XTL: A Meta-Language and Tool for Temporal Logic Model-Checking

R. Mateescu [⋆] and H. Garavel [‡]

[⋆] *CWI / SEN2 group*
413, Kruislaan
NL-1098 SJ Amsterdam, The Netherlands
Radu.Mateescu@cwi.nl

[‡] *INRIA Rhône-Alpes / VASY group*
655, avenue de l'Europe
F-38330 Montbonnot St. Martin, France
Hubert.Garavel@inria.fr

Abstract

We present a temporal logic model-checking environment based on a new language called XTL (*eXecutable Temporal Language*). XTL is a functional programming language designed to allow a compact description of various temporal logic operators, which are evaluated over a Labelled Transition System (LTS). XTL offers primitives to access the data values (possibly) contained in the states and labels of the LTS, as well as to explore the transition relation. The temporal logic operators are implemented by means of iteration expressions computing sets of states and sets of transitions. Various useful modal and temporal logics like HML, CTL, LTAC and ACTL, have been implemented in XTL, and several industrial case-studies, such as the BRP protocol designed by Philips, the IEEE-1394 serial bus standardized by IEEE, and the CFS protocol developed by Bull and INRIA, have been successfully validated using the XTL model-checker.

1 Introduction

The last years have witnessed an increasing application of formal methods in the design and validation of complex applications, such as communication protocols and distributed systems. One of the most popular techniques of program verification is the so-called *model-checking*. In this approach, the application is first described using an appropriate high-level language, such as LOTOS[1] [17] or μCRL[2] [12]. Next, the program is translated into a Labelled Transition System model (LTS for short), over which the desired correctness properties, expressed as temporal logic formulas, are verified by means of specialized tools called *model-checkers*.

The literature concerning this area is very rich in results: a large variety of temporal logics have been defined, allowing to capture different kinds of correctness properties, and several corresponding model-checking algorithms have been proposed. Also, numerous tool environments allowing verification by model-checking have been developed, such as EMC [4], CWB [6], SPIN [15], TAV [19], MEC [1], JACK [3], and Concurrency Factory [5], to mention only a few of them.

However, many of the currently available tools are either dedicated to a particular description language and/or temporal logic (e.g., the language PROMELA [15] used in

[1]Language of Temporal Ordering Specification
[2]*micro* Common Representation Language

Spin, the logic Actl [23] used in Jack, etc.), or they are based on a particular model-checking algorithm (e.g., the boolean resolution algorithm [2] used in Mec). Therefore, most of these tools have limited applicability in different contexts, and their adaptation to another setting may be overwhelming in terms of time and implementation effort.

Another important issue is the handling of data values, both at the level of the description language and of the temporal logic. For instance, the Lts models corresponding to value-passing description languages as Lotos or μCrl contain data values in the states and/or transition labels. This requires the ability to express and verify temporal properties involving data, e.g., "*after a message* `m` *has been sent, the same message* `m` *will be eventually received.*" Although studied in the theorem-proving approach [26], this issue has received little attention in the setting of automated model-checking.

In this paper, we present an approach to temporal logic model-checking that attempts to reduce the shortcomings mentioned above. Our method is based on a meta-language called Xtl (*eXecutable Temporal Language*), which is a functional programming language designed to allow a compact description of temporal logic operators. We use the term "meta-language" to emphasize that Xtl allows not only to handle, in a uniform way, the data objects (i.e., types and functions) defined in the program to be verified, but also the states, transitions, and labels of the corresponding Lts model. Furthermore, since Xtl is a programming language, it can be used to define non-standard temporal operators and, more generally, to perform any computation on an Lts model (e.g., to calculate the branching factor, print the list of labels, etc.).

The Xtl model-checker has been developed as part of the Cadp (Cæsar/Aldébaran Development Package) protocol engineering toolset [9]. We describe here the version 1.1 of the Xtl tool, which is currently integrated in Cadp[3].

The paper is organized as follows. Section 2 gives an overview of the Xtl language and shows various examples of temporal logic operators implemented in Xtl. Section 3 briefly describes the architecture of the Xtl model-checker. Section 4 presents several applications to industrial case-studies. Finally, Section 5 contains some concluding remarks and directions for future work.

2 Overview of the XTL language

In this section, we first describe the (extended) Lts models over which Xtl programs are interpreted, and next we present the basic Xtl constructs, illustrating their use by means of various examples. Due to space limitations, we cannot describe here in full detail the whole Xtl language. A more detailed presentation can be found in [22] and in the technical documentation of the Cadp toolset.

2.1 Labelled Transition Systems

In order to verify temporal properties of programs written in value-passing description languages as Lotos and μCrl, we must naturally use an adequate representation of the corresponding Lts models, the states and labels of which may contain data values. Such a representation is available within the Cadp toolset as a special file format called Bcg (*Binary Coded Graphs*) [10]. A Bcg file representing the Lts model of a program to be verified (denoted by "source program" in the remainder of the paper) contains essentially the following informations:

[3]The Cadp toolset can be obtained at the Url `http://www.inrialpes.fr/vasy/cadp.html`.

- A set of *states*, each of them being represented as a tuple containing the values of all the program variables (the so-called *state-vector*). An initial state is identified.
- A set of *actions* (also called *labels*), each of them being represented as a list of typed values. In BCG files generated from LOTOS programs, the labels have the form "$G\ v_1 \dots v_n$," where G is a gate name.
- A *transition relation*, represented as a list of *transitions* encoded as tuples of the form (s_1, a, s_2), each of them indicating that the program can move from state s_1 to state s_2 by performing action a.

Besides the above elements, a BCG file generated from a source program contains also a *type area* and a *function area* that give access to the types and functions defined in the source program, respectively. Throughout this section, we implicitly consider an LTS model represented in BCG format, over which the XTL constructs will be interpreted.

2.2 Types, expressions, and functions

As we mentioned earlier, XTL allows to handle, in a uniform way, the data values used in the source program, as well as the elements of the corresponding LTS model. To achieve this, the XTL language allows to define and use objects belonging to the following types.

external types: These are the types defined in the source program; they are exported by the type area of the BCG file encoding the LTS. The data values belonging to these types can be handled using the functions defined in the source program, which are exported by the function area of the BCG file.

internal types: These are the types predefined in XTL. The standard predefined types (`boolean`, `integer`, `character`, etc.) are provided, together with their usual operators. Beside these types, there are also the so-called *meta-types* `stateset`, `state`, `edgeset`, `edge`, `labelset`, and `label`, denoting the (sets of) states, transitions, and labels of the LTS, respectively. These types are equipped with the *meta-operators* given in Table 1, which allow to access the initial state of the LTS and to explore the transition relation (some of these operators are inspired from Dicky's calculus [8]).

OPERATOR	MEANING
`init : -> state`	initial state
`succ, pred : state -> stateset`	successors and predecessors of a state
`in, out : state -> edgeset`	incoming and outgoing transitions of a state
`source, target : edge -> state`	origin and destination states of a transition

Table 1: Basic XTL meta-operators

The basic XTL expressions are shown in Table 2. Function calls may be either in prefix, or infix notation (in the case of binary operators, such as the predefined operations "`+`", "`*`", "`<=`", etc.). The label-matching expression returns a result of type `boolean`. The construct enclosed in its brackets (called *action pattern*) allows to examine the structure of a transition label of the LTS and (possibly) to extract the values of its fields and bind them to variables. Quantifiers over finite domains and comprehensive set definitions have a syntax close to their usual mathematical notation.

The most simple way to implement in XTL temporal operators expressing action or state properties is to compute their denotational semantics, i.e., the sets of LTS labels or states satisfying them. For example, the following expression computes the set of labels corresponding to the emission of a signal with different source and destination addresses (identifiers are in upper-case letters and keywords in lower-case):

```
{ L:label where L -> [ SIGNAL ?S:Addr ?D:Addr where S <> D ] }
```

The variables `S` and `D`, initialized by pattern-matching with the corresponding values contained in the label, are used in the "**where**" clause, which allows additional filtering using a boolean condition.

EXPRESSION	MEANING
$F(E_1, ..., E_n)$	prefix function call
$E_1\ F\ E_2$	infix function call
E `->` `[` G `?`x`:`T `!`E_1 [**where** E_2] `]`	label matching
exists x:T **in** E **end_exists**	existential quantifier
forall x:T **in** E **end_forall**	universal quantifier
{ x:T **where** E }	set in comprehension
let x:T:=E **in** E_1 **end_let**	variable definition
if E **then** E_1 **else** E_2 **end_if**	conditional
for [x_0:T_0] [**in** x_1:T_1] [**while** E_1] **apply** F **from** E_2 **to** E_3 **end_for**	iteration

Table 2: Summary of the basic XTL expressions

Used together with the quantifiers, the set definition construct allows to easily express modal operators. For instance, the $\langle\alpha\rangle\,\varphi$ modality of the Hennessy-Milner logic HML [14] can be implemented by the XTL function below:

```
def Dia (A:labelset, F:stateset) : stateset =
    { S:state where
        exists T:edge among out (S) in
            (label (T) among A) and (target (T) among F)
        end_exists
    }
end_def
```

The parameters `A` and `F` denote the sets of labels and states satisfying α and φ, respectively. The function call "`Dia (A, F)`" returns the states satisfying $\langle\alpha\rangle\,\varphi$, i.e., the states having an outgoing transition whose label satisfies α and whose target state satisfies φ.

The "**let**" and "**if**" constructs shown in Table 2 have their usual meaning (e.g., as in ML). The evaluation of the iteration construct "**for**", which allows to perform repeated computations, proceeds as follows. We first assume that the declaration $x_0\!:\!T_0$ is present, but the "**in**" and "**while**" clauses are absent. The semantics of "**for**" uses an implicit variable v_{acc} (called *accumulator*) initialized with the value of E_2. For each value of x_0 (called *iteration variable*) in the finite domain T_0, an iteration is performed, that consist in evaluating the expression $F(v_{acc}, E_3)$ and assigning it to v_{acc} (note that F must be a binary function). The result of the "**for**" expression is the value of v_{acc} after the last iteration. For example, the following XTL expression computes the maximal branching factor (i.e., the maximal number of transitions going out of a state in the LTS):

```
for S:state
    apply max
    from  0
    to    card (out (S))
end_for
```

where `max` denotes the maximum of two integer numbers and `card` gives the number of elements of a transition set.

Optionally, the "**in** $x_1 : T_1$" clause allows to give a name x_1 to the accumulator v_{acc} so that it can be referenced in E_1 and/or E_3. If present, the "**while** E_1" clause allows to control the execution of the "**for**" expression: the iterations are performed as long as the boolean expression E_1 (re-evaluated before each iteration) remains true. An absence of the iteration variable x_0 means a "forever" loop: in this case, the iterations are stopped using the "**while** E_1" clause (which must be present in order to ensure termination).

Using the "**for**" construct, temporal operators can be defined in a compact form. Thus, the following XTL function implements the operator $\mathbf{EF}_\alpha\varphi$, which is a derived modality of ACTL [23]:

```
def EF_A (A:labelset, F:stateset) =
    for in    X:stateset
        while X <> (F or Dia (A, X))
        apply or
        from  false
        to    F or Dia (A, X)
    end_for
end_def
```

A state satisfies $\mathbf{EF}_\alpha\varphi$ if it is the origin of a path leading, via zero or more actions satisfying α, to a state satisfying φ. This can be characterized as the least solution of the fixed point equation $X = \varphi \vee \langle\alpha\rangle\, X$, that is iteratively computed by the "**for**" expression above. Note the overloading of the boolean operators `or` and `false`, that denote the union and the empty set of states, respectively. Alternately, the $\mathbf{EF}_\alpha\varphi$ operator could be defined using a recursive XTL function.

2.3 Macros, libraries, and programs

In order to express temporal properties involving data conveniently, a higher-order mechanism for handling predicates containing free variables is needed. For this purpose, we incorporated in XTL a macro-expansion mechanism, which covers most practical user needs and can be implemented simply and efficiently. The following example of XTL macro-definition implements the $[\alpha]\,\varphi$ modality of HML, characterizing the states from which all outgoing actions satisfying α lead to states satisfying φ:

```
macro Box (A, F) =
    { S:state where
        forall T:edge among out (S) in
            if T -> [ A ] then target (T) among F else true end_if
        end_forall
    }
end_macro
```

The `A` and `F` parameters above denote (the textual representation of) an action pattern and an expression of type `stateset`. A macro call "`Box (<textA>, <textF>)`" is replaced in the XTL program by the body of the macro, in which the occurrences of `A` and `F` have been textually substituted with `<textA>` and `<textF>`. For instance, the following XTL macro call evaluates the set of states from which every message `M` sent on gate `SEND` can be potentially received on gate `RECV`:

```
    Box (SEND ?M:Msg, EF_A (true, Dia (RECV !M, true)))
```

where `Dia` is a macro implementing the $\langle \alpha \rangle \varphi$ modality and `EF_A` is the function defined in Section 2.2. The variable `M`, which extracts the message contained in the `SEND` labels, is visible in the second argument of the `Box` operator; this is ensured by the static semantics of the "**if**" expression used in the body of the `Box` macro-definition above. Note also that the type `Msg` is external, i.e., it is defined in the source program.

XTL allows the macro-definitions to be overloaded: several macros having the same name, but different number of parameters, may be used in the same scope. This is convenient for defining derived temporal operators having the same name, for instance the $\mathbf{pot}(\varphi_1, \varphi_2)$ and $\mathbf{pot}(\varphi)$ operators of LTAC [25], which are similar to the $\mathbf{E}[\varphi_1 \, \mathbf{U} \, \varphi_2]$ and $\mathbf{EF} \, \varphi$ operators of CTL [4], respectively.

Another useful feature is the possibility to include in an XTL program other XTL files, typically containing libraries of temporal operators. This allows to reuse existing XTL code and also to provide different implementations of the same temporal logic (see Section 4). For example, the following construct denotes the textual inclusion of an XTL source file implementing the ACTL temporal logic:

```
    library actl.xtl end_library
```

An XTL program consists of an expression (the program's body) preceded by an optional list of macro-definitions and library inclusions.

3 Implementation

We developed a model-checker for XTL as part of the CADP protocol engineering toolset. The tool takes as input an XTL program and an LTS model encoded in BCG format, evaluates the program over the LTS and produces the results.

The architecture of the model-checker is shown in Figure 1. First, the XTL program is processed by an auxiliary tool called *expander*, that textually expands the macro-definitions and includes the XTL libraries used in the program. The resulting program (containing "pure" XTL code, i.e, without macro calls) is translated into a C program, which is then compiled and linked with the BCG libraries. Note that the information contained in the BCG file is used also during the static analysis, since the types and functions defined in the source program (exported by the BCG file) can be used in the XTL program. The object file obtained in this way is executed and the results are obtained on the standard UNIX output stream.

The version 1.1 of the XTL model-checker is available on SUN workstations running SUNOS or SOLARIS and PCs running LINUX. The syntax analyzer has been implemented using the SYNTAX[4] compiler generator. The semantics analyzer has been written in LOTOS abstract data types, which are translated into C code using the CÆSAR.ADT compiler of the CADP toolset. The expander, the code generator, and all the interfacing

[4]SYNTAX is a trademark of INRIA.

code with the Bcg environment have been written in C. The overall implementation consists of about 27,000 lines of code.

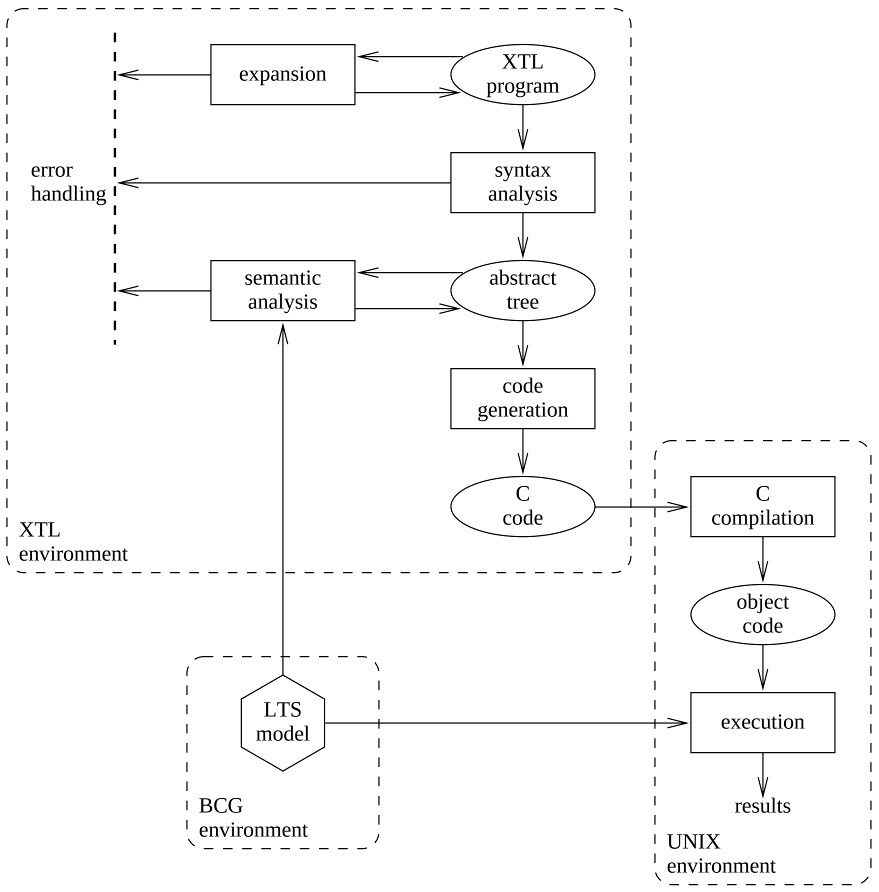

Figure 1: Architecture of the Xtl model-checker

Besides developing the model-checker, we also provided Xtl libraries implementing the operators of Hml [14], Ctl [4], Ltac [25], Actl [23], and the modal μ-calculus [18]. All these operators can be naturally used in conjunction with the built-in Xtl data-handling facilities in order to express temporal properties involving data.

4 Applications

An initial version 1.0 of the Xtl model-checker has been used to verify several Lotos descriptions of small size, such as the alternating bit protocol, a leader election algorithm, and various mutual exclusion algorithms. These exercises, together with the experience of using Xtl for teaching purposes at the University Joseph Fourier of Grenoble, provided valuable feedback, enabling us to improve the tool from the initial version 1.0 to the current version 1.1. So far, this version of the Xtl model-checker has been used to validate three medium-sized industrial case-studies.

Bounded Retransmission Protocol: The Brp protocol has been designed by Philips and is currently used in the remote control devices of television sets. It implements the transmission of (large) data packets over an unreliable communication medium by splitting them in (small) chunks that are sent sequentially. Whenever a chunk is lost, it is retransmitted over the communication medium. If a (fixed) maximal number of retransmissions is reached, the protocol gives up the transmission of the packet, appropriately informing the sender and the receiver. This protocol was proposed by Jan Friso Groote (Cwi, Amsterdam) as a verification exercise intended for the comparison of several formal methods.

Starting from a μCrl description given in [13], we produced a Lotos description of the Brp protocol, for which we identified a set of 21 safety and liveness properties, expressed in Xtl using the library of Actl operators combined with data. These properties have been successfully verified, using the Xtl model-checker, on different instances of the protocol, obtained by giving different values to the maximal number of retransmissions and to the packet length [21].

Link Layer of the IEEE-1394 Serial Bus: The Ieee-1394 serial bus ("FireWire") is a high-speed bus particularly adapted to data transmission for multimedia devices connected to computers. This bus, standardized by Ieee, is currently used by numerous constructors, such as AT&T, Canon, Compaq, Hewlett-Packard, Ibm, Kodak, Microsoft, Sony, Texas Instruments, etc.

We carried out the validation of the asynchronous part of the link layer protocol of Ieee-1394. Based upon a μCrl description provided by Bas Luttik [20] and upon the Ieee standard [16], we produced an E-Lotos description of this part of the protocol, which was subsequently translated in Lotos using the Traian prototype compiler of the Cadp toolset. We translated in Xtl (using the library of Actl operators extended with data) the 5 correctness properties of the protocol stated in natural language by Luttik. These properties have been verified, using the Xtl tool, on several instances of the protocol, obtained for different numbers of nodes connected to the bus and for various message scenarios. This allowed us to detect and correct a potential deadlock occurring in the protocol after about 50 transitions from the initial state [27].

Cluster File System: Cfs is a distributed file system developed by Bull and Inria on top of the Arias shared memory architecture [7]. Cfs was designed both to validate the Arias system itself and to experiment with distributed applications that use shared files as a programming model.

The validation of the migratory file coherency protocol of Cfs (referred to as the Cfs protocol in the sequel) has been recently carried out by Charles Pecheur. First, he produced a Lotos description modelling both the Cfs protocol and the Arias service primitives used by it. Next, he specified a set of 15 safety, liveness, and coherency properties (expressed as Actl formulas with data) of the control level (i.e., involving only the calls to Cfs synchronization primitives) and of the data level (i.e., taking into account also the access and modification of the data files). Finally, he implemented in Xtl a new library of Actl operators, able to produce diagnostic sequences explaining the truth value of a formula using the Xtl tool. These properties have been verified on various scenarios of Cfs, obtained for different application configurations on top of the Cfs protocol [24].

These experiments confirm the usefulness of the Xtl language: temporal properties involving data can be expressed in a natural way using directly the notations of the source program. Although the action-based operators of ACTL are often enough powerful to express the safety and liveness properties encountered in practice, there are situations (e.g., some of the properties of the BRP protocol) that can be handled in an easier way using more powerful constructs, such as regular expressions. These can be implemented in XTL by means of their fixed point characterizations.

5 Conclusion and future work

Formal methods have proved their usefulness in the design of complex, distributed applications. Among these methods, model-checking verification techniques are simple to use and completely automated, although limited to finite-state systems.

We presented in this paper a model-checking environment based on a special language called XTL, dedicated to the description of temporal properties involving data. A model-checker for XTL has been developed, and several widely-used temporal logics like HML, CTL, LTAC, ACTL, and the μ-calculus, have been implemented in XTL. The version 1.1 of the model-checker is available as part of the CADP protocol engineering toolset, and has been successfully used to validate several industrial case-studies [21, 27, 24].

These experiments are encouraging, confirming the advantages of the approach adopted in designing XTL, which allows to combine temporal operators and data-handling constructs. Moreover, since XTL is a programming language, it allows the user to implement new temporal logics or to extend existing ones with new operators. Indeed, Charles Pecheur developed in XTL a new library of ACTL temporal operators, able to produce diagnostic sequences [24].

The work presented here can be extended in several directions. Firstly, our experience shows that additional data types (such as sequences and subtrees of the LTS) are needed in order to facilitate the implementation of temporal operators with diagnostic features. Secondly, there is still room for improving the performances of the model-checker: XTL being a functional language, the code generator should be optimized using appropriate storage allocation techniques for the variables containing sets of states and transitions of the LTS. Finally, the implementation of on-the-fly model-checking algorithms, which do not require to generate entirely the LTS *before* evaluating a temporal formula, can be envisaged along the lines described in [22], by using the OPEN/CÆSAR approach to on-the-fly verification [11].

References

[1] A. Arnold, D. Bégay, and P. Crubillé. *Construction and Analysis of Transition Systems with MEC.* World Scientific, 1994.

[2] A. Arnold and P. Crubillé. A Linear Algorithm to Solve Fixed-Point Equations on Transition Systems. *Information Processing Letters*, 29:57–66, 1988.

[3] A. Bouali, S. Gnesi, and S. Larosa. The Integration Project for the JACK Environment. *Bulletin of the EATCS*, 54:207–223, 1994.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[5] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: a Development Environment for Concurrent Systems. In R. Alur and T. A. Henzinger, editors, *Proceedings of CAV'96*, LNCS 1102, pages 398–401, 1996.

[6] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench.* In J. Sifakis, editor, *Automatic Verification of Finite State Systems*, pages 24–37. LNCS 407. 1989.

[7] P. Dechamboux, D. Hagimont, J. Mossiere, and X. Rousset de Pina. The Arias Distributed Shared Memory: an Overview. LNCS 1175, 1996.

[8] A. Dicky. An Algebraic and Algorithmic Method for Analysing Transition Systems. *Theoretical Computer Science*, 46(2-3):285–303, 1986.

[9] J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In R. Alur and T. A. Henzinger, editors, *Proceedings of CAV'96*, LNCS 1102, pages 437–440, 1996.

[10] H. Garavel. Binary Coded Graphs — Definition of the BCG Format (version 1.0). Technical report, INRIA Rhône-Alpes, 1995.

[11] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proceedings of TACAS'98*, LNCS, 1998. Full version available as INRIA Research Report RR-3352.

[12] J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, 1990.

[13] J. F. Groote and J. C. van de Pol. A Bounded Retransmission Protocol for Large Data Packets. Technical Report Logic Group Preprint Series 100, Utrecht University, 1993.

[14] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32:137–161, 1985.

[15] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[16] IEEE. Standard for a High Performance Serial Bus. IEEE Standard 1394-1995, Institution of Electrical and Electronic Engineers, 1995.

[17] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO — OSI, Genève, 1988.

[18] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[19] K. G. Larsen. Efficient Local Correctness Checking. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of CAV'92*, LNCS 663, pages 30–43, 1992.

[20] B. Luttik. Description and Formal Specification of the Link Layer of P1394. In I. Lovrek, editor, *Proceedings of the 2nd COST 247 Int. Workshop on Applied Formal Methods in System Design*, 1997. Also available as CWI Report SEN-R9706.

[21] R. Mateescu. Formal Description and Analysis of a Bounded Retransmission Protocol. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 Int. Workshop on Applied Formal Methods in System Design*, pages 98–113. University of Maribor, Slovenia, 1996. Also available as INRIA Research Report RR-2965.

[22] R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* PhD Thesis, Institut National Polytechnique de Grenoble, 1998. To appear.

[23] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In *Proceedings of Semantics of Concurrency*, pages 407–419. LNCS 469. 1990.

[24] C. Pecheur. Advanced Modelling and Verification Techniques Applied to a Cluster File System. Research Report 3416, INRIA Rhoône-Alpes, 1998.

[25] J-P. Queille and J. Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[26] J. Rathke and M. Hennessy. Local Model Checking for a Value-Based Modal μ-calculus. Report 5/96, School of Cognitive and Computing Sciences, University of Sussex, 1996.

[27] M. Sighireanu and R. Mateescu. Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS. *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, 1998. To appear.