elements associated to features that are not handled by LASH are ignored, wherever possible. For instance, because the current representation system used by the state-space exploration module of LASH operates on integer vectors, declarations of non-integer variables and operations involving these variables --- although syntactically accepted --- are abstracted out of the compiled model.

The input language is extended with features that are essential to LASH, but that cannot be expressed in IF. The main extension concerns the definition of *meta-transitions* which are necessary to exploring infinite-state spaces with a finite amount of resources.

Parameterized Tools

RMC [Uppsala]

The Uppsala team has implemented part of the IF language with the parameterized extensions as a translator to their own language for reasoning about regular relations. The subset is large enough to cover many classes of parameterised systems, such as mutual exclusion algorithms e.g. Peterson algorithm.

The considered fragment of IF supports arrays of finite types, with assignment to individual elements as well as comparison between elements. It also supports quantification, either existential or universal. It is also possible to compare two indices for equality.

More recently, the Uppsala team improved the algorithms used in the RMC tool with a more powerful equivalence used for partitioning when computing the transitive closure. This is described in a paper published at CAV'2003, which also provides a simplified theoretical framwork for deriving equivalences that can be used in the tool [7].

The translation tool can be found as a part of the regular model checking toolset, and is available from the web site http://www.regularmodelchecking.com/.

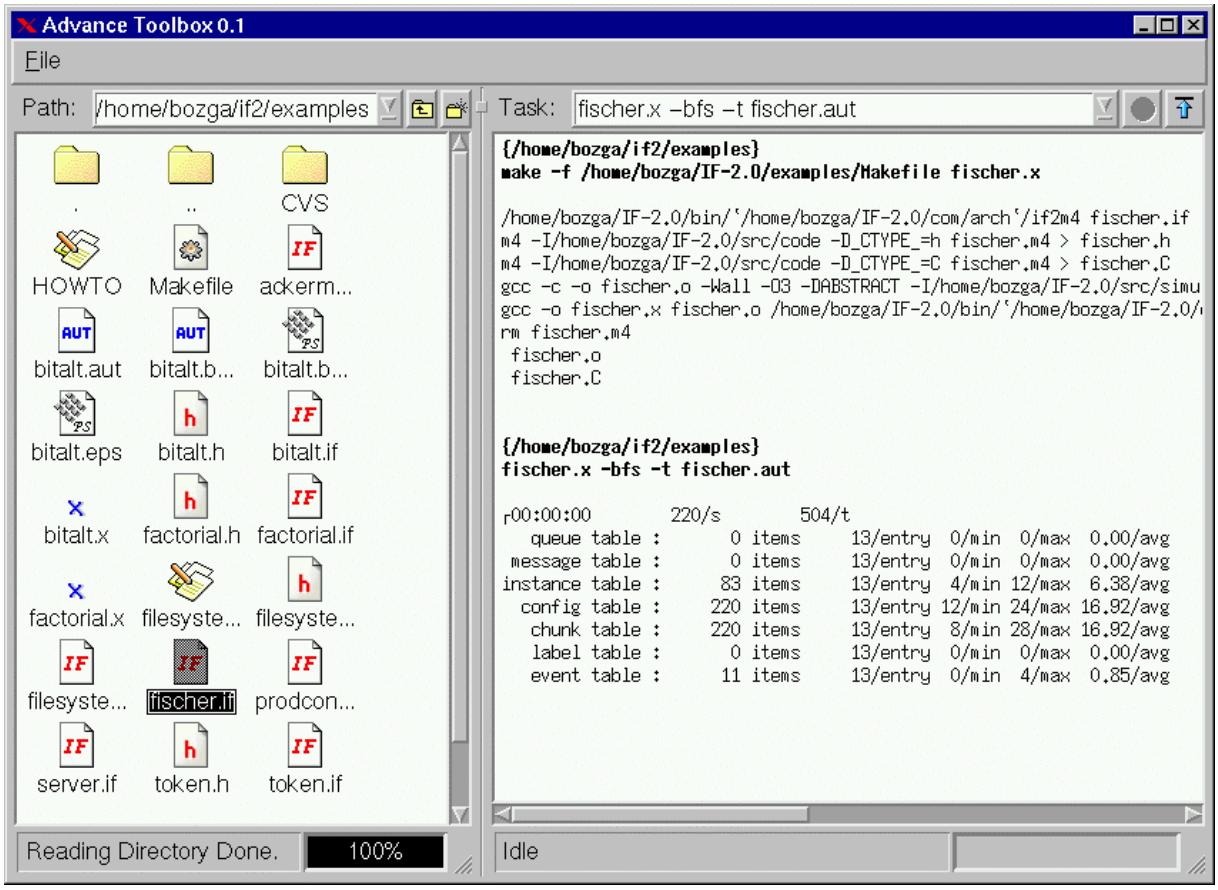
TLV [Weizmann]

The same approach has been also followed by the Weizmann team which implemented the translation of the parameterized part of the common language into the input language of the TLV tool [8]. The considered fragment is essentially the same as for RMC and includes arrays of finite types, basic operations on array elements, quantification on transitions and expression, etc.

Graphical Interface

We have implemented a first prototype of a integrated validation environment. It is built upon the common language IF and relies on two application programming interfaces:

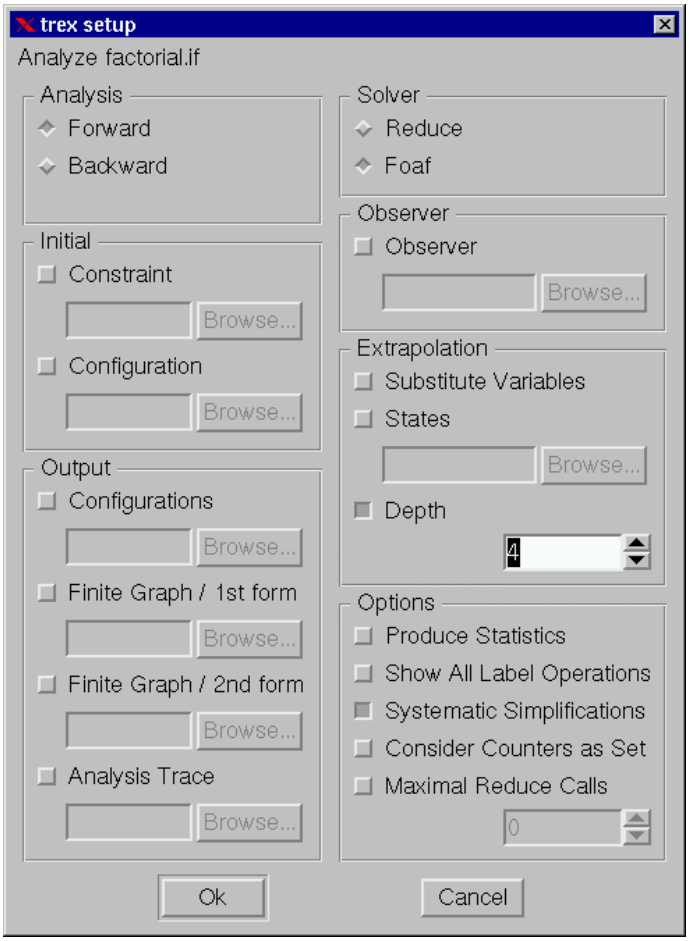
- The common language IF comes equipped with a parser allowing to construct the abstract syntax tree from textual specifications. The tree is constructed as a collection of C++ objects and can be accessed, modified or deleted by some end-user application.
- The exploration interface gives access to the semantic model of IF specifications. Semantic models are labelled transition systems. Basically, the interface provides functions to compute 1) *the initial state* and 2) *the set of successors* for some given state. Using these functions, it is possible to construct the models *on-the-fly* in an explicit manner



Currently, the environment provides *general* functionalities such as file browsing, file editing or task management as well as *specific* validation functionalities such as explicit model construction, symbolic analysis (using TREX and LASH), model minimisation (using Aldebaran), model visualisation (using dot). The figure below gives an overview of the main

window. By clicking on IF files, the tool will open a menu from where the user can invoke one of the available tools. For each tool, the environment provide a configuration window where the user may choose the right parameters for his IF file.

The next figure presents the TREX configuration window. It is rather complex but provides a complete control over all the TREX running options (analysis method, input and output files, extrapolation, initial constraints, backend solver, etc). Similar windows exist for all the currently integrated tools.



The environment is completely open, in particular, new functionalities or tools can be added with little effort. The environment (tools and interface) could be freely downloaded from the Advance project Web site http://verif.liafa.jussieu.fr/~haberm/ADVANCE/main.html.

References

1. C. Schroter, S. Schwoon and J. Esparza. **The Model Checking Kit**. In *Proceedings of Tools Day Workshop* (CONCUR'02), Masaryk University, Brno, FIMU-RS-2002-05. See also http://wwwbrauer.in.tum.de/gruppen/theorie/KIT
2. M. Bozga, S.Graf, L. Mounier. **IF-2.0: A Validation Environment for Component-Based Real-Time Systems**. In *Proceedings of CAV'02, Copenhagen, Demark*. LNCS 2404. See also http://www-verimag.imag.fr/~async/IF
3. A. Annichini, A. Bouajjani, M.Sighireanu **TreX : A tool for Reachability Analysis of Complex Systems**. In *Proceedings of CAV'01, Paris, France*. LNCS 2102. See also http://www.liafa.jussieu.fr/~sighirea/trex
4. B. Boigelot, L. Latour. **The Liege Automata-based Symbolic Handler (LASH)**. Available at http://www.montefiore.ulg.ac.be/~boigelot/research/lash
5. M. Nilsson, Julien d'Orso. **Regular Model Checking (RMC) tool**. Available at http://www.regularmodelchecking.com/
6. A. Bouajjani and T. Vojnar. **Automata with Parameterized Arrays and Parameterized Networks of Automata.** A research report of the ADVANCE project. LIAFA, University Paris 7, France 2002.
7. Parosh Aziz Abdulla, Julien d'Orso, Bengt Jonsson, and Marcus Nilsson. **Algorithmic Improvements in Regular Model Checking**. In Proc. 15th Int. Conf. on Computer Aided Verification, Lecture Notes in Computer Science, 2003.
8. E. Shahar and A. Pnueli**.** **TLV: Temporal Logic Verifier**. Available at http://www.wisdom.weizmann.ac.il/~verify/tlv/