Model checking LTL formulae in RAISE with FDR

Abigail Parisaca Vargas[1], Ana G. Garis[2], S. Lizeth Tapia Tarifa[1], and Chris George[3]

[1] San Pablo Catholic University, Arequipa, Peru
[2] University of San Luis, Argentina
[3] International Institute for Software Technology,
United Nations University, Macao

Abstract. The Raise Specification Language (RSL) is a modeling language which supports various specification styles. To apply model checking to RSL concurrent descriptions, we translate RSL specifications into the input language CSPM of FDR. FDR is the model checker for the process algebra CSP. First, we define a syntactic and semantic translation from the concurrent applicative subset of RSL to CSPM, and show that this translation is a strong bisimulation which preserves properties such as traces and deadlock. Consequently, results obtained by refinement checks in FDR are sound for the original RSL descriptions. Second, RSL uses Linear Temporal Logic (LTL) to specify desired properties, but FDR does not support LTL. LTL formulas may be translated to CSP test processes in order to check them with FDR. We build a tool which automates the translation of RSL specifications into CSPM and translates LTL formulas to CSP processes, enabling the model checking of LTL formulas over RSL descriptions with FDR.

Keywords: RAISE, RSL, CSP, FDR, formal methods, model checking, refinement, tools, LTL

1 Introduction

Concurrent systems are increasingly necessary in society. Two characteristics that make concurrent processes difficult to understand are distribution and reactivity. In order to facilitate the modeling and verification of such complicated systems, we need powerful languages and tools than can facilitate the modeling and verification of them.

Although many different kinds of modelling languages and tools are available, it can still be difficult to model and reason about these systems. Combining the use of two or more of these formalisms may be suitable to model a particular system. Moreover, it is also possible to integrate different formalisms to automate the different kinds of tasks that should be done.

In this paper, we describe the steps we followed in order to model check Raise Specification Language (RSL) descriptions using the CSP model checker FDR [10]. Section 2 discusses model checking in the context of RSL.

In Section 3 we describe our approach to develop a translation from RSL [5] to CSP [6, 16, 14], in particular to its machine-readable variant CSP_M. We establish a concurrent applicative subset of RSL that has a translation to CSP_M, then we establish a semantic link between an RSL specification and its CSP_M translation by means of

bisimulation. This translation is, actually, a strong bisimulation that preserves properties such as deadlock and divergence. Hence we show that results obtained by refinement checks in FDR are sound with respect to the original RSL description.

In Section 4 we consider how to assert and prove properties of our models. In model checking by refinement, one has to express the required property in terms of a more abstract process: the property is valid for a model if the property's description as a process is refined by the model. In CSP there are three kind of refinement: traces, failures and failures-divergences. Each one is specially useful for proving different properties; for instance, traces refinement is useful for proving safety properties. It is not always convenient to express properties in terms of more abstract processes, and temporal logic has been the traditional means of stating properties to be model checked.

The relationship between the refinement-based and temporal logic approach was studied by Leuschel, Massart and Currie [9, 8]. They show an approach for doing LTL model checking of CSP specifications using refinement checking in FDR. They present a way to handle deadlocking systems, and discuss the validity for infinite state systems. After analyzing Leuschel, Massart and Currie's approach, we take the general idea and adapt it to translate LTL formulae from RSL to CSP processes.

Section 6 summarizes the achievement: a tool that translates a subset of RSL into CSP_M in order to apply refinement checking techniques over RSL specifications using FDR, and the translation of LTL formulae from RSL descriptions to CSP tester processes, so we can apply the traditional model checking technique to applicative concurrent RSL specifications.

2 Model checking

Model checking is an automatic technique for verifying finite state systems. Enabling model checking often requires building a smaller, more abstract or simplified model of the main design that preserves its essential characteristics but avoids complexity. The idea then is to verify this model.

Applying model checking to a model includes mainly these three tasks: modeling (construction of the model), specification (definition of the properties to be checked) and verification (mechanical checking of the properties against the model). Many formal languages have been used for modeling, with corresponding tools for verification. Some form of temporal logic is the most common language for expressing properties, and the most useful for asynchronous processes is probably LTL [12], which we use.

2.1 Traditional model checking and model checking by refinement

The traditional model checking (MC) method is a verification process which decides whether p holds for M ($M, s \models p$); where M is a model structure with a initial state s and p is a desired property of the system. LTL formulae allow one to specify different requirements of systems; in particular, they are useful to correctly specify safety and liveness properties.

The FDR tool is a refinement checker for CSP models and it supports three models: traces model, failures model and failures/divergences model [10]. *Traces(P)* represents

the set of finite sequences of communications that a process P can perform. *Failures(P)* represents the set of all the pairs (s, X), where 's' is a finite trace of a process P and 'X' is a set of refusals, the events that P can refuse to participate in after doing the trace s. *Divergences(P)* represents the set of failures and also the set of divergences; while or after a divergence happens, P can perform an infinite sequence of consecutive internal actions. The forms of refinement corresponding to the traces, failures and divergences models are called *traces refinement*, *failures refinement* and *failures-divergences refinement* respectively. Traces refinement allows the checking for safety properties and failures refinement the checking of deadlock freedom, while failures-divergences refinement is the most appropriate for checking livelock freedom.

2.2 Model checking in RSL

RAISE is a formal method, with RSL as its specification language. A set of tools [4] is available for RSL, including test coverage analysis and mutation testing, translators to different languages, and a translator to PVS which allows RSL specifications to be proved by the PVS theorem prover. Model checking has been supported using the Symbolic Analysis Laboratory (SAL) as a third party model checker. This also involved adding the possibility to include LTL assertions in RSL specifications.

2.3 Using FDR for RSL

The SAL tool in the RAISE suite does not allow one to model check concurrent RSL specifications. Our purpose then was to translate the RSL applicative concurrent style process models to suitable CSP ones in order to apply tools like FDR which can help us to model RSL processes. This raises two issues

1. Translation: There are syntactic and semantic differences between RSL and CSP which we need to cope with.
2. Specification: We need to find a way to translate LTL into CSP processes.

3 Translation

We want to translate RSL applicative concurrent descriptions to corresponding CSP ones. We describe the translation in syntactic terms, and then establish formally what we mean by "corresponding": we will show that the RSL and CSP descriptions are strongly bisimilar. The details are in a technical report [18]: we give just an overview here.

3.1 The syntax

Since we are interested in the translation of RSL to CSP_M, the variant of CSP accepted as input by FDR, we need to identify the features of RSL that can be expressed in CSP_M. This translation subset includes:

– The built-in types **Bool**, **Int** and **Nat** as shown in Table 1.

Language	Type	Values	Operators
RSL	**Bool**	**true**, **false**	$\wedge, \vee, \sim, =, \neq$
CSP_M	Boolean	$true, false$	$\wedge, \vee, \neg$, $==, != $
RSL	**Int**	...,-1,0,1...	$+, -, *, /, \backslash, <, \leq, >, \geq$
CSP_M	Number	...,-1,0,1...	$+, -, *, /, \%, <, \leq, >, \geq$
RSL	**Nat**	0,1...	$+, -, *, /, \backslash, <, \leq, >, \geq$
CSP_M	Open range set	$\{0..\}$	$+, -, *, /, \%, <, \leq, >, \geq$

Table 1. Built-in types translation from RSL to CSP_M

– The compound types **Product**, **Set**, **List**, **Subtype**, **Variant** and **Record** as shown in Table 2. Since complex variants and records in RSL provide an implicit constructor, destructors, and (optionally) reconstructors, equivalent functions to these features have to be created during the translation to CSP_M.

Language	Type	Example definition	Example value	Operators
RSL	**Product**	**type** Position = **Int** × **Int**	(-6,5)	
CSP_M	Tuple		(-6,5)	
RSL	**Set**	**type** IntegerSet = **Int-set**	$\{1..5\}$	$\cup, \cap, \backslash, \in$, **card**
CSP_M	Set		$\{1..5\}$	$\cup, \cap, -, \in, card$
RSL	**List**	**type** IntegerList = **Int***	$\langle 1, 2, 3\rangle$	**hd**, **tl** , $\frown$, **len**
CSP_M	Sequence		$\langle 1, 2, 3\rangle$	$head, tail, \frown, \#$
RSL	**Subtype**	**type** Gun = {| i : **Nat** • i ∈ {0..15} |}		
CSP_M	Close range set	$nametype\ Gun = \{0..15\}$		
RSL	**Variant**	**type** ComplexColour == RGB(Red : Gun, Green : Gun, Blue : Gun) | Black | White		
CSP_M	Data type	$datatype\ ComplexColour = RGB.Gun.Gun.Gun \mid Black \mid White$ $Red(RGB.vRed.vGreen.vBlue) = vRed$ $Green(RGB.vRed.vGreen.vBlue) = vGreen$ $Blue(RGB.vRed.vGreen.vBlue) = vBlue$		
RSL	**Record**	**type** Figure :: S : Shape C : ComplexColour		
CSP_M	Data type	$datatype\ Figure = mk\_Figure.Shape.ComplexColour$ $S(mk\_Figure.vS.vC) = vS$ $C(mk\_Figure.vS.vC) = vC$		

Table 2. Compound types translation from RSL to CSP_M

– Explicit constant values of built-in types and compound types.
– Simple channels and channel arrays as shown in Table 3.
In RSL a channel array is expressed through an object array and it is translated to a complex protocol channel in CSP_M.
– Explicit function and process with **if**, **case** and/or **let** expressions as shown in Table 4.
CSP_M does not support elsif and case process expressions, but they can be simulated using If and Let expressions in the translation to CSP_M.
– The communication primitives sequence, internal and external choice, comprehended internal and external choice, parallel and comprehended parallel; and the

Channel	Language	Example definition
Simple channel	RSL	**channel** mess : Index × Data
	CSP_M	$channel\ mess : (Index, Data)$
Channel array	RSL	**object** fork[p : Index, f : Index] : **class channel** pickup, putdown : **Unit end**
	CSP_M	$channel\ fork\_pickup, fork\_putdown : Index.Index$

Table 3. Channel translation from RSL to CSP_M

Expression	Language	Example
If	RSL	**if** x > y **then** x − y **else** y − x **end**
	CSP_M	if $x > y$ then $x - y$ else $y - x$
Let	RSL	**let** p = input? **in let** (x,y) = p **in** output!x+y; PROC_PLUS() **end end**
	CSP_M	$input?p \rightarrow \mathsf{let}\ (x, y) = p\ \mathsf{within}\ \ output!x + y$ $\rightarrow PROC\_PLUS$
Case	RSL	**case** p **of** (**true**,**false**) → **true**, _ → **false** **end**
	CSP_M	$if\ let\ (x1\_, x2\_) = p$ $\ \ within\ x1\_ == true\ and\ x2\_ == false$ $then\ let\ (x1\_, x2\_) = p$ $\ \ within\ true\ else\ false$

Table 4. Function and process expression translation from RSL to CSP_M

basic processes **stop** and **skip**. To find an equivalence between these communication primitives and basic processes in RSL and CSP_M, it is necessary to evaluate the operational semantics of these two languages.

3.2 The semantics

In this section we compare the semantics of the subsets of RSL and CSP.

The operators of RSL and CSP Based on the grammar of the process expression in both process algebras, we show the operators which can be translated in both cases.

The RSL expressions are as follows:
$P =$ **skip** | **stop** | $E;\ P$ | $P \sqcap P$ | $P \Box P$ | $P \| P$ | **if** v **then** P **else** P **end**
$E =$ c? | c!v
where v is a pure value expression, and P is a process expression.

The CSP expressions are as follows:
$P = SKIP \mid STOP \mid E \rightarrow P \mid P \sqcap P \mid P \Box P \mid P \| P \mid if\ v\ then\ P\ else\ P$
$E = c? \mid c!v$
where v is a pure value expression, and P is a process expression.

Although the operators are syntactically similar, they are sometimes semantically different as discussed below.

Operational Semantics Comparison The operational semantics rules are taken from [1] in the case of the RSL rules and from [16] in the case of the CSP rules.

We do not include the details here, but simply state that the internal and external choice combinators have equivalent semantics in the two languages. The differences lie in parallelism, essentially since CSP adopts a "broadcast" semantics to communication between parallel processes, while RSL adopts a "point-to-point" semantics.

We consider two cases for parallel processes : synchronization and non-synchronization.

Synchronization The operational semantics rules for synchronization of RSL and CSP are the following:

RSL	**CSP**
$\frac{\rho \vdash \mathrm{P} \xrightarrow{a} \mathrm{P}' , \quad \mathrm{Q} \xrightarrow{\bar{a}} \mathrm{Q}'}{\rho \vdash \mathrm{P} \| \mathrm{Q} \xrightarrow{\tau} \mathrm{P}' \| \mathrm{Q}'}$ (1)	$\frac{P \xrightarrow{a} P' , \quad Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P' \| Q'}$ (2)

where if a is an input (c?x) then $\bar{a}$ is an output on the same channel (c!v), and vice versa. Note that in RSL the event is "consumed" by the synchronization, becoming a τ. The CSP rule is more general, in that both events may be inputs or outputs. The event in CSP is not consumed; other processes running in parallel may also participate in it. We see that the two rules will coincide only for matched inputs and outputs, and provided we hide synchronized events as they occur.

To deal with this problem, we adopt the following as a *design rule*:

A process can only either input or output on a channel, and at most one other process can access that channel, and the access is in the opposite direction.

This rule may seem restrictive, but is in fact advised by the RAISE Method [13], and is natural in a language with point-to-point communication. If we adopt this rule for RSL then we see that such RSL translated in the natural manner to CSP will produce CSP processes in which the two sub cases where both events are inputs, or both are outputs, cannot occur.

This rule is statically checkable provided there are no channel arrays.

Non-synchronization A non-synchronised transition between parallel processes occurs when one process makes a transition and the other does not. The relevant semantics rules for CSP and RSL show that for an internal event the rules are the same. But for a visible event CSP alone requires that the event involved in the transition of one process is not in the alphabet of the other. There is no such restriction in RSL. This is necessary in a language like RSL with point-to-point communication to preserve the associativity of the parallel operator. Suppose, for example, that P can output on channel c, and both Q and R can input on c. Then the combination (P ‖ Q) ‖ R must be able to progress by P communicating with either Q or R, and for P to communicate with R, (P ‖ Q) must be able to output on c, without Q being involved.

We can see that our design rule takes care of this problem by not allowing such a parallel combination: we cannot have both Q and R inputting on channel c.[1]

[1] We can also see that our design rule does not remove the possibility of such an architecture. We replace c by two channels cpq and cpr, say, so that Q inputs on cpq and R on cpr, and P

So we adopt the following rule for translating parallel processes:

$$(P \| Q)_T = (P_T \| Q_T) \backslash \alpha P_T \cap \alpha Q_T$$

where X_T is the CSP process translated from the RSL process X.

Soundness Soundness means establishing that the results obtained from tools applied to the CSP model are valid for the original RSL. So we need to establish the following proof rule:

$$\frac{R_T \models P_T}{R \vdash P}$$

where we want to prove property P of RSL specification R, translated to P_T and R_T respectively.

FDR is essentially a refinement checker, so P_T is typically a (traces, failures, or failures-divergence) refinement relation, but may also be an assertion of deadlock freedom. We can therefore establish soundness by establishing the following:

1. that the translation scheme is a strong bisimulation. We need strong rather than weak bisimilarity to include divergence as a property that is preserved.
2. that strongly bisimilar processes have the same traces, failures, divergences and deadlocks.

The details are in a technical report [20]: we give a brief summary here.

Bisimulation We first establish the obvious mapping of the events and basic processes of RSL and CSP. We then proceed by structural induction over the syntax of RSL processes: for each construction we assume the component processes are bisimilar to their translations, and show the constructed process to be bisimilar to its translation.

For example, to prove bisimilarity for the parallel combinator, we assume P is bisimilar to its translation P_T, and Q is bisimilar to its translation Q_T. We then show that $P \| Q$ is bisimilar to its translation (given above): call this PQ_T. To show bisimilarity we consider each possible transition of $P \| Q$ according to the RSL operational semantics to process X, say, and show there is a corresponding transition for PQ_T in the CSP operational semantics to a process which is bisimilar to X. Then we do the converse, considering each possible transition of PQ_T, finding a corresponding transition for $P \| Q$, and showing the resulting processes are bisimilar.

Properties We prove the following for a process P and its (strongly bisimilar) translation P_T:

1. Traces: P can do a trace l iff P_T can do a trace l.
2. Deadlock: P can deadlock iff P_T can deadlock.

makes an internal (non-deterministic) output choice between cpq and cpr. This is semantically equivalent to the original system (assuming no other processes access c) and obeys our design rule.

3. Refusals: x is a refusal of P iff x is a refusal of P_T
4. Failures: $failures$(P) = $failures(P_T)$.
5. Divergences: P diverges iff P_T diverges.

This means that properties we can prove of translated CSP scripts using FDR (lack of deadlock, trace-, failures- or failures-divergence-refinement) must also be true of the original RSL descriptions. In other words we have shown that FDR is a *sound* model checker for applicative concurrent RSL descriptions.

4 Specification

In this section, we deal with the modelling of LTL formulae. We first explain how LTL formulae may be modelled as tester processes in CSP, following the approach of Leuschel, Massart, and Currie [9, 8]. We then discuss how this approach can be adapted to the RSL setting.

4.1 A translation of LTL formulae to CSP

After a careful study of the relationship between the refinement-based approach and temporal logic, Leuschel, Massart and Currie [9, 8] propose to make a general solution building a tester for each possible LTL formula. Looking at the possible LTL formulae they observe that in general infinite traces have to be tested in order to infer whether a formula is satisfied. They check the satisfaction of a LTL property following the procedure defined by Vardi and Wolper [19]; that is, verifying that $[[S]]_w \cap [[\neg\phi]]_w = \emptyset$ (where $[[S]]_w$ represents all the traces of the system′s model and $[[\neg\phi]]_w$ all the traces of the negation of a LTL formula). If the intersection between $[[S]]_w$ and $[[\neg\phi]]_w$ is empty then $S \models \phi$.

The approach consists of building a tester T_ϕ from a formula ϕ, composing T_ϕ with a system S, and checking if the composition satisfies some property. T_ϕ is built by translating ϕ to the corresponding Büchi automaton and translating this automaton to CSP; finally, FDR is used for checking emptiness. Special attention is paid to deadlocking traces, so they build a tester T_ϕ which accepts infinite traces and also deadlocking traces.

Deadlocking treatment and tester building An extended LTL called LTL$_\Delta$ is defined in order to handle deadlocking traces. LTL$_\Delta$ is specified in the same way as LTL but over an extended alphabet. That is, while LTL is defined over Σ, LTL$_\Delta$ is over $\Sigma \cup \{\Delta\}$; where $\Delta \notin \Sigma$. If a valid trace π is finite then it is over Σ terminating on infinite Δ's; otherwise π is an infinite trace over Σ. Regarding the semantics of LTL$_\Delta$, two rules are defined in [9, 8] for translating LTL into LTL$_\Delta$:

1) $X\phi \quad \rightsquigarrow \quad \neg\Delta \wedge X\phi$
2) $\neg X\phi \quad \rightsquigarrow \quad \Delta \vee X\neg\phi$

Also, the definition of when an LTL$_\Delta$ formula holds is shown. Given a system specification S and a LTL$_\Delta$ formula ϕ:

$S \models \phi$ iff $\forall \pi \in [[S]]_\Delta, \pi \models \phi$,
where $[[S]]_\Delta = [[S]]_w \cup \{\gamma \Delta^w \mid (\gamma, \Sigma) \in failures(S)\}$

Following that idea, and considering that CSP system S cannot extend its traces to consider deadlocking traces, a tester T_ϕ of the LTL formula ϕ is built. T_ϕ accepts, on the one hand, infinite traces; and on the other hand, deadlocking traces. The tester is built from a Büchi automaton using the classical approach. Therefore, given an LTL formula to check, first it is negated, second it is translated to LTL$_\Delta$, and finally it is translated to a special Büchi automaton called B_Δ. B_Δ extends the traditional Büchi automaton, adding acceptance conditions to manage deadlocks. More formally, B_Δ is defined in [9] as $\mathbf{B_\Delta} = (\mathbf{\Sigma}, \mathbf{Q}, \mathbf{T}, \mathbf{Q^0}, \mathbf{F}, \mathbf{D})$ where: Σ is the alphabet, Q is the set of states, $T \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q^0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of infinite trace accepting states and $D \subseteq Q$ is a set of deadlock monitor states.

B_Δ has two acceptance conditions, the classical acceptance condition for infinite traces and acceptance conditions for deadlocking traces. The traditional Büchi automaton B over the alphabet $\Sigma \cup \{\Delta\}$ is modified into B_Δ over the alphabet Σ by:

1. identifying **deadlock monitor states** (DMS) which are reachable from an initial state by transitions in Σ and accept strings Δ^w with the classical Büchi condition,
2. removing all Δ transitions,
3. removing all transitions and states which do not lead to the acceptance of a trace.

Translation from B_Δ to CSP Each state of B_Δ is translated to a CSP process with different characteristics depending on what kind of state it is. If it is an accepting state, a CSP process with a special *success* action is created; and if it is a DMS, a special Δ transition is added. Therefore, translation from B_Δ automaton into CSP is defined as:
- map every $q \in Q$ to a CSP process name $NAME(q)$
- for every $q \in Q^0$ add the CSP definition $TESTER = NAME(q)$,
- for every non-accepting state $q \in Q \diagdown F$ and
  for all outgoing edges $(q, a, q') \in T$
  add the CSP definition $NAME(q) = a \rightarrow NAME(q')$
- for every accepting state $q \in F$ where
  $\{(q, a_1, q_1), ..., (q, a_n, q_n)\} \subseteq T$ are all the outgoing edges of q
  add the CSP definition
  $NAME(q) = success \rightarrow (a_1 \rightarrow NAME(q_1) \,\Box\, ... \,\Box\, a_n \rightarrow NAME(q_n))$
- for every state $q \in D$
  add the CSP definition $NAME(q) = deadlock \rightarrow DEADLOCK$
- add a single CSP definition of $DEADLOCK$ (where $\Sigma = \{a_1, ..., a_n\}$)
  $DEADLOCK = a_1 \rightarrow k_0 \rightarrow STOP \,\Box\, ... \,\Box\, a_n \rightarrow k_0 \rightarrow STOP$

Note that $success$, $deadlock$ and k_0 are all different and not in Σ.

If the system is not deadlocked in a deadlock monitor state then the system in parallel with the $DEADLOCK$ process will be able to perform some action in Σ and then the action k_0. On the other hand, if the system is deadlocked in a deadlock monitor state, the $DEADLOCK$ process will not be able to perform k_0, so deadlocking traces will correspond to CSP failures.

Checking emptiness using FDR The final step to verify that $[[S]]_w \cap [[\neg\phi]]_w = \emptyset$ is to check whether an infinite trace or a finite deadlocking trace of S satisfies $\neg\phi$; if there does not exist such a trace then $S \models \phi$. The way to do this check using FDR is through two kinds of refinement check.

- Traces which generate infinite successes are checked by testing whether: $SUC \sqsupseteq_T (S[[\Sigma]]\, TESTER) \setminus (\Sigma \cup \{deadlock, k0\})$ holds, where $SUC = success \rightarrow SUC$.
- Deadlocking acceptances are checked by testing whether: $deadlock \rightarrow STOP \sqsupseteq_F (S[[\Sigma]]\, TESTER) \setminus (\Sigma \cup \{success\})$ holds

If one of the tests succeeds then $S \not\models \phi$, so $S \models \phi$ only if both of them fail.

4.2 An approach to translate LTL formulae from RSL to CSP

After analyzing the approach presented in [9] and [8], we take the general idea and we adapt it in order to translate LTL from RSL to FDR. In the next subsections, our framework to translate LTL properties from RSL to FDR is detailed.

LTL specification from RSL to LTL$_{\Delta}$ We define a grammar for writing LTL assertions in RSL. LTL properties are preceded by the key word "ltl_assertion" and the definition of each property is written using an identifier tag, the process which will be tested, and the LTL property.

LTL_prop_decl ::= "ltl_assertion" {LTL_assertion}$^+$,
LTL_assertion ::= "[" LTL_tag "]" Process_name $\vdash$ LTL_prop
LTL_prop ::= Channel_name
| LTL_prefix LTL_prop | LTL_prop LTL_infix LTL_prop
| "true" | "false" | "(" LTL_prop ")"
LTL_prefix ::= "X" | "G" | "F" | "$\sim$"
LTL_infix ::= "R" | "U" | "$\vee$" | "$\wedge$" | "$\Rightarrow$"

'Process_name' and 'Channel_name' are type identifiers defined previously for a process and a channel respectively. 'LTL_tag' is also an identifier, more precisely an LTL property identifier. For example, consider the following RSL specification:

scheme VENDOR_MACHINE =
class
channel
rich,coin, choc, toff,smile: **Unit**
value
Machine: **Unit** $\rightarrow$ **in** coin **out** choc, toff **Unit**
Machine() $\equiv$ coin?;(choc!();Machine()$\,[]\,$toff!();Machine()),

Customer: **Unit** $\rightarrow$ **out** coin,smile **in** rich, choc, toff **Unit**
Customer() $\equiv$ rich?;coin!();(choc?;smile!();Customer()$\,|\,$toff?;smile!();Customer()),

SYS: **Unit** → **in** coin,choc,toff,smile,rich **out** coin,choc,toff,smile **Unit**
SYS() ≡ Machine()‖Customer()
ltl_assertion
[happy] SYS ⊢ G(rich ⇒ F(smile))
end

An ltl_assertion called "happy" is defined over the process "SYS" and the property is specified using channels "rich" and "smile". An occurrence of a channel name in an ltl_assertion indicates the corresponding event occurring. So "happy" asserts that whenever a "rich" event occurs a "smile" will eventually occur — a classical liveness property.

The translation from RSL to LTL$_\Delta$ requires that alphabets are defined by extension, i.e. as finite sets. So LTL properties only can be defined using models involving simple channels (not channel arrays).

Negating LTL properties We want to verify that $[[S]]_w \cap [[\neg\phi]]_w = \emptyset$ (where $[[S]]_w$ represents all the traces of the system's model and $[[\neg\phi]]_w$ all the traces of the negation of a LTL formula). So, the first step on the way to translate a LTL formula to LTL$_\Delta$, is the negation of the property by means of the introduction of the operator '$\sim$'. For this first translation we use standard rules such as $\sim$ G ϕ = F($\sim \phi$).

Observe that the negation of a event 'x' it is equivalent to concatenation by '$\vee$' of each alphabet's symbol, less x and plus the Δ symbol. This is because we know we have CSP events where exactly one event from the alphabet happens at any step. For the same reason, it is not possible to have more that one event concatenated by the '$\wedge$' operator either. For instance, if the alphabet is {a,b,c} then

$\sim$a = b $\vee$ c $\vee$ Δ *and* $\sim\Delta$ = a $\vee$ b $\vee$ c *and* a $\wedge$ b = false *and* a $\wedge$ Δ = false

These rules enable us to remove the negations from any formula. (We will often for clarity leave $\sim\Delta$ unexpanded in the presentation.)

Introducing the special symbol Δ Taking as models the rules defined in [9] for translating the LTL properties $X\phi$ and $\neg X\phi$ into LTL$_\Delta$ (see subsection 4.1), we analyzed the semantics for each LTL$_\Delta$ operator. Therefore, we specify a translation T of every LTL operator of a formula ϕ as follows:

T(G(ϕ)) = G(Δ) R T(ϕ)	T($\phi \wedge \psi$) = T(ϕ) $\wedge$ T(ψ)
T(F(ϕ)) = $\sim\Delta$ U T(ϕ)	T($\phi \vee \psi$) = T(ϕ) $\vee$ T(ψ)
T(X(ϕ)) = $\sim\Delta \wedge$ X(T(ϕ))	T($\phi \Rightarrow \psi$) = T(ϕ) $\Rightarrow$ T(ψ)
T(ϕ U ψ) = ($\sim\Delta \wedge$ T(ϕ)) U T(ψ)	T(a) = a, where $a \in \Sigma$
T(ϕ R ψ) = (G(Δ) $\vee$ T(ϕ)) R T(ψ)	T(Δ) = Δ

Considering the VENDOR_MACHINE example shown previously, the translation for the LTL assertion *happy* after the negation of the LTL property and the introduction of the symbol Δ is as follows:

T($\sim$(G(rich ⇒ F(smile)))) = $\sim\Delta$ U (rich $\wedge$ (G(Δ) R ($\Delta \vee$ rich)))

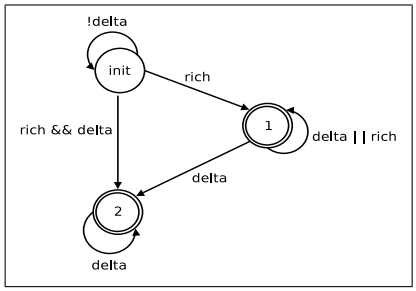

Fig. 1. Büchi automaton for the *happy* assertion

Translation from LTL$_{\Delta}$ to Büchi automata The SPIN model checker [7] is used in [9] to translate LTL$_{\Delta}$ to Büchi automata. Instead of SPIN, we use *ltl2ba* [2] to generate the Büchi automaton from an LTL$_{\Delta}$ expression. We choose ltl2ba because it is open source, which allows us to extend it. In addition, experimental work shows that it is more efficient than SPIN [3].

The ltl2ba algorithm generates a Büchi automaton from an LTL formula. First a very weak alternating automaton is built and then it is transformed it into a Büchi automaton, using a generalized Büchi automaton. Each automaton is simplified on-the-fly for saving memory and time, using iteratively three rules until no more simplification are possible [3]:

1. Inaccessible states are removed.
2. If a transition t_1 implies a transition t_2, then t_2 is removed.
3. If states q_1 and q_2 are equivalent, then they are merged.

The Büchi automaton for the *happy* assertion is shown in figure 1.

Translation from Büchi automata to Büchi delta B_{Δ} We took the source code of *ltl2ba* and extended it to generate B_{Δ} from the Büchi automaton. B_{Δ} is obtained following the steps shown in subsection 4.1, that is: 1) identifying each DMS, 2) removing all Δ transitions and finally 3) removing transitions and states which do not lead to the acceptance of a trace.

Regarding detection of DMS, in [9] they are found following an algorithm defined in [17] (adaptation of the Tarjan's search algorithm for strongly connected components). However, this one only detects states which accept strings Δ^w with the classical Büchi condition, but it does not consider which are reachable from an initial state by transitions in Σ. Therefore, we defined a complementary algorithm which detects states reachable by Σ transitions.

Consider the Büchi automaton corresponding to the VENDOR_MACHINE example. Only state 1 is detected as DMS, because it is reachable from the initial state by transitions in Σ and accepts the string Δ^w with the classical Büchi condition. Also, Δ transitions are removed, according to step 2) for building B_{Δ}. The transition labeled 'rich && delta' is removed too, because we know we have CSP events where exactly

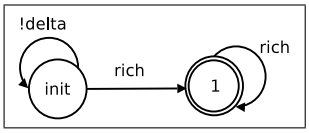

Fig. 2. B_Δ automaton for the *happy* assertion

one event from the alphabet happens at any time. Therefore, we get the B_Δ automaton shown in figure 2.

Translation from B_Δ to CSP and emptiness check After building B_Δ, we use the algorithm of section 4.1 for generating the corresponding CSP specification from B_Δ. Also, we generate FDR code for the two refinement checks as explained in subsection 4.1; i.e. checking traces which generate infinite successes and checking for deadlock.

5 Implementation of the tool

The approach described in this paper has been implemented in a tool which translates RSL specifications with temporal logic assertions into a CSP_M specification and a tester process.

In order to do so it is set up with two main components that we will call "RSL_FDR2" and "RSL_LTL_FDR2".

RSL_FDR2 takes an RSL specification as an input and performs two things. First, the RSL is transformed into an AST (abstract syntax tree) by the RSL type checker; then, applying many translation rules, the AST of RSL is transformed into an AST of CSP_M; and finally this AST is unparsed into a new output script (a `.fdr2` file) in CSP_M. Second, if one or more LTL assertions are specified (using RSL syntax), RSL_FDR2 translates them to the corresponding LTL_Δ formulae and saves them in `.ltl` files. These `.ltl` files are input to RSL_LTL_FDR2, an extension of ltl2ba, which generates for each a `TESTER` process and some CSP refinement statements, and appends them to the `.fdr2` file.

For instance, when we give the VENDOR_MACHINE example of Section 4 to the tool, it produces the following CSP_M script as output:

```
channel rich,coin,choc,toff,smile

Alph_in_Machine = {|coin|}
Alph_out_Machine = {|toff,choc|}
Machine = coin -> (choc -> (Machine) [] toff -> (Machine))

Alph_in_Customer = {|toff,choc,rich|}
Alph_out_Customer = {|smile,coin|}
Customer = rich -> (coin -> (choc -> (smile -> (Customer)) |~|
                             toff -> (smile -> (Customer))))

Alph_in_SYS = {|rich|}
```

```
Alph_out_SYS = {|smile|}
SYS = (Machine [{|coin,toff,choc|}||
                {|toff,choc,rich,smile,coin|}] Customer)\
                                              {|coin,toff,choc|}

channel success0,deadlock0,k0
Alph_SYS0 = union(Alph_in_SYS,Alph_out_SYS)

TESTER0 = State0_0
State0_0 =
  rich?x -> State1_0 [] rich?x -> State0_0 [] smile?x -> State0_0
State1_0 =
  success0 -> ( rich?x -> State1_0 ) [] deadlock0 -> DEADLOCK0
DEADLOCK0 = rich?x -> k0 -> STOP [] smile?x -> k0 -> STOP
Composition0 =
  (SYS [|Alph_SYS0|] TESTER0)\ union(Alph_SYS0,{deadlock0,k0})
DComposition0 =
  (SYS [|Alph_SYS0|] TESTER0)\ union(Alph_SYS0,{success0})

SUC0 = success0 -> SUC0
assert Composition0 [T= SUC0
RealDeadlock0 = deadlock0 -> STOP
assert DComposition0 [F= RealDeadlock0
```

The Composition0 and DComposition0 assertions are checked using FDR to determine if the property holds. Compostion0 checks for traces which generate infinite successes and Dcompositon0 checks for deadlock. Since both assertions fail, it is established that SYS satisfies the property G(rich $\Rightarrow$ F(smile)).

5.1 Efficiency

As suggested by the fact that there is a bisimulation between the RSL description and its translation, there is a one-to-one relation between the events in RSL and those in CSP, and the translation is almost certainly as good and as efficient in model checking as a hand translation would be. This has been borne out by trying the translator on a number of standard examples: cyclic scheduler, dining philosophers, railway crossing, producer-consumer, alternating bit protocol, multiplexed buffer [18].

6 Conclusions

We have shown an approach to translate a concurrent applicative subset of RSL into CSP_M, and shown the soundness of the translation through establishing a strong bisimulation. We have analyzed the approach presented in [9] and [8] and we have observed it is possible to take the general idea but it is necessary to extend it in order to translate from RSL to CSP. Therefore, we have shown the whole translation of every LTL operator to LTL_Δ operators, we have used *ltl2ba* algorithm to translate from LTL expressions to Büchi automata and we have shown how to extend *ltl2ba* to build B_Δ.

Finally, we have developed a tool for the specification of concurrent systems that allows us, first, to use the FDR tool on the CSP_M scripts, and to draw sound conclusions about the RSL descriptions and second, to translate LTL formulas from RSL to CSP that helps us to express the specification of desired properties in a friendly way enabling the model checking of LTL formulae about RSL descriptions with FDR.

A problem with this approach is that it needs failure of model checking to prove success. So if the proposed property is not proved, there is only model checking success and no trace to indicate what went wrong. By including a notion of fairness in the model checking it may be possible to prove LTL properties more directly, as hinted by Roscoe [15] and adopted in recent work in Singapore [11].

References

1. M. Debabi. The RSL semantic course, 1993.
2. P. Gastin and D. Oddoux. ltl2ba tool. http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php.
3. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation, 2001. In Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), Paris, France, July 2001, LNCS 2102, pages 53-65.
4. Chris George. RAISE Tools User Guide. Technical Report 227, UNU-IIST, P.O. Box 3058, Macau, February 2001. The tools are available from http://www.iist.unu.edu.
5. The RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, 1992.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
7. Gerard J. Holzmann. The SPIN Model Checker, 2003. Addison-Wesley.
8. Michael Leuschel, Thierry Massart, and Andrew Currie. How to Make FDR Spin LTL Model Checking of CSP by Refinement, 2001. Journal Lecture Notes in Computer Science, volumen 2021, pages 99+.
9. Michael Leuschel, Thierry Massart, and Andrew Currie. How to Make FDR Spin LTL Model Checking of CSP by Refinement, September 2000. Technical Report, Dependable Systems and Software Engineering Research Group, School of Electronics and Computer Science, University of Southampton, England.
10. Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 User Manual, 2005.
11. PAT: Process Analysis Toolkit. http://www.comp.nus.edu.sg/~pat/.
12. A. Pnueli. The temporal logic of concurrent programs, 1981. Theoretical Computer Science, 13:45-60.
13. The RAISE Method Group. *The RAISE Development Method*. Prentice-Hall International, 1995.
14. A. W. Roscoe. *Model-checking CSP*, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.
15. A. W. Roscoe. Compiling Shared Variable Programs into CSP. In *Proceedings of PROGRESS workshop 2001*, 2001.
16. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 2005.
17. R. Sedgewick. Algorithms in C++, 1992. Addison-Wesley.
18. Lizeth Tapia and Chris George. Model Checking Concurrent RSL with CSPM and FDR2. Research Report 393, UNU-IIST, P.O.Box 3058, Macau, April 2008.
19. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification, 1986. In Proccedings of LICS6, pages 332-344.
20. Abigail Parisaca Vargas and Chris George. Formalising the translation from RSL to CSP. Research Report 395, UNU-IIST, P.O.Box 3058, Macau, May 2008.