Compilation of the Intermediate Representation V1

Clément Poncelet, Florent Jacquemard

Compilation of the Intermediate Representation V1

Clement Poncelet, Florent Jacquemard

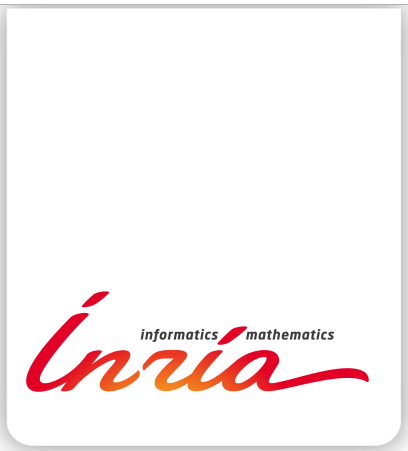
RESEARCH REPORT

N° 8701

March 2015

Project-Team MUTANT

ISSN 0249-6399 ISRN INRIA/RR--8701--FR+ENG



Compilation of the Intermediate Representation V1

Clement Poncelet, Florent Jacquemard

Project-Team MUTANT

Research Report n° 8701 — March 2015 — 10 pages

Abstract: The IR has the form of an executable code modeling the expected Antescofo's behavior on a given score. We present here a simplified version of Antescofo's IR suitable for our presentation, leaving features such as thread creation, conditional branching, and variable handling outside of the scope of this paper. In this paper, we define the construction of the first version of the IR from a given Antescofo DSL score.

Key-words: Model-based testing, automatic model construction, Antescofo

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Compilation de la Représentation Intermédiaire v1

Résumé : La Représentation intermédiaire est sous forme d'un code exécutable modèlisant le comportement attendu d'Antescofo pour une partition donnée. Nours présentons ici une version simplifiée de cette représentation, mettant de côté la création de thread, les branchements conditionnels et la gestion de variable. Dans ce papier, nous définissons la construction de la première version de la représentation intermédiaire depuis la partition Antescofo.

Mots-clés : test basé sur model, construction de modèle automatique, Antescofo

1 introduction

1.1 The score-based IMS Antescofo representation

The IR construction is based on the parsing of an Antescofo mixed score abstract syntax tree. We represent this AST (the handled part) as described Figure 1.

Let $\mathcal{O}$ be a set of *output messages* (also called *action symbols* and denoted a) which can be emitted by the system and let $\mathcal{I}$ be a set of *event symbols* (denoted e) to be detected by the LM (i.e. positions in score). An *action* is a term $\mathsf{act}(d, s, al)$ where d is the delay before starting the action, s is either an atom in $\mathcal{O}$ or a finite sequence of actions (such a sequence is called a *group*), and al is a list of attributes. A *mixed score* is a finite sequence of *input events* of the form $\mathsf{evt}(e, d, s)$ where $e \in \mathcal{I}$, d is a duration and s is the top-level group *triggered* by e. Sequences are denoted with square brackets [,] and the empty sequence is [ ].

A sequence of actions is a suite of actions that can be another group or an atomic action. An atomic action can be a message or a kill action. Messages and groups can have attributes for their synchronization strategy and/or error management (**@loose @global** by default). We distinct an atomic action of a group by the presence or not of an actions sequence in al.

2 Model construction

The construction is done through a compiler, which visits the AST (design pattern visitor) and constructs the corresponding network of machines. Currently we represent graphically these machines with extended FSMs.

2.1 FSM representation

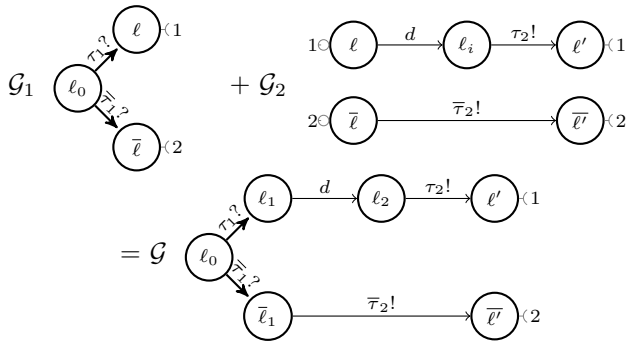
During the parsing, part of machines are built and merged with other (previous and next parts).

For this purpose we use FSMs with two distinguished sequences of locations called providers and seekers. They are marked with $i\circ$, resp. $\prec i$, where i is an index in the sequence of providers, resp. seekers. The operator $\mathcal{G}_1 + \mathcal{G}_2$ takes two FSMs with the same number of seekers for $\mathcal{G}_1$ as providers for $\mathcal{G}_2$, and returns

score	::=	ϵ | evt score
evt	::=	e d s
act	::=	d s al
s	::=	ϵ | *act* s
al	::=	sync err
sync	::=	**@loose** | **@tight**
err	::=	**@local** | **@global**

Figure 1: Grammar of the handled AST-mixed score

one FSM $\mathcal{G}$ obtained by a union of $\mathcal{G}_1$ and $\mathcal{G}_2$ where moreover, every seeker of $\mathcal{G}_1$ is merged with the provider of $\mathcal{G}_2$ with the same index. The sequence of providers (resp. seekers) of $\mathcal{G}$ is the sequence of providers of $\mathcal{G}_1$ (resp. sequence of seekers of $\mathcal{G}_2$), see the Figure below.



2.2 Inference rules

Parsing the Score Inference rules are used to explain the recursive behavior of each case, during the parsing phases. Keep in brain that a rule presents in its top what you need to do (one or more sub-functions, or itself recursively) for doing a step of the function at its bottom part. A function is represented with its inputs arguments, its type name ($\vdash$), the current AST parsed part and the IR returned. Inference rules describe the parsing and model construction of the compiler. It is depicted as:

$$\frac{\langle input\rangle \vdash \langle AST\rangle : \langle IR\rangle \qquad \cdots}{\langle input\rangle \vdash \langle AST\rangle : \langle FinalIR\rangle}\ \text{EX}$$

with $\langle FinalIR\rangle = \langle IR\rangle$ + rule step and $< input >$ = acc, sc, e, an accumulator of time elapsed since the beginning of processing of the group, a score sc and a current event e (called the next closest event). Generally the parsed AST ($\langle AST\rangle$) is an event for rules (EG,...) which contains a group (otherwise nothing is done and the event is simply past), or an actions sequence with a current atomic action or group parsed. In the latter case rules MSG or GRP respectively is executed.

2.3 Beginning

We start with two rules, which parse the entire AST ($\vdash_{\mathsf{all}}$). The first rule expresses the empty case returning an empty FSM called $\mathcal{A}_\emptyset$. The other starts by creating three parallelized machines (expressed with the operator $||$), $\mathcal{E}$ for the environment, $\mathcal{S}$ for the Specification composing by $\mathcal{P}$ the proxy and $\mathcal{G}$ the sub-network of machines modeling Antescofo score-specific behaviors. We let $\mathcal{M}$ the entire IR-model composing by the environment $\mathcal{E}$ and the specification $\mathcal{S}$ and we define the operator $||$ as a current parallelization of two states machines.

$$\frac{}{\vdash_{\mathsf{all}} \emptyset : \mathcal{A}_\emptyset}\ \text{EMPTY} \qquad \frac{\vdash_{\mathsf{env}} sc : \mathcal{E} \qquad \vdash_{\mathsf{proxy}} sc : \mathcal{P} \qquad \vdash_{\mathsf{evt}} \langle [\,], sc\rangle : \mathcal{G}}{\vdash_{\mathsf{all}} sc : \mathcal{M}}\ \text{ALL}$$

With $\mathcal{M} = \mathcal{E} \parallel \mathcal{S}$ and $\mathcal{S} = \mathcal{P} \mid\mid \mathcal{G}$

2.4 Environment and proxy

Not described here, the proxy and environment construction (resp. $\vdash_{\mathsf{env}}$ and $\vdash_{\mathsf{proxy}}$) depend on the possible number of consecutive missed events (and rate of variation for event durations). However it is a simple event parse, creating an edge (in the trivial case) for each event of the mixed score.

2.5 top Groups

The IR resulted $\mathcal{G}$ is a set of parallelized machine $\mathcal{G}_i$ describing the behavior of an actions sequence. So i is equal to the number of actions sequences of the mixed score. As $\vdash_{\mathsf{env}}$ and $\vdash_{\mathsf{proxy}}$, $\vdash_{\mathsf{evt}}$ parse the events of the mixed score. But it creates a machine $\mathcal{G}_{Ti}$ (called top group $\vdash_{\mathsf{top}}$) for each events's no empty s.

$$\frac{\vdash_{\mathsf{evt}} \langle sc{::}ev, sc'\rangle : \mathcal{G}}{\vdash_{\mathsf{evt}} \langle sc, ev{::}sc'\rangle : \mathcal{G}}\ \text{TOP}$$

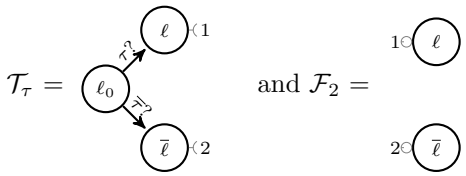
$$\frac{\vdash_{\mathsf{evt}} \langle sc{::}ev, sc'\rangle : \mathcal{G} \qquad 0, ev{::}sc', \tilde{e} \vdash_{\mathsf{top}} sa : \mathcal{G}_{sa}}{\vdash_{\mathsf{evt}} \langle sc, ev{::}sc'\rangle : \mathcal{G} \parallel \mathcal{T}_e + \mathcal{G}_{sa} + \mathcal{F}_i}\ \text{TOP}$$

where $ev = \mathsf{evt}(e, d, \epsilon)$ for the first rule and $ev = \mathsf{evt}(e, d, sa)$ for the second one. Is described here a step of the function ($\vdash_{\mathsf{evt}}$), note the step by passing ev from right to left of the parsed score (sc::sc'). A function ($\vdash_{\mathsf{top}}$), creating the related machine of sa ($\mathcal{G}_{sa}$), is called (if s is not empty). The result is parallelized with the rest of the future groups $\mathcal{G}$ (built by the recursive call).

The second above rule assumes that we have associated a FSM $\mathcal{G}_{sa}$ to a group sa triggered by the event e, using the predicate $\vdash_{\mathsf{top}}$. Implicitly, the group sa is treated like $\mathsf{act}(0, sa, [\mathsf{top}])$ where top is an adhoc strategy of sync and error handling specific to top-level groups.

Input 0 is the initialization of the accumulator, $ev{::}sc'$ the rest of the score from the current event (included) and $\tilde{e} = closest(acc + d, sc)$ the next event after the event trigger. sa is the actions sequence parsed by the function $\vdash_{\mathsf{top}}$. The function *closest* associates to a duration d and a score sc the latest event $e \in \mathcal{I}$ in sc before d in the timeline defined by sc.

The *trigger* $\mathcal{T}_\tau$ associated to the signal or input event $\tau \in \Sigma_{\mathsf{sig}} \cup \mathcal{I}$, and the *finisher* $\mathcal{F}_i$ are defined as follows (for i = 2).



The finisher, set all seeker locations to stop locations. i can vary from 2 to 5 (**@tight @global** for a killable group).

2.6 *al* Groups

We will define here how a FSM is constructed during a parse of an actions sequence. The construction depends on the context, i.e the synchronization and error management attributes (al) but a generic behavior can be defined. The current version of Antescofo defines the specific group top as a **@loose @global** group.

$$\frac{acc+d, sc, closest(acc+d, sc) \vdash_{al} sa : \mathcal{G}_{sa}}{acc, sc, e \vdash_{al} \mathsf{act}(d, a, al')::sa : \mathcal{G}(a, d, al, sc, e) + \mathcal{G}_{sa}}\ \text{MSG}$$

$$\frac{acc+d, sc, \tilde{e} \vdash_{al} sa : \mathcal{G}_{sa} \qquad acc+d, sc, \tilde{e} \vdash_{al'} sa' : \mathcal{G}'_{sa}}{acc, sc, e \vdash_{al} \mathsf{act}(d, sa', al')::sa : \tilde{\mathcal{G}} + \mathcal{G}_{sa} \parallel \mathcal{T}(g') + \mathcal{G}'_{sa}}\ \text{GRP}$$

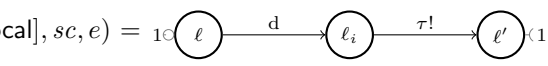
where $|sa'| > 1$, g' is a fresh signal name associated with the group sa', $\tilde{e} = closest(acc + d, sc)$, and $\tilde{\mathcal{G}} = \mathcal{G}(g', d, al, sc, e)$ is a generic function that for a pair of attributes al constructs the corresponding part of FSM (detailed latter by cases).

A step is done by the construction of the FSM-part ($\tilde{\mathcal{G}}$), the accumulator incrementation ($acc + d$) and the update of the closest event ($\tilde{e}$). A group parsing add a function ($\vdash_{al'}$), corresponding to the parsed group attributes, to construct its proper FSM ($\mathcal{G}'_{sa}$), and parallelizes it once terminated. Note that the function is initialized with the current accumulator updated, the rest of the mixed-score and the closest event.

We define the construction of the FSM for each attribute.

2.7 @loose @local

The loose local preserves the music coherence by following the musician only with the current tempo (delay in beats). It can be relevant for music accompaniment when it leads the part. Loose local groups have not late fire condition, if its trigger event is missed the group is not launch.

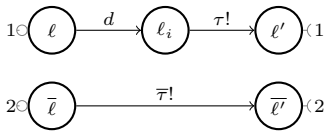


To terminate a **@loose @local** group, $\mathcal{F}_2$ or $\mathcal{F}_3$ for killable groups, is used.

2.8 @loose @global

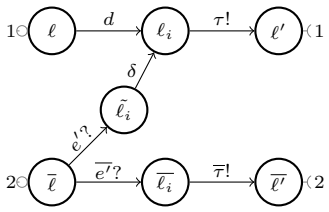
The loose global behavior is quite as the loose local one, but for missed case the detected event (latter than the triggered one) $\mathbf{current}_e$ split the group in two part. The **past** actions (from the trigger to the event) are directly fired, and other are played as a standard loose local group (= top group). The late loose global begin by a list of late actions and check if the next event is missed or not.

$\mathcal{G}(\tau, d, [\mathsf{loose}, \mathsf{global}], sc, e) =$



When $e = closest(acc + d, sc)$, and otherwise, let $e' = closest(acc + d, sc)$,

$\mathcal{G}(\tau, d, [\mathsf{loose}, \mathsf{global}], sc, e) =$

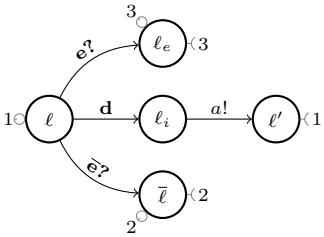


where δ is the difference between $acc + d$ and the date of e' in sc. To terminate a **@loose @global** group, $\mathcal{F}_2$ or $\mathcal{F}_3$ for killable groups, is used.

2.9 @tight @local

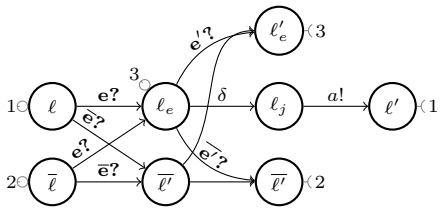
These attributes strictly synchronize the musician events played (during performance) with actions triggers. Tight machines manage standard and missed cases. We need one index more for the earlier events case related to the location l_e.

$\mathcal{G}(\tau, d, [\mathsf{tight}, \mathsf{local}], sc, e) =$



When $e = closest(acc + d, sc)$, and otherwise, let $e' = closest(acc + d, sc)$.

$\mathcal{G}(\tau, d, [\mathsf{tight}, \mathsf{local}], sc, e) =$

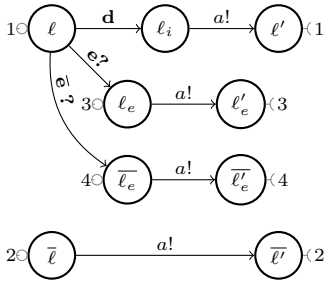


To terminate a **@tight @local** group, $\mathcal{F}_3$ or $\mathcal{F}_4$ for killable groups, is used.

2.10 @tight @global

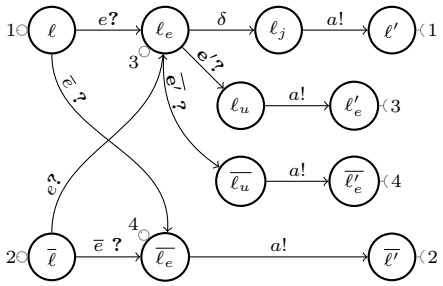
The tight global machines regroup all the specifications of previous groups. We have a great view of what is the Antescofo's behavior on a specific case: from top to bottom a branch specify: standard, earlier standard, earlier missed and missed case.

$\mathcal{G}(\tau, d, [\mathsf{tight}, \mathsf{global}], sc, e) =$



When $e = closest(acc + d, sc)$, and otherwise, let $e' = closest(acc + d, sc)$.

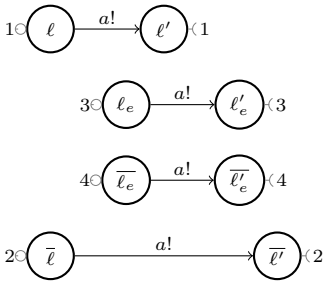
$\mathcal{G}(\tau, d, [\mathsf{tight}, \mathsf{global}], sc, e) =$

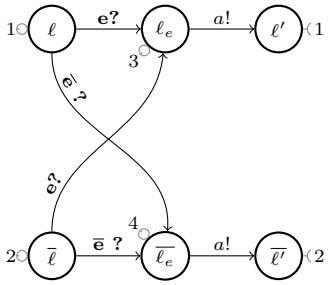


To terminate a **@tight @global** group, $\mathcal{F}_4$ or $\mathcal{F}_5$ for killable groups, is used.

2.11 special cases

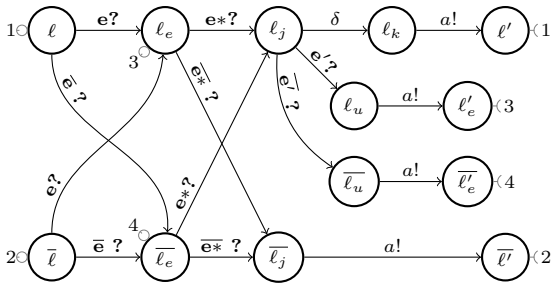
Since actions can have a delay of 0, d and δ transitions can be erased (all parallel awaits (as $e'?$) are deleted too with their branches). An example is done for a **@tight @global** group.


1
ℓ
$a!$
ℓ'
1
3
ℓ_e
$a!$
ℓ'_e
3
4
$\overline{\ell_e}$
$a!$
$\overline{\ell'_e}$
4
2
$\overline{\ell}$
$a!$
$\overline{\ell'}$
2


1
ℓ
e?
ℓ_e
$a!$
ℓ'
1
3
$\overline{e}$?
e?
4
2
$\overline{\ell}$
$\overline{e}$?
$\overline{\ell_e}$
$a!$
$\overline{\ell'}$
2

A jump can be seen for tight groups. It appends when an action delay is greater than a score event duration. 3 events are important for a jump since the previous delay wait on e, $e*$ is waited to know the start of the next action's delay that can have the earlier event e' played.

$\mathcal{G}(\tau, d, [\mathsf{tight}, \mathsf{global}], sc, e) =$



where $\langle e*, e' \rangle = closest(acc + d, sc)$.

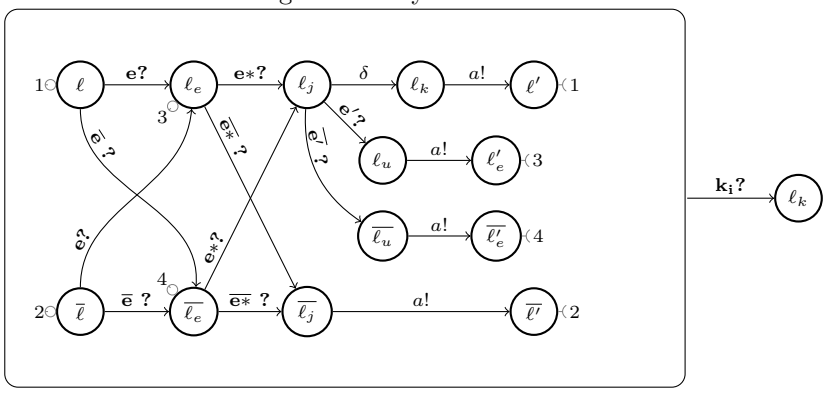
For example for the score:

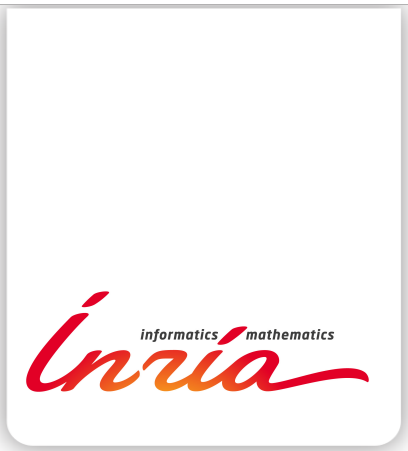
BPM 144	$s_1 =$	$\mathsf{act}(0, s_2, [\mathsf{tight}; \mathsf{global}])$
$\mathsf{evt}(e_1, 1, s_1)$;	$s_2 =$	$\mathsf{act}(0.2, [a_0], [\,])$;
$\mathsf{evt}(e_2, 0.5, [])$;		$\mathsf{act}(3, [a_1], [\,])$;
$\mathsf{evt}(e_3, 0.5, [])$;		
$\mathsf{evt}(e_4, 2, [])$;		
$\mathsf{evt}(e_5, 1, [])$;		

where

we have e = e2, $e*$ = e4, e' = e5, when a1 is parsed.

Kill actions are managed as an action (the difference is the symbol that is an internal symbol (not an output)). The variation is on targeted groups (called killable) that have in parallel an asap transition waiting for the kill symbol. When fired the machine goes directly on a final state "kill".





RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399