Joaquín Aguado, Michael Mendler (Eds.)

SLA++P 2007

Model-driven High-level Programming of Embedded Systems

Proceedings

Satellite Workshop to the Tenth European Joint Conferences on
Theory and Practice of Software (ETAPS 2007)
Braga, Portugal, March 31, 2007

Preface

This report contains the proceedings of the International Workshop on *Model-driven High-level Programming of Embedded Systems* (SLA++P 2007). The workshop is dedicated to synchronous languages and the model-driven high-level programming of reactive and embedded systems. Firmly grounded in clean mathematical semantics, synchronous languages are receiving increasing attention in industry ever since they emerged in the 80s. Lustre, Esterel, Signal are now widely and successfully used to program real-time and safety critical applications of commercial scale, from nuclear power plant management layer to Airbus air flight control systems. At the same time, model-based programming is making its way in other fields of software engineering, too, often involving cycle-based synchronous paradigms.

Transcending the former five editions of the SLAP workshop series on Synchronous Languages, Applications, and Programming, SLA++P is not limited to synchronous approaches. It is open to other engineering design techniques with strong semantical foundations providing a way to go from high-level description to provable executable code.

The workshop took place on March 31, 2007 as a satellite event to the Tenth European Joint Conferences on Theory and Practice of Software (ETAPS 2007) in Braga, Portugal. We received 16 paper submissions with authors from 6 countries on 4 continents, of which 56% were accepted for presentation. Every paper was evaluated based on 3-5 peer reviews by an international programme committee with the following members:

Benoît Caillaud *(IRISA-INRIA, .fr)*
Jean-Luis Colaço *(Esterel Technologies, .fr)*
Nicholas Halbwachs *(VERIMAG-IMAG, .fr)*
Grégoire Hamon *(Univ. of Chalmers, .se)*
Reinhard von Hanxleden *(Univ. of Kiel, .de)*
Leszek Holenderski *(Philips, .nl)*
Luciano Lavagno *(Politech. Univ. of Turin, .it)*
Gerald Lüttgen *(Univ. of York, .uk)*
Michael Mendler *(Univ. of Bamberg, .de)*
Gordon Pace *(Univ. of Malta, .mt)*
Axel Poigné *(Fraunhofer Institute, .de)*
R.K. Shyamasundar *(IBM India Res. Labs, .in)*
Robert de Simone *(INRIA Sophia-Antip., .fr)*
Satnam Singh *(Microsoft Res. Cambridge, .uk)*
Simone Tini *(Univ. dell'Insubria, .it)*
Joaquín Aguado *(Univ. of Bamberg, .de)*

The meeting of the programme committee took place electronically in January 2007. In addition to the submitted regular contributions, there was an invited presentation by Steven Miller (Advanced Technology Center of Rockwell-Collins, USA).

We would like to express our gratitude to all authors of submitted papers, to the members of the programme committee and to the referees assisting them. We greatly appreciate the efforts of the ETAPS organising committee from the University of Minho, especially Joost Visser and Luis Soares Barbosa.

February 2007

Joaquín Aguado and Michael Mendler
(SLA++P Workshop Organisers)

Contents

R. Sinha, P. S. Roop, S. Basu
A Model Checking Approach to Protocol Conversion 3

M. Boldt, C. Traulsen, R. von Hanxleden
Worst Case Reaction Time Analysis of Concurrent Reactive Programs 21

A. Ray, R. Cleaveland
Executable Specifications for Real-Time Distributed Systems 39

O. Tardieu, S. A. Edwards
Instantaneous Transitions in Esterel 55

P. Raymond, Y. Roux, E. Jahier
Specifying and executing reactive scenarios with Lutin 71

D. Stauch
Modifying Contracts with Larissa Aspects 87

F. Maraninchi, L. Samper, K. Baradon, A. Vasseur
Lustre as a System Modeling Language: Lussensor, a Case-Study with Sensor Networks 103

L. du Bousquet, M. Delaunay
Towards mutation analysis for LUSTRE programs 119

J. Gao, M. Whalen, E. Van Wyk
Extending Lustre with Timeout Automata 136

1. A Model Checking Approach to Protocol Conversion

Roopak Sinha, Partha S. Roop[1], and Samik Basu[2]

1 University of Auckland, New Zealand
2 Iowa State University, USA

Notes:

A Model Checking Approach to Protocol Conversion

Roopak Sinha, Partha S. Roop, and Samik Basu

University of Auckland rsin077,p.roop@ec.auckland.ac.nz
Iowa State University sbasu@cs.iastate.edu

Abstract. System-on-chip verification is an active research area. Of particular interest is protocol conversion, where two components with different protocols are controlled to communicate accurately. We present an approach to protocol conversion using model checking. The temporal logic ACTL is used to describe desired behaviour and finite state machines are used for protocol description. We use tableau-based converter construction and prove that a converter exists only when a successful tableau can be constructed. Liveness is incorporated so that converters satisfy additional constraints on protocol communication. A NuSMV-based implementation has been created and we present results on various problems including a large NuSMV example.

1 Introduction

A System-on-a-chip (SoC) integrates components of a computer system into a single chip with various hardware and software components connected using a central bus such as AMBA [8]. SoC verification is an active area of interest and verification strategies are based on data-flow and/or control-flow analysis of the system. The focus of this paper is *protocol conversion* for *mismatched* protocols [13]. Although *physical connectivity* (interconnection using physical channels) between components can generally be achieved, *logical connectivity*, where processes communicate in the desired fashion, cannot always be guaranteed [13]. A *mismatch* occurs when processes fail to be logically connected. The aim of protocol conversion is to synthesize extra glue-logic, called a *converter*, to control mismatched protocols to reconcile mismatches. A converter can control the communication between protocols by employing strategies such as *event hiding* [13], *event translation* [5] and *inhibition* [16]. The automatic generation of a converter is known as *converter synthesis* whereas *convertibility verification* focuses on establishing whether protocols are mismatched and whether a converter exists. Fig. 1 gives an overview of protocol conversion where a converter controls two protocols P_1 and P_2 to satisfy given specifications.

We present a technique using model checking for automatically synthesizing a converter. Protocols, in our setting, are represented using *Kripke Structures* (KS) [7] and the desired properties of the combined protocols are represented using temporal logic ACTL, a branching time temporal logic with universal path

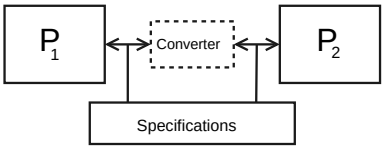


Fig. 1. Protocol conversion

quantifiers. The logic is particularly interesting and relevant for protocol conversion as mismatches in protocols must be addressed for *every* path of their KS descriptions. Given two KSs P_1 and P_2 and a set of desired properties in ACTL, Ψ, the protocol conversion via converter synthesis problem (illustrated in Fig. 1) is equivalent to checking for the existence of a converter under which the protocols satisfy all formulas in Ψ.

Central to our technique is the construction of a tableau where satisfaction of Ψ by the protocols and the converter is defined in terms of the satisfaction of its subformulas (similar to [2]). The tableau construction also results in the synthesis of a converter as the protocol-composition states are explored along with the subformulas of the desired property. The technique leads to local and on-the-fly construction of the converter, one where the state-space of the protocols and the subformulas of the property are explored and expanded as and when needed. In fact, in the event there exists no converter, i.e., the protocols cannot be matched (hard mismatch [10]), our tableau-based technique can potentially identify the failure without exploring the state-space that is irrelevant for failure inference.

The main contributions of this paper are summarized as follows:

- We present a temporal logic based formulation for protocol conversion where temporal logic formulas in ACTL are used to specify the desired communication between participating protocols.
- A tableau-based technique for identifying a converter, if one exists, as a glue-logic between composed protocols to reconcile the protocol mismatches and ensure that the desired specifications are satisfied. The tableau is sound and complete and the converter, thus synthesized, is correct by construction.
- The tableau-based technique describes a local and on-the-fly algorithm for converter synthesis—one where the state-space of the protocols being composed are explored only as and when needed to prove or disprove the existence of a converter. The algorithm is polynomial in the size of the participating protocols and the given specifications.

The rest of this paper is organized as follows. We summarize works related to our approach in section 2 and provide a motivating example in section 3. The problem of protocol conversion is described in section 4 and we provide our proposed tableau-based protocol-conversion approach in section 5. Section 7 presents implementation results with concluding remarks in section 8.

2 Related Work

A number of techniques have been developed to address the problem of protocol conversion using a wide range of formal and informal settings with varying degrees of automation—projection approach [13], quotienting [5], conversion seeds [15], synchronization [17], supervisory control theory [11], to name a few. Some techniques, like converters for protocol gateways [3] and interworking networks [4], rely on ad hoc solutions. Some approaches, like protocol conversion based on conversion seeds [15] and protocol projections [13], require significant user expertise and guidance during converter construction. While this problem has been studied in a number of formal settings [10, 13, 15, 17], only recently have some formal verification based solutions been proposed [16, 8, 11, 9].

The closest to our approach are [16, 8]. In [16], the authors present an approach towards protocol conversion using finite state machines to represent participating protocols as well as specifications employing a game-theoretic framework to generate a converter. This solution is restricted only to protocols with half-duplex communication between them. D'Silva et al [8] present synchronous protocol automata to allow formal protocol specification and matching, as well as converter synthesis. The matching criteria between protocols are based on whether events are blocking or non-blocking and no additional specifications can be used. The approach allows model checking only as an auxiliary verification step to ensure that conversion is correct.

In contrast to the above techniques, we use temporal logic to represent desired functionality of the combined protocols. Being based on temporal logic, our technique can define desired properties succinctly and with a higher-level of granularity. For example: a desired behavior of the combination may be sequencing of events such that event a in protocol P_1 always happens before event b in P_2. Also, as our technique is based on the (tableau-based) model checking algorithm, the converter synthesized is correct by construction.

The presented approach is similar to the synthesis of discrete controllers with temporal logic and Control-D system [1]. However, the approach in[1] generates controllers that can only perform *disabling*, i.e, transitions in the underlying system can be disabled that lead to the eventual failure of given `CTL` formulas. Additionally, the approach does not handle liveness properties. On the other hand, converters generated using our approach not only perform disabling, but they can also *buffer* events for later use in the communication of the protocols. Additionally, the synthesized converters can generate *extra control signals* expected as input by one protocol but not emitted by the other, in order to lead the communication between the protocols to states that conform to given specifications.

3 Illustrative Example

We motivate our approach using the following example. Fig. 2 shows the communication protocols of two devices, a producer and a consumer, which need to

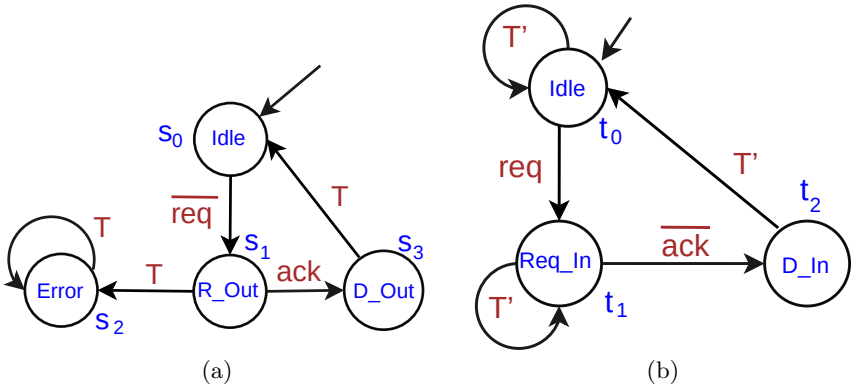


Fig. 2. The producer-consumer protocol pair. (a) Producer P_1, (b) Consumer P_2.

communicate with each other. In its initial state s_0, the producer protocol P emits a request ($\overline{req}$) and makes a transition to state s_1. In s_1, an acknowledge input (ack) is expected immediately. In case ack is not available, a transition to the error state s_2 is made. In case ack is available, a transition to state s_3 is made where one packet of data is produced (denoted by the D_Out label). From s_3, the producer resets back to its initial state s_0.

The consumer protocol P_2 operates as follows. In its initial state t_0, the consumer awaits a request from the producer protocol. Once a request is received, a transition to state t_1 is made. In state t_1, an acknowledge signal $\overline{ack}$ is emitted and a transition to state t_2 is made. In t_2, a packet of data is read (denoted by the label D_In) and a transition back to the initial state is made. Note that an event a represents an input whereas $\overline{a}$ represents an output. We specify their desired behaviour using the following ACTL formulas:

1. $\texttt{AG}\neg Error$: The communication never enters a state labelled by $Error$.
2. $\texttt{AG}\,[D\_Out \Rightarrow (\ D\_In\ \vee\ \texttt{AXA}(\neg D\_Out\ \texttt{U}\ D\_In)\ )]$: Each data packet emitted by the producer is read by the consumer before another data packet is emitted (no loss).

Given the producer-consumer protocol pair in Fig. 2, it is possible that the unrestricted behavior of the protocols may lead to states that fail to satisfy the above properties. We formalize our solution to resolve these issues in the following sections.

4 Preliminaries

Model of Protocols: Kripke Structures. Protocols are described using Kripke structures as follows:

Definition 1 (Kripke Structure). *A Kripke structure (KS) is a finite state machine represented by a tuple* $\langle AP, S, s_0, \Sigma, R, L, \rangle$ *where* AP *is a set of*

atomic propositions; S is a finite set of states; $s_0 \in S$ is the initial state; Σ is a finite set of events; $R \subseteq S \times \Sigma \times S$ is the transition relation; and $L : S \to 2^{AP}$ is the state labelling function.

We consider that the transitions in a Kripke structure trigger with respect to a clock. At each clock cycle, the KS checks for the presence of input/output events that can trigger a transition from the current state. If no input/output triggers are present, the transition using the event T (or T') is made. In case there is no T or T'-transition, the protocol remains in the current state. The relations $(s, a, s') \in R$ will be represented by $s \xrightarrow{a} s'$. Given two KS P_1 and P_2 using a shared clock, their combined behavior is given by their *parallel composition* as follows:

Definition 2 (Parallel Composition). *Given two Kripke structures $P_1 = \langle AP_1, S_1, s_{0_1}, \Sigma_1, R_1, L_1\rangle$ and $P_2 = \langle AP_2, S_2, s_{0_2}, \Sigma_2, R_2, L_2, \rangle$, their parallel composition, denoted by $P_1||P_2$, is $\langle AP_{1||2}, S_{1||2}, s_{0_{1||2}}, \Sigma_{1||2}, R_{1||2}, L_{1||2}\rangle$ where $AP_{1||2} = AP_1 \cup AP_2$; $S_{1||2} = S_1 \times S_2$; $s_{0_{1||2}} = (s_{0_1}, s_{0_2})$; and $\Sigma_{1||2} \subseteq \Sigma_1 \times \Sigma_2$. $R_{1||2} \subseteq S_{1||2} \times \Sigma_{1||2} \times S_{1||2}$ such that*

$$(s_1 \xrightarrow{\sigma_1} s_1') \wedge (s_2 \xrightarrow{\sigma_2} s_2') \Rightarrow ((s_1, s_2) \xrightarrow{(\sigma_1,\sigma_2)} (s_1', s_2'))$$

Finally, $L_{1||2}((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$.

We restrict the scope of this paper to protocols that can be represented as *deterministic* Kripke structures only. A Kripke structure is deterministic if and only if for all states s, the number of outgoing transitions on any event a is less than equal to 1. The parallel composition of P_1 and P_2 in Fig. 2 (assuming a shared clock) is $P_1||P_2$ and is shown in Fig. 3.

Model of Specifications. `ACTL` is a branching time temporal logic with universal path quantifiers. It is defined over a set of propositions using temporal and boolean operators as follows:

$$\Psi \to P \mid \neg P \mid tt \mid ff \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \mathtt{AX}\Psi \mid \mathtt{A}(\Psi\ \mathtt{U}\ \Psi) \mid \mathtt{AG}\Psi$$

Semantics of an `ACTL` formula, φ denoted by $[\![\varphi]\!]_M$ are given in terms of set of states in a Kripke structure (or a KS), M, which satisfies the formula (see Fig. 4). A state $s \in S$ is said to satisfy a `ACTL` formula φ, denoted by $M, s \models \varphi$, if $s \in [\![\varphi]\!]_M$. Typically, the context of the semantics, i.e., M in $[\![\ ]\!]_M$ is implicit, and omitted. We also say that $M \models \varphi$ to indicate $M, s_0 \models \varphi$. In this paper, we restrict ourselves to formulas where negations are applied to propositions only.

4.1 Protocol Converters

The composition $P_1||P_2$ (Fig. 3) represents the unconstrained behaviour of the protocols including undesirable paths introduced due to mismatches. A converter is needed to bridge the mismatches appropriately. In this section, we introduce

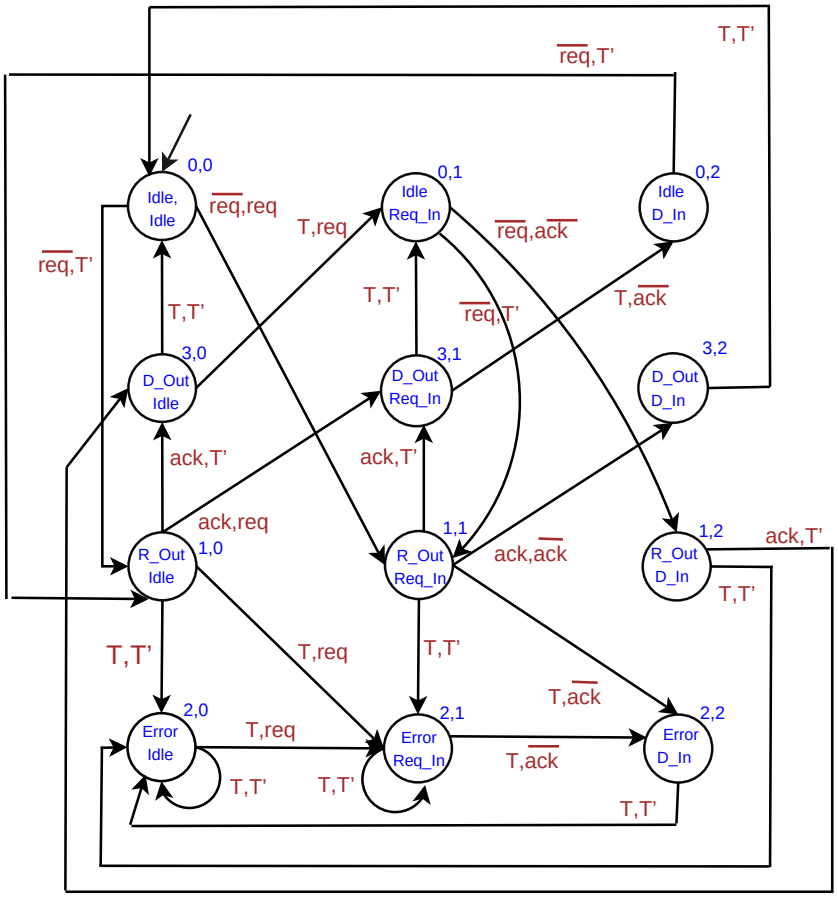


Fig. 3. Unrestricted composition of producer-consumer protocol pair: $P_1||P_2$.

converters and also the control actions of a converter (such as event blocking, event buffering and generation of extra signals) by introducing a new composition of the participating protocols with the converter.

Definition 3 (Converter). *A converter $\mathcal{C}$ for two protocols P_1 and P_2 is a Kripke structure $\langle AP_{\mathcal{C}}, S_{\mathcal{C}}, s_{\mathcal{C}0}, \Sigma_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}}\rangle$ such that $AP_{\mathcal{C}} = \emptyset$ and $\Sigma_{\mathcal{C}} = (\Sigma_1 \times \Sigma_2) \cup \{(*,*)\}$.*

In the above, the event-element $(*,*)$ is a wild-card event tuple, short-hand form of denoting any event-pairs from $\Sigma_1 \cup \Sigma_2$. The composition of a converter with the protocols is performed using the following rule: inputs to (outputs from) a protocol are outputs from (inputs to) the converter, i.e., the participating protocols communicate via the converter which acts as an intermediary. Input and output on the same event are *duals* and we will say that $\mathcal{D}(a,b)$ evaluates to true if $a = \sigma$ $(\bar{\sigma})$ and $b = \bar{\sigma}$ (σ) or if either a and/or b is the wildcard event

$$1 : [\![p]\!] = \{s \mid p \in L(s)\} \quad 2 : [\![\neg p]\!] = \{s \mid p \notin L(s)\} \quad 3 : [\![tt]\!] = S \quad 4 : [\![ff]\!] = \emptyset$$
$$5 : [\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!] \quad 6 : [\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$$
$$7 : [\![\texttt{AX}\varphi]\!] = \{s | \forall s \longrightarrow s' \wedge s' \in [\![\varphi]\!]\}$$
$$8 : [\![\texttt{A}(\varphi\ \texttt{U}\ \psi)]\!] = \{s | \forall s = s_1 \longrightarrow s_2 \longrightarrow \ldots \wedge \exists j.s_j \in [\![\psi]\!]\ \wedge \forall i < j.s_i \in [\![\varphi]\!]\}$$
$$9 : [\![\texttt{AG}\varphi]\!] = \{s | \forall s = s_1 \longrightarrow s_2 \longrightarrow \ldots \wedge \forall i.s_i \in [\![\varphi]\!]\}$$

Fig. 4. Semantics of ACTL

$*$. We extend $\mathcal{D}$ to operate on pairs of signals, where $\mathcal{D}((a,b),(c,d))$ evaluates to true *iff* both $\mathcal{D}(a,c)$ and $\mathcal{D}(b,d)$ evaluate to true.

After establishing the i/o relationship between a converter and the participating protocols, we now define the control of a converter over the protocols using the // operator as follows.

Definition 4 (Lock-Step Converter Composition). *Given the KS* $P_1||P_2 = \langle AP_{1||2}, S_{1||2}, s_{0_{1||2}}, \Sigma_{1||2}, R_{1||2}, L_{1||2}\rangle$ *and a converter* $\mathcal{C} = \langle AP_{\mathcal{C}}, S_{\mathcal{C}}, s_{\mathcal{C}0}, \Sigma_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}}\rangle$*, the lock-step composition* $\mathcal{C}//(P_1||P_2) = \langle AP_{1||2}, S_{\mathcal{C}//(1||2)}, s_{0_{\mathcal{C}//1||2}}, \Sigma_{1||2}, R_{\mathcal{C}//(1||2)}, L_{\mathcal{C}//(1||2)}\rangle$ *such that:*

1. $S_{\mathcal{C}//(1||2)} \subseteq S_{\mathcal{C}} \times S_{1||2}$*;*
2. $s_{0_{\mathcal{C}//1||2}} = (s_{0_{\mathcal{C}}}, s_{0_{(1||2)}})$*;*
3. $R_{\mathcal{C}//(1||2)} \subseteq S_{\mathcal{C}//(1||2)} \times \Sigma_{1||2} \times S_{\mathcal{C}//(1||2)}$ *where* $s_{\mathcal{C}//(1||2)} \overset{(\sigma_1,\sigma_2)}{\longrightarrow} s'_{\mathcal{C}//(1||2)} \in R_{\mathcal{C}//(1||2)}$ *when*

$$\begin{bmatrix} s_{\mathcal{C}} \overset{\sigma_1^c,\sigma_2^c}{\longrightarrow} s'_{\mathcal{C}} \ \wedge\ s_{1||2} \overset{(\sigma_1,\sigma_2)}{\longrightarrow} s'_{1||2} \\ \bigwedge \\ \mathcal{D}(\sigma_1^c,\sigma_1)\ \wedge\ \mathcal{D}(\sigma_2^c,\sigma_2) \end{bmatrix} \Rightarrow s_{\mathcal{C}//(1||2)} \overset{(\sigma_1,\sigma_2)}{\longrightarrow} s'_{\mathcal{C}//(1||2)}$$

4. $L_{\mathcal{C}//(1||2)}(s_{\mathcal{C}}, s_{1||2}) = L_{1||2}(s_{1||2})$

The transition relation of the protocols composed with a converter ensures that protocols move only when the converter allows that move. As such the lock-step composition // is different from unrestricted composition (Definition 2).

5 Tableau-Based Protocol Conversion

Protocol conversion, in addition to reconciling the mismatches, also requires that certain desired behavior is exhibited by the composition of the participating protocols. These desired functionalities are described by a set of ACTL formulas. We will denote this set as Ψ. The converter synthesis problem for protocol conversion is, therefore,

$$\exists \mathcal{C} : \forall \varphi \in \Psi :\ \mathcal{C}//(P_1||P_2) \overset{?}{\models} \varphi$$

I.e. is there a converter $\mathcal{C}$ for P_1 and P_2 such that the given protocols in the presence of $\mathcal{C}$ conforms to all the properties defined by formulas in Ψ?

We present a tableau-based technique for performing protocol conversion using ACTL specifications. This technique has the following advantages:

1. Local exploration of state-space of the protocols: the protocol transition systems are explored as and when needed to prove or disprove the existence of a converter.
2. On-the-fly synthesis of converter: generation of the tableau results in the generation of a converter if such a converter exists.
3. Sound and complete: a converter generated using the tableau is correct by construction.

The tableau rules are of the following form:

$$\frac{c//s \models \Psi}{c_1//s_1 \models \Psi_1 \ldots c_n//s_n \models \Psi_n}$$

where s is a state in $P_1||P_2$ and $s_1, s_2, \ldots, s_n$ are a function of s, while $c_1, c_2, \ldots, c_n$ are the states of the converter to be generated. Similarly, Ψ is the set of formulas to be satisfied by s whereas $\Psi_1, \Psi_2, \ldots, \Psi_n$ are some derivatives of Ψ. The numerator represents the obligation to be satisfied, i.e., s in the presence of a converter state c must satisfy the set of formulas in Ψ and in order to realize that, each obligation in the denominator must be fulfilled.

The tableau is initiated by a tableau-node resulting from the composition of the start state of the unrestricted composition of P_1 and P_2 and a generated start state c_0 of a possible converter. The construction proceeds by matching the current tableau-node with the numerator of a tableau rule and obtaining the denominator which constitutes the next set of tableau-nodes. Fig. 5 presents our tableau-rules for converter synthesis and protocol conversion.

The rule emp corresponds to the case when there is no obligation to be satisfied by the composition; any converter is possible in this case, i.e., the converter allows all possible behavior of the protocol composition at state s.

The prop rule states that a converter is synthesizable only when the obligation of satisfying the proposition is released by the protocol composition state s; otherwise there exists no converter. Once the propositional obligation is met, the subsequent obligation is to satisfy the rest of the formulas in the set Ψ.

The $\wedge$-rule states that the satisfaction of the conjunctive formula depends on the satisfaction of each of the conjuncts. The $\vee$-rules are the duals of $\wedge$-rule. The Rule $\mathtt{unr}_{au}$ depends on the semantics of the temporal operator AU. A state is said to satisfy $\mathtt{A}(\varphi\ \mathtt{U}\ \psi)$ if and only if it either satisfies ψ or satisfies φ and evolves to new states each of which satisfies $\mathtt{A}(\varphi\ \mathtt{U}\ \psi)$. These equivalences can be directly derived from the semantics of AU formulas. Similarly, $\mathtt{AG}\varphi$ is satisfied by states which satisfy φ and whose all next states satisfy $\mathtt{AG}\varphi$ (Rule $\mathtt{unr}_{ag}$).

Finally, $\mathtt{unr}_s$ is applied when the formula set in the numerator Ψ consists formulas of the form $\mathtt{AX}\varphi$. Satisfaction of these formulas demands that all next states of the $c//s$ must satisfy every φ where $\mathtt{AX}\varphi \in \Psi$, i.e., $c//s$ satisfies all elements of Ψ_{AX}.

$$\text{emp}\ \frac{c//s \models \{\}}{\bullet} \qquad \text{prop}\ \frac{c//s \models [\{p\} \cup \Psi]}{c//s \models \Psi}\ p \in L(s) \vee \models p$$

$$\wedge\ \frac{c//s \models [\{\varphi_1 \wedge \varphi_2\} \cup \Psi]}{c//s \models [\{\varphi_1, \varphi_2\} \cup \Psi]}$$

$$\vee_1\ \frac{c//s \models [\{\varphi_1 \vee \varphi_2\} \cup \Psi]}{c//s \models [\{\varphi_1\} \cup \Psi]} \qquad \vee_2\ \frac{c//s \models [\{\varphi_1 \vee \varphi_2\} \cup \Psi]}{c//s \models [\{\varphi_2\} \cup \Psi]}$$

$$\mathtt{unr}au\ \frac{c//s \models [\{\mathtt{A}(\varphi\ \mathtt{U}\ \psi)\} \cup \Psi]}{c//s \models [(\psi \vee (\varphi \wedge \mathtt{AXA}(\varphi\ \mathtt{U}\ \psi))) \cup \Psi]}$$

$$\mathtt{unr}ag\ \frac{c//s \models [\{\mathtt{AG}\varphi\} \cup \Psi]}{c//s \models [(\varphi \wedge \mathtt{AXAG}\varphi) \cup \Psi]}$$

$$\mathtt{unr}_s\ \frac{c//s \models \Psi}{\exists \pi \subseteq \Pi.\ (\forall \sigma \in \pi.\ c_\sigma//s_\sigma \models \Psi_{AX})} \begin{cases} \Psi_{\mathtt{AX}} = \{\varphi_k \mid \mathtt{AX}\varphi_k \in \Psi\} \\ \Pi = \{\sigma \mid (s) \xrightarrow{\sigma} (s_\sigma)\} \\ c_\sigma = c' : c \xrightarrow{\sigma'} c' \wedge \mathcal{D}(\sigma, \sigma') \end{cases}$$

Fig. 5. Tableau Rules for converter generation

Note that unrestricted behavior of the protocol (where c allows all the transitions from s) may not be able to satisfy this obligation; however, a converter can be generated such that c allows a *subset* (π) of all possible transitions from s (Π) and these transitions lead to states which satisfy the formulas in Ψ_{AX} (as stated by the unr_s rule). If there are k outgoing transitions from s, there are 2^k choices; however, the tableau considers k choices (one for each successor) and unr_s leads to k possible denominators–one denominator per transition from s. These choices can then be aggregated to represent all enabled transitions of s. Any denominators that return failure result in the corresponding successors of s being disabled by the converter[1].

Finitizing the tableau. It is important to note that the resulting tableau can be of infinite depth as each recursive formula expression AU or AG can be unfolded infinitely many times.

This problem arising due to unbounded unfolding of the formula expressions can be addressed using the fixed point semantics of the formulas $\mathtt{AG}\varphi$ and $\mathtt{A}(\varphi\ \mathtt{U}\ \psi)$. The former is a greatest fixed point formula while the later is a least fixed point formula.

$$\mathtt{AG}\varphi \equiv Z_{\mathtt{AG}} =_\nu \varphi \wedge AXZ_{\mathtt{AG}},$$
$$\mathtt{A}(\varphi\ \mathtt{U}\ \psi) \equiv Z_{\mathtt{AU}} =_\mu \psi \vee (\varphi \wedge AXZ_{\mathtt{AU}})$$

[1] However, instead of examining all possible subsets, it is sufficient for the converter state c to allow just one transition from s such that $c'//s'$ satisfies all formulas in Ψ_{AX}, although such a converter may be too restrictive.

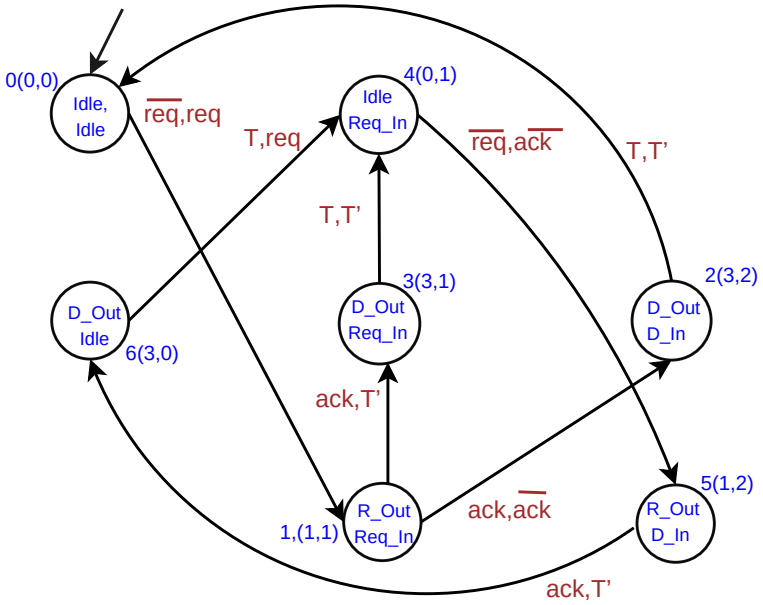


Fig. 6. The combined system $\mathcal{C}//P_1||P_2$ (Figures 2 and 3).

The greatest (least) solution for $Z_{\texttt{AG}}$ ($Z_{\texttt{AU}}$) is the semantics of $\texttt{AG}(\varphi)$. It can be shown (details are omitted) that satisfaction of the greatest fixed point formula is realized via loops in the model; while satisfaction of the least fixed point formula demands the existence of a loop-free tableau. As such, if a tableau-node $c'//s \models \Psi$ is visited and there exists a prior node $c//s \models \Psi$ i.e. the same tuple s paired with the same Ψ is seen in a tableau path, we verify whether there exists a least fixed point formula $\texttt{AU}$ in Ψ; if such a formula is present, we say that the tableau path resulted in an unsuccessful path; otherwise, we terminate the tableau path successfully and equate c' with c (a loop in the converter is generated).

Complexity. The tableau considers all possible subformulas of the given set of desired properties. Each such subformula is paired with all possible states in the protocol-pair. The complexity of the tableau construction is $O(|S| \times |\varphi|)$ where S is the number of states in the protocol pairs and $|\varphi|$ is the size of the formula expressing the desired properties (the conjunction of all properties).

The following theorem follows from the above discussion.

Theorem 1 (Sound and Complete). *Two protocols P_1 and P_2 are compatible* wrt *to a set Ψ of ACTL formulas ($\forall \varphi \in \Psi : \mathcal{C}//(P_1||P_2) \models \varphi$) if and only if there exists a successful tableau for the tableau node $c_0//s_0 \models \Psi$ where s_0 is the start state of $P_1||P_2$ and c_0 is the start state of $\mathcal{C}$.*

6 Live Converters

For two protocols P_1 and P_2 and a set of ACTL specifications Ψ, the tableau-based approach formulated above can generate multiple converters. This is because the rules $\vee$ and unr_s may lead to several choices for constructing the tableau-node denominator. Some of the generated converters, therefore, may disable protocol-behavior and lead to conformance of the desired property vacuously. For example, properties of the form $\phi \Rightarrow \psi$ will be satisfied by the converted protocol pairs if ϕ is not satisfied.

To counter this situation, we can impose further restrictions on converter generation by including *liveness* conditions that need to be satisfied by the resulting system $\mathcal{C}//P_1||P_2$. Such liveness conditions can be defined using ACTL and used as input to tableau along with desired properties. The goal will be avoid construction of converters that will lead to violation of liveness properties by the converted protocols.

For the producer-consumer example, we use the following liveness conditions:

- AGA($true$ U D_In), AGA($true$ U D_Out): $\mathcal{C}$ must allow the producer to always eventually write data and the consumer to always eventually consumer some data.
- AG[$D\_Out \Rightarrow$ (D_In $\vee$ AXA($\neg R\_Out$ U D_In))]: Once data is written, no further requests are allowed before a read operation is performed.

The converter synthesis process for the producer-consumer is presented in section 9. The combined system $\mathcal{C}//(P_1||P_2)$ is shown in Fig. 6. The converter $\mathcal{C}$ obtained for the producer-consumer example is a maximally permissive converter that ensures that $\mathcal{C}//(P_1||P_2)$ satisfies the above liveness constraints. For better readability in Fig. 6, we have annotated each state with $i(j,k)$ where i denotes the state of the generated converter while j and k are states of P_1 (producer) and P_2 (consumer) respectively.

7 Results

A protocol conversion tool employing the tableau construction approach has been implemented by extending the NuSMV model checker [6]. The implementation takes as input the Kripke structure representation of two protocols P_1 and P_2 (obtained from NuSMV models) and a set Ψ of ACTL properties from the user. It proceeds by computing the parallel composition $P_1||P_2$ and then uses the tableau rules to realize the converter, if it exists. The results table (Tab.1) contains four columns. The first two columns contain the description and size (number of states) of the participating protocols. The ACTL properties used are shown in the third column with the size of the converter shown in column 4. The first five problems are well-known protocol conversion problems [16, 13]. The next problem is a producer-consumer example where the producer can produce multiple 8-bit data after each handshake whereas the slave can only read

$P_1(\|S_{P_1}\|)$	$P_1(\|S_{P_2}\|)$	ACTL Properties	$\mathcal{C}(\|S_C\|)$
Master (3)	Slave (3)	$\mathtt{AG}(\neg Req\_InUR\_Out)$, $\mathtt{AG}[R\_Out \Rightarrow ((Req\_In) \vee \mathtt{AXA}(\neg R\_Out\ \mathtt{U}\ Req\_In))]$, $\mathtt{A}(\neg G\_Out\ \mathtt{U}\ R\_Out)$, $\mathtt{A}(\neg Gnt\_In\ \mathtt{U}\ G\_Out)$	6
ABP sender(6)	NP receiver(4)	$\mathtt{AGA}(\neg A\_Out\ \mathtt{U}\ ACC)$, $\mathtt{AG}[A\_Out \Rightarrow (ACC \vee \mathtt{AXA}(\neg A\_Out\ \mathtt{U}\ ACC))]$	8
ABP receiver(8)	NP sender(3)	$\mathtt{AGA}(\neg A\_Out\ \mathtt{U}\ ACC)$, $\mathtt{AG}[(A+ \Rightarrow (ACC \vee \mathtt{AXA}(\neg A\_Out\ \mathtt{U}\ ACC))]$	8
Poll-End Receiver(2)	Ack-Nack Sender(3)	$\mathtt{AG}[Data\_Out \Rightarrow (Data\_In \vee \mathtt{AXA}(Data\_In\ \mathtt{U}\ Data\_Out))]$	6
Handshake (2)	Serial(2)	$\mathtt{AGA}(\neg A\ \mathtt{U}\ A')$, $\mathtt{AGA}(\neg B\ \mathtt{U}\ B')$, $\mathtt{AG}(A' \Rightarrow \mathtt{AXA}(\neg A'\ \mathtt{U}\ A))$	3
Multi-write master protocol(3) 8-bit Write	Single-read slave protocol(4) 8-bit Read	$\mathtt{AG}(\neg Error)$,$\mathtt{A}(\neg D\_Out\ \mathtt{U}\ Req\_In)$ $\mathtt{A}(\neg Req\_In\ \mathtt{U}\ R\_Out)$	8
Mutex Process 1 (3)	Mutex Process 2 (3)	$\mathtt{AG}(\neg critical1 \vee \neg critical2)$	7
MCP missionaries	MCP cannibals (30)	$\mathtt{AGAF}((MCP.missionaries = 0) \wedge (MCP.cannibals = 0))$	22
4-bit ABP Sender	Modified Receiver (166432)	$\mathtt{AGAF} sender.state = get$	14312

Table 1. Implementation Results

one 8-bit data after each handshake. The generated converter controls the communication between the two components such that paths where data is lost are never reached. The final three results are well-known NuSMV examples modified to create a mismatch. Note that size entry in the second column for the final two results refers to the combined size of the system (size of $P_1||P_2$) for these examples.

8 Conclusions and Future Directions

Protocol conversion to resolve protocol mismatches is an active research area. A number of solutions have been proposed. Some approaches require significant user input and guidance, while some only partly address the protocol conversion problem. Most formal approaches work on protocols that have unidirectional communication and use finite state machines to describe specifications. In this paper we propose a formal approach to protocol conversion which alleviates the above problems. Specifications are described in temporal logic and protocols are allowed to be bidirectional. A tableau-based approach using the model checking

framework is used to generate converters in polynomial time. We prove that the approach is sound and complete and provide implementation results.

The presented approach uses ACTL to describe desired specifications. The extension to the more expressive logic CTL requires minimal effort but the presence of existential formulas in CTL will increase the complexity to $EXPTIME$-complete as protocol conversion under CTL is equivalent to *module checking* [12, 1] problem. Similarly, tableau rules for LTL will result in $PSPACE$ complexity of protocol conversion. Future work includes the unification of various protocol conversion issues under the presented framework. The technique can be extended to resolve data-width mismatches [8], clock-mismatches [14] and interface-mismatches between protocols. Data-width mismatches occur when protocols have varying word-sizes. A converter must therefore ensure that no data is lost during inter-protocol communication. Clock-mismatches occur when protocols operate using clocks that may be running at different frequencies. Interface mismatches occur when protocols use inconsistent naming conventions for control signals, thus requiring the converter to perform event translation [5]. Another issue is the handling of *uncontrollable* actions [11, 1]. Some transitions in $P_1||P_2$ may be uncontrollable and therefore cannot be disabled. An extension to the presented tableau-based converter generation approach to generate a converter, if possible, under these additional restrictions is endeavored.

References

1. M. Antoniotti. *Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the Control-D system*. PhD thesis, New York University, New York, 1995.
2. Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for CTL*. In *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 388–397, June 1995.
3. G V Bochmann. Deriving protocol converters for communication gateways. *IEEE Transactions on Communications*, 38(9):1298–1300, September 1990.
4. F M Burg and N D Iorio. Networking of networks: Interworking according to osi. *IEEE Journal on Selected Areas in Communications*, 7(7):1131–1142, September 1989.
5. Kenneth L Calvert and Simon S Lam. Formal methods for protocol conversion. *IEEE Journal on Selected Areas in Communication*, 8(1):127–142, 1990.
6. R. Cavada, Alessandro Cimatt, E. Olivetti, M. Pistore, and M. Roveri. *NuSMV 2.1 User Manual*, June 2003.
7. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
8. Vijay D'Silva, S Ramesh, and Arcot Sowmya. Synchronous protocol automata : A framework for modelling and verification of soc communication architectures. In *DATE*, pages 390–395, 2004.
9. Saurav Gorai, Saptarshi Biswas, Lovleen Bhatia, Praveen Tiwari, and Raj S. Mitra. Session 42: simulation assisted formal verification: Directed-simulation assisted formal verification of serial protocol and bridge. In *Proceedings of the 43rd annual conference on Design automation DAC '06*, pages 731 – 736, 2006.
10. P. Green. Protocol conversion. *IEEE Transactions on Communications*, 34(3):257–268, March 1986.

11. R. Kumar and S. S. Nelvagal. Protocol conversion using supervisory control techniques. In *IEEE International Symposium on Computer-Aided Control System Design*, pages 32–37, 1996.
12. O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164:322–344, 2001.
13. S Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, 1988.
14. J Lefebvre. *Esterel v7 Reference Manual-Initial Standardization Proposal*, 2005.
15. K. Okumura. A formal protocol conversion method. In *ACM SIGCOMM 86 Symposium*, pages 30–37, 1986.
16. R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *International Conference on Computer Aided Design ICCAD*, 2002.
17. J. C. Shu and Ming T. Liu. A synchronization model for protocol conversion. *Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies. Technology: Emerging or Converging? INFOCOM '89*, pages 276–284, 1989.

9 Appendix A: Tableau Generation and Converter Extraction

This section presents the tableau generation and converter extraction for the producer-consumer example in Fig.2.

A tableau consists of *nodes* and *links*. A *node* t corresponds to a state s in $P_1||P_2$ (called $t.state$) and two sets of ACTL formulas: *now* ($t.now$) and *next* ($t.next$). $t.now$ contains formulas to be satisfied by $t.state$ and $t.next$ is used to hold AX formulas that (after removing the AX operator) are to be satisfied by the successors of $t.state$ using the unr_s rule. Formulas in $t.now$ are broken down using the tableau rules in Fig. 5 and any future commitments are moved to $t.next$. Once $t.now$ is empty, all future commitments are passed to children nodes, that are essentially tableau nodes corresponding to the successors of $t.state$. Tableau nodes are connected to each other using *links*. Each node also has a *type*, used as a status flag. A node with no children is called a $PURE$ node. A $PURE$ node can be expanded into one of the following two types of nodes: OR_NODE or AX_NODE.

A node of the type OR_NODE is formed when an OR formula is encountered in the *now* set of a node. An OR formula of the type $\varphi_1 \vee \varphi_2$ formula can be satisfied by a state s in two possible ways: if s satisfies φ_1 or if s satisfies φ_2 (rules $\vee_1$ and $\vee_2$ in Fig. 5). Therefore, if such a formula is present in $t.now$ of a node t, two children nodes, corresponding to the state $t.state$ are formed. All formulas in $t.now$ and $t.next$ are copied to the *now* and *next* sets of the children except that the OR formula responsible for the split is replaced by φ_1 in child 1 and φ_2 in child 2. On the other hand, a node t is expanded to become a type AX_NODE when $t.now$ is empty and when $t.next$ contains at least once AX formula. Following rule unr_s in Fig. 5, the state $t.state$ satisfies all *next* formulas when all its successors satisfy these future commitments (all formulas

in $t.next$ after deleting the preceding AX). Hence the node is expanded to become a type AX_NODE where each child node corresponds to a successor of $t.state$ and contains in its *now* set, all formulas in $t.next$ after their preceding AX have been removed.

The tableau construction for the producer-consumer example starts at the construction of the *root* node of the tableau corresponding to the state (s_0, t_0) of $P_1||P_2$ (Fig. 2) with its *now* set containing all formulas and liveness conditions (described in sections 4 and 6). The root node is shown in Tab. 2.

now	next
$\texttt{AG}\neg Error$	
$\texttt{AG}[\neg D\_Out \vee (D\_In \vee \texttt{AXA}(\neg D\_In \texttt{ U } D\_Out))]$	
$\texttt{AGA}(true \texttt{ U } D\_In)$	
$\texttt{AGA}(true \texttt{ U } D\_Out)$	
$\texttt{AG}[\neg D\_Out \vee (\ D\_In \ \vee \ \texttt{AXA}(\neg R\_Out \texttt{ U } D\_In)\ )]$	

Table 2. Node 0. $state = (s_0, t_0)$, $type = PURE$.

After the creation of the root node, we successively apply the tableau rules given in Fig. 5 to the formulas in the *now* set and move any AX type subformulas to *next* until *now* is empty. Due to the presence of $\vee$-type subformulas (such as the sub-formula $D\_In \vee (true \wedge \texttt{AGA}(true \texttt{ U } D\_In))$ obtained from $\texttt{AGA}(true \texttt{ U } D\_In)$), the root node is expanded and re-expanded as an OR_NODE to have several children node (each corresponding to one satisfaction possibility of the various $\vee$ formulas). However, for the producer-consumer example, only one valid $PURE$ node (node $0b$) remains (shown in Tab. 3). This node is a child of the root node 0 which is now of type OR_NODE (as described above).

now	next
	$\texttt{AXAG}\neg Error$
	$\texttt{AXAG}[\neg D\_Out \vee (D\_In \vee \texttt{AXA}(\neg D\_In \texttt{ U } D\_Out))]$
	$\texttt{AXAGA}(true \texttt{ U } D\_In)$
	$\texttt{AXAGA}(true \texttt{ U } D\_Out)$
	$\texttt{AXAG}[\neg D\_Out \vee (\ D\_In \ \vee \ \texttt{AXA}(\neg R\_Out \texttt{ U } D\_In)\ )]$
	$\texttt{AXA}(true \texttt{ U } D\_In)$
	$\texttt{AXA}(true \texttt{ U } D\_Out)$

Table 3. Node $0b$. $state = (s_0, t_0)$, $type = PURE$.

We use the unr_s rule and expand the $PURE$ node $0b$ to have two children nodes, one for each of the two successors of (s_0, t_0), that have the same *now* sets as $0b.next$ (after the preceding AX). The above process of applying tableau

rules to break down formulas in a node's *now* set followed by the application is repeated until all $PURE$ type nodes have been exhausted (expanded or deleted due to failure).

9.1 Converter Extraction

The converter from a tableau is extracted by traversing the nodes of the tableau starting from its root node. A successful tableau contains only nodes of the type AX_NODE or OR_NODE as all $PURE$ type nodes are either deleted (due to failure) or expanded.

If a node of type OR_NODE is encountered, *one* valid child is chosen for further traversal. However, when a node of type AX_NODE is encountered, *all* valid children nodes are iteratively traversed. Each such child of a AX_NODE type node t corresponds to an enabled successor of the state $t.state$ which leads to the satisfaction of all future commitments contained in $t.next$. Of course, there may be some successors of $t.state$ which do no correspond to any child of t. Such successors of $t.state$ lead to the failure of future commitments and therefore, the links to their respective nodes are removed from t during converter construction. AX_NODE type nodes are also special because each such node corresponds to a state in the *converter*. All links to the children of an AX_NODE type node represent actual transitions in $P_1||P_2$ and therefore the generated converter must enable all such transitions. For example, if a node t corresponding to state s in $P_1||P_2$ has two children t' and t'' corresponding to successors s' and s'' of s, such that $s \xrightarrow{\sigma_1} s'$ and $s \xrightarrow{\sigma_2} s''$, we construct a converter state c which has two transitions $c \xrightarrow{\sigma_1'} c'$ and $c \xrightarrow{\sigma_2'} c''$ where $\mathcal{D}(\sigma_1, \sigma_1')$ and $\mathcal{D}(\sigma_2, \sigma_2')$. Further, c' (c'') corresponds to t' if it is a type AX_NODE or otherwise to some AX_NODE t_1' such that t links to t_1' through a number of OR_NODE type nodes.

For the producer consumer example in Fig. 2, Tab. 4 shows how each node of type AX_NODE corresponds to a generated converter state. The generated converter results in the combined system $\mathcal{C}//(P_1||P_2)$ in Fig. 6.

Node	$P_1\|\|P_2$ **state**	**Converter state**	**Enabled successors of** $\mathcal{C}$
Node 0b	(s_0, t_0)	c_0	c_1
Node 2b	(s_1, t_1)	c_1	c_2, c_3
Node 3b	(s_3, t_2)	c_2	c_0
Node 4b	(s_3, t_1)	c_3	c_4
Node 5b	(s_0, t_1)	c_4	c_5
Node 6b	(s_1, t_2)	c_5	c_6
Node 7b	(s_3, t_0)	c_6	c_4

Table 4. Converter Generation From Tableau

2. Worst Case Reaction Time Analysis of Concurrent Reactive Programs

Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden

Department of Computer Science,
Christian-Albrechts-Universität zu Kiel

Notes:

Worst Case Reaction Time Analysis of Concurrent Reactive Programs

Marian Boldt Claus Traulsen Reinhard von Hanxleden

Dept. of Computer Science
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
{mabo,ctr,rvh}@informatik.uni-kiel.de

Abstract

Reactive programs have to react continuously to their inputs. Here the time needed to react with the according output is important. While the synchrony hypothesis takes the view that the program is infinitely fast, real computations take time. Similar to the traditional *Worst Case Execution Time* (WCET), the *Worst Case Reaction Time* (WCRT) of a program determines the maximal time for one reaction.

In this paper, we present an algorithm to determine the WCRT of a program written in the synchronous language Esterel. This value gives an upper bound for the execution time when the program is executed on a reactive processor. Specifically, we consider the execution of the Esterel program on the *Kiel Esterel Processor* (KEP), a reactive processor that can execute Esterel-like instructions. Here the WCRT directly determines an upper bound on the instruction cycles per logical tick. The WCRT also gives a guideline for the execution time when the Esterel program is compiled to software by a simulation-based approach.

We have implemented the WCRT analysis algorithm as part of an Esterel compiler for the KEP and have measured an accuracy of analysis results of about 40% on average.

Key words: Synchronous Languages, Esterel, Worst Case Execution Time, Worst Case Reaction Time, Instantaneous Reachability

1 Introduction

Many embedded systems belong to the class of *reactive systems*, which continuously react to inputs from the environment by generating corresponding outputs. For these systems, exact timing information or at least an upper bound of the execution time is crucial. To perform an exact *Worst Case Execution Time* (WCET) analysis is difficult, and in general not possible for Turing-complete languages. It typically imposes fairly strong restrictions on the analyzed code, such as a-priori known upper bounds on loop iteration counts, and even then control flow analysis is often overly conservative [18,5]. Furthermore, even for a linear sequence of instructions, typical modern architectures make it difficult to predict how much time exactly the execution of these instructions consumes, due to pipelining, out-of-order execution,

argument-dependent execution times (*e. g.*, particularly fast multiply-by-zero), and caching of instructions and/or data. Finally, if external interrupts are possible or if an operating system is used, it becomes even more difficult to predict how long it really takes for an embedded system to react to its environment. Despite the advances already made in the field of WCET analysis, it appears that most practitioners today still resort to extensive testing plus adding a safety margin to validate timing characteristics. To summarize, performing conservative yet tight WCET analysis appears by no means trivial and is still an active research area.

One step to make WCET analysis of reactive applications more feasible is to choose a programming language that provides direct, predictable support for reactive control flow patterns. One suitable candidate for this is the synchronous language Esterel [2], which has been developed for programming control-oriented, embedded systems. It directly supports concurrency and multiple forms of preemption. Based on the *synchrony hypothesis*, it offers determinism even for concurrent components. The execution of Esterel programs is divided into (logical) ticks, each of which conceptually takes no time. Esterel forbids programs with a potentially unbounded number of statements to be performed within a tick. This is reflected in the rule that there cannot be *instantaneous loops*; within a loop body, each statically feasible path must contain at least one tick-delimiting instruction. The restricted nature of Esterel and its sound mathematical semantics allow formal analysis of Esterel programs and make the computation of a WCET for Esterel programs achievable.

In addition to choosing a suitable programming language, the feasibility of WCET analysis crucially depends on the execution platform. A relatively new approach for control-oriented reactive-systems are *reactive processors* [22,14,15]. These processors directly support reactive control flow, such as preemption and concurrency. In this paper we will use the *Kiel Esterel Processor* (KEP), a reactive processor based on the synchronous language Esterel, to show that timing analysis is practical for reactive processors, hence making the reactive processing approach particularly well suited for hard real-time systems. There are two main factors that contribute to this, on the one hand the synchronous execution model of Esterel, and on the other hand the direct implementation of this execution model on a reactive processor. Furthermore, reactive processors are not designed to optimize (average) performance for general purpose computations, and hence do not have a hierarchy of caches, pipelines, branch predictors, etc. This leads to a simpler design and execution behavior and further facilitates WCET analysis.

As we here are investigating the timing behavior for reactive systems, we are concerned with computing the maximal time it takes to compute a single reaction, that is the time from given input events to generated output events. Therefore we call this analysis a *Worst Case Reaction Time* (WCRT) analysis. The WCRT determines the maximal rate for the interaction with the environment. Whether WCRT can be formulated as a classical WCET problem or not depends on the implementation approach. If the implementation is based on sequentialization such that there exist two dedicated points of control at the beginning and the end of each reaction, respectively, then WCRT can be formulated as WCET problem; this is the case, for example, if one "automaton function" is synthesized, which is called during each reaction. If, however, the implementation builds on a concurrent model

of execution, where each thread maintains its own state of control across reactions, then WCRT requires not only determining the maximal length of pre-defined instruction sequences, as in WCET, but one also has to analyze the possible control point pairs that delimit these sequences. Thus, WCRT is more elementary than WCET in the sense that it considers single reactions, instead of whole programs, and at the same time WCRT is more general than WCET in that it is not limited to pre-defined control boundaries.

The contribution of this paper is a WCRT analysis of complete Esterel programs including concurrency and preemption. The analysis computes the WCRT in terms of KEP instruction cycles, which roughly match the number of executed Esterel statements. As part of the WCRT analysis, we also present an approach to calculate potential instantaneous paths, which may be used in compiler analyses and optimizations that go beyond WCRT analysis.

In the following section, we consider related work. In Section 3 we will give an introduction into the synchronous model of computation for Esterel and the KEP. We outline the generation of a *Concurrent KEP Assembler Graph* (CKAG), an intermediate graph representation of an Esterel program, which we use for our analysis. Section 4 explains our algorithm in detail, while Section 5 gives experimental results, comparing the computed number of reactions with values obtained from exhaustive simulation. The paper concludes in Section 6.

2 Related Work

As mentioned in the introduction, there exist numerous approaches to classical WCET analysis. For a survey see, *e. g.*, Puschner and Burns [20]. These approaches usually consider (subsets) of general purpose languages, such as C, and take informations on the processor designs and caches into account.

Regarding the analysis of synchronous programs, Logothetis, Schneider and Metzler [16,17] have employed model checking to perform a precise WCET analysis for the synchronous language Quartz, which is similar to Esterel. However, their problem formulation was different from the WCRT analysis problem we are addressing. They were interested in computing the number of ticks required to perform a certain computation, such as a primality test, which we would actually consider to be a transformational system rather than a reactive system [12]. We here instead are interested in how long it may take to compute a single tick, which can be considered an orthogonal issue.

One important problem that must be solved when performing WCRT analysis for Esterel is to determine whether a code-segment is reachable instantaneously or delayed or both. This is related to the well-studied property of *surface* and *depth* of an Esterel program, *i. e.*, to determine whether a statement is instantaneous reachable or not, which is also important for schizophrenic Esterel programs [2]. This was addressed in detail by Tardieu and de Simone [23]. They also point out that an exact analysis of instantaneous reachability has NP complexity. We, however, are not only interested whether a statement can be instantaneous, but also whether it can be non-instantaneous.

Beside being executed on a reactive processors, Esterel programs can be synthesized to hardware [1] or compiled into software, *e. g.*, C-code; see Edwards [10]

for an overview. Currently, the most efficient compilation schemes are simulation based [9,7,19,11]: the Esterel program is organized according to some kind of graphical structure and its current state is stored in a data-structure on the application level, *e. g.*, a bit-vector. Based on this vector, the current actions in the graph are triggered. While this approach produces fairly efficient code, both in size and in execution speed, it removes much of the structure from the Esterel-program, making the WCET analysis as hard as for "normal" C programs.

Ringler [21] considers the WCET analysis of C code generated from Esterel. But his approach is only feasible for the generation of circuit code [2], which scales well for large applications, but tends to be slower than the simulation based approach.

Li *et al.* [14] compute a WCRT of sequential Esterel programs directly on the source code. However, they did not address concurrency, and their source-level approach could not consider compiler optimizations. We perform the analysis on an intermediate level after the compilation, as a last step before the generation of assembler code. This also allows a finer analysis and decreases the time needed for the analysis.

The KEP contains a *TickManager* [14], which monitors how many instructions are executed in the current logical tick. To minimize jitter, a maximum number of instructions for each logical tick can be specified. If the current tick needs less instructions, the start of the next tick is delayed. If the tick needs more instructions, an error-output is set. Hence a tight, but conservative upper bound of the maximal instructions for one tick is of direct value for the KEP. See Li *et al.* [14] for details on the relation between the maximum number of instruction per logical tick and the physical timing constraints from the environment perspective.

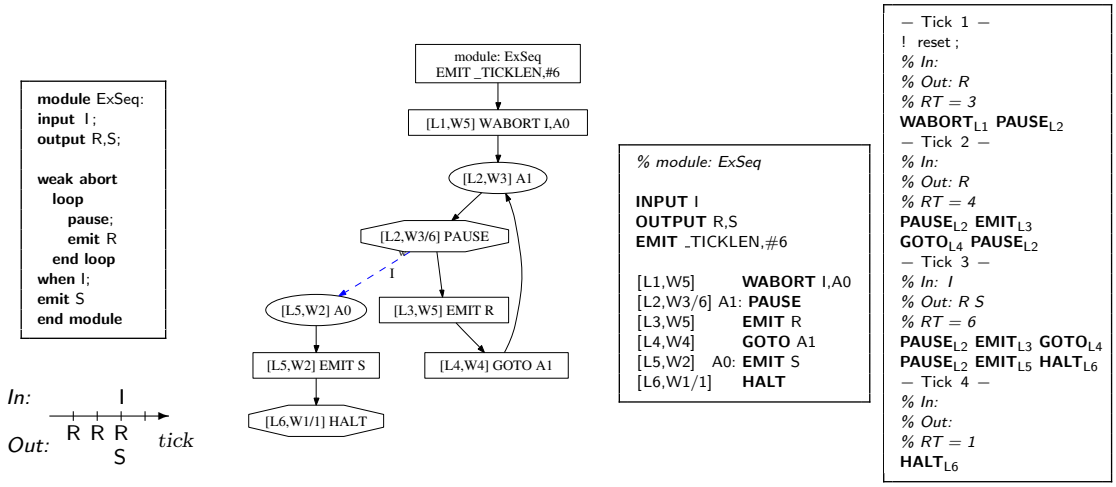
3 Esterel, KEP and the CKAG

Next we give a short overview of Esterel and the KEP. While our analysis is implemented in the compiler from Esterel to the KEP assembler, it is also of interest for other execution forms of Esterel. The analysis itself is performed on a graph representation of Esterel-programs, the CKAG.

3.1 Esterel

The execution of an Esterel program is divided into logical *instants*, or *ticks*, and communication within or across threads occurs via *signals*; at each tick, a signal is either *present* (emitted) or *absent* (not emitted). Esterel statements are either *transient*, in which case they do not consume logical time, or *delayed*, in which case execution is finished for the current tick. Per default statements are transient, and these include for example emit, loop, present, or the preemption operators. Delayed statements include pause, (non-immediate) await, and every. Esterel's parallel operator, ||, groups statements in concurrently executed threads. The parallel terminates when all its branches have terminated.

Esterel offers two types of preemption constructs. An *abortion* kills its body when an abortion trigger occurs. We distinguish *strong* abortion, which kills its body immediately (at the beginning of a tick), and *weak* abortion, which lets its body receive control for a last time (abortion at the end of the tick). A *suspension*



(a) Esterel code and sample trace (b) CKAG (c) KEP assembler (d) KEP sample trace

Fig. 1. A sequential Esterel example. The body of the KEP assembler program (without interface declaration and initialization of the TickManager) is annotated with line numbers L1–L6, which are also used in the CKAG and in the trace to identify instructions. The trace shows for each tick the input and output signals that are present and the reaction time (RT), in instruction cycles.

freezes the state of a body in the instant when the trigger event occurs.

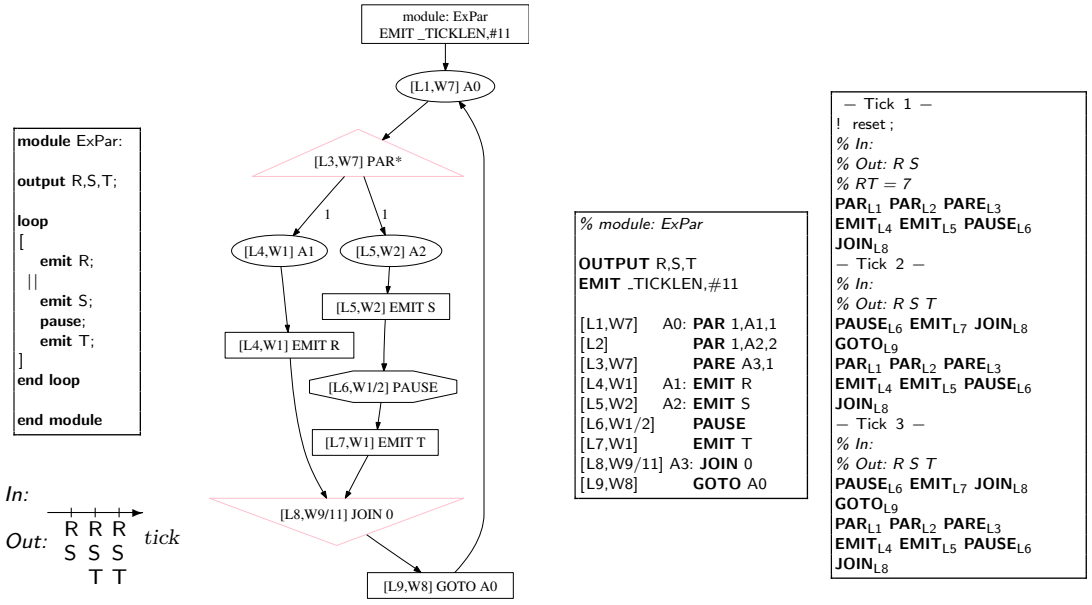
Esterel also offers an exception handling mechanism via the trap/exit statements. An exception is *declared* with a trap scope, and is *thrown* (*raised*) with an exit statement. An exit T statement causes control flow to move to the end of the scope of the corresponding trap T declaration. This is similar to a goto statement, however, there are further rules when traps are nested or when the trap scope includes concurrent threads. If one thread raises an exception and the corresponding trap scope includes concurrent threads, then the concurrent threads are weakly aborted; if concurrent threads execute multiple exit instructions in the same tick, the outermost trap takes priority.

A simple sequential Esterel example ExSeq can be found in Figure 1(a). From the second instant on it will continuously emit the signal R. When the input I occurs, it emits R one last time. In the same instant, it also emits S and terminates. This behavior can also be observed in the trace in Figure 1(a), where input I occurs in the third tick.

For another example, consider ExPar shown in Figure 2(a), which loops over two parallel threads. The program emits the signals R and S in the first instant, and since the loop instantaneously restarts its body, it will from the second instant on continuously emit all three signals R, S, and T.

3.2 The Kiel Esterel Processor

The instruction set of the KEP is very similar to the Esterel language. The Esterel language distinguishes *kernel statements* (*e. g.*, emit, pause) and *derived statements* (*e. g.*, await, every) [3]. Derived statements are in general just syntactic sugar and



(a) Esterel (b) CKAG (c) KEP assembler (d) Sample trace

Fig. 2. A concurrent example program.

can be reduced to kernel statements. The KEP Instruction Set Architecture (ISA) includes all kernel statements, and in addition some frequently used derived statements. The KEP ISA also includes valued signals, which cannot be reduced to kernel statements. The only parts of Esterel v5 that are not part of the KEP ISA are combined signal handling and external task handling, as they both seem to be used only rarely in practice; however, adding these capabilities to the KEP ISA seems relatively straightforward.

Due to this direct mapping from Esterel to the KEP ISA, most Esterel statements can be executed in just one instruction cycle. For more complicated statements, well-known translations into kernel statements exist, allowing the KEP to execute arbitrary Esterel programs. Part of the KEP instruction set is shown in Figure 3. The KEP assembler programs corresponding to ExSeq and ExPar and sample traces are shown in Figures 1(c)/(d) and 2(c)/(d), respectively. Note that PAUSE is executed for at least two consecutive ticks, and consumes an instruction cycle at each tick.

The KEP provides a configurable number of Watcher units, which detect whether a signal triggering a preemption is present and whether the program counter (PC) is in the corresponding preemption body [15]. Therefore, no additional instruction cycles are needed to test for preemption. Only upon entering a preemption scope two cycles are needed to initialize the Watcher, as for example the $\mathsf{WABORT}_{\mathsf{L1}}$ instruction in ExSeq.

To implement concurrency, the KEP employs a multi-threaded architecture, where each thread has an independent program counter (PC) and threads are scheduled according to their statuses and dynamically changing priorities. To begin of each instruction-cycle, the enabled thread with the highest priority is selected and executed. The scheduler is very light-weight. In the KEP, scheduling and context switching do not cost extra instruction cycles, only changing the priority of a thread

Mnemonic, Operands	Esterel Syntax	Cycles	Notes
PAR *prio*$_1$, *startAddr*$_1$, *id*$_1$ ... PAR *prio*$_n$, startAddr$_n$, *id*$_n$ PARE *endAddr* *startAddr*$_1$: ... *startAddr*$_2$: ... *startAddr*$_n$: ... *endAddr*: JOIN	[p$_1$ || ⋮ || p$_n$]	$\left.\begin{array}{l}\\\\\\\\\\\\\\\\\end{array}\right\} n+1$ 1	For each thread, one PAR is needed to define the start address, thread id and initial priority. The end of a thread is defined by the start address of the next thread, except for the last thread, whose end is defined via PARE. The cycle count of a *fork node* depends on the count of threads.
PRIO *prio*		1	Set current thread priority to *prio*.
[W]ABORT[I, *n*] *S*, *endAddr* ... *endAddr*:	[**weak**] **abort** ... **when** [**immediate**, *n*] *S*	2	
SUSPEND[I,*n*] *S*, *endAddr* ... *endAddr*:	**suspend** ... **when** [**immediate**, *n*] *S*	2	
startAddr: ... EXIT *exitAddr startAddr* ... *exitAddr:*	**trap** *T* **in** ... **exit** *T* ... **end trap**	1	Exit from a trap, *startAddr*/*exitAddr* specifies trap scope. Unlike GOTO, check for concurrent EXITs and terminate enclosing ||.
PAUSE	**pause**	1	Wait for a signal. AWAIT TICK is equivalent to PAUSE.
AWAIT [I, *n*] *S*	**await** [**immediate**, *n*] *S*	1	
SIGNAL *S*	**signal** *S* **in** ... **end**	1	Initialize a local signal *S*.
EMIT *S* [, {#*data*|*reg*}]	**emit** *S* [(*val*)]	1	Emit (valued) signal *S*.
SUSTAIN *S* [, {#*data*|*reg*}]	**sustain** *S* [(*val*)]	1	Sustain (valued) signal *S*.
PRESENT *S*, *elseAddr*	**present** *S* **then** ... **end**	1	Jump to *elseAddr* if *S* is absent.
HALT	**halt**	1	Halt the program.
addr: ... GOTO *addr*	**loop** ... **end loop**	1	Jump to *addr*.

Fig. 3. Overview of the KEP instruction set architecture, and their relation to Esterel and the number of processor cycles for the execution of each instruction.

costs an instruction. For each thread, a PAR instruction is executed, to initialize the program counter and the priority and to define the thread id. Thereafter one PARE instruction is executed, which denotes the end of the parallel scope. During each instant in which one parallel thread is active, also the JOIN must be executed, in order to determine whether the threads have terminated.

3.3 The Concurrent Kep Assembler Graph (CKAG)

The WCRT analysis is not directly performed on the Esterel level, but on an intermediate data structure, the CKAG. The CKAG is a directed graph composed of various types of nodes and edges to match KEP program behavior. It is used during compilation from Esterel to KEP assembler, for, *e. g.*, dead code elimination, priority assigning [13], optimizations and the WCRT analysis.

The CKAG distinguishes *transient nodes*, which represent instantaneous execution, *delay nodes*, which represent statements that may hold for more than one

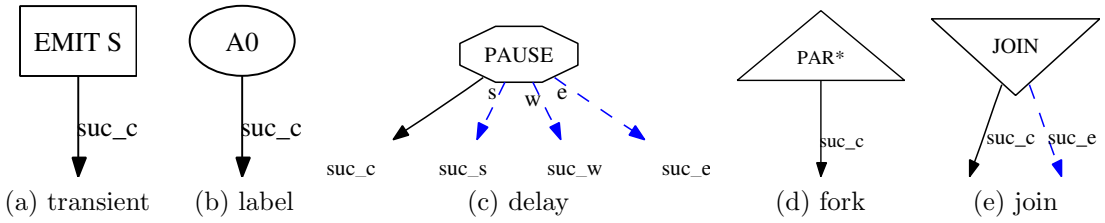


Fig. 4. Nodes and edges of a Concurrent KEP Assembler Graph (CKAG).

tick, and *fork* and *join nodes*, which represent concurrency (see Figure 4). Given a CKAG node n, the set $n.suc\_c$ denotes the set of sequential control flow successors (represented in the CKAG as solid edges). Successors reached via preemptions are $n.suc\_s$ for strong aborts, $n.suc\_w$ for weak aborts, and $n.suc\_e$ for exceptions (exit), represented as dashed edges; they are marked with small tail labels s, w and e, respectively. The CKAGs corresponding to ExSeq and ExPar can be found in Figures 1(b) and 2(b), respectively.

The CKAG is built from Esterel source by traversing recursively over its *Abstract Syntax Tree* (AST) generated by the *Columbia Esterel Compiler* (CEC) [8]. Visiting an Esterel statement results in creating the according CKAG node. A node typically contains exactly one statement, except *label nodes* containing just address labels and *fork nodes* containing one PAR statement for each child thread initialization and a PARE statement. When a *delay node* is created, additional preemption edges are added according to the abortion/exception context.

To preserve the signal-dependencies in the execution, additional priority assignments (PRIO statements) might be introduced by the compiler. To assure schedulability, the program is completely dismantled, *i. e.*, transformed into kernel statements. In this dismantled graph the priority assignments are inserted. A subsequent "undismantling" step before the computation of the WCRT detects specific patterns in the CKAG and collapses them to more complex instructions, such as AWAIT or SUSTAIN, which are also part of the KEP instruction set.

4 Worst Case Reaction Time (WCRT)

Given a KEP program we define its WCRT as the maximum number of KEP cycles executable in one instant. Thus WCRT analysis requires finding the longest instantaneous path in the CKAG, where the length metric is the number of required KEP instruction cycles. We abstract from signal relationships and might therefore consider unfeasible executions. Therefore the computed WCRT can be pessimistic. We first present, in Section 4.1, a restricted form of the WCRT algorithm that does not handle concurrency yet. The general algorithm requires an analysis of instant reachability between *fork* and *join* nodes, which is discussed in Section 4.2, followed by the presentation of the general WCRT algorithm in Section 4.3.

4.1 Sequential WCRT Algorithm

First we present a WCRT analysis of sequential CKAGs (no *fork* and *join nodes*). Consider the ExSeq example in Figure 1(a) again. The longest possible execution

```
1  int getWcrtSeq(g)      // Compute WCRT for sequential CKAG g
2    forall n ∈ Nodes do n.inst := n.next := ⊥ end
3    getInstSeq(g.root)
4    forall d ∈ DelayNodes do getNextSeq(d) end
5    return max ({g.root.inst} ⋃ {d.next : d ∈ DelayNodes})
6  end
```

```
1  int getInstSeq(n)      // Compute statements instantaneously reachable from node n
2    if n.inst = ⊥ then
3      if n ∈ TransientNodes ∪ LabelNodes then
4        n.inst := max {getInstSeq(c) : c ∈ n.suc_c} + cycles(n.stmt)
5      elif n ∈ DelayNodes then
6        n.inst := max {getInstSeq(c) : c ∈ n.suc_w ∪ n.suc_e} + cycles(n.stmt)
7      fi
8    fi
9    return n.inst
10 end
```

```
1  int getNextSeq(d)      // Compute statements instantaneously reachable from delay node d at tick start
2    if d.next = ⊥ then
3      d.next := max {getInstSeq(c) : c ∈ d.suc_c ∪ d.suc_s} + cycles(d.stmt)
4    fi
5    return d.next
6  end
```

Fig. 5. WCRT algorithm, restricted to sequential programs. The nodes of a CKAG g are given by $Nodes = TransientNodes \cup LabelNodes \cup DelayNodes \cup ForkNodes \cup JoinNodes$, $g.root$ indicates the first KEP statement. cycles($stmt$) returns the number of instruction cycles to execute $stmt$, see third column in Figure 3.

occurs when the signal I becomes present, as is the case in Tick 3 of the example trace shown in Figure 1(d). Since the abortion triggered by I is weak, the abort body is still executed in this instant, which takes four instructions: $\mathsf{PAUSE}_{\mathsf{L2}}$, $\mathsf{EMIT}_{\mathsf{L3}}$, the $\mathsf{GOTO}_{\mathsf{L4}}$, and $\mathsf{PAUSE}_{\mathsf{L2}}$ again. Then it is detected that the body has finished its execution for this instant, the abortion takes place, and $\mathsf{EMIT}_{\mathsf{L5}}$ and $\mathsf{HALT}_{\mathsf{L6}}$ are executed. Hence the longest possible path takes six instruction cycles.

The sequential WCRT is computed via a *Depth First Search* (DFS) traversal of the CKAG, see the algorithm in Figure 5. For each node n a value $n.inst$ is computed, which gives the WCRT from this node on in the same instant when execution reaches the node. For a transient node, the WCRT is simply the maximum over all children plus its own execution time.

For non-instantaneous *delay nodes* we distinguish two cases within a tick: control can *reach* a *delay node* d, meaning that the thread executing d has already executed some other instructions in that tick, or control can *start* in d, meaning that d must have been reached in some preceding tick. In the first case, the WCRT from d on within an instant is expressed by the $d.inst$ variable already introduced. For the second case, an additional value $d.next$ stores the WCRT from d on within an instant; "next" here expresses that in the CKAG traversal done to analyze the overall WCRT, the $d.next$ value should not be included in the current tick, but in a next tick. Having these two values ensures that the algorithm terminates in the case of non-instantaneous loops: to compute $d.next$ we might need the value $d.inst$.

For a *delay node*, we also have to take abortions into account. The handlers

(*i. e.*, their continuations—typically the end of an associated abort/trap scope) of weak abortions and exceptions are instantaneously reachable, so their WCRTs are added to the *d.inst* value. In contrast, the handlers of strong abortions cannot be executed in the same instant the delay node is reached, because according to the Esterel semantics an abortion body is not executed at all when the abortion takes place. On the KEP, when a strong abort takes place, the delay nodes where the control of the (still active) threads in the abortion body resides are executed once, and then control moves to the abortion handler. In other words, control cannot move from a delay node d to a (strong) abortion handler when control reaches d, but only when it starts in d. Therefore, the WCRT of the handler of a strong abortion is added to *d.next*, and not to *d.inst*.

We do not need to take a weak abortion into account for *d.next*, because it cannot contribute to a longest path. An abortion in an instant when a delay node is reached will always lead to a higher WCRT than an execution in a subsequent instant where a thread starts executing in the delay node.

The resulting WCRT for the whole program is computed as the maximum over all WCRTs of nodes where the execution may start. These are the *start node* and all *delay nodes*. To take into account that execution might start simultaneously in different concurrent threads, we also have to consider the *next* value of *join nodes*.

Consider the example ExSeq in Figure 1. Each node n in the CKAG g is annotated with a label "W$\langle n.inst\rangle$" or, for a delay node, a label "W$\langle n.inst\rangle/\langle n.next\rangle$." In the following, we will refer to specific CKAG nodes with their corresponding KEP assembler line numbers L$\langle n\rangle$. It is $g.root$ = L1. The sequential WCRT computation starts initializing the *inst* and *next* values of all nodes to $\perp$ (line 2 in getWcrtSeq, Figure 5). Then getInstSeq(L1) is called, which computes L1.*inst* := max { getInstSeq(L2) } + cycles($\text{WABORT}_{\text{L1}}$). The call to getInstSeq(L2) computes and returns L2.*inst* := cycles(PAUSE_{L2}) + cycles(EMIT_{L5}) + cycles(HALT_{L6}) = 3, hence L1.*inst* := 3 + 2 = 5. Next, in line 4 of getWcrtSeq, we call getNextSeq(L2), which computes L2.*next* := getInstSeq(L3) + cycles(PAUSE_{L2}). The call to getInstSeq(L3) computes and returns L3.*inst* := cycles(EMIT_{L3}) + cycles(GOTO_{L4}) + L2.*inst* = 1 + 1 + 3 = 5. Hence L2.*next* := 5 + 1 = 6, which corresponds to the longest path triggered by the presence of signal I, as we have seen earlier. The WCRT analysis therefore inserts an "EMIT _TICKLEN, #6" instruction before the body of the KEP assembler program to initialize the TickManager accordingly.

4.2 Instantaneous Statement Reachability for Concurrent Esterel Programs

It is important for the WCRT analysis whether a *join* and its corresponding *fork* can be executed within the same instant. The algorithm for instantaneous statement reachability computes for a *source* and a *target* node whether the *target* is reachable instantaneously from the *source*. Source and target have to be in sequence to each other, *i. e.*, not concurrent, to get correct results.

In simple cases like EMIT or PAUSE the sequential control flow successor is executed in the same instant respectively next instant, but in general the behavior is more complicated. The parallel, *e. g.*, will terminate instantaneously if all subthreads are instantaneous or an EXIT will be reached instantaneously; it is not-instantaneous if at least one sub-thread is not instantaneous.

The complete algorithm is presented in detail elsewhere [4]. The basic idea is to compute for each node three potential reachability properties: *instantaneous*, *not-instantaneous*, *exit-instantaneous*. Note that a node might be as well (potentially) *instantaneous* as (potentially) *non-instantaneous*, depending on the signal status. Computation begins by setting the *instantaneous* predicate of the *source* node to *true* and the properties of all other nodes to *false*. When any property is changed, the new value is propagated to its successors. If we have set one of the properties to *true*, we will not set it to *false* again. Hence the algorithm is monotonic and will terminate. Its complexity is determined by the amount of property changes which are bounded to three (three boolean) for all nodes, so the complexity is $O(3 * |Nodes|) = O(|Nodes|)$.

The most complicated computation is the property *instantaneous* of a *join node* because several attributes have to be fulfilled for it to be *instantaneous*:

- For each thread, there has to be a (potentially) instantaneous path to the *join node*.
- The predecessor of the *join node* must not be an EXIT, because EXIT nodes are no real control flow predecessors. At the Esterel level, an exception (exit) causes control to jump directly to the corresponding exception handler (at the end of the corresponding trap scope); this jump may also cross thread boundaries, in which case the threads that are jumped out of and their sibling threads threads terminate. To emulate this at the KEP level, an EXIT instruction does not jump directly to the exception handler, but first executes the JOIN instructions on the way, to give them the opportunity to terminate threads correctly. If a JOIN is executed this way, the statements that are instantaneously reachable from it are not executed, but control instead moves on to the exception handler, or to another intermediate JOIN. To express this, we use the third property besides *instantaneous* and *non-instantaneous*: *exit-instantaneous*.

Roughly speaking the *instantaneous* property is propagated via for-all quantifier, *not_instantaneous* and *exit_instantaneous* via existence-quantifier.

Most other nodes simply propagate their own properties to their successors. The *delay node* propagates in addition its *non-instantaneous* predicate to its delayed successors and *exit nodes* propagate *exit-instantaneous* reachability, when they themselves are reachable instantaneously .

4.3 General WCRT Algorithm

The general algorithm, which can also handle concurrency, is shown in Figure 6. It emerges from the sequential algorithm that has been described in Section 4.1 by enhancing it with the ability to compute the WCRT of *fork* and *join nodes*. Note that the instantaneous WCRT of a *join node* is needed only by a *fork node*, all *transient nodes* and *delay nodes* do not use this value for their WCRT. The WCRT of the *join node* has to be accounted for just once in the instantaneous WCRT of its corresponding *fork node*, which allows the use of a DFS-like algorithm.

The instantaneous WCRT of a *fork node* is simply the sum of the instantaneously reachable statements of its sub-threads, plus the PAR statement for each sub-thread and the additional PARE statement.

```
1  int getWcrt(g)    // Compute WCRT for a CKAG g
2    forall n ∈ Nodes do n.inst := n.next := ⊥ end
3    forall d ∈ DelayNodes do getNext(d) end
4    forall j ∈ JoinNodes do getNext(j) end // Visit according to hierarchy (inside out)
5    return max ({getInst(g.root)} ⋃ {n.next : n ∈ DelayNodes ∪ JoinNodes})
6  end
```

```
1  int getInst(n)      // Compute statements instantaneously reachable from node n
2    if n.inst = ⊥ then
3      if n ∈ TransientNodes ∪ LabelNodes then
4        t.inst := max {getInst(c) : c ∈ suc_c \ JoinNodes} + cycles(n.stmt)
5      elif n ∈ DelayNodes then
6        n.inst := max {getInst(c) : c ∈ suc_w ∪ suc_e \ JoinNodes} + cycles(n.stmt)
7      elif n ∈ ForkNodes then
8        n.inst := ∑_{t∈n.suc_c} t.inst + cycles(n.par_stmts) + cycles(PARE)
9        prop := reachability(n, n.join) // Compute instantaneous reachability of join from fork
10       if prop.instantaneous or prop.exit_instantaneous then
11         n.inst += getInst(n.join)
12       elif prop.not_instantaneous then
13         n.inst += cycles(JOIN) // JOIN is always executed
14       fi
15     elif n ∈ JoinNodes then
16       n.inst := max{getInst(c) : c ∈ suc_c ∪ suc_e} + cycles(n.stmt);
17     fi
18   fi
19   return n.inst
20 end
```

```
1  int getNext(n)      // Compute statements instantaneously reachable from delay node d at tick start
2    if n.next = ⊥ then
3      if n ∈ DelayNodes then
4        n.next := max {getInst(c) : c ∈ suc_c ∪ suc_s \ JoinNodes} + cycles(n.stmt)
5      elif n ∈ JoinNodes and (n.fork, n).not_instantaneous then
6        n.next := n.inst + ∑_{t∈n.fork.suc_c} max{n.next : t.id = n.id}
7      fi
8    fi
9    return n.next
10 end
```

Fig. 6. General WCRT algorithm.

Like *delay nodes*, *join nodes* also have a *next* value. When a *fork/join* pair (f, j) could be *non-instantaneous* then we have to compute a WCRT $j.next$ for the next instants analogously to the *delay nodes*. Its computation requires first the computation of all sub-thread *next* WCRTs. Then we simply sum the maximum value for every thread. If the parallel does not terminate instantaneously, all directly reachable states are reachable in the next instant. Therefore we have to add the execution time for all statements that are instantaneously reachable from the *join node*. Note that the computation is independent from the scheduling.

The whole algorithm computes first the *next* WCRT for all *delay* and *join nodes*; it computes recursively all needed *inst* values. Thereafter the instantaneous WCRT for all remaining nodes is computed. The result is simply the maximum over all computed values.

Consider the example in Figure 2. First we note that the *fork/join* pair is always *non-instantaneous*, due to the cycles($\mathsf{PAUSE}_{\mathsf{L6}}$) statement. We compute $\mathsf{L6}.next =$ cycles($\mathsf{PAUSE}_{\mathsf{L6}}$) + cycles($\mathsf{EMIT}_{\mathsf{L7}}$) = 2. From the *fork node* L3, the PAR and PARE

statements, the instantaneous parts of both threads and the JOIN are executed, hence $\mathsf{L3}.inst = 2 \times \mathsf{cycles(PAR)} + \mathsf{cycles(PARE)} + \mathsf{cycles(JOIN)} + \mathsf{L4}.inst + \mathsf{L5}.inst = 2+1+1+1+2 = 7$. Therefore, the WCRT of the program is $\mathsf{L8}.next = \mathsf{L6}.next + \mathsf{L8}.inst = 2+9 = 11$. Note that the JOIN statement is executed twice.

A known difficulty when compiling Esterel-programs is that due to nesting of exceptions and concurrency, statements might be executed multiple times in one instant. This problem, also known as *reincarnation*, is handled correctly by our algorithm. Since we compute nested joins from inside to outside, the same statement may effect both the instantaneous and non-instantaneous WCRT, which are added up in the next join. This exactly matches the possible control-flow in case of reincarnation. Even when a statement is executed multiple times in an instant, we compute a correct upper bound for the WCRT.

Regarding the complexity of the algorithm, let $n := |Nodes|$, $d := |DelayNodes|$, $f := |ForkNodes|$ and $j := |JoinNodes|$. For each node its WCRT's *inst* and *next* are computed at most once, and for all *fork nodes* a fork-join reachability analysis is additionally made, which has itself $O(n)$. So we get altogether a complexity of $O(n+d+j) + O(f*n) = O(2*n) + O(n^2) = O(n^2)$.

5 Experimental Results

The WCRT analysis is implemented in the KEP compiler. It automatically inserts a correct EMIT _TICKLEN instruction at the beginning of the program. To validate our approach, we used Esterel-Studio to generate test cases for Esterel programs, which cover all states and transitions. The programs were executed on the KEP with the test cases as input. We measured the maximal reaction time during these executions and compared it to the computed value. The Esterel programs in Table 1 are taken from the *Estbench* [6]. We never underestimated the WCRT, and our results are on average 38% too high. For each program, the lines of code, the computed WCRT and the measured WCRT with the resulting difference is given. We also give the average WCRT analysis time on a standard PC (AMD Athlon XP, 2.2GHz, 512 KB Cache, 1GB Main Memory); as the table indicates, the analysis takes only a couple of milliseconds.

The table also compares the Average Case Reaction Time (ACRT) with the WCRT. The ACRT is on average about two thirds of the WCRT, which is relatively high compared to traditional architectures. In other words, the worst case on the KEP is not much worse than the average case, and padding the tick length according to the WCRT does not waste too much resources. On the same token, designing for worst-case performance, as typically must be done for hard real-time systems, does not cause too much overhead compared to the typical average-case performance design. Finally, the table also lists the number of scenarios generated by Esterel-Studio and accumulated logical tick count for the test traces.

6 Conclusions and Further Work

We have presented the WCRT analysis of reactive programs written in the Esterel language. The analysis is performed on a graph representation, the *Concurrent KEP Assembler Graph* (CKAG). In a first step we compute whether concurrent threads

Module name	LoC	WCRT			t_{an}	ACRT		Test	Ticks
		WC_{est}	WC_{act}	$r_{est-act}$	[ms]	AC_{act}	AC/WC	cases	
abcd	152	56	44	27%	1.0	28	64%	161	673
abcdef	232	84	68	24%	1.5	42	62%	1457	50938
eight_buttons	312	112	92	22%	2.0	57	62%	13121	45876
channel_protocol	57	49	38	29%	0.4	19	50%	114	556
reactor_control	24	24	15	60%	0.2	12	80%	6	20
runner	26	10	7	43%	0.3	4	57%	131	2548
tcint	410	192	138	39%	2.8	86	62%	148	1325

Table 1
Experimental results. The WC_{est} and WC_{act} data denote the estimated and actual WCRT, respectively, measured in instruction cycles. The ratio $r_{est-act} := WC_{est}/WC_{act} - 1$ indicates by how much our analysis overestimates the WCRT. AC_{act} is the actual Average Case Reaction Time (ACRT), AC/WC ($= AC_{act}/WC_{act}$) gives the ratio to the WCRT. Test cases and Ticks are the number of different scenarios and logical ticks that were executed, respectively.

terminate instantaneously, thereafter we are able to compute for each statement how many instruction are maximally executable from it in one logical tick. The maximal value over all nodes gives us the WCRT of the program. The analysis considers concurrency and the multiple forms of preemption that Esterel offers. The asymptotic complexity of the WCRT analysis algorithm is quadratic in the size of the program; however, experimental results indicate that the overhead of WCRT analysis as part of compilation is negligible. We have implemented this analysis as part of a compiler from Esterel to KEP assembler, and use it to automatically compute an initialization value for the KEP's TickManager. This allows to achieve a high, constant response frequency to the environment, and can also be used to detect hardware errors by detecting timing overruns.

Our analysis is safe, *i. e.*, conservative in that it never underestimates the WCRT, and it does not require any user annotations to the program. In our benchmarks it overestimates the WCRT on average by about 40%. This is already competitive with the state of the art in general WCET analysis, and we expect this to be acceptable in most cases. However, there is still significant room for improvement. So far, we are not taking any signal status into account, therefore our analysis includes some unreachable paths. Considering all signals would lead to an exponential growth of the complexity, but some local knowledge should be enough to rule out most unreachable paths of this kind. Also a finer grained analysis of which parts of parallel threads can be executed in the same instant could lead to better results. However, it is not obvious how to do this efficiently.

Our analysis is influenced by the KEP in two ways: the exact number of instructions for each statement and the way parallelism is handled. At least for non-parallel programs our approach should be of value for other compilation methods for Esterel as well, *e. g.*, simulation-based code generation. A virtual machine with similar support for concurrency could also benefit from our approach. We would also like to generalize our approach to handle different ways to implement

concurrency. A WCRT analysis directly on the Esterel level gives information on the longest possible execution path. Together with a known translation to C, this WCRT information could be combined with a traditional WCET analysis, which takes caches and other hardware details into account.

References

[1] Berry, G. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London 339* (1992), 87–104.

[2] Berry, G. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999.

[3] Berry, G. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000.

[4] Boldt, M. Worst-case reaction time analysis for the KEP3. Study thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Nov. 2007. To appear.

[5] Burns, A., and Edgar, S. Predicting computation time for advanced processor architectures. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (EUROMICRO-RTS 2000)* (2000).

[6] Estbench Esterel Benchmark Suite. http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz.

[7] Closse, E., Poize, M., Pulou, J., Venier, P., and Weil, D. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In *Electronic Notes in Theoretical Computer Science* (July 2002), F. Maraninchi, A. Girault, and E. Rutten, Eds., vol. 65, Elsevier.

[8] Edwards, S. A. CEC: The Columbia Esterel Compiler. http://www1.cs.columbia.edu/~sedwards/cec/.

[9] Edwards, S. A. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21*, 2 (Feb. 2002).

[10] Edwards, S. A. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems 8*, 2 (Apr. 2003), 141–187.

[11] Edwards, S. A., Kapadia, V., and Halas, M. Compiling Esterel into static discrete-event code. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'04)* (Barcelona, Spain, Mar. 2004).

[12] Harel, D., and Pnueli, A. On the development of reactive systems. *Logics and models of concurrent systems* (1985), 477–498.

[13] Li, X., Boldt, M., and von Hanxleden, R. Mapping Esterel onto a multithreaded embedded processor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, USA, October 21–25 2006).

[14] Li, X., Lukoschus, J., Boldt, M., Harder, M., and von Hanxleden, R. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (New York, NY, USA, Sept. 2005), ACM Press, pp. 225–236.

[15] Li, X., and von Hanxleden, R. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques* (Dijon, France, April 23–27 2006).

[16] Logothetis, G., and Schneider, K. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)* (Munich, Germany, March 2003), IEEE Computer Society, pp. 196–203.

[17] Logothetis, G., Schneider, K., and Metzler, C. Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems. In *Forum on Design Languages (FDL)* (Frankfurt, Germany, 2003), Kluwer.

[18] Malik, S., Martonosi, M., and Li, Y.-T. S. Static timing analysis of embedded software. In *DAC '97: Proceedings of the 34th annual conference on Design automation* (1997), ACM Press, pp. 147–152.

[19] Potop-Butucaru, D., and de Simone, R. *Optimization for faster execution of Esterel programs*. Kluwer Academic Publishers, Norwell, MA, USA, 2004, pp. 285–315.

[20] Puschner, P., and Burns, A. A review of worst-case execution-time analysis (editorial). *Real-Time Systems 18*, 2/3 (2000), 115–128.

[21] Ringler, T. Static worst-case execution time analysis of synchronous programs. In *ADA-Europe- 5. International Conference on Reliable Software Technologies* (2000).

[22] Roop, P. S., Salcic, Z., and Dayaratne, M. W. S. Towards Direct Execution of Esterel Programs on Reactive Processors. In *4th ACM International Conference on Embedded Software (EMSOFT 04)* (Pisa, Italy, Sept. 2004).

[23] Tardieu, O., and de Simone, R. Instantaneous termination in pure Esterel. In *Static Analysis Symposium* (San Diego, California, June 2003).

3. Executable Specifications for Real-Time Distributed Systems

Arnab Ray[1], and Rance Cleaveland[2]

1 Fraunhofer USA Center for Experimental Softare Engineering,
University of Maryland at College Park, USA
2 Department of Computers Science,
University of Maryland at College Park, USA

Notes:

Executable Specifications for Real-Time Distributed Systems

Arnab Ray [1]

Fraunhofer USA Center for Experimental Software Engineering
University of Maryland at College Park
4321 Hartwick Road Suite 500
College Park MD 20742-3290

Rance Cleaveland [2]

Department of Computer Science
University of Maryland at College Park
AV Williams Building
College Park MD 20742

Abstract

One of the challenges in designing distributed, embedded systems is the paucity of formal, executable specification notations that provide support for both real-time and asynchronous communication. This paper describes a timed architecture design language (Timed Architecture Interaction Diagrams or TAID) that, by virtue of its formal, executable semantics, combines the benefits of synchronous specification notations with the advantages of traditional architecture description languages. In addition, TAID provides support for a variety of temporal inter-process communication (IPC) primitives as a native feature of the language, so that the encapsulated communication behavior (captured by real-time "buses" in TAID) may be re-used across designs and serve as specifications for more detailed model implementations.

Keywords: Software Architecture, Real-time, Simulations, Formal Methods, Distributed Systems.

1 Introduction

Software architectures have emerged as important artifacts of the software design process, as they support better system comprehension [12], pre-implementation analysis [16], identification of units of reuse [17] and product-line engineering [2], among other development activities. The research community has developed several formalisms for expressing architectural designs: WRIGHT [13], TRACTA [4], AADL [6], Rapide [11], and AID [15] to name a few. These notations seek to provide a (semi-)formal modeling framework wrapped inside an intuitive and expressive design language. Of these AID (Architectural Interaction Diagrams) distinguishes

[1] Email: arnabray@fc-md.umd.edu
[2] Email: rance@cs.umd.edu

itself by virtue of its ability to define different inter-process communications (IPC)s as native features of the language, thus facilitating concise and re-usable system specifications.

Most software architecture notations describe system behavior as a causal sequence of events (e.g *A causes B*). This makes them insufficiently expressive when it comes to describing real-time embedded systems, where precise temporal constraints between actions (e.g. *A causes B within 5 seconds of A*) need to be encoded. Notations like AADL do include real time, but do not have a formal semantics. In contrast, foundational synchronous notations such as timed process algebras [5] provide a rich formal framework for describing real-time systems, but these theories are often considered to be too abstract/high-level to be used for realistic system specifications.

This paper demonstrates how, by using ideas from timed process algebras, we may add a notion of discrete time to a non-timed architecture specification language (AID), resulting in Timed Architectural Interaction Diagrams (TAID). TAID is an executable notation that combines the theoretical rigor of timed process algebra with the user-friendliness of an architecture description language. In TAID, communication is defined through semantic devices called *buses*, which may be re-used across designs and serve as an abstract specification for a more detailed model implementation in notations like Simulink®/Stateflow®.

The utility of TAID specifications is that they provide a unified formalism for representing the entire system (both components as well as connectors), thus allowing embedded software engineers the flexibility of performing system simulations on a desktop computer. This leads to large savings of time and money that traditional embedded-system design processes incur, with their emphasis on prototyping, networking testbeds and hardware-in-the-loop testing as virtually the only strategy for design verification and validation.

The paper is arranged as follows. Section 2 outlines the basic concepts of the non-timed architecture design language, Architectural Interaction Diagrams (AID) which this paper extends to create a timed design language: Timed Architectural Interaction Diagrams (TAID) (Section 3). We then illustrate out formalism with two examples of timed inter-process communication. Section 5 details related work, and Section 6 concludes the paper.

2 Background: Architectural Interaction Diagrams

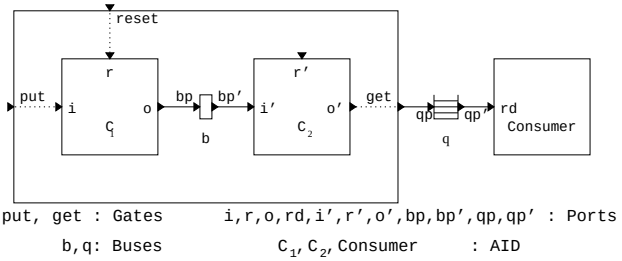


Fig. 1. A sample architecture

This section introduces some of the different elements that constitute a AID

architecture. From Figure 1, one may identify the following concepts: 1) AIDs describing subsystems/components; 2) interfaces, containing read and write ports (each port is a conduit point for data for the subsystem surrounding the interface); 3) connections between ports in a subsystem and ports in an interface (cf. the dotted lines in Figure 1) called gates, which enable one to selectively expose ports in an inner AID; 4) buses, the "connectors" through which subsystems communicate with each other; 5) links from interface ports to buses.

The execution semantics of AID are formalized in terms of a transition relation describing system-level execution steps. At the lowest level, an AID component is a state machine that comprises states and transitions. It can perform write transitions (i.e. output a value to a "write" port), read transitions (ie read in a value from a "read" port), or an internal transition.

In the remainder of the section, we provide formal definitions for some of the concepts on which TAID is based on. Let us first define formally what ports are.

- $\mathbb{W}$ is an infinite set of *write ports*.
- $\mathbb{R}$ is an infinite set of *read ports*, with $\mathbb{R} \cap \mathbb{W} = \emptyset$.
- $\mathbb{V}$ is a nonempty set of *values*.

Intuitively, $\mathbb{W}$ and $\mathbb{R}$ contain all the possible port names that may be used to define a given system, while $\mathbb{V}$ contains all the values that may be used. Our focus is on interaction rather than data manipulation, so we do not impose any additional structure on $\mathbb{V}$.

We also use the following definitions in what follows.

- $\mathbb{O} = \{w\,!\,v \mid w \in \mathbb{W}, v \in \mathbb{V}\}$ is the set of *output actions*.
- $\mathbb{I} = \{r? \mid r \in \mathbb{R}\}$ is the set of *input actions*.

The sets $\mathbb{O}$ and $\mathbb{I}$ represent interactions that a system may engage in with its environment: outputting, and inputting.

2.1 I/O Labeled Transition Systems

The basic components of the AID theory are *I/O labeled transition systems* (IOLTSs). These are defined as follows.

An *I/O labeled transition system (IOLTS)* is a tuple $\langle Q, T, q_0 \rangle$, where Q is a set of states, $q_0 \in Q$ is the start state, and $T = T_{write} \cup T_{read} \cup T_{internal}$ is the transition relation such that:

(i) $T_{write} \subseteq Q \times \mathbb{O} \times Q$,

(ii) $T_{read} \subseteq Q \times \mathbb{I} \times (\mathbb{V} \longrightarrow Q)$,

(iii) $T_{internal} \subseteq Q \times Q$.

An IOLTS encodes the operational behavior of a system, with Q being the set of system states and q_0 the initial state. State transitions may take one of three forms.

- An *output transition* $\langle q, w\,!\,v, q' \rangle \in T_{write}$ indicates a state change from q to q' when value v is written out to the environment on write port w.

- An *input transition* $\langle q, r?, f\rangle \in T_{read}$, f is a function mapping values to states. This transition indicates a state change from q to $f(v)$ if the system's environment supplies value v on read port r.
- An *internal transition* $\langle q, q'\rangle \in T_{internal}$ represents an execution step that the system can engage in without any interaction with its environment.

An IOLTS P also has an interface $I(P) \subseteq \mathbb{W} \cup \mathbb{R}$ containing the write and read ports used by transitions in P. We also write $Q(P)$ for the set of states for IOLTS P and $q_0(P)$ fot the start state of P.

2.2 Buses

Buses are the most critical elements of the AID paradigm. Mathematically, they can be seen as transducers that convert transitions of incident components (i.e. components that are connected to them) to system-level transitions. As such, they have two responsibilities: the transfer of data between senders and receivers, and the synchronization of sender/receiver transitions.

Formally, a bus in AID has form $\langle I, B, b_{init}, T\rangle$, where I is the interface, B is the set of of bus states, b_{init} is the initial bus state, and the transition relation T contains a single kind of transition – communication — which represents an instantaneous transfer of data among incident components. A communication transition of a bus M is written as:

$$b \xrightarrow[WV\ R]{W\ RV}_M b'.$$

The way to read this transition is as follows: "if the bus is in state b, and subsystems connected to the bus enable write transitions as indicated in WV and read transitions as enabled in R, then the bus fires read transitions as indicated in RV and write transitions as indicated in W and goes to state b'." This firing of selected read and write transitions in systems connected to the bus is also done atomically: thus one bus transition may consume several transitions from the components connected to it. Also, writing to a bus is interpreted with respect to components connected to a bus: so write ports on a subsystem are connected to write ports on a bus, and similarly for read ports.

The sets WV, R, W and RV deserve further comment. WV contains pairs of the form $\langle w, v\rangle$, where w is a write port on the bus and v is a value. Intuitively $\langle w, v\rangle \in WV$ if there is a writer connected to the bus on its port w that wishes to write value v to it. Similarly, $r \in R$ means that there is a reader connected to the bus on its port r that is interested in reading. The bus then, out of these enabled transitions, chooses writers whose port values are stored in W and readers as indicated by RV where $\langle r, v\rangle \in RV$ if reader connected to port r gets value v.

Like we had in the definition of IOLTS, $B(N)$ returns the set of bus states for a bus N and $b_0(N)$ the start state of N.

2.3 Architecture Interaction Diagrams

An *architecture interaction diagram (AID)* is either:

(i) an IOLTS P, with an interface $I(P)$; or

(ii) a network $N = \langle \bar{C}, \bar{M}, L \rangle$, where:
   (a) $\bar{C} = \langle C_1, \ldots, C_n \rangle$ is a tuple of components, where each $C_i = \langle S_i, I_i, G_i \rangle$ consists of an AID S_i, an interface I_i, and a *gate definition* G_i (the formal definition of gate may be found in [15]);
   (b) $\bar{M} = \langle M_1, \ldots M_k \rangle$ is a tuple of buses; and
   (c) L is the set of links. Each link connects a component port with a port on a bus. Interested readers may refer to the formal definition of links in [15]. It should be noted that for the purpose of the paper, the basic intuition behind links and gates is all that is necessary.

Intuitively, a network contains a list of components, each containing a subsystem description, an interface, and a gate definition that defines how the ports of the subsystem are connected to the interface. It also contains a list of buses and a link set connecting component ports to bus ports so that write ports are connected to write ports, and read ports to read ports, and each port (bus or component) has at most one link to it.

Mathematically, we define the semantics in the Structural Operational Semantic (SOS) style. The definition of the transition relation of an AID is given inductively using inference rules that explain how transitions of subsystems are combined to form transitions of systems.

More precisely, given an AID N the semantics associates with N an IOLTS $\langle Q_N, T_N, q_N \rangle$ describing the operational behavior of N. If $N = \langle Q, T, q_0 \rangle$ is itself an IOLTS, the association is obvious: take $Q_N = Q, T_N = T$ and $q_N = q_0$.

Now suppose that $N = \langle \bar{C}, \bar{M}, L \rangle$. What should Q_N, T_N and q_N be? In the case of Q_N, each system state should record current state information for each component and bus, and the initial state should contain the initial states of each component and bus. This leads to the following.

Let $N = \langle \langle C_1, \ldots, C_n \rangle, \langle M_1, \ldots, M_k \rangle, L \rangle$ be a network AID. Then:

(i) $Q_N = C_N \times M_N$, where:

$$C_N = \{\langle q_1, \ldots, q_n \rangle \mid q_i \in Q(S_i)\}$$
$$M_N = \{\langle b_1, \ldots, b_k \rangle \mid b_i \in B(M_i)\}$$

(ii) $q_N = C_N^0 \times M_N^0$, where $C_N^0 = \langle q_0(S_1), \ldots, q_0(S_n) \rangle$ and $M_N^0 = \langle b_0(M_1), \ldots, b_0(M_k) \rangle$.

Thus, the states for N's IOLTS consists of a state vector for N's components and another state vector for N's buses, with the start state for N containing the start states for each component and bus. We often represent these states as pairs $\langle \bar{s}, \bar{b} \rangle$, where $\bar{s}$ and $\bar{b}$ are the subsystem- and bus-state vectors, respectively.

T_N is defined by providing SOS rules that allow us to deduce the set of transitions of N from the transitions of its constituent components and buses as obtained from the structure of N. The SOS rules for AID is given in [15]

3 Timed Architectural Interaction Diagrams

In this section, we define the syntax and the semantics of TAID by augmenting the syntactic and semantic framework used for defining AID. Using concepts from

timed process algebra [5], TAID can be "layered" on top of the original language without modifying the semantics of the original theory. In other words, the additional semantics that is needed to encode time can be seamlessly superposed on the original theoretical framework. In this paper, we provide the semantics for only that *incremental* part that provides support to time. For a formal definition of the terms used and the semantics for the untimed part of TAID, the interested reader is invited to consult [15].

3.1 Timed Input-Output Labelled Systems

In TAID, we extend the original definition of IOLTSs to include time transitions (we call these T-IOLTS). A time transition may be thought of as a "clock tick" representing the passage of time.

In timed process algebra time transitions are typically required to satisfy two conditions.

Maximal progress. Time transitions are disabled when internal transitions are possible.

Time determinacy. At most one time transition is possible in any state.

The reason for these assumptions is to separate the modeling of the passage of time from the modeling of system interaction. Maximal progress guarantees that an action must occur as soon as all participants are ready to do it. In other words, enabled actions may not be delayed for even a single clock tick. Time determinacy ensures that the only ambiguity about the state a system can be in is due to the actions it performs, not just the passage of time. A fuller discussion of these issues may be found in [1].

Mathematically, the transition relation of a T-IOLTS P has a component $T_{tick} \subseteq Q \times Q$, where Q is the set of states of P. T_{tick} is also required to satisfy two conditions that are given below. In order to define these, we introduce the following notations. We write $T_{tick}(P)$ for the clock-transition relation of P and $q \xrightarrow{1}_P q'$ when $\langle q, q' \rangle \in T_{tick}(P)$. When P is evident from context, we write simply $q \xrightarrow{1} q'$. We also use $q \xrightarrow{1}$ if there is a q' such that $q \xrightarrow{1} q'$ and $q \not\xrightarrow{1}$ when this is not the case. The notation $q \not\xrightarrow{\tau}$ is used similarly, and when this holds of state q we refer to q as stable.

We can now formulate the properties that $T_{tick}(P)$ must satisfy for T-IOLTS P. Recall that $Q(P)$ is the set of states in P.

Maximal progress. $\forall q \in Q(P).\ q \xrightarrow{1}$ implies $q \not\xrightarrow{\tau}$

Time enabledness. Time is always enabled in stable states.
$\forall q \in Q(P).\ q \not\xrightarrow{\tau}$ implies $q \xrightarrow{1}$

Time determinacy. $\forall q \in Q(P).\ q \xrightarrow{1} q'$ and $q \xrightarrow{1} q''$ implies $q' = q''$

3.2 Timed Buses

Recall from the last section that buses were defined by a set of bus states (B), an interface (I), an initial bus state (b_{init}) and only one kind of transition—untimed

communication. In TAID, besides communication transitions, buses contain timed transitions of the form $T_{tick} \subseteq B \times B$, where B is the set of bus states. We write timed transitions as

$$b \xrightarrow{1}_M b'$$

and $b \xrightarrow{1}_M$ if there exists a b' such that $b \xrightarrow{1}_M b'$.

States in AID buses do not have τ-transitions, and thus maximal progress is not an issue for T_{tick} relations in buses. We do require time determinacy and time-enabledness, however; each bus state is required to have exactly one time transition. Assuming M is a timed bus and $B(M)$ its set of states, we write this condition as follows.

$$\forall b \in B(M).\ \exists! b' \in B(M).\ b \xrightarrow{1}_M b'$$

3.3 TAID

Given the notions of T-IOLTS and timed bus, we may now formally introduce Timed Architectural Interaction Diagrams (TAIDs). The definition closely follows that of AID given in Section 2.3. Specifically, a TAID is either:

(i) a T-IOLTS P with interface $I(P)$; or

(ii) a network $\langle \boldsymbol{C}, \boldsymbol{M}, L \rangle$, where $\boldsymbol{C}$ is a vector of components, $\boldsymbol{M}$ is a vector of timed buses, and L is a link relation.

The definitions of component, interface, link, etc. do not change, except that components now include TAIDs instead of AIDs, in addition to inteface and gate specifications.

3.4 TAID Semantics

As with AID, the semantics of TAID with associate with each TAID N a T-IOLTS $\langle Q_N, T_N, q_N \rangle$ describing the operational behavior of N. The structure of a TAID state (i.e, Q_N) is defined recursively on the structure of N as done for AID in Section 2.3. For T_N, we use all the SOS rules for AID [15] and add to it another SOS rule (given below) that defines the timing behavior of N in terms of the timing behavior of its constituent components and buses. More precisely the rule will allow a network to do a clock tick only if all the components and all the buses that constitute the network can do a clock transition, and if there is no enabled communication between a component and a bus inside the network.

SOS rules have the following general form.

$$\frac{\text{Premises}}{\text{Conclusion}}$$

Intuitively, a rule states that when the premises are true, the conclusion holds. In our case, a conclusion will state the existence of an element of the transition relation, T_N, while the premises will refer to transitions of subsystems and buses as well as conditions about the structure of N.

Now let us define some auxiliary notations that will enable us to define the new

SOS rule.

- Recall that a bus M includes a timed transition relation $T_{tick} \subseteq B \times B$, where B is the set of states of M, with the property that if $b \in B$ then there is a unique b' such that $\langle b, b' \rangle \in T_{tick}$. We may therefore define function $CT(b, M)$ which, given bus state b, returns the b' such that $\langle b, b' \rangle \in T_{tick}$.
- Let us now define a function $NSAT$ ("NextStateAfterTick") that, given a TAID state and a TAID, outputs the *next* state of the TAID after a clock tick has taken place.

  Mathematically, $NSAT$ is a function $(Q_N \times N) \longrightarrow Q_N$ that is defined as follows.

  If N is a T-IOLTS of form $\langle Q, T, q \rangle$ then

$$NSAT(q, N) = \begin{cases} q' & \text{if } q \xrightarrow{1}_N q' \in T \\ \textit{undefined} \text{ otherwise} \end{cases}$$

  If N is a network of form $\langle \bar{C}, \bar{M}, L \rangle$, where $\boldsymbol{C} = \langle \langle S_1, I_1, G_1 \rangle, \ldots, \langle S_n, I_n, G_N \rangle \rangle$ and $\boldsymbol{M} = \langle M_1, \ldots, M_k \rangle$, and $\boldsymbol{s} = \langle s_1, \ldots, s_n \rangle$ and $\boldsymbol{b} = \langle b_1, \ldots, b_k \rangle$, then

$$NSAT(\langle \bar{s}, \bar{b} \rangle, N) = \begin{cases} \langle \langle NSAT(s_1, S_1), \ldots, NSAT(s_n, S_n) \rangle, & \\ \quad \langle CT(b_1, M_1), \ldots, CT(b_k, M_k) \rangle \rangle & \text{if for all } i,\ NSAT(s_i, S_i) \text{ is defined} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

- A state s in TAID N is said to be stable if it cannot perform any τ transitions, i.e. $Stable(s, N)$ iff $s \xrightarrow{\tau}_N$

The SOS rule may now be given as follows.

$$\frac{Stable(q, N)}{q \xrightarrow{1}_N NSAT(q, N)}$$

This SOS rule states that if a network cannot perform any internal transitions, only then can it take a clock tick. We may now prove the following.

Lemma 3.1 *Let N be a TAID and q be a stable state of N. Then $NSAT(q, N)$ is defined.*

The lemma is useful in proving the following theorems. It also guarantees that the T-IOLTS associated with a TAID N is time-enabled.

Theorem 3.2 *Let N be a TAID. Then the T-IOLTS associated with N satisfies time determinacy.*

Theorem 3.3 *Let N be a TAID. Then the T-IOLTS associated with N satisfies maximal progress.*

The truth of these statements may be inferred from the definition of $NSAT$, the

restrictions imposed on T-IOLTSs and timed buses, and the fact that the SOS rule is the only one that can be used to infer clock-tick transitions for TAIDs.

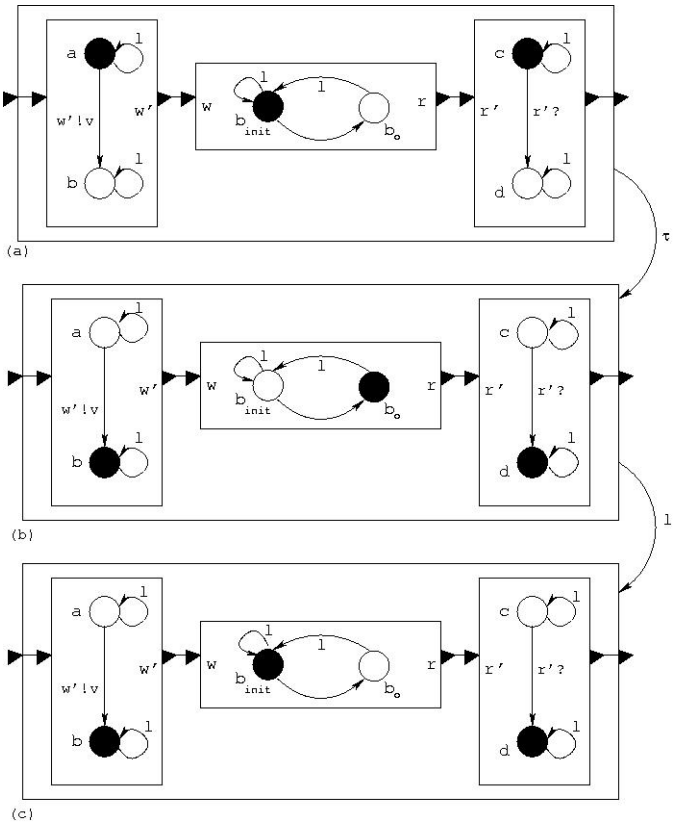


Fig. 2. A TAID execution example

Let us illustrate the concepts introduced in this section with a small example (Figure 2) of a very simple unit-delay handshake where the writer and reader block till communication is finished. Here we have a TAID network consisting of 2 T-IOLTSs and a bus that connects them. One of the T-IOLTS that has the port w' is executing a write transition labeled by $w'!v$ (to be read as: output value v to port w'); this w' port is connected to the w port of the bus via a link. The other T-IOLTS has a port r' (connected via a link to bus port r) where the T-IOLTS in question collects the value from the bus by virtue of the transition labeled by $r'?$. The states colored black represent the states where control presently resides. The bus initially has a transition from its initial state (b_{init}) to state b_0; in this transition (whose labels are omitted), the bus blocks a reader or writer until its partner is ready and then permits a synchronization of the read and write transitions. The bus then delays for one clock tick before cycling back to its initial state.

Initially in configuration (a) in Figure 2, control resides in states a and c of the writing and reading T-IOLTS and in state b_{init} for the bus. A bus communication transition takes place (denoted by the τ transition) wherein the control inside the bus passes to b_0; the reader and writer also change state, and the value v is transmitted to the reader. In (b), the system as a whole takes a clock tick (denoted by 1); the reader and writer remain in the given states, while control in the bus passes from b_0 back to b_{init}. Symbolically, this entire sequence of actions can be

Table 1
Synchronous handshake bus $\langle I, B_\delta, b_{init}, T\rangle$ with δ delay

$$I = \langle \mathbb{W}, \mathbb{R}\rangle$$
$$B_\delta = \{b_{init}, b_{err}\} \bigcup \{\langle i, w, r, v\rangle \mid i \in \{0..\delta\}, v \in \mathbb{V}, w \in \mathbb{W}, r \in \mathbb{R}\} \bigcup \{b_{i,w,r,v} \mid i \in \{0..\delta\}\}$$

T is defined as follows.

(i) $b_{init} \xrightarrow[WV\ R]{W\ RV} \langle 0, w, r, v\rangle$ iff $\langle w, v\rangle \in WV \wedge r \in R \wedge W = \emptyset \wedge RV = \emptyset$

(ii) $\langle i, w, r, v\rangle \xrightarrow[WV\ R]{W\ RV} b_{i,w,r,v}$ iff $i < \delta \wedge \langle w, v\rangle \in WV \wedge r \in R$

(iii) $b_{i,w,r,v} \xrightarrow{1} \langle i+1, w, r, v\rangle$

(iv) $\langle i, w, r, v\rangle \xrightarrow[WV\ R]{W\ RV} b_{err}$ iff $\langle w, v\rangle \notin WV \vee r \notin R$

(v) $\langle \delta, w, r, v\rangle \xrightarrow[WV\ R]{W\ RV} b_{init}$ iff $\langle w, v\rangle \in WV, r \in R \wedge W = \{w\} \wedge RV = \{\langle r, v\rangle\}$

(vi) $\langle \delta, w, r, v\rangle \xrightarrow[WV\ R]{W\ RV} b_{err}$ iff $\langle w, v\rangle \notin WV \vee r \notin R$

represented thus:

$$\langle a, b_{init}, c\rangle \xrightarrow{\tau} \langle b, b_0, d\rangle \xrightarrow{1} \langle b, b_{init}, d\rangle$$

4 Modeling Communication Primitives Using TAID

4.1 Timed Bi-party Handshake

The first example we consider is a bi-party handshake communication that takes δ time units to execute. In this kind of communication, there is no limit on how many subsystems are allowed to use the bus. The bus requires all senders and receivers to block until an exchange of data between one selected writer and selected reader occurs, after which the selected writer and reader are free to continue executing. This transfer of data consumes δ time.

In this semantics, the sequence of events are:

(i) The bus chooses a writer and a reader among the writers and readers who want to use the bus.

(ii) At each instant of time, the bus checks to see if either of the reader or the writer *timed out*, i.e. is no longer interested in the communication it initiated. If that is the case, then the bus transitions to an error state.

(iii) Once the delay is finished and the original writer and reader both are still enabled for the communication, the bus finishes the handshake and releases the writer and reader. Anything else causes the bus to transition to the error state.

Note that the semantics of timed handshake is different from the semantics of a FIFO queue with capacity 1 and propagation delay 1 because in a FIFO, a writer writes to the FIFO and is immediately unblocked while here, the writer remains blocked till a reader has read the value.

The responsibility of any bus that implements a timed handshake is twofold: 1) transfer data between a writer and a reader 2) enforce the blocking of the chosen writer and reader for the duration of the handshake. Since it is not possible for any communicating medium to physically block a writer and reader (since they may always execute a time-out transition), the bus should not complete any interaction where at some time during the δ delay, either the writer or the reader timed out. As a result, when a successful handshake is completed, we can then automatically deduce that the writer and the reader were indeed blocked for δ time units.

The bus definition is provided in in Table 1. The first line in the table indicates that the bus's interface has a set $\mathbb{W}$ of write and a set $\mathbb{R}$ of read ports. The second line in the table defines the bus states. There are two distinguished bus states: b_{init} (the initial state) b_{err} (the error state) in addition to 1) a set of general bus states, each of which is a tuple that contains a variable i i.e. the number of clock ticks already elapsed at that state, a variable w that stores the value of the selected write port, a variable v that stores the data value obtained from w and a variable r that contains the value of the read port selected 2) a set of "trap" states of the form $b_{i,w,r,v}$ which are the states at which the bus can perform a clock tick.

Rule 1 states that if there exists at least one writer and a reader, then a writer is chosen along with its value as also a reader and stored in the bus state. The counter representing time elapsed since initiation of communication is set to 0. Rule 2 says that if the chosen writer and reader are present in the set of enabled writers and readers respectively, and the upper limit of the counter (i.e. δ) has not been reached, then the bus transitions to a state where it is allowed to perform a tick (vide Rule 3). If however, the writer or reader has timed out, then the bus (vide Rule 4) transitions to an error state. If when the counter expires, the original writer and reader are still present in the interaction, the handshake is completed and the writer and reader are released (Rule 5). Else a transition to an error state is made (Rule 6).

4.2 Timed Buffered Communication

In a timed buffered communication channel, writers and readers communicate over a FIFO buffer whose capacity is assumed to be N data elements. There exists a pre-determined propagation delay (δ time units) between the tail and the head of the FIFO buffer. In other words, a data element written to the head of the FIFO buffer is read δ time units later at the tail end.

Let us now define the set of bus states and auxiliary get and put functions that enable us to add/remove data elements from the internal data structure of the bus.

- $B_{N,\delta} = 2^{\{0,\ldots,\delta\}\times\mathbb{V}}$ ie $B_{N,\delta}$ is the set of subsets of tuples, the first element being a *time in flight* (TIF) counter (with the initial TIF of any element in the FIFO being equal to δ) and the second element being the actual data value that is stored as a cell in a FIFO where the cardinality of the set is N.

- $$put(\langle t, v\rangle, b) = \begin{cases} b \bigcup \{\langle t, v\rangle\} & \text{if } \neg\exists v'.\langle t, v'\rangle \in b \\ b - \{\langle t, v'\rangle\} & \text{if } \langle t, v'\rangle \in b \\ undefined & \text{if } \mid b \mid = N \end{cases}$$

  The intuition behind this is that when an element is written to the FIFO, the *put* logic checks to see if there is another data value with the same TIF already present in the FIFO. If there is not, then the data is added to the FIFO. If there exists a data value with the same TIF (i.e. the same time-stamp), then a data collision has occurred or in other words, simultaneous writes to the same location at the same time has taken place. In this case, we allow none of the colliding values to enter the FIFO.

- We define a *tick* function that decrements the TIF associated with each element in the FIFO.
  $tick(b, n) = \{\langle t - n, l\rangle \mid \langle t, l\rangle \in b\}$

  Note that the definition of *tick* allows the TIF to become negative. A data value with an associated negative TIF can be interpreted as a value that has propagated from the head to the tail of the FIFO but has still not been read (and thus removed) from the FIFO. If more than one data value with a negative TIF exists, the one with the minimum TIF is the one that is read (since it has been in the FIFO for the longest time).

- Before we define, the *get* function for a FIFO, we need to establish some auxiliary definitions.

  lnp is a function that takes a set of tuples of the form $\langle t, v\rangle$ and returns a tuple with the lowest non-positive TIF.

  $$lnp(T) = \begin{cases} \langle t, v\rangle & \text{if } \langle t, v\rangle \in T,\ t \leq 0 \text{ and } \forall\langle t', v'\rangle \in T.t' \geq t \\ undefined \text{ otherwise} \end{cases}$$

  Now we define the *get* function.

  $$get(b) = \begin{cases} \langle v, b - \{lnp(b)\}\rangle & \text{if } lnp(b) = \langle t, v\rangle \\ undefined & \text{otherwise} \end{cases}$$

Rule 1 in Table 2 state that as long as the buffer is not full, writers can keep on writing to the bus. All such writes are instantaneous, i.e. time progresses only after all enabled writes are completed. When a data value enters the bus, there is a time-stamp (or more precisely a time in flight) that is attached to it—in this example since propagation delay is assumed to be δ, we attach the value δ to every data value once it is written to the bus. Every time a clock ticks, the "time in flight" is decremented by 1 (Rule 2). Readers who want to read are allowed to do so only if the data element at the read end of the buffer has its "time in flight" less

Table 2
FIFO bus $\langle I, B_{N,\delta}, \emptyset, T\rangle$, with capacity N and δ delay

$$I = \langle \mathbb{W}, \mathbb{R} \rangle$$
$$B_{N,\delta} = 2^{\{0,\delta\}\times\mathbb{V}}$$

T is defined as follows.

(i) $b \xrightarrow[WV\ R]{W\ RV} b'$ iff $\exists \langle w, v\rangle \in WV.\, W = \{w\} \wedge b' = put(\langle \delta, v\rangle, b)$
or $\exists r \in R, v \in \mathbb{V}.\, get(b) = \langle v, b'\rangle \wedge RV = \{\langle r, v\rangle\}$

(ii) $b' \xrightarrow{1} b''$ iff $b'' = tick(b', 1)$

than or equal to 0 which means that it has been in the buffer for at least δ time units.

5 Related Work

The SAE Architecture Analysis & Design Language (AADL) [6] (that grew out of MetaH [8]) is a textual and graphical language supports model-based engineering of embedded real-time systems. However AADL, not having an explicit *execution semantics* for inter-component communication lacks the power to package communication behavior into architectural elements which can then be used as atomic blocks for system construction. All AADL communication takes place implicitly through queued or nonqueued data at ports whereas in TAID, components perform communication via an explicit entity called a bus that encapsulates the specific communication semantics which is not limited to queued/non-queued communication only.

WRIGHT [14] and TRACTA [4] are ADLs that, like TAID, have a formal executable semantics "under the hood". However, they do not provide notions for modeling time and also do not provide for the ability to parameterize communication ie these languages have a single communication primitive that is built into it and that cannot be extended by any means. Ptolemy [10] is a modeling environment that provides support for heterogenous specifications by allowing for encodings of different models of computation. TAID distinguishes itself from Ptolemy by its use of buses as a means for encapsulating different models of computation (synchronous, asynchronous) in a concise manner.

There are several tool implementations eg Artisan Studio [7] that realize UML-RT [3] and SysML [9] constructs but this approach suffers from UML/SysML not having a standardized execution semantics.

For a more full-fledged discussion of different ADLs with respect to AID,interested readers are requested to refer to [15]

6 Conclusions and Future Work

This paper describes a executable, timed specification language, TAID, for describing real-time, distributed systems. TAID is intended for use in the design process for high-integrity embedded systems. Our future work consists of using

Simulink/Stateflow as modeling infrastructure for representing and simulating distributed real-time system designs comprising TAIDs whose components are Simulink / Stateflow models. The effort also involves the development of Simulink blocksets as specifications for standard communication platforms. A project on this topic is underway with a major international automotive company.

References

[1] J.C.M. Baeten and C.A. Middelburg. Process algebra with timing: real time and discrete time. *Handbook of Process Algebra (J. A. Bergstra and A. Ponse and S. A. Smolka, editors)*, pages 627–684, 2001.

[2] P. Clements and L. Northrop. Software product lines: Practices and patterns. *Boston, MA: Addison-Wesley*, 2002.

[3] Bruce Powel Douglass and David Harel. Real-time UML: Developing Efficient Objects for Embedded Systems. 1997.

[4] D. Giannakopoulou. The TRACTA Approach for Behaviour Analysis of Concurrent Systems. *Department of Computing, Imperial College of Science, Technology and Medicine DoC 95/16*, 1995.

[5] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Inf. Comput*, 117(2):221–239, 1995.

[6] http://www.aadl.info.

[7] http://www.artisansw.com/pdflibrary/Studio6.0.pdf.

[8] http://www.htc.honeywell.com/metah/. Honeywell corp.

[9] http://www.sysml.org/.

[10] Edward Lee. Overview of ptolemy project. *Technical Memorandum*, 6(3):213–249, 2001.

[11] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September.

[12] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[13] R.Allen and D.Garlan. Formalizing architectural connection. *16th International Conference on Software Engineering*, 1994.

[14] R.Allen and D.Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[15] Arnab Ray and Rance Cleaveland. Architectural interaction diagrams: Aids for system modeling. *Proceedings of the International Conference on Software Engineering,(ICSE)*, pages 396–406, 2003.

[16] Mary Shaw. Software Architectures for Shared Information Systems. *D.M. Steier and T.M. Mitchell (Eds.), Mind Matters: A Tribute to Allen Newell. Mahwah, NJ*, pages 219–251, 1996.

[17] Kevin J. Sullivan and John C. Knight. Experience assessing an architectural approach to large-scale systematic reuse. *Proceedings of 18th International Conference on Software Engineering*, pages 220–229, 1996.

4. Instantaneous Transitions in Esterel

Olivier Tardieu[1], and Stephen A. Edwards[2]

1 INRIA, Sophia Antipolis, France
2 Columbia University in the City of New York, USA

Notes:

Instantaneous Transitions in Esterel

Olivier Tardieu[1,3]

INRIA, Sophia Antipolis

Stephen A. Edwards[2,4]

Columbia University in the City of New York

Abstract

Esterel is an imperative synchronous programming language for the specification of deterministic concurrent reactive systems. While providing the usual control-flow constructs—sequences, loops, conditionals, and exceptions—its lack of a goto instruction makes the programming of arbitrary finite state machines awkward and hinders the design of source-to-source program transformations. We previously introduced to Esterel a non-instantaneous gotopause instruction, which prevents the synchronous execution of code before and code after the transition. Here, we tackle instantaneous transitions. Concurrency demands we assign scopes and priorities to gotos, so we extend Esterel's exception handling mechanism to allow exception handlers in arbitrary locations. We advocate for and formalize the resulting language. We observe that instantaneous gotos complement but do not replace non-instantaneous gotopauses.

Keywords: concurrency, exceptions, SyncCharts, compilation.

1 Introduction

Esterel [3,4,5,6] is a concurrent programming language. Its syntax is imperative and fit for the design of control-oriented reactive systems [10]. Its semantics are synchronous: active threads run in lockstep and communicate via instantly broadcast signals. Like most modern imperative languages, Esterel promotes structured programming. Common programming practice strongly discourages the use of gotos when they are available [8], but Esterel provides none at all.

The lack of goto is not without reason. First, gotos and concurrency do not mix well and Esterel code is hardly ever sequential. Second, loops—simple forms of jumps—already cause substantial trouble. To make a long story short, a complex loop unfolding algorithm—reincarnation [3,19]—is a mandatory step in the compilation of Esterel.

[1] Email: olivier.tardieu@sophia.inria.fr

[2] Email: sedwards@cs.columbia.edu

[3] Tardieu was at Columbia when this work was performed.

[4] Edwards and his group are supported by the NSF, Intel, Altera, the SRC, and NYSTAR.

Nevertheless, the lack of a goto instruction is a drawback. Many standards explicitly prescribe (unstructured) state machines. For example, the link layer specification of the Serial ATA standard [16] specifies a 31-state machine by listing transitions in a table. To describe such machines, many formalisms, such as SyncCharts [1,2], provide graphical modeling of reactive systems using hierarchical and parallel compositions of finite state machines. While its synchronous semantics match those of Esterel, the translation from SyncCharts to Esterel is awkward and obfuscates the programmer's intent. Transitions are encoded with signaling. Arbitrary state machines can be encoded using one concurrent process per state. But maintaining structural information about exclusive states in the generated code is not easy. In contrast, a goto allows the direct encoding of transitions and the preservation of this information.

Internally, all Esterel compilers use ad hoc intermediate languages (e.g., IC [5] and GRC [14]) that expand Esterel control-flow constructs into jump instructions. This suggests adding gotos to Esterel should not only be feasible but also have a minor impact on code generation. While for code generation, it would be reasonable to translate formalisms such as SyncCharts directly to such internal formats, this would not help users reason about specifications.

Previously, we extended Esterel with a *gotopause* instruction [17]. By design, it ensures that one instant elapses between the execution of the jump instruction and the execution of the code following the target of the jump. Thanks to the definition of well-formed programs, we were able to specify non-instantaneous jumps that are consistent with the principles of deterministic synchronous concurrency. The delay implies their semantics do not involve unfolding, making compilation trivial.

Of course, non-instantaneous jumps are no help for the programming of finite state machines with instantaneous transitions. In this paper, we introduce instantaneous jumps, which we obtain by combining features of loops, exceptions, and non-instantaneous jumps. First, like exceptions, instantaneous jumps have scopes and are prioritized accordingly. In a series of concurrent jumps, all but the highest-priority jump are ignored. Second, as with loops, the semantics of instantaneous jumps rely on unfolding. Finally, the machinery for transferring control to a distant location in the source code already exists in the formal semantics of Esterel thanks to *gotopause*.

We introduce instantaneous jumps by extending the exception handling mechanism of Esterel. Raising an exception normally jumps to the end of the exception scope. Our extension makes it possible to place the exception handler, i.e., the target of the jump, at any point within the scope of the exception. This employs an explicit *catch* instruction, which behaves like a label.

While this "exception handler within a trap" construct may appear strange, simply taking a more traditional goto-and-label approach would come with too many caveats to be any simpler. This paper aims at understanding the interactions between concurrency and gotos to provide a formal framework that can be used to add a variety of jump constructs. What if a goto attempts to exit the scope of an exception? What if concurrent gotos target exclusive program states? Our design minimizes the change to the language and its semantics. We only suggest a general, low-level syntax. Additional syntactic sugar is probably necessary.

statements		locations	compatible locations
$p, q ::=$	`nothing`	$\emptyset$	$\emptyset$
	ℓ`:pause`	$\{\ell\}$	$\emptyset$
	`gotopause` ℓ	$\emptyset$	$\emptyset$
	p `;` q	$\mathcal{L}_p \cup \mathcal{L}_q$	$\mathcal{C}_p \cup \mathcal{C}_q$
	p `||` q	$\mathcal{L}_p \cup \mathcal{L}_q$	$\mathcal{C}_p \cup \mathcal{C}_q \cup (\mathcal{L}_p \times \mathcal{L}_q) \cup (\mathcal{L}_q \times \mathcal{L}_p)$
	`[`p`]`	$\mathcal{L}_p$	$\mathcal{C}_p$
	`loop` p `end`	$\mathcal{L}_p$	$\mathcal{C}_p$
	`signal` S `in` p `end`	$\mathcal{L}_p$	$\mathcal{C}_p$
	`emit` S	$\emptyset$	$\emptyset$
	`present` S `then` p `else` q `end`	$\mathcal{L}_p \cup \mathcal{L}_q$	$\mathcal{C}_p \cup \mathcal{C}_q$
	`suspend` p `when` S	$\mathcal{L}_p$	$\mathcal{C}_p$
	`trap` T `in` p `end`	$\mathcal{L}_p$	$\mathcal{C}_p$
	`exit` T	$\emptyset$	$\emptyset$

Fig. 1. The syntax of Esterel. Compatible locations.

In particular, we show instantaneous gotos do not generalize non-instantaneous gotos but complement them: *gotopause* instructions are not simply instantaneous jumps plus delays.

We describe the syntax and semantics of the Esterel language and the *gotopause* instruction in Section 2. We introduce and formalize the *catch* instruction in Section 3. Through an example, we illustrate the encoding of state machines with instantaneous transitions. We also discuss loop elimination as an instance of a source-to-source program transformation relying on the new construct. We discuss related work in Section 4 and conclude in Section 5.

2 Esterel and *gotopause*

Without loss of generality, we focus on the pure Esterel language augmented with a *gotopause* instruction. The full language is obtained from its pure fragment by adding data-centric constructs irrelevant to our discussion.

2.1 Syntax and Intuitive Semantics

We describe the grammar of our kernel language in Fig. 1, Col. 1. The non-terminals p and q denote statements, S signal identifiers, T exception identifiers, and ℓ integer labels. The infix "`;`" operator binds tighter than "`||`."

In Cols. 2 and 3, we recursively define the locations $\mathcal{L}_p$ and the compatible locations $\mathcal{C}_p$ of the statement p. The locations of p are the labels of the *pause* instructions in p. They must be pairwise distinct. Formally, in statements when both p and q occur, the sets $\mathcal{L}_p$ and $\mathcal{L}_q$ must be disjoint. For example, `1:pause ; 1:pause` is illegal. We discuss compatible locations later.

The execution of an Esterel program, i.e., a statement, consists of a possibly infinite sequence of atomic execution steps called reactions. Each reaction is said to last for one instant. *Pause* instructions represent reaction boundaries, i.e., the progress of time.

- `nothing` does nothing; terminates instantly, that is to say a statement immediately after this instruction is run instantly.
- ℓ`:pause` suspends the execution for one instant. The statement immediately after this instruction, if any, is run in the next instant of execution.
- `gotopause` ℓ suspends the execution for one instant. The statement immediately after the *pause* instruction with label ℓ is run in the next instant of execution.
- p `;` q executes p instantly followed by q if/when p terminates; instantly terminates if/when q terminates. If the execution of p raises an exception then it is instantly propagated upward and q is not run. If the execution of q raises an exception then it is instantly propagated upward.
- p `||` q executes p in parallel with q synchronously: one reaction of p `||` q consists of one reaction of p and one reaction of q until p or q terminates. If p terminates first then q continues running and p `||` q instantly terminates when q does (and vice versa). If p and q raise exceptions in the same instant, the exception with higher priority is instantly propagated upward. If p only raises an exception then q is allowed to complete its current reaction before this exception is instantly propagated upward. Even if incomplete, the execution of q is not resumed in the next instant (and vice versa).
- `[`p`]` runs p. This allows sequences of parallel statements, e.g., `[`p `||` q`];[`r `||` s`]`.
- `loop` p `end` repeats p forever unless p raises an exception, which is instantly propagated upward. Two iterations of the loop may not complete in the same instant. E.g., `loop nothing end` is illegal. This constraint ensures that atomic execution steps (reactions) can be computed with statically bounded resources [18].
- `signal` S `in` p `end` declares the local signal S in p and executes p. Signals are lexically scoped. Signal declarations are not mandatory. Undeclared signals occurring in *emit* and *present* constructs are considered global.
- `emit` S emits signal S and terminates instantly. Global signals may be emitted by the environment in addition to the program itself.
- `present` S `then` p `else` q `end` executes p if S is emitted in this instant (by the program or the environment if global), and executes q otherwise. If the execution of the chosen branch requires more than one instant, it is continued in the next instants independently from the status of S in these instants.
- `suspend` p `when` S instantly starts executing p and ignores the status of S. However, if the execution of p does not complete instantly, it is only allowed to run in later instants in which S is not emitted (otherwise, it is suspended).
- `trap` T `in` p `end` declares exception T in p and executes p. Exceptions are lexically scoped. If p terminates or raises exception T then `trap` T `in` p `end` terminates instantly. If p raises a different exception it is propagated upward. In case of nested exception declarations, the outermost declaration has the highest priority.
- `exit` T raises exception T. We define *depth*(`exit` T) as the number of *trap* constructs enclosing the *exit* and enclosed in the declaration of T.

For example,

```
trap T
  emit A ; 1:pause ; emit B ; exit T ; emit C
||
  emit D ; 2:pause ; emit E ; emit F ; 3:pause ; emit G
end ; emit H
```

emits signals `A` and `D` in its first reaction, then `B`, `E`, `F`, and `H` in its second and final reaction. Neither `C` nor `G` is emitted. Here, the depth of `exit T` is 0.

Locations represent possible suspension points for the execution between two reactions. In previous example, after the first reaction, the execution is suspended at locations 1 and 2.

In Fig. 1, Col. 3, we define compatible locations. Two locations ℓ and ℓ' are compatible in p, i.e., $(\ell, \ell') \in \mathcal{C}_p$, iff these locations belong to concurrent branches of p. By construction, in the usual Esterel language (no *gotopause*), only compatible locations may be reached simultaneously. If L_0 is a set of pairwise compatible locations of the program p, we write p/L_0 for the program p suspended at locations L_0. We say p/L_0 is a state of the program p.

In Esterel with *gotopause*, several *gotopause* instructions may be executed concurrently. Their target locations must exist and be pairwise compatible [19]:

- `[ gotopause 1 || gotopause 2 ] ; [ 1:pause || 2:pause ]` is fine.
- `gotopause 1 ; 2:pause` is illegal because the *gotopause* instruction lacks a target *pause* instruction.
- `[ gotopause 1 || gotopause 2 ] ; 1:pause ; 2:pause` is illegal because the target *pause* instructions of the jump are not compatible.

2.2 Formal Semantics

We denote by $p\backslash X$ either the program p itself—the program p in its initial state—or the program p in some state p/L_0. Reactions of a program p are expressed via labeled transitions of the form:

$$p\backslash X \xrightarrow[I]{O,k} L$$

- $p\backslash X$ is the state from which the reaction starts;
- I is the set of signals emitted by the environment; [5]
- O is the set of signals emitted by the program;
- k is the completion code of the reaction:
  · $k = 0$ if the execution terminates instantly,
  · $k = 1$ if part of the execution is delayed due to *pause*(s) or *gotopause*(s),
  · $k \geq 2$ if an exception is reported; and
- p/L is state reached by the reaction. By construction, $L \neq \emptyset$ iff $k = 1$.

[5] This differs from the usual presentations of the language semantics, where present signals are considered instead ($E = I \cup O$). We choose such a presentation here because we find it more intuitive. This choice has no impact on the language extension we propose.

In Fig. 2, we specify the semantics of Esterel with *gotopause* as a set of facts and deduction rules in a structural operational style [13]. All but the two rules marked (∗) will be preserved unchanged in the specification of Esterel plus *gotopause* plus *catch* in Section 3.4.

Consider the rule

$$\frac{p\backslash X \xrightarrow[I\cup O']{O,0} \emptyset \qquad q \xrightarrow[I\cup O]{O',k} L}{p \;;\; q\backslash X \xrightarrow[I]{O\cup O',k} L}.$$

It specifies that $p \;;\; q$ when started (resp. restarted in state $p \;;\; q/L_0$) may react to inputs I with outputs $O \cup O'$, completion code k, and reaches the state $p \;;\; q/L$ if

- p when started (resp. restarted in state p/L_0) reacts to inputs $I \cup O'$ by terminating instantly with outputs O; and
- q when started reacts to inputs $I \cup O$ with outputs O', completion code k, and reaches the state q/L.

Because of the *synchrony hypothesis*, not only are the outputs O of p inputs of q, but reciprocally the outputs O' of q are inputs of p.

2.3 Instantaneous Loops and Reincarnation

Using the extended exception handling mechanism we propose, one can implement loops without the *loop* construct. We focus here on understanding the properties of loops, which our language extension must preserve.

The formal semantics of the *loop* construct consists of two rules so that

- `loop` p `end` $\xrightarrow[I]{O,k} L$ iff $p \xrightarrow[I]{O,k} L \wedge k \neq 0$ and
- `loop` p/L_0 `end` $\xrightarrow[I]{O,k} L$ iff $\begin{cases} \text{either } p/L_0 \xrightarrow[I]{O,k} L \wedge k \neq 0 \\ \text{or } p/L_0 \xrightarrow[I\cup B]{A,0} \emptyset \wedge p \xrightarrow[I\cup A]{B,k} L \wedge k \neq 0 \wedge O = A \cup B \end{cases}$.

When `loop` p `end` starts executing, it starts executing its body p, which may either suspend its execution (because of *pause* or *gotopause* instructions) or raise an exception; but p cannot terminate instantly. When the loop is restarted from the state L_0, it restarts its body. Now, if p terminates instantly, a new iteration—a new execution of p—is instantly started. Again, this iteration cannot terminate instantly.

First, observe that a program such as `loop nothing end` admits no possible execution: it deadlocks. In the sequel, we introduce similar safeguards to the semantics of exceptions that choose deadlocks over instantly diverging behaviors.

Second, *loop* and *signal* constructs do not commute. In Fig. 3, program (a) emits signal `A` from the second instant onwards. In contrast, program (b) never emits `A` because, in each reaction, the test applies to a fresh signal `S` distinct from the emitted signal `S`. We say signal `S` is reincarnated because of the loop. In the sequel, we implement comparable interaction rules for *signal* and *trap* scopes so loops built from *trap-exit-catch* constructs behave in the same way.

$$\texttt{nothing} \xrightarrow[I]{\emptyset,0} \emptyset$$

$$\texttt{emit}\ S \xrightarrow[I]{\{S\},0} \emptyset$$

$$\texttt{exit}\ T \xrightarrow[I]{\emptyset,\, depth(\texttt{exit}\ T)+2} \emptyset$$

$$\frac{p\backslash X \xrightarrow[I\cup O']{O,0} \emptyset \qquad q \xrightarrow[I\cup O]{O',k} L}{p\ \texttt{;}\ q\backslash X \xrightarrow[I]{O\cup O',k} L}$$

$$\frac{p\backslash X \xrightarrow[I]{O,k} L \qquad k\neq 0}{p\ \texttt{;}\ q\backslash X \xrightarrow[I]{O,k} L}$$

$$\frac{q/L_0 \xrightarrow[I]{O,k} L}{p\ \texttt{;}\ q/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{p \xrightarrow[I]{O,k} L}{\texttt{suspend}\ p\ \texttt{when}\ S \xrightarrow[I]{O,k} L}$$

$$\frac{S\notin I\cup O \qquad p/L_0 \xrightarrow[I]{O,k} L}{\texttt{suspend}\ p\ \texttt{when}\ S/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{p\backslash X \xrightarrow[I]{O,k} L \qquad k\neq 0}{\texttt{loop}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O,k} L}$$

$$\frac{p/L_0 \xrightarrow[I]{O,k} L \qquad L_0\cap\mathcal{L}_q=\emptyset}{p\ \texttt{||}\ q/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{q/L_0 \xrightarrow[I]{O,k} L \qquad L_0\cap\mathcal{L}_p=\emptyset}{p\ \texttt{||}\ q/L_0 \xrightarrow[I]{O,k} L}$$

$$\forall k,k' : k\sqcup k' = \max(k,k')$$
$$\begin{cases} \downarrow 0 = \downarrow 2 = 0 \\ \downarrow 1 = 1 \\ \downarrow n = n-1 \qquad \forall n>2 \end{cases}$$

$$\ell\texttt{:pause} \xrightarrow[I]{\emptyset,1} \{\ell\}$$

$$\texttt{gotopause}\ \ell \xrightarrow[I]{\emptyset,1} \{\ell\}$$

$$\frac{\ell\in L_0}{\ell\texttt{:pause}/L_0 \xrightarrow[I]{\emptyset,0} \emptyset}$$

$$\frac{p\backslash X \xrightarrow[I\backslash\{S\}]{O,k} L}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O\backslash\{S\},k} L}$$

$$\frac{S\in I\cup O \qquad p \xrightarrow[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O,k} L}$$

$$\frac{S\notin I\cup O \qquad q \xrightarrow[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O,k} L}$$

$$\frac{p/L_0 \xrightarrow[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{q/L_0 \xrightarrow[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{p/L_0 \xrightarrow[I\cup O']{O,0} \emptyset \qquad p \xrightarrow[I\cup O]{O',k} L \qquad k\neq 0}{\texttt{loop}\ p\ \texttt{end}/L_0 \xrightarrow[I]{O\cup O',k} L}$$

$$\frac{p \xrightarrow[I\cup O']{O,k} L \qquad q \xrightarrow[I\cup O]{O',k'} L'}{p\ \texttt{||}\ q \xrightarrow[I]{O\cup O',k\sqcup k'} \begin{cases} L\cup L' & \text{if } k\sqcup k'=1 \\ \emptyset & \text{if } k\sqcup k'\neq 1 \end{cases}}$$

$$\frac{p/L_0\cap\mathcal{L}_p \xrightarrow[I\cup O']{O,k} L \qquad q/L_0\cap\mathcal{L}_q \xrightarrow[I\cup O]{O',k'} L'}{p\ \texttt{||}\ q/L_0 \xrightarrow[I]{O\cup O',k\sqcup k'} \begin{cases} L\cup L' & \text{if } k\sqcup k'=1 \\ \emptyset & \text{if } k\sqcup k'\neq 1 \end{cases}}$$

$$\frac{S\in I \qquad L_0\neq\emptyset}{\texttt{suspend}\ p\ \texttt{when}\ S/L_0 \xrightarrow[I]{\emptyset,1} L_0} \qquad (*)$$

$$\frac{p\backslash X \xrightarrow[I]{O,k} L}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O,\downarrow k} L} \qquad (*)$$

Fig. 2. The semantics of Esterel with *gotopause*.

```
signal S in
  loop
    present S then emit A end;
    1:pause;
    emit S;
  end
end
```

(a)

```
loop
  signal S in
    present S then emit A end;
    1:pause;
    emit S;
  end
end
```

(b)

Fig. 3. Loops and reincarnation.

3 Introducing *catch* in Esterel

We now extend Esterel with a new *catch* instruction. The syntax becomes

$$p, q ::= \texttt{nothing} \mid \ell\texttt{:pause} \mid ... \mid \texttt{exit}\ T \mid \texttt{catch}\ T$$

with the constraint that there can be at most one `catch` T statement in the scope of each `trap` T `in` ... `end` construct under the usual scoping rules. For instance, if there are two nested declarations for the same exception identifier T, then there can be at most one `catch` T statement inside the inner declaration plus at most one `catch` T statement between the declarations.

If there is no such *catch* instruction, we always implicitly add one at the end of the *trap* body:

$$\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \quad \rightarrow \quad \texttt{trap}\ T\ \texttt{in}\ p\ \texttt{;}\ \texttt{catch}\ T\ \texttt{end}$$

Hence, in the sequel, we assume there is exactly one `catch` T statement for each declaration of T.

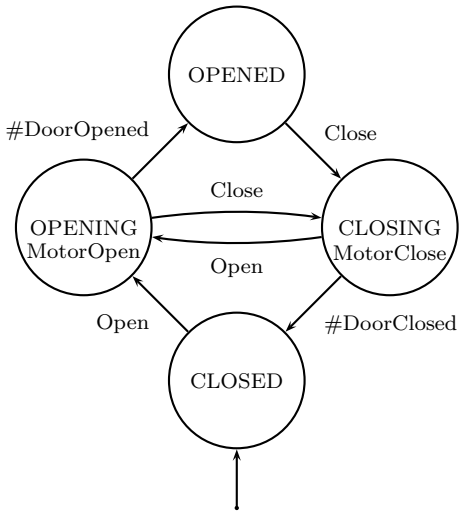
The *catch* instruction grabs control instantly when the corresponding exception occurs. Intuitively, *exit* is like a *goto* with *catch* as its label.

3.1 Example

In Fig. 4, we demonstrate the encoding of a state machine for an elevator door using *catch*. It has four states: OPENING, OPENED, CLOSING, and CLOSED the initial state of the machine. The input signals Open and Close convey user commands. The input signals DoorOpened and DoorClosed indicate the door's position. The output signals MotorOpen and MotorClose control the motor. Control signals must be sustained over a period of time for the door to fully open or fully close.

In this design, the DoorOpened and DoorClosed sensor signals must be taken into account instantly—as specified with #—so that the motor is shut down without delay. Moreover, we want instantaneous transitions to take priority over non-instantaneous transitions.

This design is implemented as follows. One exception is declared for each state. Exception priorities are irrelevant here because we never raise two exceptions simultaneously. State entry points are specified with *catch* constructs. Instantaneous transitions are encoded by *exit* constructs. Non-instantaneous transitions are delayed by *pause* instructions. Alternatively, *gotopause* instructions could be used for non-instantaneous transitions here.



(a)

```
trap OPENING in trap OPENED in
trap CLOSING in trap CLOSED in

  catch CLOSED;
    present Open then pause; exit OPENING end;
    pause; exit CLOSED;

  catch OPENING;
    present DoorOpened then exit OPENED end;
    emit MotorOpen;
    present Close then pause; exit CLOSING end;
    pause; exit OPENING;

  catch OPENED;
    present Close then pause; exit CLOSING end;
    pause; exit OPENED;

  catch CLOSING;
    present DoorClosed then exit CLOSED end;
    emit MotorClose;
    present Open then pause; exit OPENING end;
    pause; exit CLOSING;

end end end end
```

(b)

Fig. 4. Encoding an arbitrary state machine with *trap-exit-catch*. (a) A state machine for an elevator door. DoorOpened and DoorClosed are sensors that indicate the door's position; Open and Close initiate or override commands; and MotorOpen and MotorClose control the motor. (b) Coding this using the *catch* instruction.

3.2 Catch in Sequential Code

The *exit-catch* construct mimics the *goto-label* construct of C. For example,

```
trap T in exit T ; emit A ; catch T ; emit B end
```

only emits `B`. In general, the semantics of *exit* is that the body of its enclosing *trap* is terminated and restarted at the *catch*. In particular, the *catch* instruction may occur to the left of the corresponding *exit*(s). For instance,

```
trap T in emit A ; catch T ; emit B ; 1:pause ; exit T end
```

behaves just like

```
emit A ; loop emit B ; 1:pause end
```

Incidentally, this means that `catch T`, when run immediately after `emit A`, does nothing and terminates instantly.

In general, the expansion of loops

$$\texttt{loop}\ p\ \texttt{end} \quad \rightarrow \quad \texttt{trap}\ T\ \texttt{in exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T\ \texttt{end}$$

is semantics-preserving provided T is a fresh exception identifier. In particular, p cannot terminate instantly in this context. We prove the correctness of the expansion and motivate the first *exit* in Section 3.5.

Since the semantics of *exit* is that the body of its enclosing *trap* is terminated and restarted at the *catch*, the signals local to the *trap* body are reincarnated as the control jumps from *exit* to *catch*. In Fig. 5, program (a), signal `A` is emitted because the *signal* statement is not restarted. In contrast, in program (b), signal `S` is reincarnated because the *exit* statement causes the body of the *trap*, which includes the *signal* scope, to be terminated and restarted. Thus, a second, fresh incarnation of signal `S` appears and signal `A` is not emitted here.

```
signal S in
  trap T in
    emit S;
    exit T;
    catch T;
    present S then
      emit A        % runs
    end
  end
end
```

(a)

```
trap T in
  signal S in
    emit S;
    exit T;
    catch T;
    present S then
      emit A        % does not run
    end
  end
end
```

(b)

Fig. 5. The effect of scopes.

3.3 Catch and Concurrency

Several *exit*s may execute concurrently, as illustrated in Fig. 6. When program (a) runs, `exit T1` and `exit T2` both execute. However, because exception `T1` takes precedence over `T2`, only `catch T1` is relevant: control resumes from there, and `A` is emitted instantly. Signal `B` is not emitted because control is only transferred to the first parallel branch; the second parallel branch is treated as having terminated.

In contrast, in program (b), the two *gotopause* statements are equally relevant, jumping to both branches of the second parallel, meaning that both `A` and `B` are emitted in the second instant.

Furthermore, we observe that program (c) is legal whereas program (d) is not. In program (c), two *exit* statements execute instantly, but again only the outermost exception affects control, so only `B` is emitted. However, concurrent *gotopause* statements that send control into a sequence—incompatible locations—are illegal. Priorities eliminate this potential problem with *exit* statements.

Since *gotopause*(s) and *exit*(s) implement dual approaches to concurrency, *gotopause* instructions do not reduce to *trap-exit-catch* constructs plus delays. On the one hand, *trap-exit-catch* constructs cannot replace *gotopause* instructions when several targets must be reached concurrently and the scopes of the concurrent jumps intersect.[6] On the other hand, *gotopause* instructions cannot encode the instantaneous transitions of SyncCharts specifications. As a result, we believe it makes sense to retain both constructs.

3.4 Formal Semantics

Previously, we defined the states of a program p as pairs p/L_0 where L_0 is a set of compatible locations of p and also the initial state of p, which we identified with p. To express the semantics of the *catch* instruction, we now introduce exception states: for each statement in the scope of a `trap` T `in` ... `end` construct and containing a `catch` T statement, we associate the exception state $p/\#T$. In other words,

[6] The scope of a non-instantaneous jump is the least program piece that contains both the source *gotopause* and target *pause* instructions of the jump. The scopes of concurrently executed jumps are typically pairwise disjoint when using *gotopause* to encode SyncCharts non-instantaneous transitions thanks to SyncCharts restrictions on inter-level transitions. In contrast, these scopes are typically not disjoint when using *gotopause* to cure schizophrenia [19].

```
trap T1 in
  trap T2 in
    [
      exit T1
    ||
      exit T2
    ];
    [
      catch T1;
      emit A % runs
    ||
      catch T2;
      emit B % doesn't
    ]
  end
end
```

(a)

```
[
  gotopause 1
||
  gotopause 2
];
[
  1:pause;
  emit A % runs
||
  2:pause;
  emit B % runs
]
```

(b)

```
% OK
trap T1 in
  trap T2 in
    [
      exit T1
    ||
      exit T2
    ];
    catch T2;
    emit A
    catch T1;
    emit B
  end
end
```

(c)

```
% Erroneous
[
  gotopause 1
||
  gotopause 2
];
2:pause;
emit A;
1:pause;
emit B
```

(d)

Fig. 6. The difference between *gotopause* and *trap-exit-catch*.

we extend the locations of p to contain not only the locations of its *pause* instructions but also the locations of its *catch* instructions. Moreover, we consider these new locations to be first pairwise incompatible and second incompatible with *pause* locations. Now, the set L_0 in p/L_0 is either a potentially empty set of compatible *pause* labels of p or the single location $\#T$ of some `catch` T statement in p.

The formal semantics of Fig. 2 consists of twenty-four rules. To extend Esterel with the *catch* instruction, we preserve the first twenty-two rules, discard the two rules marked $(*)$, and add the six rules in Fig. 7 for *catch*, *trap*, and *suspend*:

- `catch` T does nothing and terminates instantly when started or restarted from location $\#T$.
- `trap` T `in` p `end` behaves like p if exception T is never raised. If T is raised then the *trap* construct instantly restarts p at location $\#T$. This execution cannot instantly raise T again ($k \neq 2$). Both rules for the *trap* construct carefully avoid capturing another exception with same identifier T by using the test $X \neq \#T$, which is shorthand for "if $p \backslash X$ is of the form p/L_0 then $L_0 \neq \#T$."
- `suspend` p `when` S when requested to restart from some location $\#T$, does so ignoring the status of signal S. Because the semantics of the *trap* construct consists in exiting and restarting its body if the exception occurs, inner *suspend* statements are considered to be in their first instant of execution when restarted. Thus, as usual, we want to ignore the status of S in the first instant.

By construction, the final state of any reaction cannot be an exception state: exception states are both generated and evaluated within the instant.

The `trap` T `in` p `end` statement, by preventing exception T from occurring twice instantly in p, effectively forbids instantaneous loops. Because the *trap* instruction starts a fresh incarnation of p when the exception occurs, reincarnation of signals local to p takes place as expected.

$$\texttt{catch}\ T \xrightarrow[I]{\emptyset,0} \emptyset \qquad \texttt{catch}\ T/\#T \xrightarrow[I]{\emptyset,0} \emptyset$$

$$\frac{X \neq \#T \quad p\backslash X \xrightarrow[I]{O,k} L \quad k \neq 2}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O,\Downarrow k} L} \qquad \frac{S \in I \quad L_0 \neq \emptyset \quad L_0 \neq \#T}{\texttt{suspend}\ p\ \texttt{when}\ S/L_0 \xrightarrow[I]{\emptyset,1} L_0}$$

$$\frac{X \neq \#T \quad p\backslash X \xrightarrow[I\cup O']{O,2} \emptyset \quad p/\#T \xrightarrow[I\cup O]{O',k} L \quad k \neq 2}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O\cup O',\Downarrow k} L} \qquad \frac{S \in I \quad p/\#T \xrightarrow[I]{O,k} L}{\texttt{suspend}\ p\ \texttt{when}\ S/\#T \xrightarrow[I]{O,k} L}$$

Fig. 7. The semantics of *catch*.

3.5 Correctness Results

We first check the correctness of our language extension by proving that the extended semantics matches the initial semantics for a program without *catch* instructions. We then prove the loop expansion of Section 3.2.

In this section, we denote by $\circ\!\!\rightarrow$ the reactions defined by the initial semantics and by $\rightarrow$ the reactions defined by the extended semantics.

Since we decided earlier to deal with absent *catch* instructions by inserting them at the end of their respective *trap* blocks, we consider the statements:

- initial language: p and $P = \texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}$, and
- extended language: q and $Q = \texttt{trap}\ T\ \texttt{in}\ q\ \texttt{;}\ \texttt{catch}\ T\ \texttt{end}$.

We prove that P and Q are equivalent if p and q are.

Lemma 3.1 *If* $\forall X, \forall I, \forall O, \forall k : p\backslash X \circ\!\!\xrightarrow[I]{O,k} L \Leftrightarrow q\backslash X \xrightarrow[I]{O,k} L$ *then:*
$\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \circ\!\!\xrightarrow[I]{O,k} L \Leftrightarrow \texttt{trap}\ T\ \texttt{in}\ q\ \texttt{;}\ \texttt{catch}\ T\ \texttt{end}\backslash X \xrightarrow[I]{O,k} L.$

Proof. $\forall T', \forall X \neq \#T', \forall I, \forall O, \forall k$: let $\hat{\kappa}$ be k if $k \leq 1$ or $k+1$ otherwise.

First, $\texttt{trap}\ T\ \texttt{in}\ q\ \texttt{;}\ \texttt{catch}\ T\ \texttt{end}\backslash \#T'$ deadlocks for all T' since q does.

Second, $\texttt{trap}\ T\ \texttt{in}\ q\ \texttt{;}\ \texttt{catch}\ T\ \texttt{end}\backslash X \xrightarrow[I]{O,k} L$

iff $\begin{cases} \text{either} \quad q\ \texttt{;}\ \texttt{catch}\ T\backslash X \xrightarrow[I]{O,\hat{\kappa}} L \\ \text{or} \quad O = A \cup B \wedge q\ \texttt{;}\ \texttt{catch}\ T\backslash X \xrightarrow[I\cup B]{A,2} \emptyset \wedge q\ \texttt{;}\ \texttt{catch}\ T/\#T \xrightarrow[I\cup A]{B,\hat{\kappa}} L \end{cases}$

iff $q\backslash X \xrightarrow[I]{O,\hat{\kappa}} L$ or $q\backslash X \xrightarrow[I]{O,2} \emptyset \wedge \hat{\kappa} = 0$

iff $p\backslash X \circ\!\!\xrightarrow[I]{O,\hat{\kappa}} L$ or $p\backslash X \circ\!\!\xrightarrow[I]{O,2} \emptyset \wedge \hat{\kappa} = 0$

iff $\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \circ\!\!\xrightarrow[I]{O,k} L.$ □

Theorem 3.2 *If* p *contains no catch instruction then the initial and extended semantics define the same reactions for all states of* p.

Proof. By induction on the number of nested exception declarations in p. □

We now return to the loop expansion of Section 3.2. Comparing the semantics of the *loop* and *trap* constructs, we observe that the *loop* body is never permitted to terminate instantly, neither in its first nor in subsequent iterations. The *trap* body however may instantly raise the exception. The rules only forbid the exception to occur again when restarting the body from the *catch* location. Therefore, to ensure a correct expansion of *loop*s into *trap-exit-catch* constructs in Section 3.2, we insert a second *exit* at the beginning of the *trap* body in addition to the obvious one at the end of the body.

For simplicity,[7] we establish

Theorem 3.3 *If T is a fresh identifier then these statements are equivalent:*

- `trap` T `in loop` p `end end`,
- `trap` T `in exit` T `;` `catch` T `;` p `;` `exit` T `end`.

Proof. $\forall L_0 \neq \#T, \forall I, \forall O, \forall k$: let $\hat{\kappa}$ be k if $k \leq 1$ or $k+1$ otherwise.

First, `trap` T `in exit` T `;` `catch` T `;` p `;` `exit` T `end`$/L_0 \xrightarrow[I]{O,k} L$

$$\text{iff} \begin{cases} \text{either} \quad \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T/L_0 \xrightarrow[I]{O,\hat{\kappa}} L \\ \text{or} \quad O = A \cup B \wedge \begin{cases} \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T/L_0 \xrightarrow[I\cup B]{A,2} \emptyset \\ \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T/\#T \xrightarrow[I\cup A]{B,\hat{\kappa}} L \end{cases} \end{cases}$$

$$\text{iff} \begin{cases} \text{either} \quad p\ \texttt{;}\ \texttt{exit}\ T/L_0 \xrightarrow[I]{O,\hat{\kappa}} L \\ \text{or} \quad O = A \cup B \wedge p\ \texttt{;}\ \texttt{exit}\ T/L_0 \xrightarrow[I\cup B]{A,2} \emptyset \wedge p\ \texttt{;}\ \texttt{exit}\ T \xrightarrow[I\cup A]{B,\hat{\kappa}} L \end{cases}$$

$$\text{iff} \begin{cases} \text{either} \quad p/L_0 \xrightarrow[I]{O,\hat{\kappa}} L \wedge \hat{\kappa} \neq 0 \\ \text{or} \quad O = A \cup B \wedge p/L_0 \xrightarrow[I\cup B]{A,0} L \wedge p \xrightarrow[I\cup A]{B,\hat{\kappa}} L \wedge \hat{\kappa} \neq 0 \end{cases}$$

iff `loop` p `end`$/L_0 \xrightarrow[I]{O,\hat{\kappa}} L$, thus iff `trap` T `in loop` p `end end`$/L_0 \xrightarrow[I]{O,k} L$.

Second, `trap` T `in exit` T `;` `catch` T `;` p `;` `exit` T `end` $\xrightarrow[I]{O,k} L$

$$\text{iff} \begin{cases} \text{either} \quad \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T \xrightarrow[I]{O,\hat{\kappa}} L \quad (\textit{impossible}) \\ \text{or} \quad O = A \cup B \wedge \begin{cases} \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T \xrightarrow[I\cup B]{A,2} \emptyset \\ \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T/\#T \xrightarrow[I\cup A]{B,\hat{\kappa}} L \end{cases} \end{cases}$$

iff p `;` `exit` $T \xrightarrow[I]{O,\hat{\kappa}} L$, thus iff $p \xrightarrow[I]{O,\hat{\kappa}} L \wedge \hat{\kappa} \neq 0$

iff `loop` p `end` $\xrightarrow[I]{O,\hat{\kappa}} L$, thus iff `trap` T `in loop` p `end end` $\xrightarrow[I]{O,k} L$.

Finally, both statements deadlock if required to start from location $\#T$. □

[7] The enclosing *trap* construct in the first statement ensures exception depths are identical in the two statements. Hence, there is no need to micromanage depths in the proof.

4 Related Work

The origin of this paper was the usual connection between transitions in finite state machine and gotos in imperative languages. A transition from state A to state B is nothing but a jump from block A to the beginning of block B, where blocks A and B implement the behaviors in states A and B.

While graphical design formalisms à la StateCharts [9,20] permit arbitrary, unstructured state machines, Esterel makes it awkward because of its lack of goto.

The goto-like constructs we formalize here follow directly from SyncCharts [1,2], a StateCharts-like graphical modeling language with well-defined synchronous semantics à la Esterel. But our constructs are more expressive than the collection of transitions available in SyncCharts. In particular, the trap-catch-exit construct makes it possible to exit and reenter several layers of nested macrostates at once. While SyncCharts drawings abide by rigid nesting rules and drastically restrict inter-level transitions, we allow them whenever possible.

Coding arbitrary state machines is even harder in pure dataflow synchronous languages because the programmer is responsible for specifying all sequential behavior. To address this, researchers have proposed language extensions such as mode automata [12] in Argos [11] and more recently in Lucid Synchrone [7]. Faithful to the languages they extend, these proposals restrict transitions to avoid complex causal dependencies and schizophrenia. We do not. In particular, we allow arbitrarily (finitely) many transitions to be taken in one instant.

While we want to ease the encoding of graphical synchronous specifications into textual Esterel programs, others have attempted the converse: automatically synthesizing graphical specifications from textual Esterel programs [15]. We hope to eventually combine the two to provide a user-friendly way of switching between graphical and textual representations of a specification.

5 Conclusions

We extend the *trap-exit* construct of Esterel with a new *catch* instruction that allows exception handlers to appear anywhere in the body of the *trap*. One can think of the *exit* instruction as a goto to the location of the corresponding *catch* instruction.

Simultaneous *exit*s result in a single jump to the highest-priority handler. Thus, our *trap-exit-catch* construct supplements but does not supplant the existing *gotopause* instruction for concurrent non-instantaneous jumps. We believe both must coexist in the language. Only *gotopause* can decouple the structure of program states from that of the source code while the *catch* instruction makes it possible to specify finite state machines with instantaneous transitions. In particular, it greatly simplifies the translation of SyncCharts into Esterel.

Although we did not address causality, especially constructive causality [3], we think there is no issue. The semantics of the new construct is obtained by combining existing pieces: loops for reincarnation, exceptions for priorities, and non-instantaneous jumps for locations. Synchronous digital circuit synthesis for the extended language, thus constructive semantics, should be similarly derived. For the same reason, implementing the new construct should be straightforward.

References

[1] C. André. SyncCharts: A visual representation of reactive behaviors. RR 95–52, I3S, 1995.

[2] Charles André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proceedings of Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille, France, July 1996.

[3] Gérard Berry. The constructive semantics of pure Esterel. Draft book, 1999.

[4] Gérard Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brooks, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer-Verlag, Heidelberg, Germany, 1984.

[5] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[6] Frédéric Boussinot and Robert de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[7] Jean-Loius Coalço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 173–182, Jersey City, New Jersey, September 2005.

[8] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] David Harel and Amir Pnueli. *On the Development of Reactive Systems*, volume 13 of *NATO ASI Series. Series F, Computer and Systems Sciences*, pages 477–498. Springer-Verlag, 1985.

[11] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

[12] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *Proceedings of the European Symposium on Programming (ESOP)*, Lisbon (Portugal), March 1998. Springer-Verlag.

[13] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Åarhus, Denmark, 1981.

[14] Dumitru Potop-Butucaru. Optimizations for faster execution of Esterel programs. In *Proceedings of the 1st International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 227–236, Mont St. Michel, France, June 2003.

[15] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing safe state machines from Esterel. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, page FIXME, Ottawa, Canada, June 2006.

[16] Serial ATA Workgroup. *Serial ATA: High Speed Serialized AT Attachment*, August 2001. www.serialata.org.

[17] Olivier Tardieu. Goto and concurrency: Introducing safe jumps in Esterel. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Electronic Notes in Theoretical Computer Science, Barcelona, Spain, 2004. Elsevier.

[18] Olivier Tardieu and Robert de Simone. Instantaneous termination in pure Esterel. In *Proceedings of the 10th Annual Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 91–108, San Diego, California, June 2003.

[19] Olivier Tardieu and Robert de Simone. Curing schizophrenia by program rewriting in Esterel. In *Proceedings of the 2nd International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, San Diego, California, June 2004.

[20] Michael von der Beeck. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems: Third International Symposium Proceedings*, volume 863 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

5. Specifying and executing reactive scenarios with Lutin

Pascal Raymond, Yvan Roux, and Erwan Jahier

VERIMAG (CNRS, UJF, INPG), Grenoble, France

Notes:

Specifying and executing reactive scenarios with Lutin

Pascal Raymond, Yvan Roux, Erwan Jahier [1]

VERIMAG (CNRS, UJF, INPG)
Grenoble, France [2]

Abstract

This paper presents the language Lutin and its operational semantics. This language specifically targets the domain of reactive systems, where an execution is a (virtually) infinite sequence of input/output reactions. More precisely, it is dedicated to the description and the execution of constrained random scenarios. Its first use is for test sequence specification and generation. It can also be useful for early execution, where Lutin programs can be used to simulate modules that are not yet fully developed.
The programming style mixes relational and imperative features. Basic statements are input/output relations, expressing constraints on a single reaction. Those constraints are then combined to describe non deterministic sequences of reactions. The language constructs are inspired by regular expressions, process algebra (sequence, choice, loop, concurrency). The set of statements can be enriched with user defined operators. A notion of stochastic directive is also provided, in order to finely influence the selection of a particular class of scenarios.

Keywords: Reactive systems, synchronous programming, language design, test, simulation.

1 Introduction

The targeted domain is the one of reactive systems, where an execution is a (virtually) infinite sequence of input/output reactions. Examples of such systems are control/command in industrial process, embedded computing systems in transportation etc.

Testing reactive software raises specific problems. First of all, a single execution may require thousands of atomic reactions, and thus as many input vector values. It is almost impossible to write input test sequences by hand: they must be automatically generated according to some concise description. More specifically, the relevance of input values may depend on the behavior of the program itself: the program influences the environment which in turn influences the program. As a matter of fact, the environment behaves itself as a reactive system, whose environ-

[1] Emails: Pascal.Raymond@imag.fr, Yvan.Roux@imag.fr, Erwan.Jahier@imag.fr.
[2] URL: http://www-verimag.imag.fr

ment is the program under test. This feed-back aspect makes off-line test generation impossible: testing a reactive system requires to run it in a simulated environment.

All these remarks have lead to the idea of defining a language for describing random reactive systems (in the sense that they are not fully predictable). Since testing is the main goal, the programming style should be close to the intuitive notion of test scenarios, which means that the language is imperative and sequential.

Note that, even if testing is the main goal, such a language can be useful for other purposes. In particular, for early prototyping and simulation, where constrained random programs can implement missing modules.

For programming random systems, one solution is to use a classical (deterministic) language together with a random procedure. In some sense, non-determinism is achieved by relaxing deterministic behaviors. We have adopted an opposite solution where non-determinism is achieved by constraining chaotic behaviors; in other terms, the proposed language is mainly relational, not functional.

In the language Lutin, non predictable atomic reactions are expressed as input/output relations. Those atomic reactions are combined using statements like sequence, loop, choice or parallel composition. Since simulation (execution) is the goal, the language also provides stochastic constructs to express that some scenarios are more interesting/realistic than others.

Since the first version [1], the language has evolved with the aim of being a user-friendly, powerful programming language. The basic statements (inspired by regular expressions), have been completed with more sophisticated control structures (parallel composition, exceptions and traps) and a functional abstraction has been introduced in order to provide modularity and reusability.

This work is indeed related to synchronous programming languages [2]. Some constructs of the language (traps and parallel composition) are directly inspired by the imperative synchronous language Esterel [3], while the relational part (constraints) is inspired by the declarative language Lustre [4].

Related works are abundant in the domain of models for non-deterministic (or stochastic) concurrent systems: Input/Output automata [5], and their stochastic extension [6]; stochastic extension of process algebra [7,8]. There are also relations with concurrent constraint programming [9], particularly with works that adopt a synchronous approach of time and concurrency [10,11]. A general characteristic of these models is that they are defined to perform analysis of stochastic dynamic systems (e.g., model checking, probabilistic analysis). On the contrary, Lutin is designed with the aim of being a user-friendly programming language. On one hand, the language allows to concisely describe, and then execute a large class of scenarios. On the other hand, it is in general impossible to decide if a particular behavior can be generated and even less with which probability.

The paper is organized as follows: it starts with an informal presentation of the language. Then the operational semantics is formally defined in terms of constraints generator. Some important aspects, in particular constraints solving, are parameters of this formal semantics: they can be adapted to favor the efficiency or the expressive power. These aspects are presented in the implementation section. Finally, we conclude by giving some possible extensions of this work.

2 Overview of the language

2.1 Reactive, synchronous systems

The language is devoted to the description of reactive systems. Those systems have a cyclic behavior: they react to input values by producing output values and updating their internal state. We adopt the synchronous approach, which in this case simply means that the execution is viewed as a sequence of pairs "input values/output values".

Such a `system` is declared with its inputs and output variables; they are called the *support variables* of the system.

Example 2.1 We illustrate the language with a simple example that receives a Boolean (`c`) and a real (`t`) and produces a real `x`. The high-level specification is that `x` should get closer to `t` when `c` is true, or should tend to zero otherwise. The header of the program is:

```
system foo(c: bool; t: real) returns (x: real) = statement
```

The core of the program (*statement*) will be developed later.

During the execution, input values are provided by the environment: they are *uncontrollable variables*. The program reacts by producing output values: they are *controllable variables*.

2.2 Variables, reactions and traces

The core of the `system` is a statement describing a sequence of atomic reactions. In Lutin, a reaction is not deterministic: it does not define precisely the output values, but just states *constraints* on these values. For instance, the constraint `((x > 0.0) and (x < 10.0))` states that the current output should be some value comprised between 0 and 10.

Constraints may involve inputs, for instance: `((x > t - 2.0) and (x < t))`. In this case, during the execution, the actual value of `t` is substituted, and the resulting constraint is solved.

In order to express temporal constraints, *previous values* can be used: `pre` *id* denotes the value of the variable *id* at the previous reaction. For instance `(x > pre x)` states that `x` must increase in the current reaction. Like inputs, `pre` variables are uncontrollable: during the execution, their values are inherited from the past and cannot be changed: this is the *non-backtracking principle*.

Performing a reaction consists in producing, if it exists, a particular solution of the constraint. Such a solution may not exist:

Example 2.2 In the constraint:

```
(c and (x > 0.0) and (x < pre x + 10.0))
```

`c` (input) and `pre` x (past value) are uncontrollable, so, during the execution, it may appear that `c` is false and/or that `pre` x is less than -10.0. In those cases, the constraint is unsatisfiable: we say that the constraint *deadlocks*.

Local variables may be useful auxiliaries for expressing complex constraints. They can be declared within a program:

```
local ident : type in statement
```

A local variable behaves as a hidden output: it is controllable and must be produced as long as the execution remains in its scope.

2.3 Composing reactions

A constraint (Boolean expression) represents an atomic reaction: it is in some sense a snapshot of the current variable values. Scenarios are built by combining such snapshots with *temporal statements*. We use the term *trace* to design expressions made of temporal statements; constraints are implicitly traces of length 1.

The basic trace statements are inspired by regular expression: sequence (`fby`), unbounded loop (`loop`) and non-deterministic choice (`|`).

Because of this design choice, the notion of sequence differs from the one of Esterel, which is certainly the reference in control-flow oriented synchronous language[3]. In Esterel, the sequence (semicolon) is instantaneous, while the Lutin construct `fby` "takes" one instant of time.

Example 2.3 With those operators, we can propose a first version for our example. In this version, the output tends to 0 or `t` according to a first order filter. The non-determinism resides in the initial value, and also in the fact that the system is subject to failure and may miss the `c` command.

```
((-100.0 < x) and (x < 100.0)) fby -- initial constraint
loop {
     (c and (x = 0.9*(pre x) + 0.1*t)) -- x gets closer to t
|    ((x = 0.9*(pre x))                 -- x gets closer to 0
}
```

Initially, the value of `x` is (randomly) chosen between -100 and +100, then, forever, it may, tend to `t` or to 0.

Note that, inside the loop, the first constraint (x tends to t) is not satisfiable unless `c` is true, while the second is always satisfiable. If `c` is false, the first constraint *deadlocks*. In this case, the second branch (x gets closer to 0) is necessarily taken. If `c` is true, both branches are feasible: one is randomly selected, and the corresponding constraint is solved.

This illustrates an important principle of the language: the *reactivity principle* states that a program may only deadlock if all its possible behaviors deadlock.

2.4 Traces, termination and deadlocks

Because of non-determinism, a behavior has in general several possible first reactions (constraints). According to the reactivity principle, it deadlocks only if all those constraints are not satisfiable. If at least one reaction is satisfiable, it must "do something": we say that it is *startable*.

Termination, startability and deadlocks are important concepts of the language; here is a more precise definition of the basic statements according to those concepts:

- A constraint c, if it is satisfiable, generates a particular solution and terminates,

otherwise it deadlocks.
- *st1* `fby` *st2* executes *st1*, and, if and when it terminates, executes *st2*. If *t1* deadlocks, the whole statement deadlocks.
- `loop` *st*, if *st* is startable, behaves as *st* `fby loop` *st*, otherwise it terminates. Intuitively, the meaning is "loop as far as possible".
- `{`*st1* `|`... `|`*stn* `}` randomly chooses one of the *startable* statements from *st1*...*stn*. If none of them are startable, the whole statement deadlocks.
- The priority choice `{`*st1* `|>`... `|>`*stn* `}` behaves as *st1* if *st1* is startable, otherwise behaves as *st2* if *st2* is startable, etc. If none of them are startable, the whole statement deadlocks.
- `try` *st1* `do` *st2* catches any deadlock occurring *during the execution* of *st1* (not only at the first step). In case of deadlock, the control passes to *st2*.

2.5 Well-founded loops

Let's denote by ε the identity element for `fby` (i.e. the unique behavior such that b `fby` ε = ε `fby` b = b). Although this "empty" behavior is not provided by the language, it is helpful for illustrating a problem due to the loops.

As a matter of fact, the simplest way to define the loop is to state that "`loop` c" is equivalent to "c `fby loop` c `|>`ε", that is, try in priority to perform one iteration, and if it fails, stop. According to this definition, nested loops may generate infinite, instantaneous loops, as shown in the following example:

Example 2.4 `loop {loop` *c*`}`

Performing an iteration of the outer loop consists in executing the inner loop `loop` *c*. If *c* is not currently satisfiable, `loop` *c* terminates immediately, and thus, the iteration is actually "empty": it generates no reaction. However, since it is not a deadlock, this strange behavior is considered by the outer loop as a normal iteration. As a consequence, another iteration is performed, which is also empty, and so on: the outer loop keeps the control forever but does nothing.

One solution is to state that such programs are incorrect. Statically checking whether a program will infinitely loop or not is impossible. Some over-approximation is necessary, which will reject all the incorrect programs, but also lots of correct ones. For instance, a program as simple as: "`loop` { {`loop` a} `fby` {`loop` b } }" will certainly be rejected as potentially incorrect.

We think that such a solution is far to restrictive and tedious for the user, and we prefer to slightly modify the semantics of the loop. The solution retained is to introduce the *well-founded loop principle*: a loop statement may stop or continue, but if it continues it must do something. In other terms, empty iterations are forbidden.

The simplest way to explain this principle is to introduce an auxiliary operator $st\backslash_\varepsilon$: if *st* terminates immediately, $st\backslash_\varepsilon$ deadlocks, otherwise it behaves as *st*. The correct definition of `loop` *st* follows:
- if $st\backslash_\varepsilon$ is startable, behaves as $st\backslash_\varepsilon$ `fby loop` *st*,
- otherwise terminates.

2.6 Influencing non-determinism

When executing a non-deterministic statement, the problem of which choice should be preferred arises. The default is that, if k of the n choices are startable, each of them is chosen with a probability $1/k$.

In order to influence this choice, the language provides a mechanism of *relative weights*:

```
{st1 weight w1 |...  |stn weight wn }
```

Weights may be integer constants, or, more generally, *uncontrollable integer expressions*. In other terms, the environment and the past may influence the probabilities.

Example 2.5 In a first version (example 2.3), our example system may ignore the command `c` with a probability 1/2. This case can be made less probable by using weights (when omitted, a weight is implicitly 1):

```
loop {
     (c and (x = 0.9*(pre x) + 0.1*t)) weight 9
|    ((x = 0.9*(pre x))
}
```

In this new version, a true occurrence of `c` is missed with the probability 1/10.

Note that weights are not commands, but rather directives. Even with a big weight, a non startable branch has a null probability to be chosen, which is the case in the example when `c` is false.

2.7 Random loops

We want to define some loop structure where the number of iterations is not fully determined by deadlocks. Such a construct can be based on weighted choices, since a loop is nothing but a binary choice between stopping and continuing. However, we found it more natural to define it in terms of expected number of iterations. Two loop "profiles" are provided:

- `loop[`*min*`,`*max*`]`: the number of iterations should be between the constants *min* and *max*
- `loop~`*av*`:`*sd*: the average number of iteration should be *av*, with a standard deviation *sd*.

Note that random loops, just like other non-deterministic choices, follow the *reactivity principle*: depending on deadlocks, looping may sometimes be required or impossible. As a consequence, during an execution, the actual number of iterations may significantly differ from the "expected" one (see §3,§4.2).

Moreover, just like the basic `loop`, they follow the *well-founded loop principle*, which means that, even if the core contains nested loops, it is impossible to perform "empty" iterations.

2.8 Parallel composition

The parallel composition of Lutin is synchronous: each branch produces, at the same time, its local constraint. The global reaction must satisfy the conjunction of all

those local constraints. This approach is similar to the one of temporal concurrent constraint programming [11].

The termination of the concurrent execution is directly inspired by the language Esterel. During the execution:

- if one or more branches abort (deadlock), the whole statement aborts,
- otherwise, the parallel composition terminates if and when all the branches have terminated.

The concrete syntax may seem strange since it suggests a non-commutative operator, this choice is explained in the next section.

`{`*st1* `&>`... `&>`*stn* `}`

2.9 Parallel composition versus stochastic directives

It is impossible to define a parallel composition which is fair according to the stochastic directives, as shown in the following example.

Example 2.6 Consider the statement:
`{ {`X `weight` 1000 `|` Y `} &> {`A `weight` 1000 `|` B `} }`
where X, A, $X \wedge B$, $A \wedge Y$ are all startable, but not $X \wedge A$.
The priority can be given:

- to $X \wedge B$, but it does not respect the stochastic directive of the second branch,
- to $A \wedge Y$, but it does not respect the stochastic directive of the first branch.

In order to solve the problem, the stochastic directives are not treated in parallel, but in *sequence*, from left to right:

- the first branch "plays" first, according its local stochastic directives,
- the second one makes its choice, according to what has been chosen by the first one etc.

In the example, the priority is then given to $X \wedge B$.

Note that the concrete syntax (`&>`) has been chosen to reflect the fact that the operation is not commutative: the treatment is parallel for the constraints (conjunction), but sequential for stochastic directives (weights).

2.10 Exceptions

User-defined exceptions are mainly a means for by-passing the normal control flow. They are inspired by exceptions in classical languages (Ocaml) and also by the trap signals of Esterel.

Exceptions can be globally declared outside a system (`exception` *ident*) or locally within a statement, in which case the standard binding rules hold:

`exception` *ident* `in` *st*

An existing exception *ident* can be raised with the statement:

`raise` *ident*

and caught with the statement:

`catch` *ident* `in` *st1* `do` *st2*

If the exception is raised in *st1*, the control immediately passes to *st2*. The `do` part may be omitted, in which case the control passes in sequence.

2.11 Modularity

An important point is that the notion of `system` is not a sufficient modular abstraction. In some sense, systems are similar to main programs in classical languages: they are entry point for the execution but are not suitable for defining "pieces" of behaviors.

Data combinators.

A good modular abstraction would be one that allows to enrich the set of combinators. Allowing the definition of data combinators is achieved by providing a functional-like level in the language. For instance, one can program the useful "within an interval" constraint:

```
let within(x, min, max : real) : bool =
          (x >= min) and (x <= max)
```

Once defined, this combinator can be instantiated, for instance:
`within(a, 0.8, 0.9)` or `within(a + b, c - 1.0, c + 1.0)`
Note that such a combinator is definitively not a function in the sense of computer science: it actually computes nothing. It is rather a well typed *macro* defining how to build a Boolean expression with three real expressions.

Reference arguments.

Some combinators specifically require support variables as argument (input, output, local). This is the case for the operator `pre`, and, as a consequence, for any combinator that uses `pre`. This problem is solved by adding the flag `ref` to the type of such parameters.

Example 2.7 The following combinator defines the generic first order filter constraint. The parameter `y` must be a real support variable (`bool ref`) since its previous value is required. The other parameters can be any real expressions.

```
let fof (y: real ref; gain, x : real) : bool =
        (y = gain*(pre y) + (1.0-gain)*x)
```

Trace combinators.

User defined combinators are extended to temporal statements. A dedicated type `trace` is introduced, which is associated to any statement expressions, and by extension, to the result and the parameters of behavior combinators.

Example 2.8 The following combinator is a binary parallel composition where the termination is enforced when the second argument terminates:

```
let as_long_as(X, Y : trace) :  trace =
    exception Stop in
    catch Stop in {
          X &> {Y fby raise Stop}
    }
```

Note that the type `trace` is generic: it denotes behaviors on any set of support variables.

Local combinators.

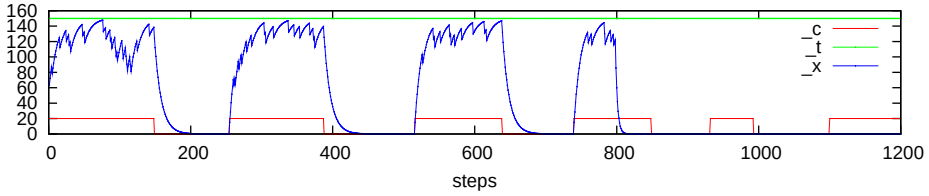
A macro can be declared within a statement, in which case the classical binding rules hold; in particular, it may have no parameter at all.

`let` *id* `(`*params*`):` *type* `=` *statement* `in` *statement*

Example 2.9 We can now write more elaborated scenarios for the system of example 2.3. In this new version, the system works almost properly for about 1000 reactions: if `c` is true, `x` tends to `t` 9 times out of 10, otherwise it tends to 0. During this phase, the gain for the filters (`a`) randomly changes each 30 to 40 reactions. At last, the system breaks down and `x` quickly tends to 0.

```
system foo(c:  bool; t:  real) returns (x:  real) =
   within(x, -100.0, 100.0) fby
   local a:  real in
   let gen_gain() :  trace = loop {
      within(a, 0.8, 0.9) fby loop[30,40] (a = pre a)
   } in
   as_long_as (
      gen_gain(),
      loop~1000:100 {
         (c and fof(x, a, t)) weight 9
      |  fof(x, a, 0.0)
      }
   } fby
   loop fof(x, 0.7, 0.0)
```

The following timing diagram shows an execution of this program where the input `t` is constant (150), and the command `c` toggles each about 50 steps.



3 Operational semantics

3.1 Abstract syntax

For the sake of simplicity, the semantics is given on the flat language (user defined macros are inlined). We use the following abstract syntax, where the intuitive meaning of each construct is given as a comment:

$$
\begin{aligned}
t ::= {} & c \ \textit{(constraint)} \mid \varepsilon \ \textit{(empty behavior)} \mid t \setminus \varepsilon \ \textit{(empty filter)} \mid t \cdot t \ \textit{(sequence)} \\
\mid {} & t^* \ \textit{(priority loop)} \mid t_k^{(\omega_c,\omega_s)} \ \textit{(random loop)} \mid \uparrow^x \ \textit{(raise)} \mid [t \overset{x}{\hookrightarrow} t'] \ \textit{(catch)} \\
\mid {} & \succ_{i=1}^{n} \ t_i \ \textit{(priority)} \mid |_{i=1}^{n} \ t_i/w_i \ \textit{(choice)} \mid \&_{i=1}^{n} \ t_i \ \textit{(parallel)}
\end{aligned}
$$

This abstract syntax slightly differs from the concrete one on the following points:

- the empty behavior (ε) and the empty behavior filter ($t \setminus \varepsilon$) are internal constructs that will ease the definition of the semantics,
- random loops are *normalized* by making explicit their weight functions:
  - the stop function ω_s takes the number of iteration already performed and returns the relative weight of the "stop" choice,
  - the continue function ω_c takes the number of iteration already performed and returns the relative weight of the "continue" choice.

  These functions are completely determined by the loop profile in the concrete program (interval or average, together with the corresponding static arguments). See §4.2 for a precise definition of these weight functions.
- the number of already performed iterations (k) is syntactically attached to the loop; this is convenient to define the semantics in terms of rewriting (in the initial program, this number is obviously set to 0).

Definition 3.1 $\mathcal{T}$ denotes the set of trace expressions, and $\mathcal{C}$ the set of constraints.

3.2 The execution environment

The execution takes place within an environment which stores the variable values (inputs and memories). Constraint resolution, weight evaluation and random selection are also performed by the environment. We keep this environment abstract. As a matter of fact, resolution capabilities and (pseudo)random generation may vary form one implementation to another, and they are not part of the reference semantics.

The semantics is given in term of constraints generator. In order to generate constraints, the environment should provide the following procedures:

Satisfiability. The predicate $e \models c$ is true iff the constraint c is satisfiable in the environment e.

Priority sort. Executing choices first requires to evaluate the weights in the environment. This is possible because weights may dynamically depends on uncontrollable variables (memories, inputs), but not on controllable variables (outputs, locals). Some weights may be evaluated to 0, in which case the corresponding choice is forbidden. Then a random selection is made, according to the actual weights, to determine a total order between the choices.

For instance, consider the following list of pairs (trace/weight), where x and y are uncontrollable variables:

$$(t_1/x+y), (t_2/1), (t_3/y), (t_4/2)$$

In a environment where $x = 3$ and $y = 0$, weights are evaluated to:

$$(t_1/3), (t_2/1), (t_3/0), (t_4/2)$$

The choice t_3 is erased, and the remaining choices are randomly sorted according to their weights. The resulting (total) order may be:

- t_1, t_2, t_4 with a probability $3/6 \times 1/3 = 1/6$
- t_1, t_4, t_2 with a probability $3/6 \times 2/3 = 1/3$
- t_4, t_1, t_2 with a probability $2/6 \times 3/4 = 1/4$
- etc.

All these treatments are "hidden" within the function $Sort_e$ which takes a list of pairs (choice/weights) and returns an ordered choices list.

3.3 The step function

An execution step is performed by the function $Step(e, t)$, taking an environment e and a trace expression t. It returns an *action* which is either:

- a transition $\xrightarrow{c} n$, which means that t produces a *satisfiable* constraint c and rewrite itself in the (next) trace n,
- a termination $\updownarrow^x$, where x is a termination flag which is either ε (normal termination), δ (deadlock) or some user-defined exception.

Definition 3.2 $\mathcal{A}$ denotes the set of actions, and $\mathcal{X}$ denotes the set of termination flags.

3.4 The recursive step function

The run function is defined via a recursive function $\mathcal{S}_e(t, g, s)$ where the parameters g and s are continuation functions returning actions.

- $g : \mathcal{C} \times \mathcal{T} \mapsto \mathcal{A}$ is the *goto* function, defining how a local transition should be treated according to the calling context.
- $s : \mathcal{X} \mapsto \mathcal{A}$ is the *stop* function, defining how a local termination should be treated according to the calling context.

At the top-level, $\mathcal{S}_e$ is called with the trivial continuations:

$$Step(e, t) = \mathcal{S}_e(t,\ \ g, s) \quad \text{with} \quad g(c, v) = \xrightarrow{c} v \quad \text{and} \quad s(x) = \updownarrow^x$$

Basic traces. The empty behavior raises the termination flag in the current context. A raise statement terminates with the corresponding flag. At last, a constraint generates a goto or raises a deadlock, depending on its satisfiability.

$$\mathcal{S}_e(\varepsilon, g, s) = s(\varepsilon)$$
$$\mathcal{S}_e(\updownarrow^x, g, s) = s(x)$$
$$\mathcal{S}_e(c, g, s) = \text{if } e \models c \text{ then } g(c, \varepsilon) \text{ else } s(\delta)$$

Sequence. The rule is straightforward:

$$\begin{aligned}\mathcal{S}_e(t \cdot t', g, s) &= \mathcal{S}_e(t, g', s') \qquad \text{where:}\\ g'(c, n) &= g(c, n \cdot t')\\ s'(x) &= \text{if } x = \varepsilon \text{ then } \mathcal{S}_e(t', g, s) \text{ else } s(x)\end{aligned}$$

Priority choice. We only give the definition of the binary choice, since the operator is right-associative. This rule formalizes the reactivity principle: all possibilities in t must have deadlock before t' is taken into account.

$$\mathcal{S}_e(t \succ t', g, s) = \text{if } r \neq \Updownarrow^{\delta} \text{ then } r \text{ else } \mathcal{S}_e(t', g, s) \quad \text{where} \quad r = \mathcal{S}_e(t, g, s)$$

Empty filter and priority loop. The empty filter intercepts the termination of t and replaces it by a deadlock:

$$\mathcal{S}_e(t \setminus \varepsilon, g, s) = \mathcal{S}_e(t, g, s') \quad \text{where} \quad s'(x) = \text{if } x = \varepsilon \text{ then } s(\delta) \text{ else } s(x)$$

The semantics of the loop is then defined according to the equivalence:

$$t^* \Leftrightarrow (t \setminus \varepsilon) \cdot t^* \succ \varepsilon$$

Catch. This case covers the operators `try` ($z = \delta$) and `catch` (z is an exception):

$$\begin{aligned}\mathcal{S}_e([t \overset{z}{\hookrightarrow} t'], g, s) &= \mathcal{S}_e(t, g', s') \qquad \text{where:}\\ g'(c, n) &= g(c, [n \overset{z}{\hookrightarrow} t'])\\ s'(x) &= \text{if } x = z \text{ then } \mathcal{S}_e(t', g, s) \text{ else } s(x)\end{aligned}$$

Parallel composition. We only give the definition of the binary case, since the operator is right-associative.

$$\begin{aligned}\mathcal{S}_e(t \,\&\, t', g, s) &= \mathcal{S}_e(t, g', s') \qquad \text{where:}\\ s'(x) &= \text{if } x = \varepsilon \text{ then } \mathcal{S}_e(t', g, s) \text{ else } s(x)\\ g'(c, n) &= \mathcal{S}_e(t', g'', s'')\\ s''(x) &= \text{if } x = \varepsilon \text{ then } g(c, n) \text{ else } s(x)\\ g''(c', n') &= g(c \wedge c', n \,\&\, n')\end{aligned}$$

Weighted choice. The evaluation of the weights, and the (random) total ordering of the branches, are both performed by the function $Sort_e$ (cf. §3.2).

$$\begin{aligned}\text{if } Sort_e(t_i/w_i) = \emptyset\text{:}\quad & \mathcal{S}_e(|_{i=1}^{n}\ t_i/w_i, g, s) = s(\delta)\\ \text{otherwise:}\quad & \mathcal{S}_e(|_{i=1}^{n}\ t_i/w_i, g, s) = \mathcal{S}_e(\succ Sort_e(t_1/w_1, \cdots, t_n/w_n), g, s)\end{aligned}$$

Random loop. The semantics is defined according to the equivalence:

$$t_i^{(\omega_c, \omega_s)} \quad \Leftrightarrow \quad (t \setminus \varepsilon) \cdot t_{i+1}^{(\omega_c, \omega_s)}/\omega_c(i) \quad | \quad \varepsilon/\omega_s(i)$$

3.5 A complete execution

Solving a constraint. The main role of the environment is to store the values of uncontrollable variable: it is a pair of stores "(past values, input values)". For such an environment $e = (\rho, \iota)$, and a satisfiable constraint c, we suppose given a procedure able to produce a particular solution of c: $Solve_{\rho,\iota}(c) = \gamma$ (where γ is a store of controllable variables). We keep this *Solve* function abstract, since it may vary from one implementation to another (see §4).

Execution algorithm. A complete run is defined according to:

- a given sequence of input stores $\iota_0, \iota_1, \cdots, \iota_n$,
- an initial (main) trace t_0,
- an initial previous store (values of `pre` variables) ρ_0

It produces a sequence of (controllable variables) stores $\gamma_1, \gamma_2, ..., \gamma_k$, where $k \leq n$. For defining this output sequence, we use intermediate sequences of traces $(t_1, \cdots, t_{k+1})$, previous stores $(\rho_1, \cdots, \rho_k)$, environments $(e_0, \cdots, e_k)$, and constraints $(c_0, \cdots, c_k)$. The relation between those sequences are listed below, for all step $j = 0 \cdots k$:

- the current environment is made of previous and input values: $e_j = (\rho_j, \iota_j)$
- the current trace makes a transition: $e_j : t_j \xrightarrow{c_j} t_{j+1}$
- a solution of the constraint is elected: $\gamma_j = Solve_{e_j}(c_j)$
- the previous store for the next step is the union of current inputs/outputs: $\rho_{j+1} = (\iota_j \oplus \gamma_j)$

At the end, we have:

- either $k = n$, which means that the execution has run to completion,
- or $(\rho_{k+1}, \iota_{k+1}) : t_{k+1} \Updownarrow^x$ which means that it has been aborted.

4 Implementation

A prototype has been developed, which implements the operational semantics presented in the previous section. This tool can:

- interpret/simulate Lutin programs in a file-to-file (or pipe-to-pipe) manner. This tool serves for simulation/prototyping: several Lutin simulation sessions can be combined with other reactive process in order to animate a complex system.
- compile Lutin programs into the internal format of the testing tool Lurette. This format, called Lucky, is based on flat, explicit automata [12]. In this case, Lutin serves as a high level language for designing test scenarios.

4.1 Notes on constraint solvers

The core semantics only defines how constraints are generated, not how they are solved. This choice is motivated by the fact that there is no "ideal" solver.

A required characteristic of such a solver is that it must provide a constructive, complete decision procedure: methods that can fail and/or that are not able to exhibit a particular solution are clearly not suitable. Basically, a constraint solver should provide:

- a syntactic analyzer for checking if the constraints are supported by the solver (e.g. linear arithmetics); this is necessary because the language syntax allows to write arbitrary constraints,
- a decision procedure for the class of constraints accepted by the checker,
- a precise definition of the election procedure which selects a particular solution (e.g. in terms of fairness).

Even with those restriction, there is no obvious best solver:

- it may be efficient but limited in terms of capabilities,
- it may be powerful, but likely to be very costly in terms of time and memory.

The idea is that the user should choose between several solvers (or several options of a same solver) the one which best fits his needs.

Actually, we use the solver that have been developed for the testing tool Lurette [13,14]. This solver is quite powerful, since it covers Boolean algebra and linear arithmetics. Concretely, constraints are solved by building a canonical representation which mixes Binary Decision Diagrams and convex polyhedra.

Because it is very powerful, this method is also costly. However the solver benefits from several years of experimentation and optimizations (partitioning, switch form polyhedra to intervals whenever it is possible).

The election of a particular solution is also quite sophisticated:

- The basic rule is to ensure some fairness between the solutions. This is achieved by simulating a uniform choice among the solutions domain. Since uniform selection within a polyhedron is a complex problem, several options are available ranging from a simple, rough approximation to a very accurate, and thus costly one.
- The solver can also be parameterized to select some class of interesting solutions (e.g. limit values corresponding to the polyhedra vertices).

4.2 Notes on predefined loop profiles

In the operational semantics, loops with iteration profile are translated into binary weighted choices. Those weights are dynamic: they depend on the number of (already) performed iterations k.

Interval loops. For the "interval" profile, those weights functions are formally defined, and thus, they could take place in the reference semantics of the language. For a given pair of integer (min, max), such that $0 \leq min \leq max$, and a number k of already performed iterations, we have:

- if $k < min$ then $\omega_s(k) = 0$ and $\omega_c(k) = 1$ (loop is mandatory),
- if $k \geq max$ then $\omega_s(k) = 1$ and $\omega_c(k) = 0$ (stop is mandatory),
- if $min \leq k < max$ then $\omega_s(k) = 1$ and $\omega_c(k) = 1 + max - k$

Average loops. There is no obvious solution for implementing the "average" profile in terms of weights. A more or less sophisticated (and accurate) solution should be retained, depending on the expected precision.

In the actual implementation, for an average value av and a standard variation sv, we use a relatively simple approximation:

- First of all, the underlying discrete repartition law is approximated by a continuous (Gaussian) law. As a consequence, the result will not be accurate if av is too close to 0, and/or if st is too big comparing to av. Concretely we must have $10 < 4 * sv < av$.
- The Gaussian repartition, for which it is well known that there is no algebraic form, is itself approximated by using an interpolation table (512 samples with a fixed precision of 4 digits).

5 Conclusion

We propose a language for describing constrained-random reactive system. Its first purpose is to describe test scenarios, but it may also be useful for prototyping and simulation.

We have developed a compiler/interpreter which strictly implements the operational semantics presented here. Thanks to this tool, the language is integrated into the framework of the tool Lurette, where it is used to describe test scenarios. Further works concerns the integration of the language within a more general prototyping framework.

Other works concern the evolution of the language. We plan to introduce a notion of signal (i.e. event), which is useful for describing values that are not always available (this is related to the notion of clocks in synchronous languages). We also plan to allow the definition of (mutually) tail-recursive traces. Concretely, that means that a new programming style would be allowed, based on explicit concurrent, hierarchic automata.

References

[1] Raymond, P., Roux, Y.: Describing non-deterministic reactive systems by means of regular expressions. In: First Workshop on Synchronous Languages, Applications and Programming, SLAP'02, Grenoble (2002)

[2] Halbwachs, N.: Synchronous programming of reactive systems. Kluwer Academic Pub. (1993)

[3] Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152

[4] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language Lustre. Proceedings of the IEEE **79** (1991) 1305–1320

[5] Lynch, N.A., Tuttle, M.R.: An introduction to Input/Output automata. CWI-Quarterly **2** (1989) 219–246

[6] Wu, S.H., Smolka, S.A., Stark, E.W.: Composition and behaviors of probabilistic I/O automata. Theoretical Computer Science **176** (1997) 1–38

[7] Jonsson, B., Larsen, K., Yi, W.: Probabilistic extensions of process algebras (2001)

[8] Bernardo, M., Donatiello, L., Ciancarini, P.: Stochastic process algebra: From an algebraic formalism to an architectural description language. Lecture Notes in Computer Science **2459** (2002)

[9] Saraswat, V.A., ed.: Concurrent Constraint Programming. MIT Press (1993)

[10] Saraswat, V.A., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: LICS. (1994) 71–80

[11] Nielsen, M., Palamidessi, C., D.Valencia, F.: Temporal concurrent constraint programming: Denotation, logic and applications. Nord. J. Comput **9** (2002) 145–188

[12] Raymond, P., Jahier, E., Roux, Y.: Describing and executing random reactive systems. In: SEFM 2006, 4th IEEE International Conference on Software Engineering and Formal Methods, Pune, India (2006)

[13] Raymond, P., Weber, D., Nicollin, X., Halbwachs, N.: Automatic testing of reactive systems. In: 19th IEEE Real-Time Systems Symposium, Madrid, Spain (1998)

[14] Jahier, E., Raymond, P., Baufreton, P.: Case studies with lurette v2. In: Software Tools for Technology Transfer. Volume 8. (2006) 517–530

6. Modifying Contracts with Larissa Aspects

David Stauch

VERIMAG, Centre équation - 2, GIÈRES, France

Notes:

Modifying Contracts with Larissa Aspects

David Stauch

Verimag, Centre équation - 2, avenue de Vignate, 38610 GIÈRES — France

Abstract

This paper combines two successful techniques from software engineering, aspect-oriented programming and design-by-contract, and applies them in the context of reactive systems. For the aspect language Larissa and contracts expressed with synchronous observers, we show how to apply an aspect *asp* to a contract C and derive a new contract C', such that for any program P which fulfills C, P with *asp* fulfills C'. We validate the approach on a medium-sized example.

Keywords: Aspect-oriented programming, Design-by-contract, synchronous languages

1 Introduction

1.1 Synchronous Languages and Aspect-Oriented Programming

Aspect-oriented programming (AOP) offers facilities to a base language which aim at encapsulating *crosscutting concerns*. These are concerns that cannot be properly captured into a module by the decomposition offered by the base language. AOP languages express crosscutting concerns in *aspects*, and *weave* (i.e. compile) them in the program with an aspect weaver.

All the aspect extensions of existing languages (like AspectJ [7]) share two notions: pointcuts and advice. A *pointcut* describes, with a general property, the program points (called *join points*) where the aspect should intervene (e.g., all the methods of the class `X`, all the methods whose name contains `visit`, etc.). The *advice* specifies what has to be done at each join point (execute a piece of code before the normal code of the method, for instance).

Most existing aspect languages cannot be used in the context of reactive systems, because they lack the semantic properties needed for formal verification, and the programming languages used for reactive systems are often different from general-purpose programming languages. Therefore, we developed the aspect language Larissa [1] as an extension to the synchronous programming language Argos. Argos is a hierarchical automata language, based on Mealy machines. It seems a good candidate as a base language, as it is the simplest language with the parallel structure which we want to crosscut, and which is typical for synchronous languages. Larissa

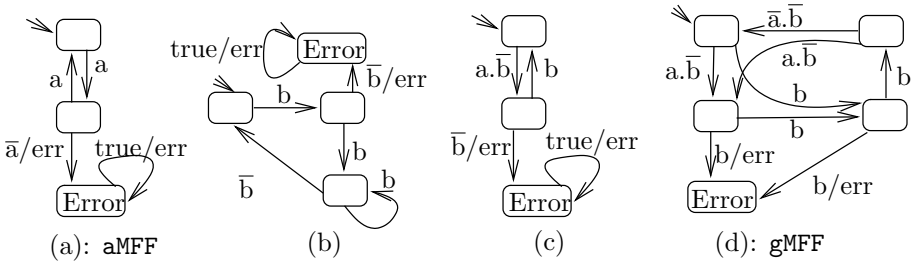


Fig. 1. The contract for the MFF. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions are expressed by "`triggering condition / outputs`", e.g. a transition labelled by "`a/b`" is triggered when `a` is true and emits `b`. Negation is expressed with an overbar, and conjunction with a dot. The observers accept all traces that do not lead to state Error.

has strong semantic properties, like the preservation of equivalence between programs. The approach presented in this paper strongly depends on these properties.

1.2 Synchronous Languages and Design-by-Contract

Design-by-Contract [14] is a design principle, originally introduced for object-oriented systems, where a method is specified by a contract. A contract is a specification in form of an implication between an assumption clause and a guarantee clause. A method fulfills its contract if after its execution, the guarantee holds if the assumption was true when the program was called.

Contracts have been adapted to reactive systems by [12]. Reactive systems constantly receive inputs from their environment, and emit outputs to it. Therefore, it seems natural to let assumptions restrict the inputs, and let guarantees ensure properties on the outputs. Additionally, what a program is allowed to do often depends to a large extent on previous occurrences of signals. A convenient way to express such temporal properties over input and output traces are observers. An observer [6] is a program that observes the inputs and the outputs of the program, without modifying its behavior, and computes a safety property (in the sense of safety/liveness properties as defined in [8]). Observers have a single output `err`, which is emitted to show that a trace is not accepted. They can be expressed in the same language as the program.

As an example, consider the following contract for a mono-stable flip-flop (MFF) with one input `a` and one output `b`. The contract is composed of an assumption, shown in Figure 1(a), which states that `a`'s always occur in pairs, and a guarantee consisting of two automata, shown in Figures 1(b) and (c), which are composed in parallel. The automaton in Figure 1(b) guarantees that a single `b` is never emitted, and the automaton in Figure 1(c) guarantees that when `a` occurs while no `b` is emitted, `b` is emitted in the next instant. The product of Figure 1(b) and Figure 1(c) is shown in Figure 1(d).

1.3 Combining Contracts and Aspects

AOP and design-by-contract can hardly be used concurrently. Obviously, the contract of a program is invalidated when an aspect is applied to it. Consider the AspectJ example in Figure 2. The pointcut (line 7) intercepts calls to method `m` (line 4), and the around advice (lines 9–11) modifies the intercepted calls by adding

```
1  class c{
2    /* @assume i < 10 */
3    /* @guarantee \result < 10 */
4    int m(int i){...}
5  }
6
7  pointcut pcm(int i) : call(int c.m(int)) && args(i);
8
9  int around(int i) : pcm(i){
10   return 1 + proceed(i+1);
11 }
```

Fig. 2. Example of a contract in presence of an AspectJ aspect.

`1` to the argument, then calling `m` through the proceed statement, and adding `1` to the result. This modifies both the initial assumption (line 2) and guarantee (line 3) of `m`. However, we can give a new contract for `m` in this case. To ensure that `m` is called according to its initial specification, the assumption must be changed to $i < 9$. On the other hand, the value returned by `m` may be higher than specified by the original guarantee in the presence of the aspect: we can only guarantee that $\backslash result < 11$, provided `m` does not call itself recursively.

Deriving such new contracts appears to be an interesting approach to combine AOP and contracts. However, this seems very difficult for contracts for Java programs and AspectJ, and it is not clear if meaningful contracts could be derived. In this paper, we present a way to derive new contracts for Argos programs and Larissa aspects. The idea is to apply an aspect *asp* to a contract C and obtain a new contract C', such that if P fulfills C, then $P \triangleleft asp$ fulfills C'.

The remainder of the paper is structured as follows: Section 2 defines Argos and Larissa; Section 3 describes how to derive a new contract from a contract and an aspect; Section 4 validates the approach on a larger example; Section 5 describes related work; and Section 6 concludes. An extended version of this paper can be found at [15].

2 Argos and Larissa

This section presents a restriction of the Argos language [13], and the Larissa extension [1]. Argos is defined as a set of operators on complete and deterministic input/output automata communicating via Boolean signals. The semantics of an Argos program is given as a trace semantics that is common to a wide variety of reactive languages.

2.1 Traces and Trace Semantics

Definition 2.1 [Traces] Let $\mathcal{I}$, $\mathcal{O}$ be sets of Boolean input and output variables representing signals from and to the environment. An *input trace*, *it*, is a function: $it : N \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$. An *output trace*, *ot*, is a function: $ot : N \longrightarrow [\mathcal{O} \longrightarrow \{\texttt{true}, \texttt{false}\}]$. We denote by $InputTraces$ (resp. $OutputTraces$) the set of

all input (resp. output) traces. A pair (it, ot) of input and output traces (i/o-traces for short) provides the valuations of every input and output at each instant $n \in N$. We denote by $it(n)[i]$ (resp. $ot(n)[o]$) the value of the input $i \in \mathcal{I}$ (resp. the output $o \in \mathcal{O}$) at the instant $n \in N$.

A set of pairs of i/o-traces $S = \{(it, ot) \mid it \in InputTraces \wedge ot \in OutputTraces\}$ is *deterministic* iff $\forall (it, ot), (it', ot') \in S \,.\, (it = it') \implies (ot = ot')$, and it is *complete* iff $\forall it \in InputTraces \,.\, \exists ot \in OutputTraces \,.\, (it, ot) \in S$.

A set of traces is a way to define the semantics of an Argos program P, given its inputs and outputs. From the above definitions, a program P is *deterministic* if from the same sequence of inputs it always computes the same sequence of outputs. It is *complete* whenever it allows every sequence of every eligible valuations of inputs to be computed.

2.2 Argos

The core of Argos is made of input/output automata, the synchronous product, and the encapsulation. The synchronous product allows to put automata in parallel which synchronize on their common inputs. The encapsulation is the operator that expresses the communication between automata with the synchronous broadcast: if two automata are put in parallel, they can communicate via a signal s. The semantics of an automaton is defined by a set of traces, and the semantics of the operators is given by translating expressions into flat automata.

Definition 2.2 [Automaton] An *automaton* $\mathcal{A}$ is a tuple $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where $\mathcal{Q}$ is the set of states, $s_{\text{init}} \in \mathcal{Q}$ is the initial state, $\mathcal{I}$ and $\mathcal{O}$ are the sets of Boolean input and output variables respectively, $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ is the set of transitions. $\mathcal{B}ool(\mathcal{I})$ denotes the set of Boolean formulas with variables in $\mathcal{I}$. For $t = (s, \ell, O, s') \in \mathcal{T}$, $s, s' \in \mathcal{Q}$ are the source and target states, $\ell \in \mathcal{B}ool(\mathcal{I})$ is the triggering condition of the transition, and $O \subseteq \mathcal{O}$ is the set of outputs emitted whenever the transition is triggered. Without loss of generality, we consider that automata only have complete monomials as input part of the transition labels.

The *semantics* of an automaton $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ is given in terms of a set of pairs of i/o-traces. This set is built using the following functions:

$$S\_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times N \longrightarrow \mathcal{Q}$$
$$O\_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times N \setminus \{0\} \longrightarrow 2^{\mathcal{O}}$$

$S\_step(s, it, n)$ is the state reached from state s after performing n steps with the input trace it; $O\_step(s, it, n)$ are the outputs emitted at step n:

$$\begin{array}{ll} n = 0: & S\_step_{\mathcal{A}}(s, it, n) = s \\ n > 0: & S\_step_{\mathcal{A}}(s, it, n) = s' \qquad O\_step_{\mathcal{A}}(s, it, n) = O \\ & \text{where } \exists (S\_step_{\mathcal{A}}(s, it, n-1), \ell, O, s') \in \mathcal{T} \\ & \wedge\ \ell \text{ has value true for } it(n-1)\,. \end{array}$$

We note $Traces(\mathcal{A})$ the set of all traces built following this scheme: $Traces(\mathcal{A})$ defines the semantics of $\mathcal{A}$. The automaton $\mathcal{A}$ is said to be *deterministic* (resp.

complete) iff its set of traces $\mathit{Traces}(\mathcal{A})$ is deterministic (resp. complete) (see Definition 2.1). Two automata $\mathcal{A}_1$, $\mathcal{A}_2$ are *trace-equivalent*, noted $\mathcal{A}_1 \sim \mathcal{A}_2$, iff $\mathit{Traces}(\mathcal{A}_1) = \mathit{Traces}(\mathcal{A}_2)$.

Definition 2.3 [Synchronous Product] Let $\mathcal{A}_1 = (\mathcal{Q}_1, s_{\text{init}1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1)$ and $\mathcal{A}_2 = (\mathcal{Q}_2, s_{\text{init}2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2)$ be automata. The *synchronous product* of $\mathcal{A}_1$ and $\mathcal{A}_2$ is the automaton $\mathcal{A}_1 \| \mathcal{A}_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{\text{init}1}, s_{\text{init}2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T})$ where $\mathcal{T}$ is defined by:

$$(s_1, \ell_1, O_1, s_1') \in \mathcal{T}_1 \wedge (s_2, \ell_2, O_2, s_2') \in \mathcal{T}_2 \iff (s_1 s_2, \ell_1 \wedge \ell_2, O_1 \cup O_2, s_1' s_2') \in \mathcal{T} \ .$$

The synchronous product of automata is both commutative and associative, and it is easy to show that it preserves both determinism and completeness.

Definition 2.4 [Encapsulation] Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ be a set of inputs and outputs of $\mathcal{A}$. The *encapsulation* of $\mathcal{A}$ w.r.t. Γ is the automaton $\mathcal{A} \setminus \Gamma = (\mathcal{Q}, s_{\text{init}}, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}')$ where $\mathcal{T}'$ is defined by:

$$(s, \ell, O, s') \in \mathcal{T} \wedge \ell^+ \cap \Gamma \subseteq O \wedge \ell^- \cap \Gamma \cap O = \emptyset \iff (s, \exists \Gamma \ . \ \ell, O \setminus \Gamma, s') \in \mathcal{T}'$$

ℓ^+ is the set of variables that appear as positive elements in the monomial ℓ (i.e. $\ell^+ = \{x \in \mathcal{I} \mid (x \wedge \ell) = \ell\}$). ℓ^- is the set of variables that appear as negative elements in the monomial l (i.e. $\ell^- = \{x \in \mathcal{I} \mid (\overline{x} \wedge \ell) = \ell\}$).

Intuitively, a transition $(s, \ell, O, s') \in \mathcal{T}$ is still present in the result of the encapsulation operation if its label satisfies a local criterion made of two parts: $\ell^+ \cap \Gamma \subseteq O$ means that a local variable which needs to be true has to be emitted by the same transition; $\ell^- \cap \Gamma \cap O = \emptyset$ means that a local variable that needs to be false should *not* be emitted in the transition.

If the label of a transition satisfies this criterion, then the names of the encapsulated variables are hidden, both in the input part and in the output part. This is expressed by $\exists \Gamma \ . \ \ell$ for the input part, and by $O \setminus \Gamma$ for the output part.

In general, the encapsulation operation does not preserve determinism nor completeness. This is related to the so-called "causality" problem intrinsic to synchronous languages (see, for instance [2]).

2.3 Contracts for Argos

An observer is an automaton which specifies a class of programms fulfilling a certain safety property. It is formally defined as follows.

Definition 2.5 [Observer] An observer is an automaton $(\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T)$ which observes an automaton that has inputs $\mathcal{I}$ and outputs $\mathcal{O}$. When an observer emits `err`, it will go to state Error and also emit `err` in the next instant. A program P is said to *obey* an observer *obs* (noted $P \models \mathit{obs}$) iff $P \| \mathit{obs} \setminus \mathcal{O}$ produces no trace which emits `err`.

Transitions leading to the Error state are called *Error transitions*.

A contract specifies a class of programs with two observers, an assumption and a guarantee. Definition 2.6 is an auxiliary definition, used to formally define contracts in Definition 2.7. ϵ denotes the empty trace.

Definition 2.6 [Trace Combination] Let $it : N \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ and $ot : N \longrightarrow [\mathcal{O} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ be traces, with $\mathcal{I} \cap \mathcal{O} = \emptyset$. Then, $it.ot : N \longrightarrow [\mathcal{I} \cup \mathcal{O} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ is a trace s.t. $\forall i \in \mathcal{I} \,.\, it.ot(n)(i) = it(n)(i) \wedge \forall o \in \mathcal{O} \,.\, it.ot(n)(o) = ot(n)(o)$.

Definition 2.7 [Contract] A contract over inputs $\mathcal{I}$ and outputs $\mathcal{O}$ is a tuple (A,G) of two observers over $\mathcal{I} \cup \mathcal{O}$, where A is the assumption and G is the guarantee. A program P fulfills a contract (A,G), written $P \models (A,G)$, iff

$$(it.ot, \epsilon) \in \mathit{Traces}(A) \wedge (it, ot) \in \mathit{Traces}(P) \Rightarrow (it.ot, \epsilon) \in \mathit{Traces}(G) \,.$$

Intuitively, a guarantee G should only restrict the outputs of a program and an assumption A should only restrict the inputs. We do not require this formally, but contracts which do not respect this constraint are of little use. Indeed, if G restricts the inputs more than A, it follows from Definition 2.7 that there exists no program P s.t. $P \models (A,G)$. Conversely, a program is usually placed in an environment E, s.t. $E \models A$. If A restricts the outputs, no such E exists, as the outputs are controlled by P.

2.4 Larissa

Argos operators are already powerful. However, there are cases in which they are not sufficient to modularize all concerns of a program: a small modifications of the global program's behavior may require that we modify all parallel components, in a way that is not expressible with the existing operators.

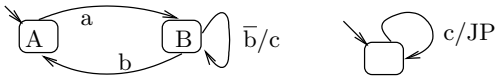
The goal of aspects being precisely to specify such cross-cutting modifications of a program, we proposed an aspect-oriented extension for Argos [1], which allows the modularization of a number of recurrent problems in reactive programs, like the reinitialization. This leads to the definition of a new operator (the aspect weaving operator), which preserves determinism and completeness of programs, as well as semantic equivalence between programs.

Similar to aspects in other languages, a Larissa aspect consists of a pointcut, which selects a set of join points, and an advice, which modifies these join points.

2.4.1 Join Point Selection

To preserver semantical equivalence, pointcuts in Larissa are not expressed in terms of the internal structure of the base program (as for instance state names), but refer to the observable behavior of the program only, i.e., its inputs and outputs.

Therefore, observers are well suited to express pointcuts. A pointcut is thus an observer which selects a set of *join point transitions* by emitting a single output *JP*, the *join point signal*. A transition T in a program P is selected as a join point transition when in the concurrent execution of P and the pointcut, *JP* is emitted when T is taken.



(a) base program (b) pointcut

Fig. 3. Example pointcut.

Technically, we perform a parallel product between the program and the pointcut and select those transitions in the product which emit *JP*. However, if we simply put a program P and an observer PC in parallel, P's outputs $\mathcal{O}$ will become synchronization signals between them, as they are also inputs of PC. They will be encapsulated, and are thus no longer emitted by the product. We avoid this problem by introducing a new output o' for each output o of P: o' will be used for the synchronization with PC, and o will still be visible as an output. First, we transform P into P' and PC into PC$'$, where $\forall o \in \mathcal{O}$, o is replaced by o'. Second, we duplicate each output of P by putting P in parallel with one single-state automaton per output o defined by: $dupl_o = (\{q\}, q, \{o'\}, \{o\}, \{(q, o', o, q)\})$. The complete product, where $\mathcal{O}$ is noted $\{o_1, ..., o_n\}$, is given by:

$$\boldsymbol{\mathcal{P}}(P, \mathrm{PC}) = (P' \| \mathrm{PC}' \| dupl_{o_1} \| \ ... \ \| dupl_{o_n}) \setminus \{o'_1, ..., o'_n\}$$

The join point transitions are those transitions of $\boldsymbol{\mathcal{P}}(P, \mathrm{PC})$ that emit *JP*.

Figure 3 illustrates the pointcut mechanism. The pointcut (b) specifies any transition which emits c: in base program (a), the loop transition in state B is selected as a join point transition.

2.4.2 Specifying the Advice

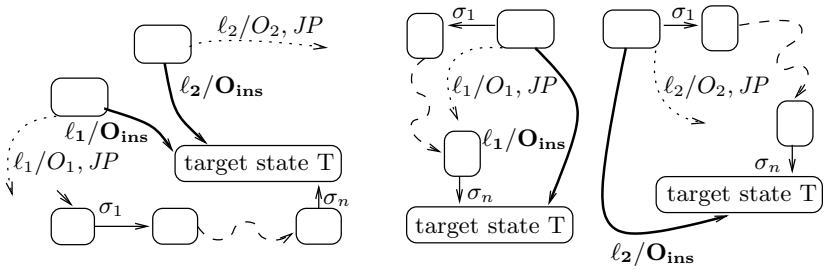
In aspect oriented languages, the advice expresses the modification applied to the base program. In Larissa, we define two types of advice: in the first type, an advice replaces the join point transitions with *advice transitions* pointing to an existing target states; in the second type, an advice introduces a Argos program between the source state of the join point transition and an existing target state. In both cases, target states have to be specified without referring explicitly to state names.

An advice *adv* has two ways of specifying the target state T among the existing states of the base program P. T is the state of P that would be reached by executing a finite input trace from either the initial state of P, *adv* is then called *toInit* advice, or from the source state of the join point transition, *adv* is then called *toCurrent* advice. As the base program is deterministic and complete, executing an input trace from any of its states defines exactly *one* state.

The advice weaving operator $\triangleleft\, adv$ weaves a piece of advice *adv* in a program. Definition 3.2 in the following section gives a formal definition for toInit advice. The remainder of this section describes the different kinds of advice informally.

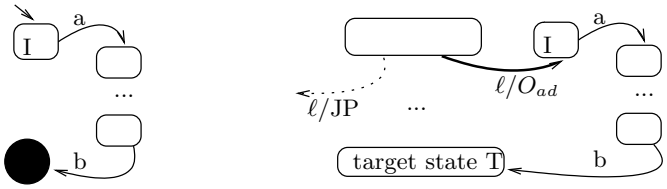
Advice Transitions

The first type of advice consists in replacing each join point transition with an advice transition. Once the target state is specified by a finite input trace $\sigma = \sigma_1 \dots \sigma_n$, the only missing information is the label of these new transitions. We do not change the input part of the label, so as to keep the woven automaton



(a) toInit aspect (b) toCurrent aspect

Fig. 4. Schematic toInit and toCurrent aspects. Advice transitions are in bold, join point transitions are dotted.



(a) inserted automaton A_{ins} (b) woven program

Fig. 5. Inserting an advice automaton.

deterministic and complete, but we replace the output part by some *advice outputs* O_{ad}. These are the same for every advice transition, and are thus specified in the aspect. Advice transitions are illustrated in Figure 4.

Advice Programs

It is sometimes not sufficient to modify single transitions, i.e. to jump to another location in the automaton in only one step. It may be necessary to execute arbitrary code when an aspect is activated. In these cases, we can insert an automaton between the join point and the target state.

Therefore, we use an *inserted automaton* A_{ins} that *terminates*. Since Argos has no built-in notion of termination, the programmer of the aspect has to identify a final state `F` (denoted by filled black circles in the figures).

We first specify a target state T as explained above. Then, for every T, a copy of the automaton A_{ins} is inserted, which means: 1) replace every join point transition J with target state T by a transition to the initial state I of this instance of A_{ins}. As for advice transitions, the input part of the label is unchanged and the output part is replaced by the *advice outputs* O_{ad}; 2) connect the transitions that went to the final state `F` in A_{ins} to T. Advice programs are illustrated in Figure 5.

2.4.3 Fully Specifying an Aspect

An aspect is given by the specification of its pointcut and its advice: asp = (PC, adv), where PC is the pointcut and adv is the advice. adv is a tuple which contains 1) the advice outputs O_{ad}; 2) the *type* of the target state specification (*toInit* or *toCurrent*); 3) the finite trace σ over the inputs of the program; and optionally, 4) P_{adv}, the advice program. Thus, advice can be a tuple $< O_{ad}$,*type*, $\sigma >$, or, with an advice program, a tuple $< O_{ad}$,*type*,σ, $P_{adv} >$, with $type \in \{toCurrent, toInit\}$. An aspect is woven into a program by first determining the join point transitions

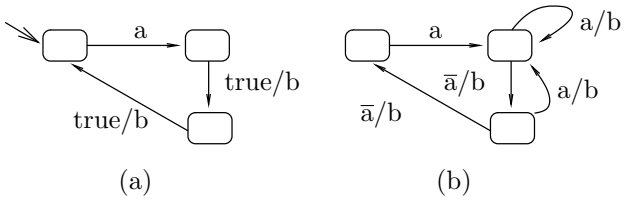


Fig. 6. A possible implementation of the MFF (a), with the retriggerable aspect applied to it (b).

and then weaving the advice.

Definition 2.8 [Aspect weaving] Let P be a program and $asp = (\mathrm{PC}, adv)$ an aspect for P. The weaving of asp on P is defined by

$$P \triangleleft asp = \boldsymbol{\mathcal{P}}(P, \mathrm{PC}) \triangleleft adv.$$

2.4.4 Example

Consider the MFF example from Section 1.2. We now want to make the MFF re-triggerable, meaning that if an `a` is emitted during several following instants, the MFF continues emitting `b`. We do this by applying the aspect `ret`= (PC, $< b, toInit, (a) >$) to the MFF, where PC =({S},S,{a,b},{JP}, {(S,a.b,JP,S)}) is a pointcut which selects all occurrences of `a.b` as join points. Figure 6(a) shows a sample implementation of the MFF, and Figure 6(b) shows the result of applying `ret` to it.

3 Weaving Aspects in Contracts

We want to apply an aspect asp not to a specific program, but to a class of programs defined by a contract C, and obtain a new class of programs, defined by a contract C', such that $P \models C \Rightarrow P \triangleleft asp \models C'$. To construct C', we simulate the effect that the aspect has on a program as far as possible on the assumption and the guarantee observers of C. However, an aspect cannot be applied directly to an observer, because the aspect has been written for a program with inputs $\mathcal{I}$ and outputs $\mathcal{O}$, whereas for the observer, $\mathcal{O}$ are also inputs.

Therefore, we transform the observers of the contract first into non-deterministic automata (NDA), which produce exactly those traces that the observer accepts. We then weave the aspects into the NDA, with a modified definition of the weaving operator. The woven NDA are then transformed back into observers. The obtained observers may still be non-deterministic, and are thus determinized.

Except for the aspect weaving, all of these steps are different for the assumption and the guarantee, as far as the Error transitions are concerned. This is because the assumption and the guarantee have different functions in a contract: the assumption states which part of the program is defined by the contract, and the guarantee gives properties that are always true for this part. Indeed, a contract (A,G) can be rewritten as (`true`,$A \Rightarrow G$). Thus, the assumption can be considered as a negated guarantee.

After weaving an aspect, the assumption must exclude the undefined part of *any* program which fulfills the contract. Therefore, it must reject a trace (by emitting

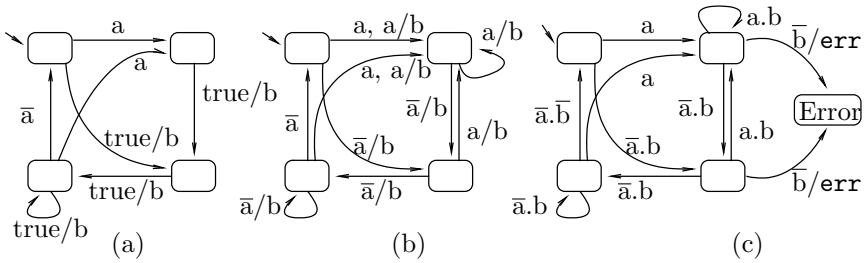


Fig. 7. a: ND_G(gMFF), b: ND_G(gMFF)$\triangleleft$ret, c: $OBS_G(ND_G$(gMFF)$\triangleleft$ret).

`err`) as soon as there exists a program for which it cannot predict the behavior. The guarantee, on the other hand, emits `err` only for traces which cannot be emitted by any program which fulfills the contract. Therefore, after weaving an aspect, the new guarantee may only emit `err` if it is sure that there exists no program that produces the trace.

3.1 Formal Definitions

This paragraph describes the weaving of aspects into contracts in detail, and illustrates it on our running example. First, Definition 3.1 defines the transformation of an observer into a NDA through two functions, one for guarantee observers and one for assumption observers.

Definition 3.1 [Observer to NDA transformation] Let $obs = (\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T)$ be an observer with an error state Error over inputs $\mathcal{I}$ and outputs $\mathcal{O}$, with $\mathcal{I} \cap \mathcal{O} = \emptyset$. $ND_G(obs) = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, T_{ND_G})$ defines a NDA, where T_{ND_G} is defined by $(s, \ell_\mathcal{I} \wedge \ell_\mathcal{O}, \emptyset, s') \in T \Rightarrow (s, \ell_\mathcal{I}, \ell^+_\mathcal{O}, s') \in T_{ND_G}$. $ND_A(obs) = (\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I}, \mathcal{O}, T_{ND_A})$ defines a NDA, where T_{ND_A} is defined by $(s, \ell_\mathcal{I} \wedge \ell_\mathcal{O}, o, s') \in T \Rightarrow (s, \ell_\mathcal{I}, \ell^+_\mathcal{O} \cup o, s') \in T_{ND_G}$.

Note that the transitions in *obs* which emit `err` (i.e. the Error transitions) have no corresponding transitions in $ND_G(obs)$. In the guarantee, these transitions correspond to input/output combinations which are never produced by the program and must not be considered by the aspect. As an example, consider the guarantee of the MFF (Figure 1(d)). Its transformation into a NDA is shown in Figure 7(a).

In the assumption, on the other hand, the Error transition correspond to inputs from the environment to which the program may react arbitrarily. If the aspect replaces these transitions in the assumption, they are also replaced in the program, and can thus be accepted from the environment by the woven program. Thus, error transitions are not removed in $ND_A(obs)$, so that the aspect weaving can modify them. The transformation of the assumption of the MFF (Figure 1(a)) is shown in Figure 8(a).

We can now apply an aspect to a NDA. However, a trace may lead to several states. Thus, for each join point transition, several advice transitions must be created, one for each target state. We only give a definition for toInit advice, but the extension to toCurrent advice and advice programs is straightforward.

Definition 3.2 [toInit weaving for NDA] Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $adv = (O_{adv}, \text{toInit}, \sigma)$ a piece of toInit advice, with $\sigma : [0, ..., \ell_\sigma] \longrightarrow$

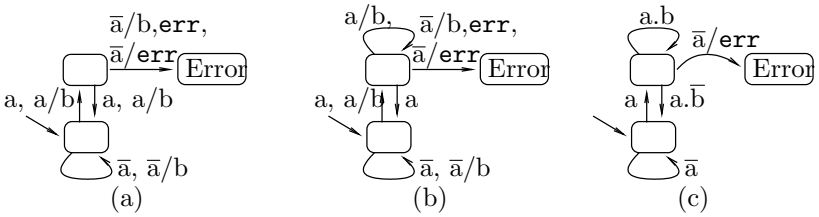


Fig. 8. a: ND_A(aMFF), b: ND_A(aMFF)$\triangleleft$ ret, c: $OBS_A(ND_A(\text{aMFF})\triangleleft \text{ret})$.

$[\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ a finite input trace of length $\ell_\sigma + 1$. Let $TARG = \{s | s = S\_step_{\mathcal{A}}(s_{\text{init}}, \sigma, \ell_\sigma)\}$ be the set of all states reachable with σ. The advice weaving operator $\triangleleft$, weaves adv into $\mathcal{A}$ and returns the automaton $\mathcal{A}\triangleleft\, adv = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O} \cup O_{adv}, \mathcal{T}')$, where $\mathcal{T}'$ is defined as follows:

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}' \tag{1}$$
$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \in O\big) \implies \forall targ \in TARG\,.\,(s, \ell, O_{adv}, targ) \in \mathcal{T}' \tag{2}$$

Transitions (1) are not join point transitions and are left unchanged. Transitions (2) are the join point transitions, their final state *targ* is specified by the finite input trace σ. $S\_step_{\mathcal{A}}$ (which has been naturally extended to finite input traces) executes the trace during ℓ_σ steps, from the initial state of $\mathcal{A}$. Figure 7(b) and Figure 8(b) show the NDAs from our example with the retriggerable aspect from Section 2.4.4 woven into them. For both NDAs, the trace leads to a single state, thus only one advice transition is introduced per join point transition.

Transforming a NDA back into an observer is different for assumptions and guarantees. In the assumption, we do not add additional error transitions, but only leave those already there. In the guarantee, we add transitions to the error state from every state where the automaton is not complete. This is correct, as these transitions correspond to traces that are never produced by any program.

Definition 3.3 [NDA to guarantee transformation] Let $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, T)$ be a NDA. $OBS_G(nd) = (\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T' \cup T'')$ defines an observer, where T' and T'' are defined by

$$(s, \ell, o, s') \in T \Rightarrow (s, \ell \wedge \ell_o \wedge \ell_{\overline{\mathcal{O} \setminus o}}, \emptyset, s') \in T' \tag{3}$$
$$\begin{aligned}&(s, \ell, \emptyset, s') \notin T' \wedge s \in \mathcal{Q} \wedge \ell \text{ is a complete monomial over } \mathcal{I} \cup \mathcal{O} \\ &\qquad \Rightarrow (s, \ell, \{\texttt{err}\}, \text{Error}) \in T''\end{aligned} \tag{4}$$

where $l_O = \bigwedge_{o \in O} o$ and $l_{\overline{O}} = \bigwedge_{o \in O} \overline{o}$ for a set O of variables.

Definition 3.4 [NDA to assumption transformation] Let $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O} \cup \{\texttt{err}\}, T)$ be a NDA. $OBS_A(nd) = (\mathcal{Q}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T')$ defines an observer, where T' is defined by

$$(s, \ell, o \cup e, s') \in T \wedge o \subseteq \mathcal{O} \wedge e \subseteq \{\texttt{err}\} \Rightarrow (s, \ell \wedge \ell_o \wedge \ell_{\overline{\mathcal{O} \setminus o}}, e, s') \in T'$$

Figure 7(c) and Figure 8(c) show the NDAs from our example transformed back into observers. As expected, the obtained guarantee in Figure 7(c) tells us that whenever the program receives an `a`, it emits `b`'s the two following instants. The

assumption, however, requires that if an `a` is emitted, it continues to be emitted until there is no `b`.

The resulting observer may not be deterministic. However, it can be made deterministic, as observers are acceptor automata. Determinization for guarantees and assumptions is different: a guarantee must only emit `err` for a trace σ if all programs fulfilling the contract never emit σ, and an assumption must emit `err` if there exists a program fulfilling the contract which is not defined for σ.

Existing determinization algorithms can be easily adapted to fulfill these requirements. We do not detail such algorithms here, but instead give conditions the determinization for assumptions and guarantees must fulfill. The new assumption and the new guarantee in the example are already deterministic, thus there is no need to determinize them.

Definition 3.5 [Assumption Determinization] Let M be a NDA with outputs $\{\texttt{err}\}$. $Det_A(M)$ is a deterministic automaton such that

$$(it, ot) \in \mathit{Traces}(\mathit{Det}_A(M)) \Leftrightarrow \\ (it, ot) \in \mathit{Traces}(M) \wedge \nexists ot' \,.\, ot'(n)[\texttt{err}] = \texttt{true} \wedge ot(n)[\texttt{err}] = \texttt{false} \,.$$

Definition 3.6 [Guarantee Determinization] Let M be a NDA with outputs $\{\texttt{err}\}$. $Det_G(M)$ is a deterministic automaton such that

$$(it, ot) \in \mathit{Traces}(\mathit{Det}_G(M)) \Leftrightarrow \\ (it, ot) \in \mathit{Traces}(M) \wedge \nexists ot' \,.\, ot'(n)[\texttt{err}] = \texttt{false} \wedge ot(n)[\texttt{err}] = \texttt{true} \,.$$

We can now state the following theorem. See [15] for a proof.

Theorem 3.7 *Let P be a program and let (A, G) be a contract. Then,*

$$P \models (A, G) \\ \Rightarrow P \triangleleft \mathit{asp} \models (\mathit{Det}_A(\mathit{OBS}_A(\mathit{ND}_A(A) \triangleleft \mathit{asp})), \mathit{Det}_G(\mathit{OBS}_G(\mathit{ND}_G(G) \triangleleft \mathit{asp})))$$

4 Example: The Tramway Door Controller

We implement and verify a larger example, taken from the Lustre tutorial [11], a controller of the door of a tramway. The door controller is responsible for opening the door when the tram stops and a passenger wants to leave the tram, and for closing the door when the tram wants to leave the station. Doors may also include a gateway, which can be extended to allow passengers in wheelchairs enter and leave the tram.

We implement the controller as an Argos program. We first develop a controller for a door without the gangway, and then add the gangway part with aspects. Figure 9 gives the in- and outputs of the controller with their specifications, and also the in- and outputs which are added by the gangway. The controller uses additional inputs, called Helper Signals, which are also shown in Figure 9. They are calculated from the original inputs, by a program given in [15].

It is important for the safety of the passengers that the doors are never open outside a station. We give a contract for the door controller, which focuses on this

Controller Inputs:		**Controller Outputs:**	
inStation	Tram is in station	doorOK	door is closed and ready to leave
leaving	Tram wants to leave station	openDoor	opens the door
doorOpen	the door is open	closeDoor	closes the door
doorClosed	the door is closed	beep	emits a warning sound
askForDoor	a passenger wants to leave the tram	setTimer	starts a timer
timer	the timer set by setTimer has run out		

Gangway Inputs:		**Gangway Outputs:**	
gwOut	the gangway is fully extended	extendGW	extends the gangway
gwIn	the gangway is fully retracted	retractGW	retracts the gangway
askForGW	a passenger wants to use the gangway		

Helper Signals Outputs:	
acceptReq	the passenger can ask for the door or the gw
doorReq	the passenger has asked for the door to open
gwReq	the passenger has asked for the gangway
depImm	the tramway wants to leave the station

Fig. 9. The interfaces of the controller and the gangway, and the helper signals.

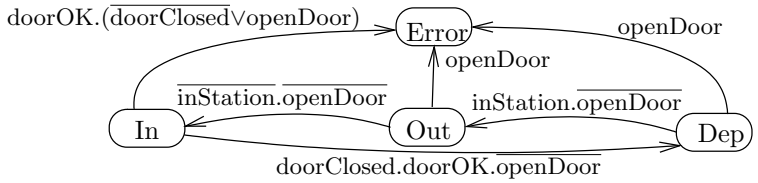


Fig. 10. The guarantee of the contract of the controller.

property. The guarantee of the contract is shown in Figure 10, it ensures that the controller emits doorOK only if the doors are closed, and openDoor only if the tram is in a station. The contract has also an assumption, which requires that the door behaves correctly (e.g., the door only opens if openDoor has been emitted). It is given in [15], along with an implementation of the controller.

To formally verify that a tram door is always closed outside a station, we develop a model that describes the possible behavior of the physical environment of the controller, i.e. the door and the tramway. These models are expressed as Argos observers, and are given in [15]. We then prove that the controller satisfies the contract, and that the contract in the environment never violates the safety property.

4.1 Adding The Gangway

Two aspects are used to add support for the gangway: one aspect that extends the gangway before the door is opened if a passenger has asked for it, and one aspect that retracts the gangway when the tram is about to leave, if it is extended.

The pointcut $\mathrm{PC}_{\mathrm{ext}}$ of the extension aspect selects all transitions where $openDoor.doorReq.doorClosed.\overline{gwOut}$ is true, and the pointcut $\mathrm{PC}_{\mathrm{ret}}$ of the retraction aspect selects all transitions where $doorOK.\overline{gwIn}$ is true.

Both aspects insert an automaton and return then to the initial state of the join point transitions. The inserted automata for the aspects are shown in Figure 11. The extension aspect is specified by $(\mathrm{PC}_{\mathrm{ext}}, < \{\}, toCurrent, (), I_{\mathrm{ext}} >)$, and the retraction aspect by $(\mathrm{PC}_{\mathrm{ret}}, < \{retractGW\}, toCurrent, (), I_{\mathrm{ret}} >)$.

We want to check that the new controller still verifies the safety property from

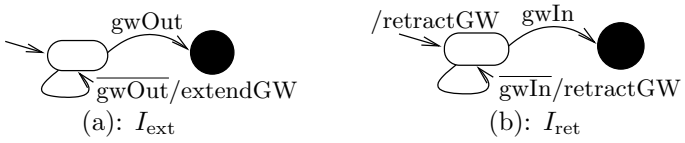


Fig. 11. Inserted automata for the extension (a) and the retraction (b) aspect.

above, and also verifies two new safety properties, which require that the gangway is always fully retracted while the tram is out of station, and that the gangway is never moved when the door is not closed. Therefore, we weave the aspects into the contract, and thus obtain a new contract that holds for controller with the aspects. Finally, we check then that the environment, to which we added a model of the gangway, satisfies the new assumption, and that the new guarantee satisfies the safety requirements in the environment.

An alternative to this modular approach is to verify directly that the sample controller with the aspects does not violate the given safety properties. One disadvantage of the alternative approach is that the woven controller may be much bigger than the woven contract. To illustrate this problem, we verified the safety properties using our implementation [9]. The source code of the door controller example is available at [10]. Verifying the woven program takes 11.0 seconds [1]. On the other hand, weaving the aspects into the guarantee of the controller contract and verifying against the environment takes 3.7 seconds [1], and verifying that the sample controller verifies the contract and verifying that the environment fulfills the assumption with the aspects takes < 0.5 seconds [1]. Thus, using this modular approach to verify the safety properties of the controller is significantly faster than verifying the complete program. Although the size of the woven controller is not prohibitive in this example, this indicates that larger programs can be verified using the modular approach.

5 Related Work

Goldman and Katz [5] modularly verify aspect-oriented programs using a LTL tableau representation of programs and aspects. As opposed to ours, their system can verify AspectJ aspects, as tools like Bandera [4] can extract suitable input models from Java programs. It is, however, limited to so-called *weakly invasive* aspects, which only return to states already reachable in the base program.

Clifton and Leavens [3] noted before us that aspects invalidate the specification of modules, and propose that either an aspect should not modify a program's contract, or that modules should explicitly state which aspects may be applied to them.

6 Conclusion

We proposed a way to show exactly how a Larissa aspect modifies the contract of a component to which it is applied. This allows us to calculate the effect of an aspect on a specification instead of only on a concrete program. This approach has several advantages. First, aspects can be checked against contracts even if the

[1] Experiments were conducted on an Intel Pentium 4 with 2.4GHz and 1 Gigabyte RAM.

final implementation is not yet available during development. Furthermore, if the base program is changed, the woven program must not be re-verified, as long as the new base program still fulfills the contract. Finally, woven programs can be verified modularly, which may allow for larger program to be verified, as indicates the example in Section 4.

We believe that the approach is exact in that it gives no more possible behaviors for the woven program than necessary. I.e., for a contract C and a trace $t \in \mathit{Traces}(C \triangleleft \mathit{asp})$, there exists a program P s.t. $P \models C$ and $t \in \mathit{Traces}(P \triangleleft \mathit{asp})$. This remains however to be proven. A more interesting direction for future work would be to derive contracts the other way round. Given a contract C and an aspect asp, can we automatically derive a contract C' such that $C' \triangleleft \mathit{asp} \models C$? Finally, the proposed approach works only because we have restricted Argos and Larissa to Boolean signals. It would be interesting to see if this approach can be extended to programs with valued signals or variables.

References

[1] K. Altisen, F. Maraninchi, and D. Stauch. Aspect-oriented programming for reactive systems: a proposal in the synchronous framework. *Science of Computer Programming, Special Issue on Foundations of Aspect-Oriented Programming*, 63(3):297–320, 2006.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programming*, 19(2):87–152, 1992.

[3] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report 03-15, Iowa State University, Department of Computer Science, Dec. 2003.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, June 2000.

[5] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect-Oriented Languages (FOAL)*, Mar. 2006.

[6] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology, AMAST'93*, June 1993.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–353, 2001.

[8] L. Lamport. Proving the correctness of multiprocess programs. *ACM Trans. Prog. Lang. Syst.*, SE-3(2):125–143, 1977.

[9] Compiler for Larissa. `http://www-verimag.imag.fr/~stauch/ArgosCompiler/`.

[10] Argos source code for the tram example. `http://www-verimag.imag.fr/~stauch/ArgosCompiler/contracts.html`.

[11] The Lustre tutorial. `http://www-verimag.imag.fr/~raymond/edu/tp.ps.gz`.

[12] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, Aug. 2004.

[13] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001.

[14] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[15] D. Stauch. Modifying contracts with Larissa aspects. Technical Report TR-2006-10, Verimag, Dec. 2006.

7. Lustre as a System Modeling Language: Lussensor, a Case-Study with Sensor Networks

Florence Maraninchi, Ludovic Samper[1],
Kevin Baradon, and Antoine Vasseur[2]

1 VERIMAG and INPGrenoble/ENSIMAG, VERIMAG and France Telecom R&D Grenoble, France
2 INPGrenoble/Telecom, Grenoble, France

Notes:

Lustre as a System Modeling Language: Lussensor, a Case-Study with Sensor Networks

Florence Maraninchi, Ludovic Samper[1,2,3]

VERIMAG and INPGrenoble/ENSIMAG, VERIMAG and France Telecom R&D Grenoble, France

Kevin Baradon, Antoine Vasseur[1]

INPGrenoble/Telecom, Grenoble, France

Abstract

We describe how we use Lustre to build global and accurate executable models of energy consumption in sensor networks, intended to be used for both simulations and formal validation. One of the key ideas is to build a component-based global model, in such a way that various abstractions of the same model can be derived by unplugging a component and plugging a more abstract (or more detailed) one. This ability to play with various abstractions that can be formally compared with one another is essential for a virtual prototyping approach connected to formal validation tools. We comment on the properties of Lustre and its development environment that make this approach feasible.

Keywords: sensor networks, formal modeling, simulation, energy consumption

1 Introduction

1.1 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are distributed computer systems composed of a large number of small sensor nodes. There are many potential application areas [3]. The nodes have three main tasks: sensing their environment, processing the data and communicating with the other nodes. All the nodes are identical, except one or several *sink* nodes. A sink monitors the network. It collects the data and sends requests to the sensor nodes. Nodes do not have enough power to reach the sink directly with their radios, therefore communications are performed in a multi-hop

[1] This work has been partially supported by the French RNRT project ARESA

[2] Email:Florence.Maraninchi@imag.fr Florence.Maraninchi@imag.fr

[3] Email:Ludovic.Samper@imag.fr Ludovic.Samper@imag.fr

way. A wireless sensor network has no infrastructure, i.e., nodes do not have any a priori knowledge about the rest of the network. Hence A WSN must be able to self-organize. Nodes cooperate for this self-organization and also along the whole life of the network to achieve the requested service. Finally, routing protocols have to be tailored for WSNs (see, for instance [2]), in order to take new constraints into account, among which: in WSNs, data flow from a particular region to the sink(s) whereas, in traditional networks, any node may need to communicate and establish a route with any other; it is usually not possible in WSNs to rely on a global addressing mechanism; in the vicinity of a phenomenon, several nodes will sense the same values and thus the same data may be generated; etc.

1.2 Virtual Prototypes of Sensor Networks

A sensor network may be considered as a whole, as a new kind of computer system dedicated to one particular application. It is an *embedded* system, reacting to the stimuli of some physical environment. It is also subject to the usual constraints of embedded system design: resources are scarce, and it is very difficult, if not impossible, to modify a sensor network's behavior once it has been deployed. Moreover, the sensors are usually powered by a battery that cannot be recharged. They should therefore have the lowest consumption possible to maximize the network lifetime.

One of the main challenges is to perform energy-aware design. The problem is difficult because all the elements of a sensor network have an influence on energy consumption: the hardware of a node, the sensors, the medium-access-control and routing protocols, the application itself, the initial self-organization phase, and even the physical environment that stimulates the sensors (see, for instance [23], where we showed that a precise modeling of the physical environment is compulsory for a realistic estimation of the energy consumption).

The design of an energy-"optimal" solution is probably out of reach because of all the interacting criteria. One has to build complete solutions and then to evaluate them. Since a sensor network includes dedicated hardware, it may be long and costly to build a complete solution before evaluating it.

For all these reasons, the usual approach is to build a *virtual prototype* of a sensor network, and then to perform simulations or mathematical analyzes in order to evaluate the energy consumption. This is the approach taken by people who design new protocols, and show their benefits using a network simulator. In all these approaches, a lot of *abstractions* are necessary, in order to build manageable models of very large systems (thousands of nodes). For instance, the energy consumption may be evaluated by counting packets, and associating a worst-case estimated energy with the transmission of one individual packet. In the section "related work" below, we review the main approaches for the virtual prototyping of sensor networks.

1.3 Contribution of the paper

In this paper, we describe our experiments in using a synchronous language, namely Lustre, to build a global and executable model of a sensor network, including all the elements that influence energy consumption: the details about the hardware of

a node, the code of the protocols and the application, an executable model of the physical environment that stimulates the sensors, and also a model of the physical medium in which the radio communication occurs. Lustre is an appropriate language for building such a detailed model, especially when it comes to describing the detailed energy consumption of the hardware and its relationship with time.

The second very important point is that Lustre allows to build a clean component-based model. This is crucial because complete models of sensor networks are huge, and it is always necessary to abstract them, even for simulation purposes only. If a global model is clearly structured into well-defined components, it means that one can hope to replace one component by a more abstract one, and get a new global model, more abstract then the original one. We show that Lustre provides such a modular-abstraction framework.

Finally, Lustre is connected to various validation tools, ranging from automatic test case generation to formal verification by means of model-checking or abstract interpretation techniques. This means the model we build can be directly given as input to these tools.

A first experiment in writing global models of sensor networks has given Glonemo (for GLObal NEtwork MOdel), and was conducted using the language ReactiveML [18,23]. It is itself inspired from a first use of ReactiveML for modeling networks [19]. Glonemo is quite efficient, but not precise enough on the details of the hardware. Moreover, ReactiveML is not connected to validation tools.

The Lustre model is 1500 lines long. It has been developed by K. Baradon and A. Vasseur, two master students of the Telecom department of INPGrenoble. It includes detailed energy models for all hardware parts that have a significant energy consumption. The connection to validation tools has been established.

The paper is organized as follow: section 2 lists the elements that have to be taken into account when building an accurate global model of a sensor network; section 3 presents the main structure of the Lustre model; section 4 details the components of the model and the way they are coordinated; section 5 presents existing tools and methods designed to study sensor networks; section 6 lists the current uses of our model, and section 7 concludes.

2 Aspects to be Taken into Account in a Realistic Model

2.1 Consumption of the radio and Behavior of the MAC protocol

The radio is the part of the node that consumes most, and mainly in emitting mode. It is clear that the radio should not function at maximum power all the time. In sensor networks, the Medium Access Control (MAC) protocol layer (the one that monitors the radio) is designed in such a way that the radio spends a lot of time in some idle mode, and very short periods in emitting mode.

In an accurate sensor network model, the various *modes* of the radio and their associated consumption should be detailed. Moreover, the way the MAC protocol triggers mode changes has to be described. This is because we want to observe properties related to energy consumption. Other properties like latency, throughput, bandwidth utilization or fairness are secondary.

2.2 The CPU and the Memory

Even if the radio consumes a lot, the energy used to process data cannot be neglected. According to Yuan and Qu [27], the processor is responsible for a consumption of 30 percent of the total consumption of the node. Moreover for some MAC protocols the micro-controller can be responsible for more than 90 % of the total energy needed to receive one data packet [20]. A technique used to optimize the energy consumed by the micro-controller is *Dynamic Voltage Scaling* (DVS). DVS consists in adapting dynamically the voltage of the micro-controller according to the load. This modifies the tradeoff between the consumption and the efficiency of the MCU. When the voltage is low, the consumption is low too but the micro-controller works slowly. This idea can be used in sensor networks [27].

If we want to reflect such technical solutions in our global models, this means that we should also detail the running modes of the CPU (a small number of discrete voltages is enough, the DVS is not driven in a continuous way). We should also describe how the mode changes are triggered, and by whom.

The CPU DVS may be driven by explicit operations in the object code of an application program, if a static analysis has identified pieces of the program where the load is low. If there is no such sophisticated analysis available, the CPU DVS is usually controlled by the operating system. In sensor networks, it may also be the case that the CPU is awaken by some activity on the radio (when there will be some data to process). In the sequel, we consider commands from the application. Commands from the radio could be modeled with the same technique.

The consumption of the memory is less important but researches are conducted on this topic [5,15]. If we want to take memory consumption into account, we should include the description of the memory consumption, depending on the type (RAM, Flash, ...). Some memories can have a `standby` mode in which they cannot be read or written to, but consume less. In order to read or write, one has to put the memory in normal mode first. Such a mechanism may also be driven by explicit operations in the object code of an application program, if a static analysis has identified pieces of the program during which some variables need not be accessed.

3 Overview of the Lustre Model

3.1 Principles and Main Structure

The structure of the Lustre model, called Lussensor, is inherited from the Glonemo model written in ReactiveML.But the model has been enriched with more details on the hardware of a node (DVS for the CPU, memory consumption of various kinds of memories, etc.).

The idea is to include one modeling component for each source of energy consumption we may want to model, even if we want to consider simple models where not all the sources are described in full details. Each element of the model is a data-flow box, also called *node* in Lustre. It has several input flows, several output flows, and some internal memory. Nodes are connected together as in synchronous circuits. We comment on synchrony and asynchrony in section 7. The main structure of the model is given by Figure 1.

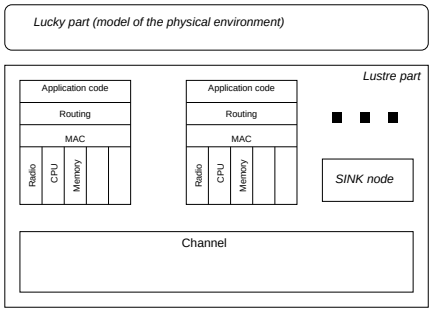


Fig. 1. Main Structure of the Model

The Lucky [10,11] part is used to model the physical environment, following the ideas described in [23]. We will concentrate on the Lustre part here. There are n instances of the same Lustre node `sensor`, representing the n identical sensors of a network, plus a special sink node. n has to be chosen statically, but we could choose n as the maximum number of sensors potentially present in the system; in this case, the model of a single sensor has an additional state "non existing" and it can be "created" or "destroyed" during the simulation.

All the components of the model are *deterministic* Lustre programs, although some of them need random values (e.g., the protocols). All the random values needed in the components are exposed as explicit inputs, connected to global inputs of the model, and then to an external generator. We could use a call to an external C function locally, but exposing the random value as an input is better for analysis purposes, because explicit abstractions can be made on its value.

Inside the Lustre node that represents one `sensor`, everything is *synchronous*. It is the right modeling since the physical node itself is a synchronous circuit. Between the sensors however, it is not the case. Although the physical nodes of a sensor network do have a physical clock, these clocks cannot be assumed to be synchronized during the whole lifetime of the network. Modeling the whole network is therefore one particular instance of the famous problem: *how to model asynchrony in a synchronous language?*. The general framework has been studied a lot (see [16,4,7]) and consists in equipping each asynchronous process with an additional input that plays the role of an activation condition for it. A specific global constraint on these activations conditions represents one special form of asynchrony. No constraint at all means pure asynchrony. A similar desynchronization mechanism is implemented in Lussensor but we do not detail it in the sequel.

The model should describe what happens precisely in the communication medium, i.e., the air in which the radio transmission occurs. We could even include electromagnetic perturbations, or other similar phenomena. All these modeling aspects are grouped in a Lustre node called `channel` that knows about the topology of the network. When a sensor emits something with its radio, this is modeled by the corresponding Lustre node sending a signal to the `channel` node, which may compute which of the other nodes will hear something, depending on their relative positions, and possibly integrating perturbations of the channel.

Each *sensor* instance is structured into several components: the application software; the routing protocol (usually software); the MAC protocol (could be software or hardware, at least partially); energy models for all the significant pieces of hardware (radio, CPU, memories, sensor, etc.).

The same principle is applied for all the *energy models*: we identify a (small) set of discrete significant values for the energy consumption of the device, corresponding to its well-identified *running modes*. Then we list all the possible mode changes.

Physically, these transitions between modes may take some time and energy too. For instance, switching the radio from sleeping to emitting mode has a cost, in both time and energy. We decide to encode all this phenomena into usual automata: spending time and energy is associated with states, the transitions are instantaneous and consume nothing. This means that we add some fictitious "states" to model the time and consumption of physical mode changes. Once these automata have been designed, the encoding into Lustre is very systematic.

Exploiting the various energy modes of hardware devices may be done in several ways. Our global model should provide a way to model any solution. Consequently, we provide a coordination between the model component that represents the application code, and the energy models of the CPU and the memory (see also section 4.3).

4 The Model Details

We then describe the components of the model in more details. The hierarchic structure of the Lustre part is described below. For each element we list the inputs and then the outputs, between parentheses. All these communications between the elements are signals, or flows, in the sense of Lustre, i.e., sequences of values over time. Technically, the node `channel` is the main node of the Lustre model, and its code contains the many instantiations of the sensor node.

The channel (sensor values, random data) (array of energies spent) =
- The sensor nodes 1, 2, ... N.
  Each node (sensor value, random data, radio inputs) (energy spent) =
  - Application (sensor value)
    (commands for Sensor, CPU, RAM, Flash energy models)
  - Routing
    (requests from appli, info from MAC) (requests to MAC, info to appli)
  - MAC (random data, radio input, requests from routing)
    (radio output, info to routing)
  - Sensor, CPU, RAM, Flash energy models
    (commands from appli) (energy spent)
  - Summation of the energies spent in this node
- The sink node (radio input)
- Summation of the energies spent since the beginning, for each node
- The data structures for the topology and state of the channel

4.1 Hardware Components

The energy models of the hardware parts are small automata, that can be encoded systematically into Lustre. The general form of the Lustre encoding can be observed on the partial model of the RAM given in Figure 2. Such a component outputs the current energy, i.e., the energy spent during one instant of the basic clock. Some other components will gather all these values and sum them to compute

```
node RAM (mode_change: int) returns (energy: real);
var current_mode: int;
let
  -- Encoding of the transitions
  current_mode = RAM_MODE_OPERATE ->
          if mode_change = MODE_DONTCHANGE
          then pre(current_mode)
          else if mode_change = RAM_MODE_OPERATE
               then RAM_MODE_OPERATE
          else ...
  -- Computation of the energy spent
  energy = if  current_mode = RAM_MODE_OPERATE
           then RAM_POWER_OPERATE
           else if current_mode = RAM_MODE_STANDBY
           then RAM_POWER_STANDBY
           else ...
tel
-- somewhere else in the global model, summation of
-- the ''instantaneous'' energy values computed by RAM:
sum = 0.0 -> RAM (...) + pre (sum) ;
```

Fig. 2. Example Lustre encoding for an automaton modeling energy consumption (all the capitalized words are constants).

the global consumption of the network. The input is a mode change request, given as the identity of the mode to reach. These components will be connected to other parts of the global model, in which the decisions for changing modes can be taken (for instance in the application software, see section 4.3).

The state is encoded by an integer or by a vector of Boolean values, depending on what we want to do with the model. For simulation purposes, it is better to use an `int`, but for validation purposes it is usually better to exhibit Boolean encodings wherever it is possible (because the exploration of the model becomes decidable). The transformation between the two forms can be done automatically in Lustre.

4.1.1 RAM and Flash Memory

The RAM memory usually has 4 modes, and not all mode-changes are possible. The modes are: `Off`, `Idle` (the memory can be read and written to normally), `Standby` (the memory cannot be read nor written to; its consumption is low; it takes some time to put the memory in `idle` mode) and `Deep-standby` (same behavior as the previous one,it consumes even less, and it takes more time to put the memory in `idle` mode).

In the model of Figure 3, for sake of simplicity, we did not model the time needed to switch between modes, because of its very small order of magnitude, compared to other times in the global model. But it could be done easily (see the principle on the radio model). The consumptions to be attached to the states are taken from the documentation of the STMicroelectronics SRAM DS2016.

For the Flash memory (Fig. 4), it is even simpler, because it will be used to store the program to be loaded. It does not need to be written to. The modes are: `Standby` and `read`. The consumptions are taken from the documentation of the SGS-THOMSON M28F256.

4.1.2 The CPU and the Sensor

The model of the CPU (Fig. 5) is a simple DVS model, corresponding to most existing DVS mechanisms. The model of the sensor (Fig. 6) is included in our

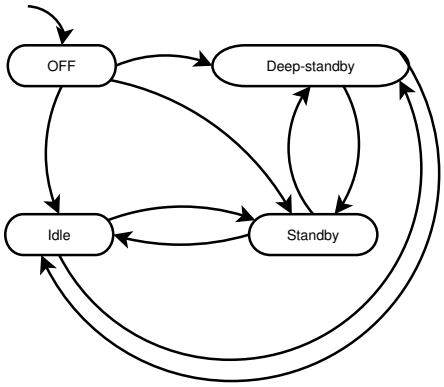


Fig. 3. Model of the RAM

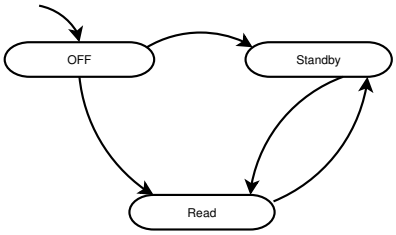


Fig. 4. Model of the Flash

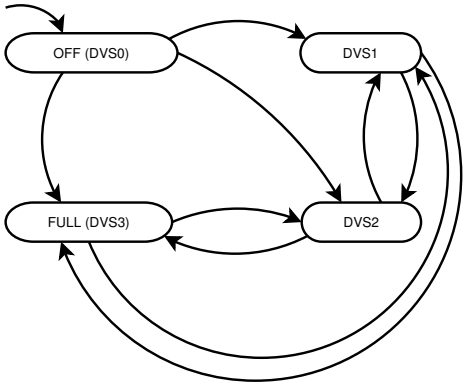


Fig. 5. The Model of the CPU

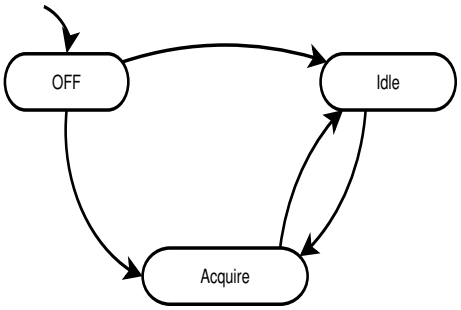


Fig. 6. Model of the Sensor

model because we suspect that intelligent sensors have several energy modes, but we did not find appropriate documentation yet. Anyway, a sensor could have at least two modes, depending on the fact that it is activated or not. This is reflected in the simple model given here, and could be enriched to take into account a more accurate sensor documentation.

4.1.3 The Radio

The radio (Fig. 7) is the most interesting energy model. The modes are: `Off`, `Idle`, `Hibernate`, `Transmit`, `Doze` and `Receive`, depending on the activity of the radio. The states denoted with dashed lines do not correspond to these modes, but they are added in order to be able to attach all timing and energy consumption information to *states*, whereas all the transitions are instantaneous and consume nothing. In the Lustre encoding, all the 10 states are represented. The information to be attached to the states is taken from the documentation of the Freescale MC13192, which implements the 802.15.4 norm.

4.2 Protocol Layers

In this paper we consider that the protocol layers are implemented in software. In order to include them in our global model, with the appropriate level of detail, we need to consider their object code. Indeed, when some software element drives

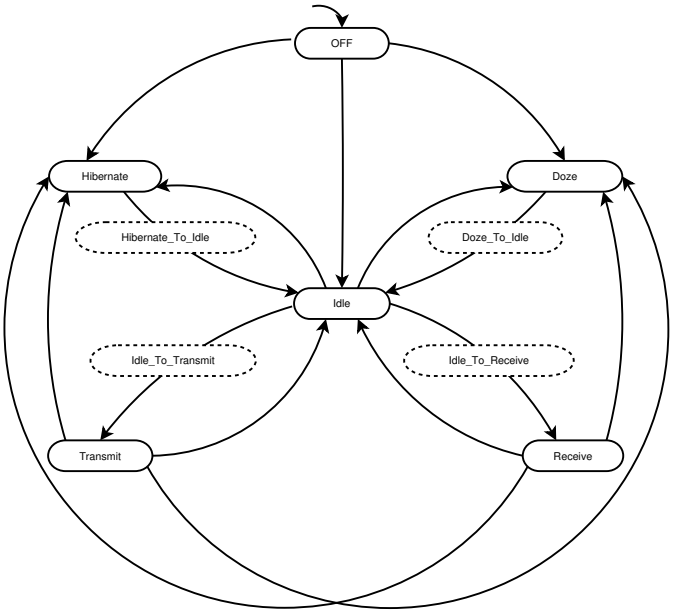


Fig. 7. Model of the Radio

the energy-saving mechanisms of the hardware, it is visible at the granularity level of the machine instructions. An assembly-line code can be easily described by an automaton (the control graph of the program), and that is what we do here. The automaton is then encoded into Lustre.

In order to give an idea of the levels of details that need to be modeled, we give a brief description of the MAC and routing algorithms included in Lussensor.

4.2.1 Medium Access Control

The MAC protocol implemented in Lussensor is a preamble MAC protocol (see, for instance, WiseMAC [6]). Each node periodically checks whether the channel is free. If the channel is busy, the node will let its radio on to get the packet that follows the preamble. Otherwise, it goes back to sleep mode.To avoid collisions we implement a *back-off*: the sender has to wait for a random time before emitting anything, then it scans the channel and if the channel is clear (Clear Channel Assessment, CCA), it sends the preamble and then the message. Otherwise, it delays the emission by setting a timer at random between 0 and $\mathrm{cw}_{\max}$. A preamble precedes each data packet for alerting the receiving node. All nodes in the network sample the medium with a common period.

The control automaton corresponding to this algorithm has 10 states. The Lustre encoding of this component will be connected to the component representing the application code, and to the component representing the channel. It also receives a random `int` value from the outside (see comment in section 3.1). For the connection to the channel, we model the radio phenomena by a pair `(signal, packet_data)`, where `packet_data` encodes the data transmitted (a Lustre array of `int`s) and `signal` is a value (`int` or `Bool` encoding the elements of the set:
{ `RF_SIGNAL_NONE`, `RF_SIGNAL_PREAMBLE`,
`RF_SIGNAL_PACKET`, `RF_SIGNAL_COLLISION` }.

```
node MAC (
  -- to be left as a global input of the model:
  random_mac: int;
  -- From application code
  start_mac: bool;
  want_to_transmit: bool; -- the appl.  wants to transmit a packet
  packet_to_transmit: useful_packet_data; -- data to be trans.
  -- From channel
  rfin_signal: int; -- type of the signal received on the radio
  rfin_packet_data: packet_data -- data received
) returns (
 energy: real;
 -- To application code
 busy: bool; -- the MAC is busy, cannot transmit now
 packet_received: bool; -- MAC has received a packet
 packet_transmitted: bool;  --  MAC has transmitted a packet
 received_data: useful_packet_data; -- the packet received
 -- To Channel
 rfout_signal: int; -- type of the signal emitted on the radio
 rfout_packet_data: packet_data; -- data emitted );
```

Fig. 8. Interface of the MAC component

This represents the fact that, from the point of view of the MAC, the radio is able to give the following information: either there is no signal, or there is a signal corresponding to a preamble, or there is a signal corresponding to a packet, or there is something that cannot be interpreted, meaning there is a collision. As an example, the interface of the Lustre node for the MAC is given by Figure 8.

4.2.2 Routing

The routing protocol included in Lussensor is the two-phase directed diffusion described in [9]. The sink first broadcasts an "interest" message to the whole network. The request can be *"send the temperature once an hour"*, or *"if the temperature increases sharply, send a message"* . This interest message is sent using a flooding routing mechanism: each node retransmits all the packets it receives except the ones it has already forwarded. When a node receives an interest, it checks whether it is concerned by the request and then forwards the packet. It will always send the values through the route that was used to reach it from the sink. The algorithm is encoded into an automaton, and then in Lustre.

4.3 The Application code and the Model of the Channel

The application code is a simple algorithm that emits the value sensed on a regular basis. It has 8 control states, and computes the commands `mode_sensor`, `mode_cpu`, `mode_flash` and `mode_ram` to be connected to the corresponding inputs of the hardware device models. For the moment, the values of these commands are entirely defined by the control state. The effect of any static analysis that would insert such commands in the object code of the application can be easily included in our model.

In Lussensor, the channel is the part of the global model that takes care of the air where communications take place, and knows about the topology of the network. The corresponding Lustre node `channel` computes which nodes receive a correct signal, which nodes are jammed, etc. It is quite complex because the main algorithms involved are iterative algorithms on matrices, which have to be encoded into the Lustre-V4 array operators (originally defined for circuit design, and based on static recursion, see [22]). But there is no intrinsic difficulty here.

5 Related Work

The first category of "virtual prototyping" approaches corresponds to the definition of formal models for performance analysis. These models are usually quite simple, and this is the reason why they are used mainly to compare protocols on one link. Since Kleinrock and Tobagi [12], they have been used extensively for the evaluation of MAC protocols. However, they cannot be used to compare complete protocol stacks.

All other virtual prototyping approaches are developed in order to include some description of the network behavior in the model. This gives more complex models, of course, for which there is no simple set of equations that could be solved. These models may be used for simulation, but we can also hope to use then for formal validation, if they are described in well-defined languages or formalisms.

Because network simulators are extensively used in the network community research, many relevant simulators have been developed. NS-2 [1] is a packet-level simulator that was first designed for wired networks. NS-2 is a discrete event simulator. The interest of having one single simulator is to enable comparisons between different protocols without the need to implement the protocol we want to compare with. Indeed, NS offers a large protocol library. However, NS is not really scalable: it is convenient for simulating a few hundred nodes only. Because one of the key issues in sensor networks is power consumption, people began to develop simulators that take the energy consumption into account.

Avrora [26] is written in Java and is cycle-accurate. It is able to execute the binary code of an application. The efficiency of the simulation relies on a quite complex synchronization pattern which in fact constitutes the model of the radio. For the environment, models are still needed, and the interaction between a model of some component and the exact description of another component is not formalized. It would be hard to use this framework to play with various abstractions.

Atemu [21] executes binary code and synchronizes the nodes on the clock cycle of the processor. Fine grain properties can be obtained up to 120 nodes. To our opinion, simulating the hardware at this level of detail is probably hopeless.

TOSSIM [17] is the simulator dedicated to TinyOS [25] applications. TOSSIM does not provide a model of the consumption. To overcome this limitation, it has been extended with PowerTOSSIM [24]. In PowerTOSSIM, each state of the CPU, Radio and EEPROM is associated with a cost. Running the simulator computes the energy consumption of each node.

AEON [13] proposes to build an energy model by running a real network, and then to include this model in a simulator like AVRORA, to do some profiling. AEON allows to observe the impact of the energy management primitives of TinyOS.

None of these simulators uses a formal model that could be used for validation.

On the other hand, the formal validation community does not seem to have started working specifically on sensor networks. To our knowledge, there is no other approach for the formal and global modeling of sensor networks, for which we can hope to use validation tools. Some experiments in modeling and analyzing sensor networks have been made with tools like HyTech [8] or Uppaal [14], but the models are still very abstract.

6 Current Uses of the Model

6.1 Validation by Simulations

The model has been developed progressively, and each component tested before integration. The complete model has been simulated with the Lustre interpreter, but it is quite slow (even for a small number of sensors, typically 10), and the graphical interface is poor, compared with what we can do in ReactiveML. The intended use is as follows: for any energy-related property one would want to observe on the model, design a Lustre *observer* (a special node that may read all the values of the input, output and local variables, but has no effect on the behavior) that outputs numbers; compile the Lustre model together with the observer; connect the code to the environment model in Lucky (this part generates values for the sensors, and also the random values needed by the protocol parts); run this a large number of times, storing the outputs; draw curves from the output sequences.

This method makes the Lustre model comparable to tools like `ns2`. We are currently investigating the "observer" version of the main quantitative evaluations usually found in the papers of the network community.

6.2 Uses of Lucky models

Lucky [10,11] belongs to the Lustre toolbox. It allows to describe non-deterministic reactive behaviors as sets of parallel communicating automata with weights representing probabilities. A Lucky component may be used in our global model to replace any of the Lustre components, provided it has the same input/output interface.

The first use of Lucky is to model the physical environment, i.e., the non-deterministic process that generates spatially and temporally correlated stimuli for the sensors. This is the same approach as in [23].

The next thing we will do is to use Lucky to model perturbations in the channel (shadowing, fading, path-loss). We'll have to encode in Lucky the accurate probabilistic modelings that have been proposed for these phenomena in the network community. The Lustre model is an appropriate platform for these experiments.

A similar use of Lucky would be to replace a part of the network (a subset of the nodes) by a *traffic generator*, i.e., a non-deterministic process that generates the states of the channel for the remaining nodes. This is related to the next point, since it is a way of abstracting the global model.

6.3 Modular Abstractions and Formal Analyzes

As mentioned in the introduction, analyzing formal models of sensor networks means we are able to perform quite drastic abstractions, but these abstractions may depend on the kind of property to be analyzed. We are interested in properties that talk about the energy consumption, for instance: "*is it possible to spend more than energy E in less that time T?*". This is a safety property. The Lustre model in which both time and energy are encoded into some numbers that behave as counters may theoretically be fed into a verification tool that deals with numbers

symbolically (abstract interpretation for instance). But the model for thousands of nodes is huge.

We propose to use the Lustre model as a modular-abstraction framework. The idea is the following: replacing a component C in a global model M by a more abstract version C' should yield a new global model M' which is indeed more abstract than M. This abstraction preservation property is essential when playing with various abstractions of the individual components. We should also be able to prove the property: C' is more abstract than C.

For the components modeling the energy consumption, the notion of abstraction has to be defined precisely. In such a model C, the energy attached to a "state" is in fact a worst-case estimation of the energy spent by unit of time while the system is in this state. Such a model, reacting to an input sequence I, produces a sequence of these "instantaneous" energies, than can be summed up. Let us note $\Sigma(C, I)$ the sum of the energy outputs produced when C reacts to I. A model C' is more abstract than C iff: for all I, $\Sigma(C, I) \geq \Sigma(C', I)$. "More abstract" means that the worst-case estimation is less precise, hence greater.

We are currently experimenting various abstract-interpretation tools to help verify automatically that an energy model C' is more abstract than a model C. The case study is the model of the radio (see Figure 7), for which various approximations can be derived.

7 Conclusions and Perspectives

We have designed the architecture of a global and accurate model of sensor networks, in Lustre. All the elements are taken into account, except the operating system. We could have included one more component for the OS without difficulty, but a lot of WSN solutions are considering static scheduling instead of using an OS, which is probably a good choice for energy consumption. Hence the "application" component of our model is sufficient. The software parts may be included at the level of detail of machine instructions, which gives a fine-grain modeling of energy consumption. Any hardware device can be modeled by a dedicated energy-model, as we did for the radio, the CPU, the memories, and the sensor. The values taken form the data-sheets of current technology devices can be directly included in our models.

We think that our model is as precise as the cycle-accurate models obtained with tools like Avrora or Atemu (see related work). Lussensor is not intended for debug simulations (it does not provide graphical outputs), but the compiled code may be used for batch simulations. Moreover, we think that Lussensor has several other important qualities:

- Lussensor is a modular model that may serve as a common platform for several abstractions. Moreover, the various abstractions can be compared, thanks to the abstraction partial order on the energy components, as described above.
- Adding observers to compute quantitative measures of the network behavior is very easy
- Lustre being a declarative language, the global model is essentially a set of

Boolean and numerical equations, for which we can hope to use a large set of symbolic verification tools.
- The Lussensor platform may be used to include existing probabilistic models as components, if we are able to describe them in Lucky.

Lussensor is a first step, for which the main perspectives are the following. First, we will use Lussensor as a case-study for our "modular worst-case energy models" approach; second, we will investigate the combined use of Lucky and Lussensor to design performance models of sensor networks that contain some details on the behavior of the computing parts. Indeed, as mentioned in section 5, the mathematical models used for the performance evaluation of protocols are too simple when it comes to representing complete protocol stacks or complex radio channel behaviors (i.e., collisions).

References

[1] The Network Simulator - ns2. http://www.isi.edu/nsnam/ns/.

[2] Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325–349, May 2005.

[3] Ian F. Akyildiz, W. Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[4] Paul Caspi, Christine Mazuet, and Natacha Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. In Udo Voges, editor, *Computer Safety, Reliability and Security, 20th International Conference, SAFECOMP 2001, Budapest, Hungary, September 26-28, 2001, Proceedings*, volume 2187 of *Lecture Notes in Computer Science*, pages 215–226. Springer, 2001.

[5] Victor Delaluz, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Anand Sivasubramaniam, and Mary Jane Irwin. Hardware and software techniques for controlling DRAM power modes. *IEEE Trans. Computers*, 50(11):1154–1173, 2001.

[6] Christian C. Enz, Amre El-Hoiydi, Jean-Dominique Decotignie, and Vincent Peiris. Wisenet: An ultralow-power wireless sensor network solution. *IEEE Computer*, 37(8):62–70, 2004.

[7] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, Grenoble, October 2002. LNCS 2491, Springer Verlag.

[8] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to hytech. Technical Report TR95-1532, Cornell University, Computer Science Department, August 29, 1995.

[9] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MOBICOM*, pages 56–67, 2000.

[10] E. Jahier, P. Raymond, and P. Baufreton. Case studies with lurette v2. *International Journal on Software Tools for Technology Transfer (STTT)*, 2006.

[11] E. Jahier, P. Raymond, and Y. Roux. Describing and executing random reactive systems. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, Pune, India, 2006.

[12] Leonard Kleinrock and Fouad A. Tobagi. Packet switching in radio channels: Part i–carrier sense multiple-access modes and their throughput-delay characteristics. In *IEEE Transactions on Communications*, volume 23, pages 1400–1416, december 1975.

[13] O. Landsiedel, K. Wehrle, and S. Gtz. Accurate prediction of power consumption in sensor networks. In *Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, 2005.

[14] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[15] Hyung Gyu Lee and Naehyuck Chang. Energy-aware memory allocation in heterogeneous non-volatile memory systems. In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 420–423, New York, NY, USA, 2003. ACM Press.

[16] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[17] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA. ACM Press.

[18] Louis Mandel and Marc Pouzet. Reactive ML.

[19] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, April 2005. ENTCS.

[20] Sylvain Plancoulaine. Architecture logicielle-matrielle pour protocole mac dans un rseau de capteurs. Rapport de stage de master de recherche csina, 5 mois, encadrant a. bachir, Université Joseph Fourier, Marseille, 2006.

[21] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, and John S. Baras. ATEMU: A Fine-grained Sensor Network Simulator. *Secon*, 2004.

[22] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits, a hardware implementation of Lustre. In Rex *Workshop on Real-Time: Theory in Practice, DePlasmolen (Netherlands)*, pages 195–208. LNCS 600, Springer Verlag, June 1991.

[23] Ludovic Samper, Florence Maraninchi, Laurent Mounier, Erwan Jahier, and Pascal Raymond. On the importance of modeling the environment when analyzing sensor networks. In *Proceedings of International Workshop on Wireless Ad-Hoc Networks 2006 (IWWAN 2006)*, page 7, New York, United States, June 2006.

[24] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *SynSys*, pages 188–200, 2004.

[25] TinyOS Team. Tinyos.

[26] Ben L Titzer, Daniel K Lee, and Jens Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. *Proceedings of IPSN*, 2005.

[27] Lin Yuan and Gang Qu. *Energy-Efficient Design of Distributed Sensor Networks*, chapter 38. CRC press, October 2004.

8. Towards mutation analysis for LUSTRE programs

Lydie du Bousquet, and Michel Delaunay

LSR-IMAG (UJF-CNRS), France

Notes:

Towards mutation analysis for LUSTRE programs

Lydie du Bousquet, Michel Delaunay [1]

LSR-IMAG (UJF-CNRS)
BP72, 38402 St Martin d'Hères Cedex, France

Abstract

Mutation analysis is usually used to provide indication of the fault detection ability of a test set. It is mainly used for unit testing evaluation. This paper describes mutation analysis principles and their adaptation to the LUSTRE programming language. Alien-V, a mutation tool for LUSTRE is presented. LESAR model-checker is used for eliminating equivalent mutant. A first experimentation to evaluate LUTESS testing tool is summarized.

Key words: Mutation analysis, data-flow programming language, LUSTRE, test, LESAR, LUTESS.

1 Introduction

In recent years, software quality assurance has received growing attention since it is recognized that a high level of quality is necessary to make both the client and the supplier confident in the product, and to reduce the maintenance costs. Testing is the most used technique for checking whether the required quality has been achieved. The testing purpose is to uncover the largest possible number of faults which have crept into the product during the construction stages. Due to the increasing complexity of software, the testing efficiency/quality has to be controlled.

During the last decade, the growing interest in synchronous languages from large companies has initiated significant contributions to the practical validation problem of synchronous software. Contrary to many other areas, and thanks to the rigorous mathematical semantics of this approach, much of current synchronous software testing theory and practice is not built on wishful thinking: several specification-based testing methods have been designed, implemented and have shown to be effective at revealing errors [13,33,21,4,24].

[1] Email: {lydie.du-bousquet,michel.delaunay}@imag.fr

Furthermore, all these methods allow to automate the test data generation process.

In this context, our previous works on testing concerned validation of Lustre programs. For this purpose, we have elaborated and we are currently improving Lutess a testing tool dedicated to synchronous programs [13,32,35]. Lutess produces randomly and dynamically test sequences. A natural question is then to evaluate the "quality" of the test data produced.

Mutation analysis has been introduced by De Millo in 1978 [11]. Its main purpose is to evaluate the quality/adequacy of a test set with respect to a fault model. It is mainly used for unit testing evaluation. The original work concerns Fortran programs. Since 1978, mutation analysis has been widespread, improved and evaluated [10,28,15,23,34,36]. Briand *et al.* have demonstrated that mutants can provide a good indication of the fault detection ability of a test suite [3].

In the following, section 2 details mutation analysis. Sect. 3 describes the adaptation of mutation operator for Lustre and introduces our mutation tool. Sect. 4 deals with equivalent mutant detection using the Lesar model-checker. Sect. 5 describes a first evaluation of Lutess testing tool. Sect. 6 concludes and draws some perspectives.

2 Mutation analysis

2.1 Principles

Mutation analysis consists in introducing a small syntactic change in the source code of a program in order to produce a *mutant* [11] (for instance, replacing one operator by another or altering the value of a constant). Then the mutant behavior is compared to the original program. If a difference can be *observed*, then the mutant is marked as *killed*. If the mutant has exactly the same observable behavior as the original program, it is *equivalent*.

The original aim of the mutation analysis is the evaluation of a test set. To do that, one has to produce all mutants corresponding to a predefined fault model. If the test set can kill all non-equivalent mutants, the test set is declared *mutation-adequate*. This means that the tests are able to discriminate the behavior of all faulty programs from the original program.

Adequacy of the test set is evaluated thanks to the *mutation score* (also called *adequacy score*). The mutation score is the percentage of non-equivalent mutants killed. For a program P, let M_T be the total number of mutant produced with respect to a particular fault model F. Let M_E and M_K be the number of equivalent and killed mutants. The mutation score of the test set T with respect to the fault model F is defined as:

$$MS(P,T,F) = \frac{M_K}{M_T - M_E}$$

A test set is *mutation-adequate* if the mutation score is equals to 1 [2] . Briand *et al.* have demonstrated that mutation analysis can provide a good indication of the fault detection ability of a test suite [3].

Mutation analysis relies on two assumptions. The first one is called *the programmer competent hypothesis*. It assumes that the programs are "nearly correct", that is to say, mostly correct with possibly simple faults. The second assumption is *the coupling assumption*. It assumes that a test set covering simple faults is able to detect more complex ones.

2.2 Tools for mutation

Mutation analysis is usually used to evaluate the adequacy of test data set produced during unit testing. It has been adapted for several programming languages, and lots of tools have been proposed [36]. For instance, Mothra tool supports Fortran 77 and ADA [10,31]. MuJava [23] and JMutator [3] are tools for Java. C-Patrol system is for C [1], NMutator for C#[3], Alien for VHDL [26]. Mutation analysis was also applied to Lustre [25], Petri-Nets, Final State Machine (FSM), Statecharts, and Estelle [17,16,15,12].

2.3 Mutation operators

The key of mutation analysis is the fault model. Fault model is expressed as a set of *mutation operators*. The original mutation operator set was proposed for Fortran 77. It was derived from studies of programmer errors. This mutation operator set has been refined during more than 15 years [10,28]. It is given Table 1.

Twenty-two operators were defined. Those operators were classified into eight classes and three levels (SAL, PDA and CCA) [10]. Statement AnaLysis (SAL) replace each statement by a TRAP [4] , by a CONTINUE or by a RETURN (for subprogram). It also replaces the (target) label in each GOTO and each DO statement.

The Predicate and Domain Analysis (PDA) takes the absolute value or the absolute negative value of an expression. It replaces one arithmetic/relational/logical operator by another, inserts a unary operator preceding an expression, alters a value of a constant or alter a DATA statement.

Coincidental Correctness Analysis (CCA) replaces a scalar variable, an array reference or a constant by another scalar variable, array reference or constant. It also replaces a reference to an array name by a reference of another array name.

[2] *Mutation testing* aims at producing tests until the maximal mutation score is obtained.
[3] http://www.inria.fr/rapportsactivite/RA2002/triskell/module7.html
[4] executing the TRAP will kill the mutant.

Mutation operator	Description	Levels
AAR	array reference for array reference replacement	CCA
ABS	absolute value insertion	PDA
ACR	array reference for constant replacement	CCA
AOR	arithmetic operator replacement	PDA
ASR	array reference for scalar variable replacement	CCA
CAR	constant for array reference replacement	CCA
CNR	comparable array name replacement	CCA
CRP	constant replacement	PDA
CSR	constant for scalar variable replacement	CCA
DER	DO statement end replacement	SAL
DSA	DATA statement alteration	PDA
GLR	GOTO label replacement	SAL
LCR	logical connector replacement	PDA
ROR	relational connector replacement	PDA
RSR	RETURN statement replacement	SAL
SAN	statement analysis (replacement by TRAP)	SAL
SAR	scalar variable for array reference replacement	CCA
SCR	scalar for constant replacement	CCA
SDL	statement deletion	SAL
SRC	source constant replacement	CCA
SVR	scalar variable replacement	CCA
UOI	unary operator insertion	PDA

Table 1
Mutation operator types for Fortran 77

2.4 Weaknesses of mutation analysis

Beyond the relevance of the mutation operators, the two main weaknesses of mutation analysis are (1) the cost and (2) equivalence decision.

Mutation cost

Mutation analysis is generally very expensive: lots of mutants are produced. Time is required to execute them in order to kill them. The number of mutant produced depends on the fault model and the program. Budd found that the number of mutants is roughly proportional to the number of reference times the number of data objects [8]. Acree *et al* estimate the number of mutants to be on the order of the square of the number of source lines [2].

Two main strategies have been proposed to reduce this cost: weak and se-

lective mutation. *Weak mutation* consists in comparing *internal states* (instead of outputs) of both program and mutant in order to detect differences. The observation can be done after the execution of the faulty expression, instruction, basic block and program. Weak mutation testing requires to produce less test that classical (strong) mutation.

N-selective mutation is mutation omitting the N most productive mutation operators. In [30], 2-selective was defined by omitting SVR (scalar variable replacement) and ASR (array reference for scalar variable replacement) operators. 4-selective mutation also omits CSR (scalar for constant replacement) and SCR (scalar for constant replacement).

Equivalence decision

A mutant which has exactly the same behavior as the original program is considered to be equivalent to the orignal program. Deciding if a mutant is equivalent to the program is an important step before computing the mutation score. Otherwise, it would not be possible to reach a mutation score of 1.

At the beginning of mutation analyis, equivalence decision was a complete manual process. It has been proved that "in general there cannot be a complete algorithmic solution to the equivalence problem" [27]. However, some works have been done to detect equivalent mutants automatically as much as possible. For instant, compiling technics or domain-constraints analysis were used [27,29], but they detect only a part of equivalent mutant set.

3 Applying mutation analysis to Lustre

3.1 Brief presentation of Lustre *language*

Lustre [18] is a synchronous declarative data flow language. The synchronous hypothesis considers the program reaction time to be negligible with respect to the reaction time of its environment.

The synchronous data flow approach consists in presenting a temporal dimension into the data flow model. A flow or stream (basic entity) includes two parts: a sequence of values of a given type, and a clock representing a sequence of instants (on the discrete temporal scale).

A Lustre description, structured in a network of nodes, represents the relations between the inputs and the outputs of a system. These relations are expressed by means of operators (nodes or basic operators), of intermediate variables and of constants.

A node is defined by a set of equations. Any local variable or output must be defined by one and only one equation. The equations can be written in any order without changing the behavior of the program.

Lustre offers usual arithmetic, boolean and conditional operators and two specific operators: **pre**, the "previous" operator, and -> the "followed-by"

```
node chrono (raz : bool)
returns (n : int);
let
  n = 0 -> if raz then 0 else ( pre(n) + 1 ) ;
tel;
```

Fig. 1. A simple Lustre program

operator [5]. Fig. 1 gives a Lustre program implementing a simple stopwatch (chronometer). The output `n` is set to 0 at the first step or when the `raz` input is true. It is incremented by one otherwise: the value of `n` at the current top is equal to the value of `n` at the previous top (pre `n`) plus one. `Current` and `When` are two other temporal specific operators of Lustre used for sampling signals.

3.2 Fault model

Mutation operators proposed for Fortran, Java or C are not completly reusable for Lustre, since it is a data-flow language. A first step of our work was to select a subset mutation operators that was compatible with Lustre language specificities. As we mentioned Sect. 2.3, mutation operators are classified into three groups: statement analysis (SAL), predicate analysis (PDA), coincidental correctness (CCA).

All operators from the class CCA were selected, except those dealing with array reference. In dataflow languages, and especially in Lustre, arrays are much more than a data structure. They are a powerfull way of constructing programs and define regular networks [6]. Simple syntactic change in the array reference usually produce an incorrect Lustre program. This is due the fact that Lustre is a strongly type language. For instance, let us consider the example given Fig. 2. In this node, the input and the output are two arrays of integers. The equation states that for (`i=0 to 5, b[i]=a[i]+1`). For this equation, replacing an array reference by a constant (in (`b` or `a`)) or modifing the size of one tabular will lead to an error.

```
node example(a : int^6)   -- array of integers
returns (b : int^6 );
let
  b[0..5] = a[0..5] + 1^6 ;
tel;
```

Fig. 2. An other Lustre program (with arrays)

All mutation operators of the PDA type except DSA were selected. Indeed Data Statement Alteration (DSA) can not directly be applied for Lustre,

[5] Let E and F be two expressions of the same type denoting the sequences $(e_0, e_1, ..., e_n...)$ and $(f_0, f_1, ..., f_n, ...)$; **pre**$(E)$ denotes the sequence $(nil, e_0, e_1, ..., e_{n-1}...)$ where nil is an undefined value. $E \rightarrow F$ denotes the sequence $(e_0, f_1, ..., f_n...)$.

since there is no data statement. In node given Fig 1, Arithmetic Operator Replacement (AOR) would replace `+` by `-`. The specific Lustre operator `pre` is considered for Unary Operator Insertion (UOI).

No SAL operators were selected. A Lustre node is a set of equations which can be written in any order. Since there should be exactly one equation for each output and local variable, it is not possible to delete a statement (SDL mutation operator). Moreover, Lustre has no DO, GOTO and RETURN statement (or similar ones). So related mutation operators (DER, DSA, GLR and RSR) have no sense here.

Lustre language has four specific temporal operators (`pre`, `followed-by`, `current` and `when`). As said previously, the operator `pre` is considered for UOI mutation operator. For `followed-by`, `current` and `when` opertaors, we are currently searching adequate mutation operator. However, thanks to "classical" mutation operators (changes in variables, constants and non-temporal operators), it is possible to alterate the behavior of sequential programs. For instance, for node `chrono`, it is possible to replace the variable `raz` or the constants 0 or 1, which will modify the behavior of `chrono`.

3.3 Mutation tool for Lustre

Alien-V [6] is a tool we built for mutating Lustre nodes. This tool was produced within a collaboration between LSR (team VASCO) and LCIS (team VALSYS). The multi-language mutant generator for VHDL and C developped by LCIS (Alien) [26] was extended to Lustre.

The mutation analysis is mainly a lexical process, since it is a multi-language tool. A mutation operator table is an input of the tool. It is possible to adapt this table to define specific mutation operators. For the moment, only AOR, LOR, and ROR are defined by default (Arithmetic/Logical/Relational Operator Replacement). CSR (Constant for Scalar variable Replacement) and SCR (Scalar for Constant Replacement) have to be manually parameterized for each program.

Some work is currently undertaken to improve Alien-V. We have used our mutation tool on several examples:

- an *Air-conditionner controller system* (Conditionner) specified in [5], and described in Lustre in [22],
- a *subway U-turn section* (UMS) [19],
- a *simplified monitoring of accelerometer sensors* (Monitoring) [7],
- a *water supplying system* (Supplying) [14]
- four examples of *8-integer sorting applications* (Sorting)
- a *lift* system [32].

[6] Lustre also means "5 years long" in French.

Program	# lexemes	# node	# operands	# operators	# mutants
Conditionner	61	1	30	20	32
UMS	66	1	32	18	16
Sorting 1	184	3	127	15	5
Sorting 2	369	1	221	104	100
Sorting 3	489	1	289	140	140
Sorting 4	932	1	562	343	924
Monitoring	268	6	86	111	3
Supplying	555	8	296	126	68
Lift	905	5	421	259	220

Table 2
Quantitative elements about mutated programs

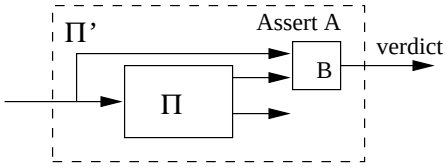


Fig. 3. Verification program structure

We have selected those 9 programs since they present different properties. Sorting programs are combinatorial examples. Conditionner and UMS are simple sequential boolean one-node programs. Monitoring is a simple sequential boolean program calling library nodes. Supplying is a more complex sequential boolean program, and Lift is a more complex boolean program which uses arrays. Most of these programs were provided with environment descriptions and safety properties.

Table 2 presents some quantitative elements about those examples and the results of the mutation analysis. As it can be noticed, mutation analysis for Lustre programs produces proportionnally less mutants than mutation analysis for imperative programming languages (see Sect. 2.4).

4 Detecting equivalent mutants

4.1 Lesar: a model-checker for Lustre

Lesar [19,20] is a model-checker for Lustre. It can be used to prove the correctness of a Lustre program with respect to some safety properties or to compare two programs.

As input, Lesar need a *verification program* [19]. A *verification program* is a specific Lustre program Π' built out of three elements (fig. 3):

- a program Π to be verified,
- a property P expressed by a boolean expression B which should be invariably *true*,
- some assumptions on the environment (environment constraints); those assumptions are boolean expressions (A) which can be assumed to be always true.

The verification is performed on a finite state abstraction Π'' of the program Π'. The verification principle is the following: proving that Π'' holds is equivalent to enumerating its finite set of states, checking that in each state (belonging to a path starting from initial state and on which the assertions are always true) and for each input vector, Π'' output evaluates to true. Lesar was originally a boolean tool. A special algorithm has been added into Lesar in order to treat contraints on numerical values.

4.2 Applying Lesar *for detecting equivalent mutant*

As previously said, mutation analysis can generate mutants equivalent to the initial program. It is the case when both mutant and original program have always the same observable behavior. Eliminating equivalent mutants is required, otherwise a maximal mutation score can not be reached.

```
node VerifPgm(in_0,..,in_n )   --inputs
returns (ok: bool);
var outo_0,..,outo_m           -- output for the original node
    outm_0,..,outm_m           -- output for the mutant
let
  (outo_0,..,outo_m) = original_node(in_0,..,in_n );
  (outm_0,..,outm_m) = mutant(in_0,..,in_n );

     -- property to be checked
   ok = (outo_0=outm_0) and ... and (outo_m=outm_m);
tel;
```

Fig. 4. A verification program to detect mutant equivalence

Lesar can be used to detect equivalent mutants produced for a Lustre program. To do that, one has to construct a verification program that is the comparison of the mutant and original programs, as it is done Fig. 4. When some environment description is provided with the original program, it is possible to consider the mutant-equivalency with respect to the environment description (using the `assert` operator) or without considering environment (unconditionnal mutant-equivalency).

We have applied Lesar to detect equivalent mutants for our 9 examples (see sect. §3.3). Although we generally have the environment descriptions for these examples, we wanted to demonstrate unconditionnal mutant-equivalency.

Program	# mutants	# eq. mutants
Conditionner	32	2
UMS	16	0
Sorting 1	5	-
Sorting 2	100	-
Sorting 3	140	-
Sorting 4	924	-
Supplying	68	15
Lift	220	0

Table 3
Mutant and equivalency

Lesar was very quick to detect equivalent mutants for Conditionner, UMS, Monitoring and Supplying. However, Lesar does not provide any result for the four 8-integer Sorting examples. We initially thought it was due to integer values. However, it was not possible to detect equivalent mutants with 8-boolean Sorting programs (same programs, with boolean inputs and outputs and same mutants produced). Lift program is composed of 5 nodes (one main node calling once time each of the four other nodes). Mutation analysis was done on each node. It was possible to detect equivalent mutant considering each node separatly.

5 Evaluating test data

As said in the introduction, mutation analysis was proposed to evaluate the quality/adequacy of a test set. So, to evaluate Alien-V, we wanted to evaluate test data produced by our testing tool Lutess. In the first part of this section, we briefly present the testing tool, and then we describe a first experiment using Alien-V to determine test data mutation-adequacy.

5.1 Lutess *testing tool: an overview*

Lutess [13,32] is a testing tool which we developed to validate reactive synchronous software. It requires three elements: an environment description written in Lustre (Δ), a program under test (Σ) and an oracle (Ω) providing the program requirements (fig. 5). Lutess builds a random generator from the environment description and constructs automatically a test harness which links the generator, the program under test and the oracle. The program under test and the oracle are both synchronous executable programs, with boolean inputs and outputs. They can be supplied as Lustre programs.

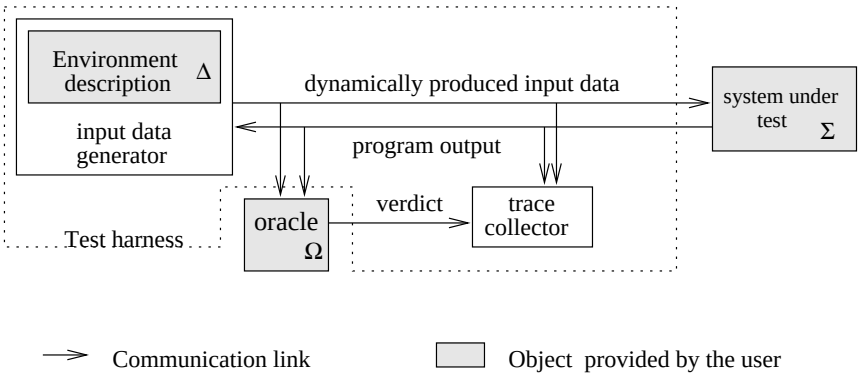


Fig. 5. Lutess

The test is operated on a single action-reaction cycle, driven by the generator. The generator randomly selects an input vector for the program under test and sends it to this latter. The program under test reacts with an output vector and feeds back the generator with it. The generator proceeds by producing a new input vector and the cycle is repeated. The oracle observes the program inputs and outputs, and determines whether the software specification is violated. The testing process is stopped when the user-defined length of the test sequence is reached.

Basically, the Lutess generator selection algorithm chooses a valid [7] input vector in an equally probable way. In each environment state, any valid input vector has the same probability to be selected. Lutess offers also various facilities to guide the generation (with property or statistical descriptions) and replay some test sequences (re-do) [13,22].

```
node OracleMutant(in_0,..,in_nn, outo_0,..,outo_m)
                --inputs and outputs for the original node
returns (ok: bool);
var
   outm_0,..,outm_m           -- output for the mutant
let
   (outm_0,..,outm_m) = mutant(in_0,..,in_n );

   -- property to be checked
   ok = (outo_0=outm_0) and ... and (outo_m=outm_m);
tel;
```

Fig. 6. A oracle program for Lutess to kill mutants

5.2 Test generation evaluation

To evaluate the mutation-adequacy of a test set, our general process is the following. On one hand, we produce one or several testing sequences with a tool

[7] An input is valid if and only if it is complying with the environment description.

Mu-	set A			set B			set C			set D		
tant	min	avg	max	min	avg	max	min	avg	max	min	avg	max
15	663	719.97	776	667	727.90	834	2	17.73	63	0	8.87	121
31	678	735.07	813	3	10.83	25	93	399.70	936	6	256.97	727
32	1416	1477.33	1580	9	19.83	34	101	407.13	937	7	256.87	727
47	3568	3790.83	3958	2074	5122.97	7101	9618	9812.03	9930	9973	9990.43	9999
53	0	3.03	11	0	4.17	13	11	333.47	899	0	254.83	724

min/max are the minimum/maximum numbers of differences observed for one sequence among the 30 of a set.
Avg is the sum of differences observed for the 30 sequences divided by 30.

Table 4
Some results for supplying example

(here Lutess). On the other hand, we produce "oracle" programs for Lutess (one for each non-equivalent mutant). Such an oracle program takes as inputs the original program inputs and outputs; it returns one boolean, which value is false each step the considered mutant produces different outputs than the original program (see Fig. 6). We then use the re-do function of Lutess, which feeds an oracle program with inputs/outputs previously obtained with the original program.

To carry out a first evaluation of Lutess, we focus on the Supplying example. Four sets of data were produced with environment constraints: without any guiding (A), with property guiding (B), with statistical guiding (C and D). Each time, 30 sequences of 10000 steps were generated. For each set, all mutants were killed. But not all test sequences killed every mutant. For some mutants, there were some test-sequences for which we could not observed any differences between the original program and the mutants.

For each set of data, we count how many steps a difference between the mutant and the original program could be observed. It was then possible to "compare" the sets of data (see Table 4). We call "mutant difficult to kill with a method" mutant for which differences could be observed in less than 1% of steps in average on the 30 sequences.

Mutants that are difficult to kill are usually those concerning "initial state" or "limit situation". Mutants difficult to kill are not the same in the different test data sets. This suggests that the generation methods of Lutess produce different types of data.

6 Conclusion and perspectives

Summary of the work

Mutation analysis aims at the evaluation of the adequacy of a test set with respect to a fault model. To do that, one has to produce all mutants corresponding to this fault model. If the test set can kill all non-equivalent mutants, the test set is declared *mutation-adequate*. This means that the tests

are able to discriminate the behavior of all faulty programs from the original program. Mutation analysis has been introduced by DeMillo in 1978 for Fortran 77 programs [11]. It has been widespread for different types of languages. [10,28,15,23,34,36]; and Briand *et al.* have demonstrated that mutants can provide a good indication of the fault detection ability of a test suite [3].

In this paper, we have adapted mutation analysis to Lustre programs. Lustre is a synchronous data-flow language. The data-flow nature of this language requires to adapt mutation operators that were originally proposed. Alien-V, the tool we built for Lustre program is presented. It has been experimented on 9 programs, from simple to more complex ones.

The main difficulty with mutation analysis is the detection of equivalent mutants. Equivalent mutants have exactly the same observable behavior than the original program. Eliminating equivalent mutant is required to obtain a maximum *mutation-score*. Since Lustre is based on a solid mathematical foundation, it is possible to construct proofs about the programs. For instance, Lesar is a model-checker for Lustre. To detect equivalent mutants, we have used Lesar. Unfortunately, it was not possible to kill mutants for some programs dealing with integers.

Finally, we have used mutant produced by Alien-V to evaluate *mutation-adequacy* of test data produced by Lutess our testing tool with different guides. First results show that the fact that a mutant is "difficult" to kill depends on the guides used for the generation. This means that Lutess generation method is really influenced by the guides.

Perspectives for Alien-V

For the moment, the fault model we used is mainly a selection of mutation-operator previously defined for imperative languages. We are currently defining adequate mutation operators for temporal Lustre operators (`followed-by`, `current` and `when`).

Mutation analysis was initially defined to evaluate adequacy of test data produced during unit-testing. Recently, some works have been proposed to extend mutation analysis to evaluate data set produced during integration testing [9]. A fault model specific to integration test evaluation was proposed. In Lustre, programs are oftenly structured with several nodes. Integration testing is therefore required. The next step for Alien-V is to define/adapt mutation operators to integration testing as it is done in [9].

Using mutation for test data adequacy evaluation

A first experimentation has been done to evaluate the mutation-adequacy of test data produced by Lutess. We would like to use mutation as a help to decide the end of testing. Moreover, we want to evaluate the influence (1) of the environment constraints and (2) of Lutess guiding methods on the mutation-adequacy of test data.

References

[1] Agrawal, H., R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur and E. Spafford, *Design of Mutant Operators for the C Programming Language*, Technical Report SERC-TR-41-P, Soft. Eng. Research Center, Dep. of Computer Science, Purdue Univ., Indiana (1989).

[2] Agree, A. T., T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward, *Mutation analysis*, Technical report git-ics-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA (1979).

[3] Andrews, J. H., L. C. Briand and Y. Labiche, *Is mutation an appropriate tool for testing experiments?*, in: *27th International Conference on Software Engineering (ICSE'05)* (2005), pp. 402–411.

[4] Arditi, L., A. Bouali, H. Boufaied, G. Clave, M. Hadj-Chaib, L. Leblanc and R. de Simone, *Using Esterel and Formal Methods to Increase the Confidence in the Functional Validation of a Commercial DSP*, in: *ERCIM workshop on Formal Methods for Industrial Critical Systems*, Trento, Italy, 1999.

[5] Atlee, J. M. and J. D. Gannon, *State-based model checking of event-driven system requirements.*, IEEE Trans. Software Eng. **19** (1993), pp. 24–40.

[6] Benveniste, A., P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic and R. de Simone, *The synchronous languages 12 years later.*, Proceedings of the IEEE **91** (2003), pp. 64–83.

[7] Bourret, P., A. Fernandez and C. Seguin, *Statistical criteria to rationalize the choice of run-time observation points in embedded software*, in: *1st International Workshop on Testability Assessment (IWoTA'04)* (2004), pp. 41–49.

[8] Budd, T. A., *Mutation analysis of program test data*, Phd thesis, Yale University, New Haven, CT, USA (1980).

[9] Delamaro, M. E., J. C. Maldonado and A. P. Mathur, *Interface mutation: An approach for integration testing.*, IEEE TSE **27** (2001), pp. 228–247.

[10] DeMillo, R., D. Guindi, K. King, M. M. McCracken and J. Offutt, *An extended overview of the mothra software testing environment*, in: *2nd Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, 1988, pp. 142–151.

[11] DeMillo, R., R. Lipton and F. Sayward, *Hints on test data selection: Help for the practicing programmer*, Computer, 11 (1978), pp. 34–41.

[12] do Rocio Senger de Souza, S., J. C. Maldonado, S. C. P. F. Fabbri and W. Lopes de Souza, *Mutation testing applied to estelle specifications*, in: *HICSS*, 2000.

[13] du Bousquet, L., F. Ouabdesselam, J.-L. Richier and N. Zuanon, *Lutess: a specification-driven testing environment for synchronous software*, in: *21st International Conference on Software Engineering* (1999), pp. 267–276.

[14] Duc, B. M., "Conception et modélisation objet des systèmes temps réel," Eyrolles, 1998, 341 pp.

[15] Fabbri, S. C. P. F., J. C. Maldonado, M. E. Delamaro and P. C. Masiero, *Mutation Testing applied to Validate Specifications Based on Statecharts*, in: *10th International Symposium on Software Reliability Engineering*, Boca Radon, FL, USA, 1999.

[16] Fabbri, S. C. P. F., J. C. Maldonado, P. C. Masiero and M. E. Delamaro, *Proteum/fsm: A tool to support finite state machine validation based on mutation testing.*, in: *19th International Conference of the Chilean Computer Science Society (SCCC '99)* (1999), pp. 96–104.

[17] Fabbri, S. C. P. F., J. C. Maldonado, P. C. Masiero, M. E. Delamaro and W. E. Wong, *Mutation testing applied to validate specifications based on petri nets.*, in: *Formal Description Techniques VIII, Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques (FORTE)*, IFIP Conference Proceedings **43** (1995), pp. 329–337.

[18] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, *Programmation et Vérification des Systèmes Réactifs : le langage LUSTRE*, Technique et Science Informatique **10** (1991).

[19] Halbwachs, N., F. Lagnier and C. Ratel, *Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Programming Language LUSTRE*, IEEE Transactions on Software Engineering (1992), pp. 785–793.

[20] Halbwachs, N., D. Pilaud, F. Ouabdesselam and A.-C. Glory, *Specifying, Programming and Verifying Real-Time Systems, using a synchronous declarative language*, in: *Workshop on automatic verification methods for finite state systems, LNCS 407* (1989).

[21] Jagadeesan, L., A. Porter, C. Puchol, J. Ramming and L. Votta, *Specification-based Testing of Reactive Software: Tools and Experiments*, in: *19th International Conference on Software Engineering*, 1997.

[22] Lakehal, A., F. Ouabdesselam, I. Parissis and J. Vassy, *Models for synchronous software testing*, in: *First International Workshop on Model, Design and Validation, 2004* (2004), pp. 41–50.

[23] Ma, Y.-S., J. Offutt and Y. R. Kwon, *Mujava: an automated class mutation system.*, Softw. Test., Verif. Reliab. **15** (2005), pp. 97–133.

[24] Marre, B. and A. Arnould, *Test Sequences Generation from Lustre Descriptions: GATeL*, in: *15th IEEE International Conference on Automated Software Engineering (ASE)* (2000).

[25] Mazuet, C., *Stratégies de Test pour des Programmes Synchrones - Application au Langage Lustre*, Technical report, Institut National Polytechnique de Toulouse, Toulouse, France (1994).

[26] Nguyen, T. B. and C. Robach, *Mutation Testing Applied to Hardware: the Mutants Generation*, in: *Proceedings of the 11th IFIP International Conference on Very Large Scale Integration*, Montpellier, France, 2001, pp. 118–123.

[27] Offutt, A. J. and W. M. Craft, *Using compiler optimization techniques to detect equivalent mutants*, Softw. Test., Verif. Reliab. **4** (1994), pp. 131–154.

[28] Offutt, A. J. and H. J. H., *A semantic model of program faults*, in: S. J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, San Diego, Californy, USA, 1996, pp. 195–200.

[29] Offutt, A. J. and J. Pan, *Automatically detecting equivalent mutants and infeasible paths*, Softw. Test., Verif. Reliab. **7** (1997), pp. 165–192.

[30] Offutt, A. J., G. Rothermel and C. Zapf, *An experimental evaluation of selective mutation*, in: *ICSE*, 1993, pp. 100–107.

[31] Offutt, A. J., J. Voas and J. Payne, *Mutation Operators for Ada*, Technical Report ISSE-TR-96-06, George Mason University (1996).

[32] Parissis, I. and F. Ouabdesselam, *Specification-based Testing of Synchronous Software*, in: *4th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, San Francisco, USA, 1996.

[33] Raymond, P., D. Weber, X. Nicollin and N. Halbwachs, *Automatic testing of reactive systems*, in: *19th IEEE Real-Time Systems Symposium (RTSS'98)* (1998).

[34] Scholivé, M. and C. Robach, "Simulation-based fault injection and testing using the mutation technique," Kluwer Academic Publishers, 2003 .

[35] Seljimi, B. and I. Parissis, *Using CLP to Automatically Generate Test Sequences for Synchronous Programs with Numeric Inputs and Outputs*, in: *17th Int. Symposium on Software Reliability Engineering (ISSRE'06)* (2006), pp. 105–116.

[36] Wong, W. E., "Mutation Testing for the New Century," Kluwer Academic Publishers, 2001.

9. Extending Lustre with Timeout Automata

Jimin Gao[1], Mike Whalen[2], and Eric Van Wyk[1]

1 Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, USA
2 Advanced Technology Center Rockwell Collins, Inc., Cedar Rapids, IA, USA

Notes:

Extending Lustre with Timeout Automata

Jimin Gao [a,1,2] Mike Whalen [b,3] Eric Van Wyk [a,1,4]

[a] *Department of Computer Science and Engineering*
University of Minnesota
Minneapolis, MN, USA

[b] *Advanced Technology Center*
Rockwell Collins, Inc.
Cedar Rapids, IA, USA

Abstract

This paper describes an extension to Lustre to support the analysis of globally asynchronous, locally synchronous (GALS) architectures. This extension consists of constructs for directly specifying the *timeout automata* used to describe asynchronous communication between processes represented by Lustre nodes. It is implemented using an extensible language framework based on attribute grammars that allows such extensions to be modularly defined so that they may be more easily composed with other language extensions.

Key words: synchronous languages, extensible languages, attribute grammars, composable language extensions

1 Introduction

Synchronous languages [2] have been successfully used to describe and reason about a wide variety of systems, including hardware design and synthesis [24], embedded software control [2], and modeling and analysis of globally asynchronous, locally synchronous (GALS) architectures [15]. These can be seen as *domain-specific languages* that address the concurrency and synchronization concerns of embedded systems and hardware at a high-level of abstraction.

The Lustre language [14], in particular, has been used in a wide range of academic and industrial projects. To better suit specific communities, the Lustre language has evolved into different dialects that further specialize the

[1] This work is partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.
[2] Email: jgao@cs.umn.edu
[3] Email: mwwhalen@rockwellcollins.com
[4] Email: evw@cs.umn.edu

language. These dialects have evolved from a simple "kernel" language that has been fairly stable throughout the development of Lustre. For example, to better support safety-critical software development, activation conditions (`condact`) and initialized delay (`fby`) constructs were added to the variant of the language used by the SCADE toolset [11], and a richer type system and modularity constructs have been proposed in Lustre v6. Other examples include `with` expressions and array slicing and composition operators in Lustre v4, `case`, `_TO_` and `_FROM_` expressions and support for generic types in the SCADE textual syntax, and different packages for statecharts-like extensions to the language [8,20]. In recent work [12], we have extended Lustre with condition tables like those found in RSML^{-e} [25], state variables for building simple state machines, and a notion of events.

There are many more domain-specific features that would make Lustre easier to use in new domains. For example, Lustre has been used for the analysis [15] and code generation [5,6] of GALS architectures. Our interests here are in using Lustre to specify and analyze (but not generate code from) the behavior of GALS architectures. Previous explorations of this idea, such as [15], assume that users manually construct a scheduler node and use it to manage the clocks of all of the asynchronous processes in the model. However, a scheduler could be automatically derived using a language extension, given the rates and drift of the asynchronous processes in the model. To support this process, we add to Lustre a *timeout_condact* construct that defines the behavior of an asynchronous process within the architecture as follows:

$$a, b = \mathit{timeout\_condact}(\mathit{rate}, \mathit{min\_drift}, \mathit{max\_drift}, \mathit{channel}(x, y), \mathit{init\_a}, \mathit{init\_b});$$

This construct (defined in Section 2) specifies that node *channel* representing a periodic process within the architecture is to be executed every *rate* milliseconds subject to clock drift in the range $\mathit{min\_drift}..\mathit{max\_drift}$. Like a *condact* expression, if the node does not evaluate, then the result of the expression is the value from the most recent evaluation, and before the first evaluation, the values $\mathit{init\_a}$ and $\mathit{init\_b}$ are used. Using this construct, a scheduler (implemented in the kernel Lustre language) can be automatically derived.

Extending a language using traditional techniques often requires a large development and tooling effort. Thus, there has been much research in programming languages communities on the development of techniques and tools for implementing languages that reduce the costs associated with adding new features to languages. There are (at least) two important criteria for extensions to a language. First, the new language constructs should have the same "look and feel" as the host language constructs. That is, they should support the same type of error-checking, optimization, and translations as do the host language constructs. Second, it should be possible to combine implementations of different extensions to the same host language to create a new language which incorporates the constructs in both. Furthermore, such a composition should require little or no implementation-level knowledge of the language

extensions. When this second criteria (referred to as the "composability criteria") is not met, users may be forced to chose between incompatible dialects of Lustre that individually have only some of the desired language constructs.

In previous work [28], we raised this issue of incompatible dialects and the traditionally high cost of language development. We proposed an extensible language framework for Lustre based on attribute grammars as a possible alternative approach to language development that satisfies the two criteria mentioned above. This approach is used to implement timeout automata as language constructs in Lustre. The primary contributions of this paper are the specification and implementation of timeout automata as first class language constructs in Lustre. Section 2 describes the GALS approach to development and defines the timeout automata construct. Section 3 describes some aspects of the implementation of the timeout automata as a language extension in our extensible languages approach. Section 4 discusses related work and concludes.

2 Timeout Automata and GALS architectures

2.1 GALS and Flight Guidance Synchronization Example

To illustrate our approach to the analysis of GALS architectures, we describe the synchronization logic in a Flight Guidance System (FGS). The FGS compares the measured state of the aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS subsystem accepts input about the aircraft's state from several other subsystems and computes the pitch and roll guidance commands provided to the autopilot.

The FGS system has two physical sides corresponding to the left and right sides of the aircraft. These provide redundant implementations that communicate over a cross-channel bus. Normally, only one FGS instance (the pilot flying side) is active, with the other FGS instance operating as a silent, hot spare. A *transfer switch* button on the flight control panel (FCP) can be used to toggle the pilot flying side. In some critical flight modes, both sides are active and independently generate guidance values for the autopilot, so that the autopilot can verify that they agree within a predefined tolerance value. [5]

To make the example of this paper tractable, we restrict ourselves to a simplified specification that deals only with the logic determining whether an FGS instance is active. This example captures critical functionality for the FGS, e.g., at least one side is active at all times, and illustrates some of the communication and coordination problems that can occur in GALS systems. In our analysis [21], we prove that this simplified model simulates the behavior of the full FGS w.r.t. synchronization, thereby ensuring that the results proven about the simplified specification also apply to the full specification.

[5] A more detailed description of the FGS can be found in [21].

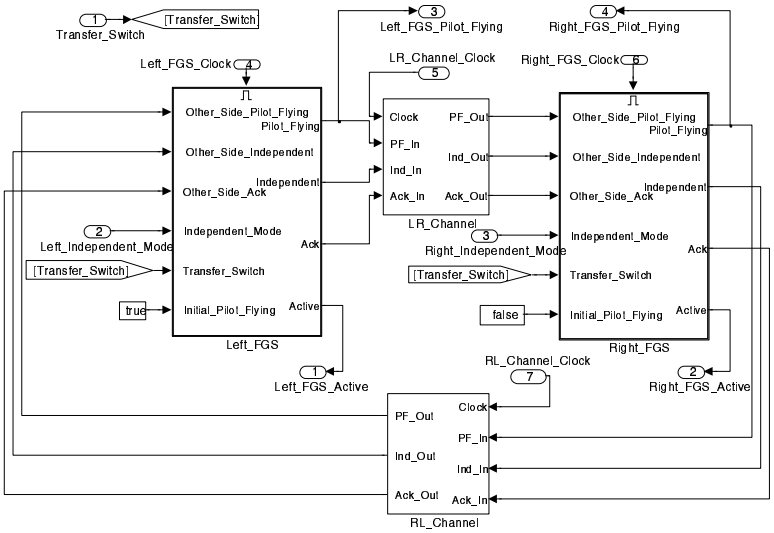


Fig. 1. Two FGS Synchronization Architecture in Simulink

A graphical model of the system architecture is shown in Figure 1. The system inputs are:

- the *Transfer Switch* input (1), which switches the *pilot flying* side,
- the *Independent Mode* inputs (2, 3), which are Boolean signals that determine whether each side believes it is in the independent mode of operation, [6] and
- the *Clock* inputs (4 – 7). Clocks are Boolean signals that enable the execution of the processes within the architecture. Treating the clock signals as unconstrained inputs allows us to model GALS systems within a synchronous paradigm [15]. By embedding this model inside a model that constrains the clocks, we can model a variety of different physical architectures and reason about their behavior.

The system outputs are:

- the *Active* outputs, which are Boolean signals that describe whether each side believes itself to be active, i.e., computing pitch and roll commands for the autopilot, and
- the *Pilot_Flying* outputs, which are Boolean signals that describe whether each side believes itself to be the pilot flying side.

2.2 Timeout Automata

A *timeout automaton* is a mechanism for constraining the Boolean clocks of processes to match a notion of real (calendar) time. As described in [9], the

[6] As discussed in [21], in an actual system these are not inputs to the FGS but are instead computed. However, the system synchronization properties do not depend on the details of this computation.

automata consists of a set of processes, each of which run at a certain rate. A *scheduler* (also called an event list) stores the times at which each of the processes will next execute. Evaluation of the system consists of advancing time to the next instant in which a process (or processes) can execute. Given a set of processes P, we assume that each process $p \in P$ has an associated rate r_p, a time until next execution t_p, and a Boolean clock signal c_p, and that there is a distinguished variable ci that records the increment of time since the last instant. Then, given a state σ mapping identifiers to values, we generate a new state σ' as follows:

- $\sigma'(ci)$ equals $\min(\sigma(t_p))$ where $p \in P$
- $\sigma'(c_p)$ is true iff $\sigma'(ci) - \sigma(t_p) = 0$
- $\sigma'(t_p)$ equals r_p if $\sigma'(c_p)$; otherwise $\sigma'(ci) - \sigma(t_p)$.

Each process "fires" (executes) when its clock signal c_p is true. Time always advances by some positive increment described by ci. If multiple processes share the same value for t_p and $t_p = ci$, then they execute simultaneously within the step. Clock drift between processes can be introduced to the model by allowing the rate r_p of each process to vary within some specified range.

In [9,3,4] this approach has been shown to be amenable to model-checking using SMT-based solvers for interesting GALS problems. We have also used it for system simulation and testing. The primary advantage of timeout automata for analysis that maximal time progress is made on each step (i.e., there are no "stuttering" steps in which the clock ticks but no other changes occur), and each step consumes a varying amount of real time as described by the clock increment ci.

In the Dual FGS example, we have used timeout automata to prove the correctness of the synchronization logic between the two FGSs. Properties proved include: (1) at least one FGS is always *Active*, and (2) at most one FGS is the *Pilot Flying* side. Other properties of interest are described in [21]. The proofs follow the process described in [9].

2.3 Implementation of Timeout Automata In Lustre

Timeout automata can be described as an extension to Lustre with the addition of a new expression construct:

$$timeout\_condact(rate, min\_drift, max\_drift, \langle node \rangle, \langle init\_vals \rangle);$$

This construct specifies that node *node* representing a periodic process within the architecture is to be executed every *rate* milliseconds subject to clock drift in the range *min_drift..max_drift*. Like a *condact* expression, if the node does not evaluate, then the result of the expression is the value from the most recent evaluation, and before the first evaluation of the node, the initial values *init_vals* are used. It is assumed (and checked by the compiler) that *timeout_condact* expressions are not nested within other clocked expressions; this matches the expectation within GALS systems in that the asynchrony

occurs at the global level and synchronous clocking mechanisms are local to one of the modeled processes.

In the Dual FGS model, the left and right FGSs run every 100 ms with a +/- 1 ms drift and communications between the two FGSs requires 15 to 25 ms. The expression of this architecture in Lustre is shown in Figure 2.

```
    type fgs_data = ... ; /* contains PF, Independent, and Ack data */
    const lft_fgs_init = ... ;   lr_init = ... ;
          rht_fgs_init = ... ;   rl_init = ... ;

    node fgs ( other_fgs_in: fgs_data, ind_mode: bool,
               transfer_switch: bool, init_pilot_flying: bool)
         returns ( fgs_out: fgs_data ) ;
     let ... tel ;

    node channel ( channel_in: fgs_data)
         returns ( channel_out: fgs_data ) ;
     let ...  tel ;

    node main ( trans: bool, lft_ind_mode: bool, rht_ind_mode: bool )
          returns ( lft_fgs_pilot: bool, lft_fgs_active: bool,
                rht_fgs_pilot: bool, rht_fgs_active: bool ) ;
    var
      lft_fgs_out: fgs_data ;  lr_chan_out: fgs_data ;
      rht_fgs_out: fgs_data ;  rl_chan_out: fgs_data ;
    let
(1)   lft_fgs_out = timeout_condact(100.0, -1.0, 1.0,
                      fgs(rl_chan_out, trans, lft_ind_mode , true),
                      lft_fgs_init) ;
(2)   lr_chan_out = timeout_condact(20.0, -5.0, 5.0,
                      channel(lft_fgs_out),lr_init);
(3)   rht_fgs_out = timeout_condact(100.0, -1.0, 1.0,
                      fgs(lr_chan_out, trans, rht_ind_mode , false),
                      rht_fgs_init) ;
(4)   rl_chan_out = timeout_condact(20.0, -5.0, 5.0,
                      channel(rht_fgs_out),rl_init);
    tel;
```

Fig. 2. FGS Synchronization Architecture using timeout_condact.

The semantics of the timeout condact specifications match the formalization in Section 2.2. Each timeout condact expression becomes a process in the timeout automata model, and a global scheduler is synthesized in Lustre from the set of these processes. As an example, consider Figure 3, the automatically generated implementation in Lustre for the FGS timeout condact in Figure 2.

The timeout node (line 8 of Figure 3) is used to define the r_p, c_p, and t_p variables for a process within the model. The `rate` and `drift` inputs set r_p, the `init_time` input sets the initial value of t_p, and the `time_decrement` input corresponds to the global time decrement between steps ci. The timeout node contains an individual count-down timer `time_remaining` that corresponds to t_p, and generates a Boolean signal `fired` that corresponds to c_p.

The expansion of the timeout condacts in `main` creates instances of the timeout node for each process and define constraints that describe the legal values for timers and drift inputs within the model. Line (3) in Figure 3 is the translation of the first timeout condact in Figure 2 to its implementation as a kernel language condact construct. The component node call to `fgs` and the initial values are the same; but the rate and drift parameters have been replaced by a clock variable (corresponding to c_p in the formal model) named `fired_1`. This variable is set on line (6) by a call to the `timeout` node that implements the time keeping operations of the timeout condacts.

On Figure 3 line (7), the model then selects the smallest time-remaining as the amount to advance each component clock (`time_decrement`) and feeds that value back to each individual component timer for use in computing the next clock tick. The definition of the node `min` is not shown but is what one would expect. Since the `time_decrement` value specifies the elapsed global time since the last clock tick, it is also output from the `main` node to allow a model checker to check properties involving global time (for example, the maximum time that some property P can be false is less than some time t).

Assert statements are also generated to restrict the new input drift values to be within the originally specified ranges of possible clock drift specified in the original timeout condact constructs. For the first timeout condact, the generated `assert` statements are shown in Figure 3 lines (4) and (5). Additional input parameters for the unconstrained input drift values are also added to the interface of `main`. The translation also adds new local variables in the line following the label (1).

The translation of the timeout condact constructs involves more that just local transformations that are possible with macro processing. The translation needs to generate new equations for each timeout condact and for defining `time_decrement` based on a global analysis that determines how many timeout condacts were used in the original code and what the generated time-remaining variables for each one are. Note that the original type declarations for `fgs_data`, the four constant *init* values and the declarations of the `fgs` and `channel` nodes are not changed in the translation and appear in the translated code as they did in the original. Thus, they are not repeated in Figure 3.

As there are only four processes in this model, the automata is relatively simple. However, with larger number of processes, it can become unwieldy. It is "boilerplate" code that must be re-written for each GALS system to be analyzed. Also, it is cumbersome to experiment with different architectural configurations (e.g., changing the rates and drift) in the translated model. We wish to encourage this kind of experimentation and formal analysis in the early stages of system design. Finally, there are hints that can be provided to aid analysis based on the structure of the automata (for example, the minimum and maximum possible values that the system clock can advance within a step). To make it easy to analyze these kinds of models, we would prefer to add a language construct to automatically construct the automata.

```
    node main (drift_4: real, init_time_4: real, drift_3: real, init_time_3: real,
               drift_2: real, init_time_2: real, drift_1: real, init_time_1: real,
               trans: bool, lft_ind_mode: bool, rht_ind_mode: bool)
         returns (time_decrement: real,
                  lft_fgs_pilot: bool, lft_fgs_active: bool,
                  rht_fgs_pilot: bool, rht_fgs_active: bool);
(1) var fired_4: real; time_remaining_4: real;
        fired_3: real; time_remaining_3: real;
        fired_2: real; time_remaining_2: real;
        fired_1: real; time_remaining_1: real;
(2)     lft_fgs_out: fgs_data ;  lr_chan_out: fgs_data ;
        rht_fgs_out: fgs_data ;  rl_chan_out: fgs_data ;
    let
(3)   lft_fgs_out = condact(fired_1, fgs(rl_chan_out, trans,lft_ind_mode, true),
                            lft_fgs_init);
      lr_chan_out = condact(fired_2, channel(lft_fgs_out), lr_init);
      rht_fgs_out = condact(fired_3, fgs(lr_chan_out, trans,rht_ind_mode, false),
                             rht_fgs_init);
      rl_chan_out = condact(fired_4, channel(rht_fgs_out), rl_init);
(4)    assert(((drift_1 <= 1) && (drift_1 >= -1)));
(5)    assert(((init_time_1 >= 0.0) && (init_time_1 <= (100.0 + 1))));
       assert(((drift_2 <= 5) && (drift_2 >= -5)));
       assert(((init_time_2 >= 0.0) && (init_time_2 <= (20.0 + 5))));
       assert(((drift_3 <= 1) && (drift_3 >= -1)));
       assert(((init_time_3 >= 0.0) && (init_time_3 <= (100.0 + 1))));
       assert(((drift_4 <= 5) && (drift_4 >= -5)));
       assert(((init_time_4 >= 0.0) && (init_time_4 <= (20.0 + 5))));
(6)    fired_1, time_remaining_1 = timeout(100.0, drift_1, init_time_1,
                                           time_decrement);
       fired_2, time_remaining_2 = timeout(20.0, drift_2, init_time_2,
                                           time_decrement);
       fired_3, time_remaining_3 = timeout(100.0, drift_3, init_time_3,
                                           time_decrement);
       fired_4, time_remaining_4 = timeout(20.0, drift_4, init_time_4,
                                           time_decrement);
(7)    time_decrement = min(time_remaining_4, min(time_remaining_3,
                           min(time_remaining_2, time_remaining_1)));
    tel;

(8) node timeout (rate: real, drift: real, init_time: real,
                  time_decrement: real)
      returns (fired: real, time_remaining: real);
    let
       time_remaining = init_time -> if fired then rate + drift
                        else pre(time_remaining) - pre(time_decrement);
       fired = (pre(time_remaining) <= pre(time_decrement));
    tel;
```

Fig. 3. Translated FGS Timeout Automata Model

3 Timeout Automata as a Language Extension

Implementing the timeout automata described in Section 2 by translation to the kernel Lustre language does not, per se, pose any exceptionally difficult challenges. Any solution, including ours, will (*i*) add a `timeout` node like the one in Figure 3 to the specification, (*ii*) add the equations that call to the `timeout` node and calculate the `time_decrement` value, and (*iii*) replace all `timeout_condact` constructs with the appropriate `condact` constructs that use the new Boolean *fired* flag. The main challenges arise in satisfying the look-and-feel and composability criteria described in Section 1. We have built [12] an extensible language framework based on higher-order attribute grammars (AGs) [17,30] and implemented an AG specification language called Silver [26] that supports the building of languages and extensions that satisfy these criteria. In this approach a host language and language extensions are implemented as individual Silver AG modules. The supporting tools allow the composition of these modules to define new extended languages with little or no implementation-level knowledge of the host or languages extensions [12]. In this section we give a brief overview of how the *timeout_condact* extension is constructed using this approach. Due to space constraints this is necessarily cursory and a number of simplifications and omissions have been made, but the full Silver specifications can be found at `www.melt.cs.umn.edu`.

3.1 Mini-Lustre as the Host Language

The specification for Mini-Lustre (a subset of Lustre) is written in Silver, a portion of which is shown in Figure. 4. A Silver specification for a language consists of an unordered series of declarations that define its concrete and abstract syntax as well as rules which assign values to attributes associated with non-terminals in the abstract syntax tree (AST). Since concrete syntax is defined as expected for traditional parser and scanner generators we do not show those and only discuss abstract syntax. To define the (abstract) syntax, there are declarations for terminals, non-terminals (keyword `nt`), and productions (`prod`), following standard AG terminology [17]. Synthesized attributes (`syn`) propagate information up the abstract syntax tree; inherited attributes (`inh`) propagate information down the AST. Equations defining attribute values are used to specify the semantic analyses, such as type checking. [7]

The first line in Figure 4 provides the name of this grammar. These are used in import statements to compose attribute grammar specifications to create the specification for an extended language. Next, are declarations for nonterminals. Synthesized attributes `pp`, `errors`, and `ctrans` of type `String` are declared; these attributes, respectively, define a node's pretty-print or "unparsed" representation, the errors occurring on the node and its children, and its translation to C. The attribute `typerep` is used to represent the type

[7] This is meant broadly and can include causality and initial-state-definedness checks.

```
grammar lustre ;
nt Root, DclList, Dcl, VarDclList, VarDcl, Locals, EqList, Eq, IdList, Expr;
syn attr pp      :: String occurs on Root, Dcl, Expr, VarDcl, ... ;
syn attr errors  :: String occurs on Root, Dcl, Expr, ... ;
syn attr ctrans  :: String occurs on Root, Dcl, Expr, ... ;
syn attr typerep :: TypeRep occurs on Expr, ExprList ;

prod root r::Root ::= dl::DclList
  { r.pp = dl.pp;  r.errors = dl.errors;  r.ctrans = ... dl.ctrans ...; }
prod dclListCons dl::DclList ::= d::Dcl dltail::DclList { ... }
prod dclListOne  dl::DclList ::= d::Dcl  { ... }
prod nodeDcl n::Dcl ::= name::Id inputs::VarDclList outputs::VarDclList
                        locals::VarDclList eqs::EqList
 { n.pp = "node " ++ name.lexeme ++ " (" ++ inputs.pp ++ ") " ++ "returns"
        ++  " (" ++ outputs.pp ++ ") " ++ "\n" ++ locals.pp ++ "\nlet\n"
        ++ eqs.pp ++ "\ntel;\n";
   n.errors = inputs.errors ++ outputs.errors ++ ... ; n.ctrans = ... ;  }
prod varDcl vd::VarDcl ::= var::Id type::Type  { ... }
prod equation eq::Eq ::= ids::IdList expr::Expr
 { eq.pp = ids.pp ++ " = " ++ expr.pp ++ ";\n" ;
   eq.errors = ... ;  /* ensure ids and expr have same type(s) */ }
prod idExpr e::Expr ::= id::Id
 { ...;  e.ctrans=...; e.errors = ... ;  /* ensure id is declared */ }
prod condactExpr e::Expr ::= f::Expr call::Expr init_vals::ExprList { ... }
```

Fig. 4. A portion of the Silver specification of Mini-Lustre.

of an expression or expression/id list. The `occurs on` clause specifies which nonterminals an attribute decorates. We will elide other nonterminal and attribution declarations as they can be inferred from the specification.

A Mini-Lustre program (represented by `Root`) is a series of declarations (`DclList`). The nonterminal `Root` on the left hand side of production `root` is named `r` ("`::`" reads as "has type"); the right hand side has a single `DclList` nonterminal named `dl`. Equations defining the synthesized attributes of `r` are listed in curly brackets. For example, the first equation defines the `pp` attribute on `r` to be the value of `pp` on `dl`. A node, defined by `nodeDcl`, has a name (`name`), a list of input parameter declarations (`inputs:: VarDclList`), a list of output parameters (`outputs`), a list of local variable declarations (`locals`), and a list of equations (`eql:: EqList`). The production `varDcl` binds identifier names to types. These bindings are stored in a symbol table that is passed to the equations in `eqs` and used for type-checking the expressions and equations following rules specified by the `errors` attributes. For example, the production `equation` checks that the identifier `id` and expression `expr` have the same type and generates an error message if they do not.

3.2 Timeout Automata as a Language Extension

To add the *timeout_condact* construct to the extensible Lustre framework we must write a Silver attribute grammar specification that will specify the con-

crete and abstract syntax of the new construct, perform error checking and other analyses on the *timeout_condact*, specify its translation to a *condact* construct, and for each use of a *timeout_condact* in a node add additional equations and parameters to that node. Further we must add the definitions for the *timeout* and *min* node to the Lustre specification. Using language features provided by Silver, all these tasks can be specified in a single grammar module, thus making this extension a stand-alone unit that can be optionally composed with other similarly-defined language extensions.

Fig. 5 shows the Silver production `tmoCondactExpr` that specifies the abstract syntax of the *timeout_condact* construct. To maintain the native look-and-feel, the `pp`, `errors`, and `typerep` attributes are defined explicitly in this production. Explicitly defining `errors` ensures that type errors are detected and reported on the *timeout_condact*, not its kernel Lustre translation. Though elided, the definition of `errors` checks that the types of values returned by the node call `call` match the types of the initial values `init_vals`.

Although `tmoCondactExpr` explicitly defines some attributes, it does not do so for attributes such as `ctrans` (or attributes for translating to the input languages of different model checkers). These attributes are *implicitly* defined using *forwarding* [27] through translation to a *condact* (`condactExpr`) in the host language by using the `forwards to` clause. When a `tmoCondactExpr` node in the AST is queried for an attribute that is not explicitly defined by an attribute definition, it forwards that query to the forwards-to construct. The value defined there is returned as the value of that attribute for the *timeout_condact*. Thus, the value of `ctrans` on a *timeout_condact* is the value of the `ctrans` attribute on the generated (translated-to) *condact* construct. Therefore, all back-end tools only see the generated *condact* calls while Lustre programmers see the *timeout_condact* calls they write.

In addition, the Silver specification assigns a unique integer identifier (attribute `num`) to each *timeout_condact* call. The identifier for each call is used in generated local variable names such as `fired_1` and `fired_2` as seen in Figure 3. Furthermore, relevant information regarding this *timeout_condact* call is gathered and propagated up the AST to the enclosing node declaration using the synthesized attribute `tmoCallInfoList`. This information is used for generating the added equations for variables such as `time_remaining_1` and `time_decrement` also seen in Figure 3. The additional attribute definitions of `tmoCallInfoList` on existing host language productions can be all specified in the grammar module of the *timeout_condact* extension by using the Silver language feature *aspect productions*, and no changes to the host language specification need to be made [13]. Once the information of all *timeout_condact* calls in a node is gathered to the level of node declaration (production `nodeDecl`), it is used to generate additional equations and parameters to be inserted into the node. This step is a global transformation that is simplified and modularly defined by using the Silver language feature *collections*. Its mechanism is not further elaborated here and interested readers

```
grammar timeout;    import lustre;
prod tmoCondactExpr  e::Expr ::= rate::Expr min_drift::Expr max_drift::Expr
                                 call::Expr init_vals::ExprList
{ e.pp = "timeout_condact(" ++ rate.pp ++ ", " ++ min_drift.pp ++ ... ;
  e.typerep = call.typerep ;
  e.errors = ...; /* check that call and init_vals have the same type */
  forwards to condactExpr(idExpr(mkTerminal(Id, "fired_" ++ toString(e.num))),
                          call, init_vals);
  e.num = gen_unique_int ( ) ;
  /* gathering information of timeout_condact calls */
  e.tmoCallInfoList = [tmoInfo(rate, min_drift, max_drift, e.num)];  }
```

Fig. 5. Silver specification for the *timeout_condact* construct.

may refer to [13] for detailed explanations.

4 Conclusion

4.1 Discussion

In this paper we have defined a timeout condact construct useful in specifying and analyzing GALS architectures. It has been implemented as a language extension in an extensible Lustre framework. Timeout automata is one of several approaches for modeling asynchrony within synchronous languages. It has been used successfully on several protocol examples (e.g. [9,3,4]) and allows a natural expression of interesting safety and bounded liveness properties over GALS architectures. However, in the simplistic translation described in this paper, it adds a significant amount of additional state into the model, which makes formal analysis more expensive. Abstractions of the possible real-time evolutions of the architecture, such as those described by [15] may yield more tractable analysis. The use of extensible languages opens up several possible directions for future research. First, we plan to investigate whether abstractions can be performed as part of the compilation step to "kernel" lustre. Second, we plan on investigating techniques for describing clock relations (such as in [15]) directly through language extensions.

Our initial efforts in extensible languages were in the domain of programming languages. We have built an extensible specification of Java 1.4 and specified a number of non-trivial language extensions [29]. One extension embeds the database query language SQL into Java so that queries can be written naturally and syntax and type errors in SQL queries can be detected at compile-time, instead of run-time, as is the case in library-based approaches.

4.2 Related Work

There have been many other efforts to extend Lustre with new language features. Many of these features can also be implemented by translation to the a kernel Lustre language. For example, recent work to add state machines to

Lustre [8] translates the state machine constructs into a kernel Lustre language and the addition of modules and generics proposed for Lustre v6.

Extensions for synthesizing Lustre logical clocks from Simulink models with "real-time" rates for blocks are proposed in [5,6]. This work is similar in that it moves from a notion of real-time to logical time. Unlike timeout automata, it imposes a fixed real-time value on the base rate of the model; this allows for code generation but makes it more difficult to analyze processes with non-harmonic periods or arbitrarily small amounts of process drift.

Embedded domain specific languages [16], higher-order extensions to Lustre [23], and reactive extensions to ML [19] can be used to build extensible language frameworks for synchronous languages [7]. But composition of language features typically requires some implementation level understanding of the language extension and thus various extensions cannot be as freely composed as in our approach [12].

More generally, several approaches have been described for extending languages with new features. Macros systems (lexical, syntactic, hygienic [18], etc) do allow new languages constructs to be specified but they lack an effective means for performing the static analysis used to, for example, generate domain specific error messages. Note that some modern macro systems (*e.g.* [1] however do a some limited facilities for error processing. Object-oriented frameworks, such as Polyglot [22], have also been proposed for building extensible languages, but they do not support the automatic composition of language extensions that is provided by the attribute grammar-based approach.

Modular language definition and extensibility has received a significant amount of attention from the AG community. Other attribute grammar approaches lack forwarding and the default definition of attributes that it provides - thus the reuse of language features specified as AG fragments is achieved only by writing attribute definitions that "glue" new fragments into the host language AG. However, a particularly interesting approach is the rewritable reference attribute grammars [10] in the JastAddII system. New constructs are translated to host language constructs by destructive rewrites on the syntax tree. Although forwarding is similar to rewriting, it is non-destructive; the original tree and the forwards-to tree exist simultaneously. This allows both the explicit and implicit (via forwarding) specification of semantics, a capability that we have found to be crucial in the highly modular language specifications required for extensible languages and composable language extensions. Some modularity is lost when the rewrites are destructive.

References

[1] Batory, D., D. Lofaso and Y. Smaragdakis, *JTS: tools for implementing domain-specific languages*, in: *Proc. 5th Intl. Conf. on Software Reuse* (1998).

[2] Benveniste, A., P. C. an S. Edwards, N. Halbwachs, P. L. Guernic and

R. de Simone, *The synchronous languages 12 years later*, Proceedings of the IEEE **91** (2003), pp. 64–83.

[3] Brown, G. M. and L. Pike, *Easy parameterized verification of biphase mark and 8N1 protocols*, in: *Proc. of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, LNCS **3920** (2006), pp. 58–72.

[4] Brown, G. M. and L. Pike, *"Easy" parameterized verificaton of cross clock domain protocols*, in: *Seventh International Workshop on Designing Correct Circuits DCC: Participants' Proceedings*, 2006, satellite Event of ETAPS.

[5] Caspi, P., A. Curic, A. Maigna, C. Sofronis and S. Tripakis, *Translating discrete-time simulink to lustre*, in: *International Conference on Embedded Software (EMSOFT'03)* (2003).

[6] Caspi, P., A. Curic, A. Maignan, C. Sofronis, S. Tripakis and P. Niebert, *From simulink to scade/lustre to tta: A layered approach for distributed embedded applications*, in: *LCTES* (2003).

[7] Claessen, K. and G. J. Pace, *An embedded language framework for hardware compilation*, in: *Proceedings of Designing Correct Circuits*, 2002.

[8] Colaco, J.-L., B. Pagano and M. Pouzet, *A conservative extension of synchronous data-flow with state machines*, in: *Proceedings of the 5th ACM International Conference on Embedded Software* (2005), pp. 173–182.

[9] Dutertre, B. and M. Sorea, *Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata*, in: *Formal Techniques in Real-Time and Fault Tolerant Systems*, LNCS **3253**, 2004, pp. 199–214.

[10] Ekman, T. and G. Hedin, *Rewritable reference attributed grammars.*, in: *Proc. of ECOOP '04 Conf.*, 2004, pp. 144–169.

[11] *SCADE suite product description—by Esterel technologies.*, http://www.esterel-technologies.com.

[12] Gao, J., M. Heimdahl and E. Van Wyk, *Flexible and extensible notations for modeling languages*, in: *Fundamental Approaches to Software Engineering, FASE 2007*, LNCS **4422** (2007), pp. 102–116.

[13] Gao, J., M. Heimdahl, M. Whalen and E. Van Wyk, *A flexible and extensible framework for modeling languages*, Technical report, University of Minnesota (2006), available at www.melt.cs.umn.edu.

[14] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, *The synchronous dataflow programming language Lustre*, Proc. of the IEEE **79** (1991), pp. 1305–1320.

[15] Halbwachs, N. and L. Mandel, *Simulation and verification of asynchronous systems by means of a synchronous model*, in: *ACSD 2006, Sixth Intl. Conf. on Application of Concurrency to System Design*, Turku, Finland, 2006.

[16] Hudak, P., *Building domain-specific embedded languages*, ACM Computing Surveys **28** (1996).

[17] Knuth, D. E., *Semantics of context-free languages*, Mathematical Systems Theory **2** (1968), pp. 127–145, corrections in **5**(1971) pp. 95–96.

[18] Kohlbecker, E., D. P. Friedman, M. Felleisen and B. Duba, *Hygienic macro expansion*, in: *Proceedings of the 1986 ACM conference on LISP and functional programming* (1986), pp. 151–161.

[19] Mandel, L. and M. Pouzet, *ReactiveML: a reactive extension to ML*, in: *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2005), pp. 82–93.

[20] Maraninchi, F. and Y. Rémond, *Mode-automata: About modes and states for reactive systems*, in: *European Symposium On Programming* (1998).

[21] Miller, S. P., M. W. Whalen, D. O'Brien, M. P. Heimdahl and A. Joshi, *A methodology for the design and verification of globally asynchronous/locally synchronous architectures*, Technical Report Contractor Report NASA/CR-2005-213912, NASA (2005).

[22] Nystrom, N., M. R. Clarkson and A. C. Myer, *Polyglot: An extensible compiler framework for java*, in: *Proc. 12th International Conf. on Compiler Construction*, LNCS **2622** (2003), pp. 138–152.

[23] Pouzet, M., *Lucid synchrone, version 3. tutorial and reference manual.*, distribution available at: www.lri.fr/~pouzet/lucid-synchrone.

[24] Rocheteau, F. and N. Halbwachs, *Pollux, a Lustre-based hardware design environment*, in: P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, 1991.

[25] Thompson, J. M., M. P. Heimdahl and S. P. Miller, *Specification based prototyping for embedded systems*, in: *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, LNCS **1687**, 1999.

[26] Van Wyk, E., D. Bodin, L. Krishnan and J. Gao, *Silver: an extensible attribute grammar system*, in: *Proc. of LDTA 2007, 7^{th} Workshop on Language Descriptions, Tools, and Analysis*, 2007, to appear.

[27] Van Wyk, E., O. de Moor, K. Backhouse and P. Kwiatkowski, *Forwarding in attribute grammars for modular language design*, in: *Proc. 11th Intl. Conf. on Compiler Construction*, LNCS **2304**, 2002, pp. 128–142.

[28] Van Wyk, E. and M. Heimdahl, *Flexibility in modeling languages and tools: A call to arms*, in: *Proc. of IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, 2005.

[29] Van Wyk, E., L. Krishnan, A. Schwerdfeger and D. Bodin, *Attribute grammar-based language extensions for java*, in: *European Conference on Object Oriented Programming (ECOOP)*, LNCS (2007), to Appear.

[30] Vogt, H., S. D. Swierstra and M. F. Kuiper, *Higher-order attribute grammars*, in: *ACM PLDI Conf.*, 1990, pp. 131–145.