A Verification Methodology for Infinite-State Message Passing Systems *

Christoph Sprenger
Projet Lemme, INRIA Sophia Antipolis
2004 route des Lucioles, BP 93
06902 Sophia Antipolis, France
sprenger@sophia.inria.fr

Krzysztof Worytkiewicz
2 av. des Planches
1820 Montreux
Switzerland
krisw@bluewin.ch

Abstract

The verification methodology studied in this paper stems from investigations on respectively deduction-based model checking and semantics of concurrency. Specifically, we consider imperative programs with CSP-like communication and use a categorical semantics *as foundation to extract from a program a* control graph *labelled by* transition predicates. *This logical content acts as system description for a* deduction-based model checker *of LTL properties. We illustrate the methodology with a concrete realisation in form of the **Mc5** verification tool written in Ocaml and using the theorem prover PVS as back-end.*

1. Introduction

Formal verification of concurrent systems has been studied for more than two decades and is still considered as far from being simple. The difficulties stem essentially from the conjunction of two characteristics most concurrent systems exhibit: non-deterministic behaviour on one hand and huge state spaces on the other. The first characteristic makes pure *proof-theoretic* approaches too tedious. The second one, also known as *state space explosion*, eliminates even state-of-the-art automatic *model-checking*, at least as far as realistic applications are of concern. One remedy for this situation is to get the best of both proof-theoretic and *model-theoretic* worlds by using model-checking algorithms as *decision procedures* within a theorem prover [16]. We advocate an alternative approach where model checkers are devised to operate by application of *deduction rules*. The latter are designed to incrementally construct a *proof object*, possibly associated to some *validity conditions* of logical nature. The approach is still *model-based* since the deduction rules in question are formulated w.r.t. a class of models. Model checkers based on deductive methods are able to cope with systems of arbitrary size. However, they require expert knowledge to be taken advantage of.

In our case, the models are graphs representing control, with edges labelled by *transition predicates*. They are extracted from a program text written in the concurrent programming language **Mc** including imperative features and synchronous message-passing communication infrastructure à la CSP [10]. We apply a categorical semantics of the language to extract the transition predicates in question, a novel approach to the best of our knowledge. The individual *processes* and communication channels occurring in a program determine a *diagram* in a category [12] where the objects are essentially graphs with edges labelled by spans i.e. pairs of functions with common domain. The composed behaviour is calculated as a *limit* of this diagram. Finally, the transition predicates are calculated from the spans by *image factorisation*, the categorically inclined reader will identify the whole construction as a *reflection*. This kind of semantics scales up to more complex object-based languages [21] and has the advantage of being *compositional*.

Given a model, our set of deduction rules is designed to construct a *proof structure* i.e. a graph with nodes labelled with *sequents* associating *control points* with logical content in form of predicates and LTL formulas [14]. Once constructed, a proof structure *per se* does in general *not* represent a *proof*. In order to do so, it needs to be validated by proving *side conditions* and *discharge conditions.* The former guarantee the *coherence* of the proof structure w.r.t. to the model while the latter guarantee that no run in the system is a counterexample to the LTL property to be proved. Discharge conditions are constructed from a Büchi automaton associated to the proof structure along with user-supplied *rankings,* which can be understood as a measure of failure potential. Side conditions and discharge conditions are first-order predicates. Due to the nature of the models however, it turns out to be convenient to work in a higher-order logic framework. This temporal deduction system is novel to the best of our knowledge and scales up to CTL*

*main part of work done while both authors were members of the Computer Networking Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland

$$\textbf{Asg}\ \frac{\Gamma, \mathtt{x} : \tau \vdash \mathtt{t} : \tau}{\Gamma, \mathtt{x} : \tau \vdash \mathtt{x} := \mathtt{t} : \langle\,\rangle} \qquad \textbf{Nop}\ \frac{}{\mathtt{nop} : \langle\,\rangle}$$

$$\textbf{Out}\ \frac{\Gamma \vdash \mathtt{t} : \tau}{\Gamma \vdash \mathtt{c!t} : \langle \mathtt{c} : \tau \rangle} \qquad \textbf{In}\ \frac{}{\Gamma, \mathtt{x} : \tau \vdash \mathtt{c?x} : \langle \mathtt{c} : \tau \rangle}$$

Table 1. Well-typed atomic statements.

$$\textbf{If}\ \frac{\Gamma \vdash \mathtt{p} : \mathtt{bool} \quad \Gamma \vdash stat_1 : \Delta \quad \Gamma \vdash stat_2 : \Theta \quad \Delta \otimes \Theta : \mathtt{ok}}{\Gamma \vdash \mathtt{if}\ \mathtt{p}\ \mathtt{then}\ stat_1\ \mathtt{else}\ stat_2\ \mathtt{end} : \Delta \otimes \Theta}$$

$$\textbf{Whl}\ \frac{\Gamma \vdash \mathtt{p} : \mathtt{bool} \quad \Gamma \vdash stat : \Delta}{\Gamma \vdash \mathtt{while}\ \mathtt{p}\ \mathtt{do}\ stat\ \mathtt{end} : \Delta}$$

$$\textbf{Seq}\ \frac{\Gamma \vdash stat_1 : \Delta \quad \Gamma \vdash stat_2 : \Theta \quad \Delta \otimes \Theta : \mathtt{ok}}{\Gamma \vdash stat_1 ; stat_2 : \Delta \otimes \Theta}$$

Table 2. Well-typed compound statements.

and models including fairness constraints [19].

It is fair to say that this work stems from a productive blend of two generally disjoint disciplines. The concepts sketched above led us to devise the verification tool **Mc5**, written in Ocaml [11] and using PVS [17] as back-end for proving side and discharge conditions.

The paper is structured as follows: section 2 elaborates on the language **Mc** and its semantics, section 3 is about the temporal deduction system, section 4 presents the **Mc5** tool, while concluding remarks are to be found in section 5. We include an appendix containing an example proof of a liveness property.

2. A Concurrent Programming Language

The concurrent programming language **Mc** acts as input language for the temporal deduction tool to be introduced in section 4. It is essentially a variant of concurrent Pascal with message-passing, enriched by a rigorously designed theory of types essentially reflecting the connectivity of the communication channels.

2.1. Syntax

The rules for well-typed **Mc** programs in-context are summarised in tables 1, 2, 3 and 4. Types are lists of *interface points* or *ip*'s, the latter being pairs associating a type to a name. The operator $\otimes$ on types performs list concatenation followed by removal of duplicates while the `ok`-judgement (on types as well as on contexts) says that no name occurs twice. Finally, notice that the ground types as well as the ground function symbols are taken in an *algebraic theory* [5].

The type system reflects the interfaces of the programs to their environment. Specifically, two ip's with matching types may form a *channel*. The **MCut**-rule expresses composition: its application creates an arbitrary number of *private* channels between two processes. In this sense, **MCut** roughly corresponds to *parallel composition* followed by *hiding* as encountered in conventional process calculi like CCS [15] or CSP [10].

2.2. Semantics

In this subsection, we present the semantic concepts underlying **Mc**.

Definition 2.1. (i) A *graph* G is pair (E, V) along with functions $dom, cod : E \to V$. We write $l \xrightarrow{e} m$ for $dom\,(e) = l \ \wedge\ cod\,(e) = m$ and $E\,(l, m) \stackrel{\text{def}}{=} \left\{ e \in E \,|\, l \xrightarrow{e} m \right\}$

(ii) A *relational* graph is a graph $G = (E, V)$ *s.t.* $E \subseteq V \times V$.

(iii) A *path* π in a graph $G = (E, V)$ is a (finite or infinite) sequence $v_1 e_1 \ldots v_i e_i v_{i+1} \ldots$ *s.t.* $dom\, e_i = v_i$ and $cod\, e_i = v_{i+1}$ for all indices i. The *trace* of π is the sequence $v_1 v_2 \ldots v_i \ldots$ of vertices occurring in π.

Twisted systems are graphs with edges labelled by *spans* i.e. by pairs of functions with common domain. They rely upon the concept of control locations and transitions on one

$$\textbf{MCut}\ \frac{\begin{array}{ll}\Gamma \vdash stat_1 : \langle \Delta, \mathtt{c}_1 : \tau_1; \cdots; \mathtt{c}_n : \tau_n \rangle & \Delta \otimes \Theta : \mathtt{ok} \\ \Xi \vdash stat_2 : \langle \Theta, \mathtt{d}_1 : \tau_1; \cdots; \mathtt{d}_n : \tau_n \rangle & \Gamma, \Xi : \mathtt{ok}\end{array}}{\Gamma, \Xi \vdash stat_1 \, \| c_1 \asymp d_1, \ldots, c_n \asymp d_n \| \, stat_2 : \Delta \otimes \Theta}$$

Table 3. The composition rule.

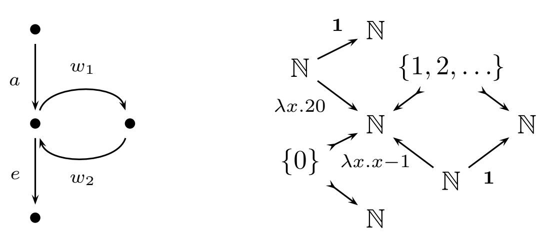
$$\textbf{Weak}\ \frac{\Gamma \vdash stat : \Delta}{\Gamma, x : \tau \vdash stat : \Delta} \qquad \textbf{Perm}\ \frac{\Gamma \vdash stat : \Delta}{\pi_1 \Gamma \vdash stat : \pi_2 \Delta}$$

Table 4. The structural rules.

hand and on (possibly explicit) state and computation on the other. In line with the underlying intuition we dub the graph *control,* its vertices *control points* and its edges *transitions*. Twisted systems are a platform of choice for handling semantics of imperative programs with CSP-style message-passing. For instance, the **Mc**-program (without communication)

$$\begin{array}{l} \mathtt{x} : \mathtt{nat} \vdash \\ \quad \mathtt{x} := 20; \\ \quad \mathtt{while}\, \mathtt{x} > 0 \,\mathtt{do}\, \mathtt{x} := \mathtt{x} - 1 \mathtt{end} : \langle\rangle \end{array}$$

can be translated to the twisted system



In particular, the program's syntactic locations are one-to-one with the control points and each transition carries a computation reflecting an assignment or the evaluation of a branching condition. Given a twisted system arising from a meaning of an **Mc**-program, there is a set D s.t. the labels of all transitions are of type $D \times D \to \{\mathsf{tt}, \mathsf{ff}\}$. D is a product where each factor corresponds as usual to an individual program variable.

Labelling of graphs with spans can be formulated in terms of graphs, (lax) functors and (lax) natural transformations determining a category $\mathbf{Ts}(\mathbf{Sets})$ with *finite limits*, the latter being the ingredient required for the semantics of the **MCut**-rule [23, 22]. However, a simpler category $\mathcal{T}(\mathbf{Sets})$ turns out to be more convenient for the present setup.

Proposition 2.2. *Let*

$$\mathbf{Osp} = (\mathbf{Osp}_0, \mathbf{Osp}_1, dom, cod, comp)$$

be the category having spans as objects and the obvious triples of functions as morphisms. The category $\mathcal{T}(\mathbf{Sets})$ given by the data

- *Objects:* (G, s) *with* $G = (E, V)$ *a reflexive graph and* $s : E \to \mathbf{Osp}_0$
- *Morphisms:* $(G, s) \xrightarrow{(h,\kappa)} (H, t)$ *with* $h = (h_0, h_1) : G \to H$ *a homomorphism of reflexive graphs and* $\kappa : E \to \mathbf{Osp}_1$ *s.t. everything commutes in*

$$\begin{tikzcd}
\mathbf{Osp_0} & E \arrow[l, "s"'] \arrow[r, shift left, "dom"] \arrow[r, shift right, "cod"'] & V \arrow[dd, "h_0"] \\
\mathbf{Osp_1} \arrow[u, "dom"] \arrow[d, "cod"'] & E \arrow[l, "\kappa"'] \arrow[u, "id"'] \arrow[d, "h_1"] & \\
\mathbf{Osp_0} & E' \arrow[l, "t"'] \arrow[r, shift left, "dom"] \arrow[r, shift right, "cod"'] & V'
\end{tikzcd}$$

- *Composition:* $(l, \kappa') \circ (h, \kappa) = (l \circ h,\, comp \circ \langle \kappa,\, \kappa' \circ h_1 \rangle)$
- *Units:* $id_{(G,s)} = (id_G,\, K_1)$ *where* $K_1(e) = id_{s(e)}$

has finite limits.

It is not hard to see how finite limits are calculated in $\mathcal{T}(\mathbf{Sets})$ and that the constructions can be carried out in the internal language of $\mathbf{Sets}$.

2.3. Systems

In this subsection we introduce the format of the systems to be used by the temporal deduction engine.

Definition 2.3. A *system*

$$\mathcal{S} = (T, C, D, dom, cod, \rho)$$

is a graph with edges labelled by predicates where $dom, cod : T \to C$ are the graph data and $\rho : T \to \mathbf{Pred}\,(D \times D)$ is the labelling.

Systems determine a *reflective subcategory* of $\mathcal{T}\,(\mathbf{Sets})$ where the legs of the labelling spans are *jointly monic* i.e. they determine a predicate. The basic underlying fact is that logical information can be extracted from a span by *image factorisation* using the familiar correspondence $\mathbf{Sub}\,(S) \cong \mathbf{Pred}\,(S)$ between subsets of S and their *classifying predicates*:

$$\begin{array}{l} A \xleftarrow{f} X \xrightarrow{g} B \\ \quad \mapsto \quad A \times B \xrightarrow{\exists x \in X.\,(a,b)=(f(x),g(x))} \{\mathsf{tt}, \mathsf{ff}\} \end{array}$$

In other words, any twisted system admits an associated system, canonically. The construction can be carried out in the internal language of (the topos) $\mathbf{Sets}$, essentially a higher-order logic over a simple type theory. The practice-oriented reader may safely skip this part of the discussion keeping in mind that the relevant constructions can be carried out in a higher-order logic over a simple type theory as implemented in PVS.

3. Deductive Temporal Verification

We introduce a tableau-based proof method for the verification of LTL properties of systems (stemming from **Mc**-programs). The method consists of two parts: a set of local proof rules used to construct *proof structures* and a global *discharge condition* that serves to establish that a proof structure is a proper *proof* of a property. For the remainder of this section, let $\mathcal{S} = (T, C, D, dom, cod, \rho)$ stand for an arbitrary but fixed system.

Definition 3.1. A *run* of $\mathcal{S}$ is an infinite sequence $\sigma = (l_0, d_0) \cdots (l_i, d_i) \cdots \in (C \times D)^\omega$ of *states* s.t. for all $i \in \mathbb{N}$ there is some $t \in T$ with $l_i \xrightarrow{t} l_{i+1}$ and $\rho(t)(d_i, d_{i+1}) = \mathsf{tt}$. A run σ is a *p-run* for $p \in \mathsf{Pred}(C \times D)$ provided $p\,(l_0, d_0) = \mathsf{tt}$. Finally, $\mathcal{R}\,(\mathcal{S}, p)$ stands for the set of p-runs of $\mathcal{S}$.

Definition 3.2. Let $\mathcal{S}$ be a system and $p, q \in \mathbf{Pred}\,(C \times D)$. Then

$$\begin{array}{rcl} \{p\}\,t\,\{q\} & \stackrel{\text{def}}{=} & \forall\,(c, c' \in C)\,, (d, d' \in D)\,. \\ & & \quad p\,(c, d) \wedge \\ & & \quad c = dom\,(t) \wedge \\ & & \quad \rho\,(t)\,(d, d') \wedge \\ & & \quad c' = cod\,(t) \;\Rightarrow\; q\,(c', d') \end{array}$$

and $\{p\}\,U\,\{q\} \stackrel{\text{def}}{=} \bigwedge_{t \in U} \{p\}\,t\,\{q\}$ are *Hoare triples* w.r.t. $\mathcal{S}$ for a transition $t \in T$ resp. for a set of transitions $U \subseteq T$.

3.1. Linear-Time Temporal Logic

The formulas of linear-time temporal logic (LTL) over X are given by

$$\varphi ::= p \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \mid \varphi_1 \,\mathsf{V}\, \varphi_2 \mid \mathsf{X}\,\varphi$$

where $p \in \mathsf{Pred}(X)$ is a predicate. As usual, some derived connectives can be defined from the basic ones, e.g. $\mathsf{F}\,\varphi \stackrel{\text{def}}{=} \mathsf{tt}\,\mathsf{U}\,\varphi$ and $\mathsf{G}\,\varphi \stackrel{\text{def}}{=} \mathsf{ff}\,\mathsf{V}\,\varphi$. These formulas are interpreted over infinite sequences $\sigma \in X^\omega$. For $\sigma \in X^\omega$ and a LTL formula φ over X, the *satisfaction relation* $\models$ is inductively defined by

$$\begin{array}{lll} \sigma \models p & \text{iff} & p(\sigma(0)) = \mathsf{tt} \\ \sigma \models \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\ \sigma \models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ or } \sigma \models \varphi_2 \\ \sigma \models \mathsf{X}\,\psi & \text{iff} & \sigma^1 \models \psi \\ \sigma \models \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists k \in \mathbb{N}.\,(\sigma^k \models \varphi_2 \\ & & \quad \text{and } \forall i < k.\,\sigma^i \models \varphi_1) \\ \sigma \models \varphi_1 \,\mathsf{V}\, \varphi_2 & \text{iff} & \forall k \in \mathbb{N}.\,(\forall i < k.\,\sigma^i \not\models \varphi_1) \\ & & \quad \text{implies } \sigma^k \models \varphi_2 \end{array}$$

where $\sigma(i)$ denotes the ith position and σ^i the ith suffix of σ. For a predicate $p \in \mathsf{Pred}(C \times D)$ and a LTL formula ϕ over $C \times D$, the notation $p \models_{\mathcal{S}} \phi$ means that all p-runs of S satisfy ϕ. For more details on LTL we refer the interested reader to [4, 14].

3.2. Proof Structures

Definition 3.3. A *sequent over $C \times D$* is a triple $l, p \vdash \Phi$ where $l \in C$ is a control point, $p \in \mathsf{Pred}(C \times D)$ is a predicate and Φ is a *finite, non-empty* set of LTL formulas over $C \times D$. By abuse of language we use $l \in C$ as predicates with $l(c, d) = \mathsf{tt}$ iff $l = c$. A sequent $l, p \vdash \Phi$ is called *valid* if $l \wedge p \models \bigvee_{\phi \in \Phi} \phi$. $\mathbf{Seq}\,(C, D)$ is the set of all sequents over $C \times D$.

Definition 3.4. Let $\Pi = (V, E, v_r, \eta)$ where (E, V) is a relational graph, $v_r \in V$ is a distinguished vertex called *root* and $\eta : V \to \mathbf{Seq}\,(C, D)$ labels vertices with sequents. Π is a *proof structure* for a sequent $l, p \vdash \phi$ w.r.t. system $\mathcal{S}$ provided $\eta(v_r) = l, p \vdash \phi$ and for all $v \in V$

(i) v is *reachable* from v_r

(ii) if v has $n \geq 0$ successors $v_1, \ldots, v_n$ then $\eta(v)$ is the conclusion and $\eta(v_1), \ldots, \eta(v_n)$ are the premises of a correct instantiation of some rule R of table 5 and the side condition of rule R holds

(iii) if $(v, v') \in E$ then rule $\mathsf{R}(sp)$ is not applied at both v and v'

We often write $l_v, p_v \vdash \Phi_v$ for $\eta(v)$.

$\mathsf{R}(ax)$	$\dfrac{l, p \vdash \Phi, p}{\cdot}$	
$\mathsf{R}(bsf)$	$\dfrac{l, p \vdash \Phi, q}{l, p \vdash \Phi}$	$(l \wedge p) \rightarrow \neg q$
$\mathsf{R}(\vee)$	$\dfrac{l, p \vdash \Phi, \phi_1 \vee \phi_2}{l, p \vdash \Phi, \phi_1, \phi_2}$	
$\mathsf{R}(\wedge)$	$\dfrac{l, p \vdash \Phi, \phi_1 \wedge \phi_2}{l, p \vdash \Phi, \phi_1 \quad l, p \vdash \Phi, \phi_2}$	
$\mathsf{R}(\mathsf{U})$	$\dfrac{l, p \vdash \Phi, \phi_1 \,\mathsf{U}\, \phi_2}{l, p \vdash \Phi, \phi_2, \phi_1 \quad l, p \vdash \Phi, \phi_2, \mathsf{X}(\phi_1 \,\mathsf{U}\, \phi_2)}$	
$\mathsf{R}(\mathsf{V})$	$\dfrac{l, p \vdash \Phi, \phi_1 \,\mathsf{V}\, \phi_2}{l, p \vdash \Phi, \phi_2 \quad l, p \vdash \Phi, \phi_1, \mathsf{X}(\phi_1 \,\mathsf{V}\, \phi_2)}$	
$\mathsf{R}(\mathsf{X})$	$\dfrac{l, p \vdash \mathsf{X}\,\Phi}{m_1, q_1 \vdash \Phi \quad \cdots \quad m_n, q_n \vdash \Phi}$	$\{p\}\, T(l, m_i)\, \{q_i\}$ for all i
$\mathsf{R}(sp)$	$\dfrac{l, p \vdash \Phi}{l, q_1 \vdash \Phi \quad \cdots \quad l, q_n \vdash \Phi}$	$(l \wedge p) \rightarrow \bigvee_{i=1}^{n} q_i$

Table 5. The local proof rules; in Rule $\mathsf{R}(\mathsf{X})$ we write $\mathsf{X}\,\Phi$ for $\{\mathsf{X}\,\phi \mid \phi \in \Phi\}$ and $m_1, \ldots, m_n$ is the set of successor vertices of l in the control graph.

The aim of a proof structure is to establish the validity of its root sequent. Clause (iii) in definition 3.4 prevents circular propositional reasoning. However, the latter is necessary but not sufficient for a proof structure to be a *proof* (of the validity of the root sequent). We have to make sure that formulas of the form $\phi_1 \,\mathsf{U}\, \phi_2$ eventually satisfy their promises ϕ_2 which is a non-local condition that cannot be guaranteed by the construction of a proof structure alone. Therefore, we need a global criterion to identify proofs.

3.3. Proofs

We formulate a criterion for a proof structure Π to be *proof* in terms of the language accepted by an automaton derived from Π and equipped with a generalised Büchi acceptance condition. For the rest of this section we consider an arbitrary but fixed proof structure $\Pi = (V, E, v_r, \eta)$ w.r.t. $\mathcal{S}$ with root sequent $\eta(v_r) = l, p \vdash \phi$.

Definition 3.5. A vertex v of Π is called *ψ-safe* for a V-formula $\psi = \phi_1 \,\mathsf{V}\, \phi_2$ if $\Phi_v \cap \{\psi, \mathsf{X}\,\psi\} \neq \varnothing$ and $\phi_2 \notin \Phi_v$. By abuse of language, a trace in Π is a trace in the underlying graph and it is *ψ-safe* if this is the case elementwise.

Definition 3.6. A trace π is called *next-free* if the Next rule $\mathsf{R}(\mathsf{X})$ is applied at most at the last vertex of π. Let l_π be the (unique) control point appearing in labels along a next-free trace π.

Definition 3.7. Let $\{\psi_1, \ldots, \psi_n\}$ be the set of V-subformulas of ϕ. The *discharge automaton*

$$\mathcal{A}^{\Pi} \stackrel{\text{def}}{=} (Q, Q_0, \Delta, \mu, \mathcal{F})$$

is given by the data

$$\begin{array}{rcl}
Q & \stackrel{\text{def}}{=} & \{\pi \mid \pi \text{ a maximal next}-\text{free trace in } \Pi\} \\
Q_0 & \stackrel{\text{def}}{=} & \{\pi \in Q \mid \pi(0) = v_r\} \\
\Delta & \stackrel{\text{def}}{=} & \{(\pi, \pi') \in Q \times Q \mid (\pi(|\pi| - 1), \pi'(0)) \in E\} \\
\mu(\pi) & \stackrel{\text{def}}{=} & l_\pi \wedge \bigwedge_{0 \leq k < |\pi|} p_{\pi(k)} \\
\mathcal{F} & \stackrel{\text{def}}{=} & \{F_1, \ldots, F_n\} \\
& & \text{where } F_i \stackrel{\text{def}}{=} \{\pi \mid \pi \text{ not } \psi_i - \text{safe}\}
\end{array}$$

A maximal trace r in (Δ, Q) with $r(0) \in Q_0$ is called a *run of $\mathcal{A}^{\Pi}$*. Such an r is a run over a sequence $\sigma \in (C \times D)^{\omega}$ if $\mu(r(i))(\sigma(i)) = \mathtt{tt}$ for all $i < |r|$. The sequence σ is *accepted* by $\mathcal{A}^{\Pi}$ if there is a run r of $\mathcal{A}^{\Pi}$ s.t. for all $F \in \mathcal{F}$ there are infinitely many $j \in \mathbb{N}$ such that $r(j) \in F$. The set of accepted sequences is called the *language of* $\mathcal{A}^{\Pi}$ and is denoted by $\mathcal{L}(\mathcal{A}^{\Pi})$.

Note that due to the side conditions of the Split and Next rules there is a run of $\mathcal{A}^{\Pi}$ over any $(l \wedge p)$-run of the system. Observe also that an accepting run of $\mathcal{A}^{\Pi}$ corresponds to a trace in Π containing infinitely many ψ-unsafe vertices for each V-subformula ψ of ϕ.

Lemma 3.8. *A $(l \wedge p)$-run σ of $\mathcal{S}$ accepted by $\mathcal{A}^{\Pi}$ if and only if σ is a counterexample to the validity of the root sequent $l, p \vdash \phi$ of Π.*

Definition 3.9. A proof structure Π is a *proof,* if $\mathcal{L}(\mathcal{A}^{\Pi}) \cap \mathcal{R}(\mathcal{S}, l \wedge p) = \varnothing$.

Definition 3.9 goes in pair with Lemma 3.8 since the latter carves effectively out those proof structures which can be considered as proofs of the property ϕ.

Discharge Conditions Proposition 3.10 provides us with a method to establish that a proof structure is a proof. As we can apply the well-known construction to transform a generalised Büchi automaton ($|\mathcal{F}| \geq 0$) into a standard Büchi automaton ($|\mathcal{F}| = 1$) accepting the same language [20], we assume here that $\mathcal{F}$ is a singleton set.

Proposition 3.10. *Let Π and $\mathcal{A}^{\Pi}$ be as in definition 3.7 except that $\mathcal{F} = \{F\}$. Π is a proof if there is a well-founded domain $(W, \prec)$ and a ranking function $\delta : Q \to (D \to W)$ such that for each $(q, r) \in \Delta$ the condition*

$$\{\mu(q) \wedge \delta(q) = w\} \;\; T(l_q, l_r) \;\; \{\mu(r) \to \delta(r) \prec w\}$$

holds if $q \in F$ and

$$\{\mu(q) \wedge \delta(q) = w\} \;\; T(l_q, l_r) \;\; \{\mu(r) \to \delta(r) \preceq w\}$$

holds if $q \notin F$.

Intuitively speaking, the ranking function measures progress towards the non-acceptance of system runs by $\mathcal{A}^{\Pi}$: the ranking is required to decrease at accepting vertices and to not increase at other vertices. Recall that there is a run of $\mathcal{A}^{\Pi}$ over any $(l \wedge p)$-run of $\mathcal{S}$. Hence, provided that the discharge conditions hold, the well-foundedness of $(W, \prec)$ implies that no $(l \wedge p)$-run of the system can be accepted by $\mathcal{A}^{\Pi}$.

It is important to remark that the discharge automaton $\mathcal{A}^{\Pi}$ of a proof structure Π for a *safety formula*, that is, a formula not containing any U-subformulas, always accepts the empty language. The reason is that all vertices of a given strongly connected component of Π are ψ-safe for some V-formula ψ. In other words, a proof structure for a safety formula is always a proof.

Soundness and Relative Completeness Our proof method is sound and complete *relative* to the *validity* of the side conditions of the local rules and validity of the discharge conditions.

Theorem 3.11. *$l \wedge p \models_{\mathcal{S}} \phi$ if and only if there is a proof structure Π for $l, p \vdash \phi$ w.r.t. $\mathcal{S}$ and a ranking function δ such that the discharge conditions of Proposition 3.10 hold for $\mathcal{A}^{\Pi}$.*

4. The Mc5 Verification Tool

In this section we report on our preliminary experience with a prototype tool implementing the framework described in the previous two sections. Its overall architecture is displayed in Figure 1. Two basic parts can be distinguished: (1) a *front-end* consisting of the **Mc** language compiler, the proof manager, the tactical language interpreter and the graphical user interface, and (2) the *back-end* theorem prover PVS [17] used to discharge first-order verification conditions. The implementation language for the front-end is Ocaml [11].

Mc Compiler The functionality of the back-end w.r.t. a given program is to extract a system from the syntactical information collected by the parser. In section 2, we argued that all the underlying constructions can be represented in PVS. However, taking the idea *literally* from an implementation perspective would result in significant inefficiencies since direct calculation is not possible in PVS but has to be performed by proving series of lemmas. We therefore opted for the following trade-off: graph-related calculations e.g. the limit construction are carried out *explicitly* while the labelling is handled in a *symbolic* way, *assembling* relevant λ-terms. So what is handed over to the proof manager is an explicit representation of the control graph and a function on edges returning the abstract syntax tree of a predicate. Not surprisingly, we had to implement “optimisations” in form of β- and δ-reductions in order to obtain human-readable PVS-terms. Finally, it is worth to notice that the compilation of individual processes can be done separately and also decoupled from the compilation of communicating systems built with such processes. It is possible by virtue of the compositionality that a semantics based on twisted systems exhibits.

The Proof Manager This part handles proof structures and discharge automata by maintaining the *proof state* including the set of open goals while providing additional functionality like undoing/redoing of elementary proof steps and postponing the subgoal in focus. When the construction of a proof structure is finished, the discharge automaton is generated and the user provides an appropriate

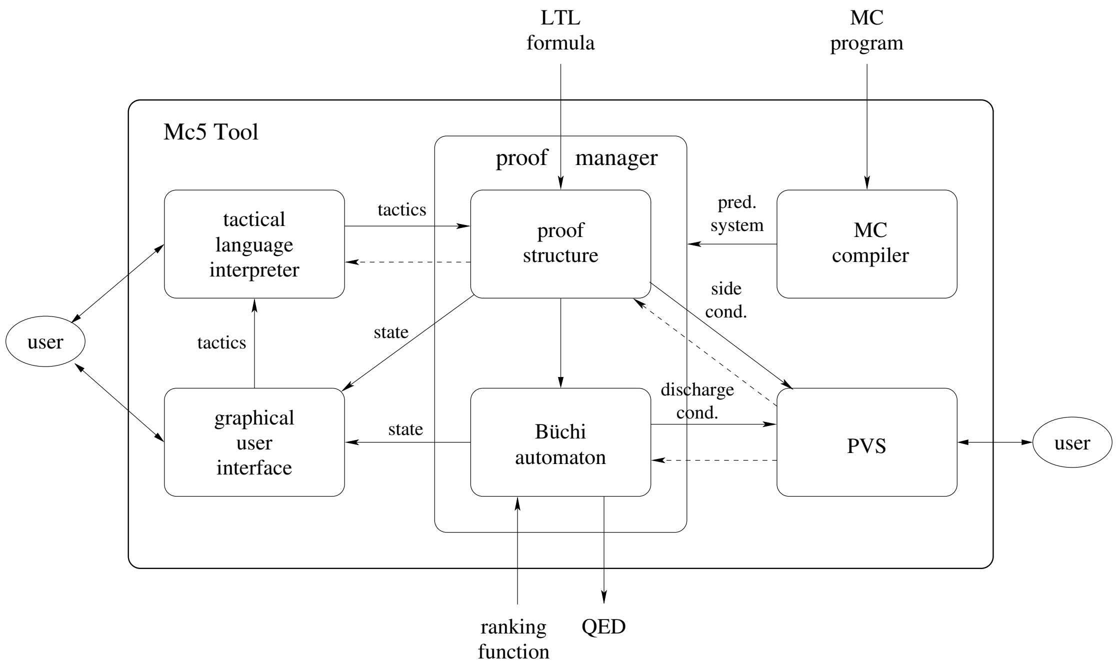


Figure 1. Mc5 Tool Architecture

well-founded relation and ranking functions for the generation of the discharge conditions. Side conditions as well as discharge conditions are proved using PVS. The interaction with PVS is very rudimentary in this first prototype implementation in that all verification conditions are passed to PVS in the form of a theory file. In particular, there is no interaction with PVS during the construction of the proof structure.

Tactics and Tacticals The manipulation of proof structures is done via a tactics module providing the user with a simple but powerful interface. The basic proof rules are mapped to tactics and there is a number of standard tacticals i.e. higher-order tactics. In the current state of development, the tactical commands are directly interpreted by the Ocaml shell. In order not to falsify the generated discharge automaton, we keep all intermediate subgoals a tactic may produce by making the basic proof rules directly act on the state of the proof structure. Each tactic returns the *number* of *new* subgoals. Counting the number k of basic rules applied to the proof structure by a tactic allows us to undo the effects of the latter in case of failure by calling the undo method of the proof structure k times. In case of success, k is pushed onto a stack thus implementing a simple tactics-level undo facility.

Graphical User Interface The DaVinci graph visualisation tool [6] has proved to be extremely helpful as a rudimentary graphical user interface for our tool. It is currently used only for displaying system specifications, proof structures and discharge automata.

5. Concluding Remarks

We propose a verification methodology building on original results in the relevant fields of semantics of concurrency respectively deduction-based model checking. Model generation relies on a novel categorical semantics of processes with synchronous communication. The semantics is compositional, which has to be contrasted with other approaches known in the literature [14]. The deduction system, presented here in a version geared towards the class of models under consideration, extends the model checker presented in [1] to cope with infinite-state systems. Although stemming from usually separated areas of computer science, those results coalesce smoothly as documented by the discussed application, the **Mc5** verification tool.

Related Work Expressing a system of communicating processes as diagram and the composed behaviour as its limit goes back to Goguen's work in the late seventies [9]. Errington recently refined the idea by introducing twisted systems in his doctoral dissertation [8] and in [7]. The second author of the present paper applied a slightly simplified version of the setup to the semantics of a concurrent programming language based on *active objects* and made the link with conventional operational semantics precise [21].

Closest to our work on verification are the diagram-based techniques of deductive model checking (DMC) [18] and generalised verification diagrams (GVDs) [3, 13]. They use automata (called *diagrams*) very similar to our discharge automata. In contrast to our method, DMC repeatedly *refines* an initial diagram by applying transformations preserving the *invariant* that each diagram accepts all potential counterexamples of the system. It is noteworthy that discharge automata satisfy this invariant (cf. Lemma 3.8) and can hence be refined using DMC transformations. The GVD method consists of the construction of an abstraction of the system, followed by automatic model checking of the property on the abstraction. Whereas GVDs can be constructed independently of the property to be verified, a proof structure can be seen as an abstraction that is constructed hand-in-hand with the tableau of the property and is thus *tailored* to the latter.

Future Work Clearly, a lot of work remains to be done. The most apparent limitation of the current prototype is its lack of automation. Automatic invariant generation techniques [2] could be applied with great benefit to produce the predicates to be supplied to the Next rule. These need to be strong enough for the discharge conditions to be provable. At the same time the discharge conditions could be weakened by replacing the acceptance condition by an equivalent (language-preserving) one with less accepting states[1].

Another issue concerns the condition for looping back to an existing vertex in the proof structure. The current implementation requires that a sequent generated by a rule *equals* a sequent already present. Equality is a much too strong condition on predicates and should be replaced by *implication*. Each new subgoal should be checked automatically against loopback candidates using for example PVS' batch mode.

The interaction with PVS needs a more interactive shape. In particular, side conditions should be submitted to PVS immediately in an attempt to prove them automatically and feedback about their status should be handed back from PVS to the proof manager.

References

[1] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *Logic in Computer Science, LICS 95*, pages 388–397, 1995.

[2] N. S. Bjorner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[3] I. A. Browne, Z. Manna, and H. B. Sipma. Generalized temporal verification diagrams. In *Foundations of Software Technology and Theoretical Computer Science, FSTTCS '95, Bangalore, India*, volume 1026 of *Lecture Notes in Computer Science*, pages 484–498. Springer-Verlag, 1995.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[5] R. L. Crole. *Categories for Types*. Cambridge University Press, 1993.

[6] daVinci home page. `http://www.tzi.de/daVinci`, 2001. University of Bremen, Germany.

[7] L. Erington. On the semantics of message passing processes. In *Proceedings of CTCS99*, 1999.

[8] L. Erington. *Twisted Systems*. PhD thesis, Department of Computing, Imperial College, London, 1999.

[9] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series on Computer Science. Prentice-Hall, 1985.

[11] INRIA. *The Objective Caml system, release 3.00*, Apr. 2000.

[12] S. M. Lane. *Categories for the Working Mathematician*. Springer, 2 edition, 1997.

[13] Z. Manna, A. Browne, H. B. Sipma, and T. E. Uribe. Visual abstractions for temporal verification. In *AMAST '98*, volume 1548 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, 1998.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[15] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 1202–1242. Elsevier Science Publishers, 1990.

[16] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[17] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[18] H. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15:49–74, 1999.

[19] C. Sprenger. *Deductive Local Model Checking – On the Verification of CTL* Properties of Reactive Systems*. Ph.D. Thesis no. 2215, Department of Computer Science, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2000.

[20] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers , Amsterdam, 1990.

[21] K. Worytkiewicz. *Components and Synchronous Communication in Categories of Processes*. Ph.D. Thesis no. 2131, Department of Computer Science, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2000.

[1] cf. the example in Appendix A, where a single accepting state would be sufficient and obviate the need for the second component in the ranking

[22] K. Worytkiewicz. Concrete process categories. In A. Kurz, editor, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier Science Publishers, 2002.
[23] K. Worytkiewicz. Paths and simulations. In P. S. Richard Blute and P. Selinger, editors, *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003. To appear.

A. Example

As a simple example consider the program below. It is built from two processes, P and Q, running in parallel and communicating over a synchronous channel. Control points are commented as a and b in P and by x, y and z in Q.

```
P =  x:nat ⊢ while true do
                (*a*) cp!x+x;
                (*b*) cp?x
             end : <cp:nat>

Q =  y:nat ⊢ while true do
                (*x*) cq?y;
                (*y*) y:=y+1;
                (*z*) cq!y
             end : <cq:nat>
```

For the parallel composition $S = \texttt{x,y} : \texttt{nat} \vdash (P \, \|\mathsf{cp} \asymp \mathsf{cq}\| \, Q)$ <> we verify that the variable y grows without bound whenever x equals y in the initial state. The liveness property of unbounded growth is expressed in LTL by the formula

$$\phi = \mathsf{F}(y \geq n)$$

where n is an arbitrary but fixed parameter.

Model Generation The control graph with control points called ax, by and bz of the system generated by the **Mc** compiler is shown in Figure 2.

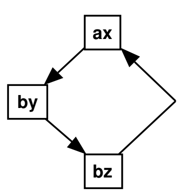


Figure 2. Control graph of system specification

while the transition predicates labeling the edges are

$$\begin{array}{lcl} \texttt{ax} \rightarrow \texttt{by} & \mapsto & x' = x \wedge y' = x + x \\ \texttt{by} \rightarrow \texttt{bz} & \mapsto & x' = x \wedge y' = y + 1 \\ \texttt{bz} \rightarrow \texttt{ax} & \mapsto & x' = y \wedge y' = y \end{array}$$

expressed in the primed/unprimed notation ubiquitous in the model-checking litterature.

Proof The proof structure for this system and the sequent $\mathsf{ax}, y = x \vdash \mathsf{F}(y > n)$ is displayed in Figure 3.

In this case, all side conditions validating this proof structure (totally 9) are proved without user intervention using PVS's `grind` command.

Having constructed the proof structure, the next step is to generate the derived discharge automaton (see Figure 4). It reflects the structure of the system: each node corresponds to a control point and each transition to a transition of the system[2]. As there is no V-subformula in the property formula, all states of the automaton are accepting. Let us call its nodes ax, by and bz according to the associated control points.

Clearly, all sequences of states accepted by this automaton violate our property, since $y \leq n$ holds in all states of such a sequence. Contemplating the automaton for a moment, we easily come up with the rankings

$$\begin{array}{lcl} \delta_{\mathsf{ax}}(x, y) & \stackrel{\text{def}}{=} & \texttt{lex2}(n - y, 1) \\ \delta_{\mathsf{by}}(x, y) & \stackrel{\text{def}}{=} & \texttt{lex2}(n - y, 0) \\ \delta_{\mathsf{bz}}(x, y) & \stackrel{\text{def}}{=} & \texttt{lex2}(n - y, 2) \end{array}$$

where `lex2` is a PVS function mapping a lexicographically ordered pair of natural numbers to (the PVS representation of) an ordinal. The first argument measures the distance between y and n, while the second one measures progress to the next decrease of the first component.

As an example, for the automaton transition from by to bz the tool generates the PVS discharge condition[3]

```
(FORALL (pc:string),(x:nat),(y:nat),
  (pc':string),(x':nat),(y':nat):
    ((pc = "by") AND (y = (x + x))
      AND NOT(y > n) AND (x = x')
      AND (y + 1 = y')
      AND (pc' = "bz")
      AND (y' = x' + x' + 1)
      AND NOT(y' > n))
    IMPLIES
      (lex2(n - y, 0) >
       lex2(n - y', 2)))
```

which is easily seen to be valid. All of the totally three discharge conditions are proved in PVS without user intervention using the command `grind`.

[2] Terminal nodes with no outgoing edges can safely be removed without changing the language accepted by the automaton.

[3] slightly edited to improve readability

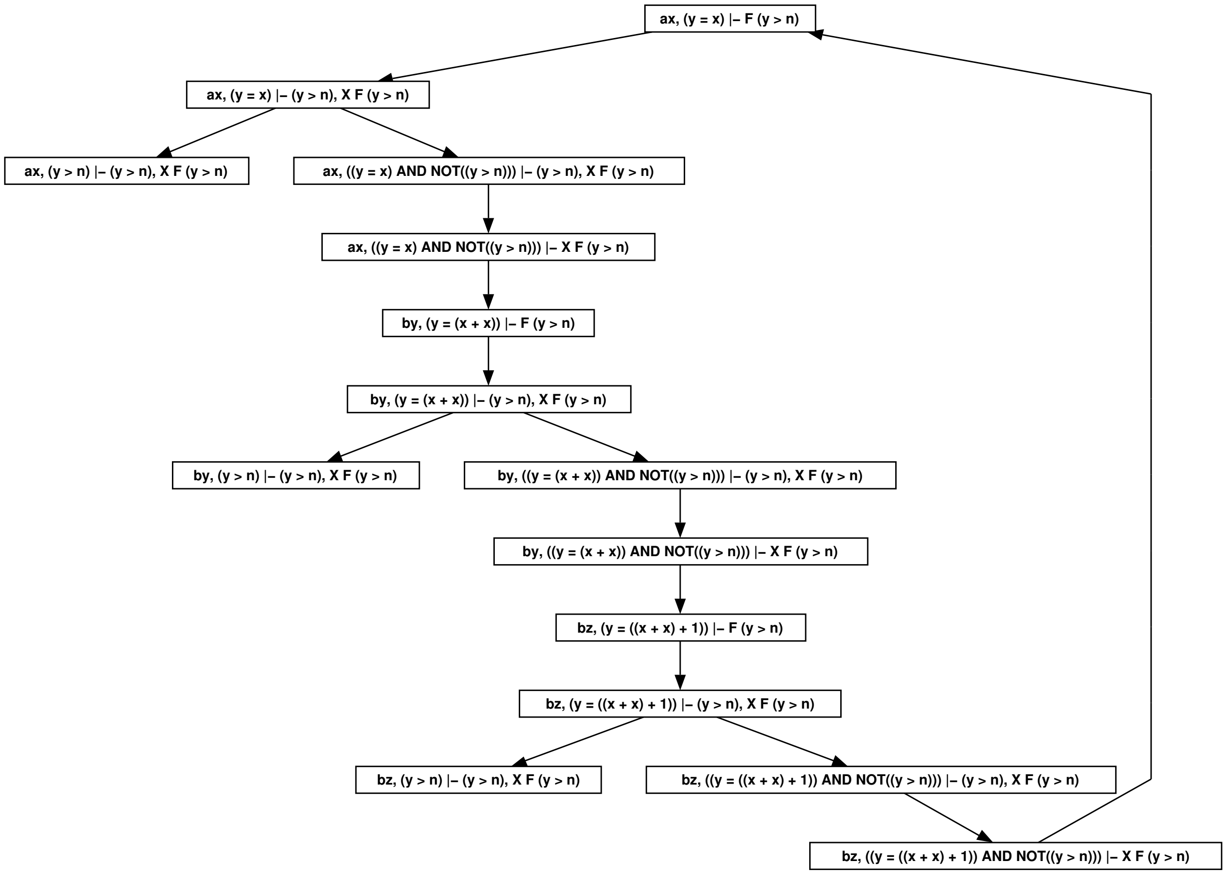


Figure 3. Proof Structure

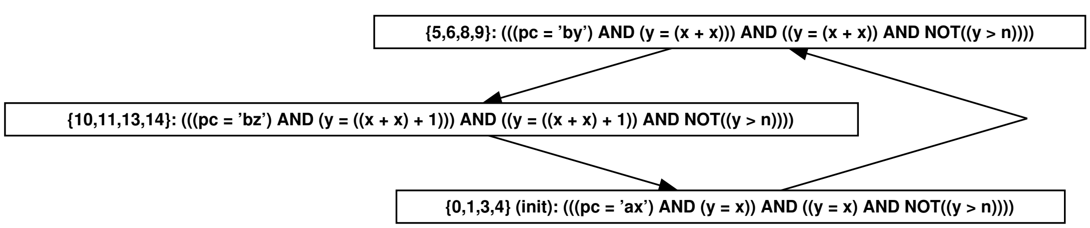


Figure 4. Discharge automaton