Guarded Action Language

Yann Thierry-Mieg[1]

LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6
4 place Jussieu, F-75252 Paris Cedex 05, France `yann.thierry-mieg@lip6.fr`

Abstract. This paper presents the Guarded Action Language, a simple yet expressive and compact language to represent the semantics of concurrent specifications. GAL is natively supported by ITS-tools, an efficient symbolic model-checker supporting both CTL and LTL. GAL is designed to be the target in a transformation process from diverse formalisms. This document is a reference for the actual semantics.

1 Introduction

We present the Guarded Action Language (GAL), specifically designed to easily express a wide range of concurrent semantics. It is supported natively by the efficient symbolic model-checker ITS-tools. GAL defines no high-level concepts (no explicit notion of process, channel, . . .), but it offers a lot of flexibility when defining the atomicity of operations and compactly expresses non determinism.

We first give an overview of our toolset and its model-checking features in section 2. We then formally define GAL and its semantics, then its parametric features in section 3.

2 Tool principles

The *ITS-tools* form a toolset allowing symbolic model-checking (LTL, CTL, reachability) for a variety of input formalisms. It supports a composition model called Instantiable Transition Systems (ITS [16]), as well as a variety of Petri net extensions or dialects, and their compositions. We focus in this paper on GAL, a new input language, both flexible and expressive, for the *ITS-tools*.

2.1 Back-end: ITS-tools

The *ITS-tools* rely on Data Decision Diagrams (DDD [7]) and hierarchical Set Decision Diagrams (SDD [8]) to provide efficient representations of sets of states. So far, GAL is encoded in DDD only. Operations on these decision diagrams are encoded using homomorphisms, which the library can automatically and dynamically rewrite to produce saturation effects in least fixpoint computations [11] and to enable evaluation of complex expressions [5].

The tool *its-reach* can compute reachable states, and shortest witness paths (one or more if so desired) to target states designated by a boolean predicate. It can also perform bounded depth exploration of a state space (a.k.a. bounded model-checking). It implements several heuristics to compute a static variable order for the input model.

The tool *its-ctl* performs verification of CTL properties (though fairness constraints are currently not supported). It reuses a component of VIS [3] to transform input formulae into forward CTL form [12]. Forward CTL often allows (but not always) to use the forward transition relation alone, which is easier to compute than the backward (predecessor) transition relation. When a statement destroys information (i.e. is not reversible), backward exploration requires to compute potential domains for variables, based on the set of reachable states. This leads to an over-approximation, whose refinement is costly as it requires to intersect decision diagrams (see [4] for more details). Hence forward CTL verification is more efficient in general, and furthermore many subproblems can be solved using least fixpoints (e.g. Forward Until) that benefit from saturation at DD level.

The tool *its-ltl* performs hybrid (i.e. that build an explicit graph in which each node stores a set of states as a decision diagram) or fully symbolic verification of LTL properties. The transformation of the formula into a Büchi automaton and the emptiness checks of the product for hybrid approaches rely on Spot [13], considered one of the best tools for this job [14]. Fully symbolic model-checking uses forward variants of Emerson-Lei [10] or One-Way Catch Them Young [15]. The hybrid approaches efficiently exploit saturation and often outperform fully symbolic ones [9]. When the property is stuttering invariant (e.g. $LTL \setminus X$) we also offer optimized fully symbolic algorithms that exploit saturation [2].

The *ITS-tools* is free software and was first released in 2009, and it contains roughly 170 klocs of C++. Through our continuous integration platform, we package and distribute up-to-date binary distributions for common platforms (MacOS 32 or 64, Linux 32 or 64, Windows 64). A set of user-friendly Eclipse plugins embedding the *ITS-tools* is also freely available. It includes an Xtext based rich editor (on-the-fly syntax check, auto-complete, quickfix...) for textual GAL models (and now Timed Automata in Uppaal XTA format) as well as a graphical tool for drawing Petri nets, and a wrapper invoking the *ITS-tools* from Eclipse.

To ease the interaction with this efficient symbolic back-end, we recently focused on building a general-purpose language to model concurrent specifications with data (arrays, arithmetic...). The bridge between this language and the back-end notably relies on the expressive symbolic operations defined in [5].

2.2 Front-end: Guarded Action Language

To allow the *ITS-tools* to capture several formalisms we define a simple yet expressive general-purpose language, that essentially describes a generator for a finite Kripke structure, with as few assumptions as possible on existence of high-level concepts such as processes or channels. As such, although direct modeling in GAL is possible, it is mainly intended to be the target of a model transformation from a language closer to the end-users.

This choice is aligned with current software engineering trends. Model-driven engineering (MDE) proposes to define domain specific languages (DSL), which contain a limited set of domain concepts [17]. This input is then transformed using model transformation technology to produce executable artifacts, tests, documentation or to perform specific validations. In this setting, GAL is designed as a convenient target formally expressing model semantics. Its syntax respects standards in programming languages (close to C or Java syntax), but has features expressing concurrency, synchronizations and fine control over atomicity of operations. With its symbolic back-end, it helps to bridge the gap between industrial specifications expressed using a DSL and symbolic model-checking tools.

With the rising importance of MDE in industrial practice, supporting such transformation based approaches helps to reduce adoption cost of formal methods in software engineering processes. From this point of view, the user builds a translation to GAL from another language. Textual GAL files can of course be built directly, but we also offer an EMF [1] compliant meta-model of GAL that can be used in conjunction with MDE tools and manipulated programmatically in Java. Several rewriting rules then perform simplification and optimization of GAL models, that benefit all input formalisms. The reduced GAL model can then be used for model-checking using *ITS-tools*.

In this paper, we show through two examples (the input languages of Divine and Uppaal) how to build such a transformation to GAL. They highlight the semantic features of GAL that make it a good target for such a transformation approach.

3 Guarded Action Language

3.1 Definition

Syntax. GAL offers a rich signature consisting of all C operators for manipulation of the `int` data type and of arrays (including nested array expressions). There is no explicit support for pointers, though they can be simulated with an array *heap* and indexes into it. An example of a GAL system that uses its concrete textual syntax is given in Fig. **??**.

We omit the definition details related to arithmetic and Boolean expressions, that have the same syntax and semantics as in the C language. We use C 32 bit signed integer semantics, with overflow effects; this ensures all variables have a finite (if large 2^{32}) domain. We assume $\mathbb{Z} = [-2^{31}, \dots, 2^{31} - 1]$ in the following.

Given a set of variables and of arrays, the set Int_e of integer expressions is the smallest set containing integers (constants), variables, array access expressions (an array name and an expression for the target index), the combinations of expressions with binary operators $+$ (add), $*$ (multiply), $/$ (divide), % (modulo), $-$ (minus), $\ll$ (left-shift), $\gg$ (right-shift), $^\wedge$ (bitwise xor), $|$ (bitwise or), & (bitwise and), unary minus $-$ and bitwise complement ˜ of an integer expression.

Boolean expressions $Bool_e$ are inductively defined using constants true, false, comparisons using $<$ (strictly less than), $<=$ (less than), $==$ equals, $!=$ (differs), $>=$ (greater than), $>$ (strictly greater) between two integer expressions, as well as Boolean combinations using && (and), $||$ (or), $=>$ (implies) of two Boolean expressions, or the ! (negation) of a Boolean expression. They can also be embedded into integer expressions, true being interpreted as 1 and false as 0.

GAL transition effects are described by statements in $Stat$, a set inductively built:

- $\langle lhs = rhs \rangle$ an assignment of an integer expression rhs to a variable or to the cell of an array designated by lhs,

- $\langle swap(l,r)\rangle$ swaps the values of variables or array cells designated by l and r,
- $\langle \sigma_1;\ldots;\sigma_k\rangle$ a sequence of k semi-colon separated statements. We note *nop* (no-operation) the empty sequence of statements,
- $\langle ite(c,\sigma_t,\sigma_f)\rangle$ a conditional if-then-else statement, with $c \in Bool_e$ and two statements σ_t and σ_f to be executed depending on the truth value of c,
- $\langle fixpoint(l)\rangle$ a fixpoint or transitive closure statement, where l (loop body) is a statement to be repeatedly executed until the behavior converges,
- $\langle abort\rangle$ a statement that drops current exploration, yielding no successors,
- $\langle call(\lambda)\rangle$ a call statement to a label λ, that non-deterministically invokes one of the transitions labeled λ. Note that circular *call* expressions are forbidden, disallowing recursion.

Assignments and control structures defined above are very standard and behave as usual. The swap statement is introduced because GAL lack temporary variables. The *call(λ)* statement models non-determinism; the transition chosen is any transition bearing label λ, allowing a state to have several successors. The *abort* statements produces the empty set of successors. More formally:

Definition 1. *A GAL system over a set Lab of labels containing* $\top$ *is a tuple* $\mathcal{G} = \langle Vars, Arrays, Trans\rangle$ *where:*

- *Vars is a set of integer variables,*
- *Arrays is a set of integer arrays; for* $a \in Arrays$*, we let* $|a|$ *designate its fixed size,*
- $Trans \subseteq Lab \times Bool_e \times Stat$ *is a set of transitions; for* $t = \langle \lambda, g, \sigma\rangle \in Trans$, $\lambda \in Lab$ *is the label of* t, $g \in Bool_e$ *is the* guard *of* t*, and* $\sigma \in Stat$ *is the* body *of* t.

Semantics. The semantics of GAL is defined as a transition system whose set of states is $\mathcal{S} = \mathbb{Z}^{|Vars|} \times \Pi_{a\in Arrays}\mathbb{Z}^{|a|}$, giving a value to each variable and array cell. Because $\mathbb{Z}$ is assumed to be finite, $\mathcal{S}$ is finite (resolving most decidability issues). Interpretation $e(s)$ of an expression e in a state s yields a constant value belonging to the range of e (integer or boolean). When lhs designates the left-hand side of an assignment, it must be either a variable (x) or an array access expression ($tab[i]$). In either case we let $lhs(s)$ denote the fully resolved variable, when interpreting the array index expression in state s. Conversely for any variable or array cell v, we note $s[v]$ the value of v in state s.

The semantics of statements and transitions are defined co-inductively.

Definition 2. *Let* $\mathcal{G} = \langle Vars, Arrays, Trans\rangle$ *be a GAL system over Lab. We note* $\mathcal{S} = \mathbb{Z}^{|Vars|} \times \Pi_{a\in Arrays}\mathbb{Z}^{|a|}$ *the set of states of* $\mathcal{G}$*. We define co-inductively semantics of statements* $\Delta_{stat} \subseteq 2^{\mathcal{S}} \times Stat \times 2^{\mathcal{S}}$ *and semantics of transitions* $\Delta_{trans} \subseteq 2^{\mathcal{S}} \times Trans \times 2^{\mathcal{S}}$ *as follows:*

- *for* $S, S' \subseteq \mathcal{S}$ *and* $\sigma \in Stat$, $\langle S, \sigma, S'\rangle \in \Delta_{stat}$ *if and only if*

$$\begin{cases}
\textit{if } \sigma = \langle lhs = rhs\rangle, S' \textit{ is the least subset of } \mathcal{S} \textit{ satisfying } \forall s \in S, \exists s' \in S' \textit{ such that} \\
\quad s'[lhs(s)] = rhs(s) \wedge \forall v \neq lhs(s), s'[v] = s[v] \\
\textit{if } \sigma = \langle nop\rangle, \quad S' = S \\
\textit{if } \sigma = \langle \sigma_1; \ldots; \sigma_k\rangle, \exists S_0 \ldots S_k \subseteq \mathcal{S} \\
\quad S_0 = S, S_k = S', \forall i \in [0 \ldots k-1], \langle S_i, \sigma_i, S_{i+1}\rangle \in \Delta_{stat} \\
\textit{if } \sigma = \langle swap(l,r)\rangle, S' \textit{ is the least subset of } \mathcal{S} \textit{ satisfying } \forall s \in S, \exists s' \in S' \textit{ such that} \\
\quad s'[l(s)] = r(s) \wedge s'[r(s)] = l(s) \wedge \forall v \neq l(s) \wedge v \neq r(s), s'[v] = s[v] \\
\textit{if } \sigma = \langle ite(c, \sigma_t, \sigma_f)\rangle, \textit{let } S_c = \{s \in S \mid c(s)\} \textit{ and } S_{\neg c} = S \setminus S_c, \\
\quad S' = S'_c \cup S'_{\neg c}, \textit{ where } \langle S_c, \sigma_t, S'_c\rangle, \langle S_{\neg c}, \sigma_f, S'_{\neg c}\rangle \in \Delta_{stat} \\
\textit{if } \sigma = \langle fixpoint(l)\rangle, \exists\, n \in \mathbb{N}, \exists\, S_0 \ldots S_n \subseteq \mathcal{S}, \\
\quad S_0 = S, S_n = S', \forall i \in [0 \ldots n-1], \langle S_i, l, S_{i+1}\rangle \in \Delta_{stat} \\
\quad \textit{and } \langle S_n, l, S_n\rangle \in \Delta_{stat} \textit{ and } \nexists k < n, \langle S_k, l, S_k\rangle \in \Delta_{stat} \\
\textit{if } \sigma = \langle call(\lambda)\rangle, \textit{ let } T_\lambda \subseteq Trans \textit{ designate transitions with label } \lambda, \\
\quad S' = \bigcup_{t \in T_\lambda} S'_t \mid \langle S, t, S'_t\rangle \in \Delta_{trans} \\
\textit{if } \sigma = \langle abort\rangle, \quad S' = \emptyset
\end{cases}$$

– *for* $S, S' \in \mathcal{S}$ *and* $t = \langle \lambda, g, \sigma\rangle \in Trans$, $\langle S, t, S'\rangle \in \Delta_{trans}$ *if and only if* $\langle \{s \in S \mid g(s)\}, \sigma, S'\rangle \in \Delta_{stat}$

We now define the semantics of $\mathcal{G}$ *as a transition system* $[\![\mathcal{G}]\!] = \langle \mathcal{S}, T\rangle$ *such that* $T \subseteq \mathcal{S} \times \mathcal{S}$*. We define* T *as union of transitions labelled by* $\top$*:*
$T = \bigcup_{s \in \mathcal{S}} \{\langle s, s'\rangle | \exists \langle \{s\}, t, S'\rangle \in \Delta_{trans} \wedge s' \in S' \wedge t \textit{ has label } \top\}$.

Linearity. The semantics of Δ_{stat} is defined in terms of sets of states in $2^{\mathcal{S}}$, thus closely matching the underlying symbolic operations. The definitions of statement semantics are all linear (with respect to $\cup$) to the input set, i.e. $\forall S_1, S_2 \subseteq \mathcal{S}$, $\langle S_1, \sigma, S'_1\rangle$ and $\langle S_2, \sigma, S'_2\rangle \in \Delta_{stat} \implies \langle S_1 \cup S_2, \sigma, S'_1 \cup S'_2\rangle \in \Delta_{stat}$. This is obvious for the abort statement, which produces no successor, and for the assignment and swap statements, which are deterministic and produce one image per input state. Linearity of transitions and statements is co-inductively proven, but well-founded since cyclic label calls are forbidden. If Δ_{stat} is linear, so is the if-then-else statement since it acts on both halves (hence a partition) of the input set. Similarly, linearity of the call statement stems from its union semantics and the linearity of called transitions. The sequence statement is linear since it amounts to a composition of linear effects. Finally, provided that n exists, the fixpoint statement is equivalent to a finite sequence of linear effects, hence is linear itself.

Fixpoint. The fixpoint is the only statement that really requires a definition in terms of sets (to enforce $\langle S_n, l, S_n\rangle \in \Delta_{stat}$), the other statements semantics could otherwise be defined directly as subsets of $\mathcal{S} \times \mathcal{S}$ rather than as subsets of $2^{\mathcal{S}} \times 2^{\mathcal{S}}$. The fixpoint statement offers a high expressive power, and fine control of the transition relation. However care must be taken to ensure its convergence.

If misused, the fixpoint may not terminate, either because it is attempting to build an infinite set (or rather too large to fit in memory, since $\mathcal{S}$ is finite), or because the sequence of sets of states it defines is somehow oscillating. Since we lack a structural criterion to decide whether a fixpoint will terminate (defining one is difficult in general even with finite $\mathcal{S}$, and undecidable if integers are unbounded), divergent fixpoints are considered malformed and will hopefully be detected at runtime. To avoid restricting the expressiveness of GAL, we leave to the user the responsibility of ensuring that fixpoints are well-formed (i.e. n exists). Several classical features can be described with a correct use of the fixpoint statement:

- When the body statement l expresses a non deterministic choice between an action a and an empty statement (identity), the effect is that of the least fixpoint μ of lambda-calculus.
- It may also be used to compute a greatest fixpoint ν of lambda-calculus, for instance to identify states belonging to a strongly connected component of the transition system.
- In other cases it can be used to iterate a transformation or rewriting to stability, for instance if the action is to decrement a given variable until 0 is reached. This can be used to map several states onto a single image, allowing to express directly in GAL classical state-space reductions, such as symmetry reduction [6].

Call. The non-deterministic call construct combined with the sequence is particularly important to allow expression of transition relations that are a composition of sum of effects (e.g action "a or a'" followed by action "b or b'"). Making all the alternatives explicit ($ab, ab', a'b, a'b'$) could lead to an exponential blowup of the representation size. A sequence of if-then-else constructs can also avoid an exponential blowup of the representation size with respect to an explicit modeling of all the alternatives. These exponential pitfalls unfortunately cannot be avoided by a plain support based approach like in LTSmin (see section **??**).

3.2 Parametric GAL

GAL also features parametric constructs to comfortably express common patterns. They can be degeneralized, and amount to syntactic sugar.
Range. We let a GAL definition contain the definition of named subsets of $\mathbb{Z}$ called *ranges*. A transition t can bear an arbitrary number of formal parameters, each having a name and range. The parameters can be used like ordinary variables within the definitions of the guard and body of t, though they cannot be assigned new values. They can also be used within the definition of the label of t, and in the definition of labels used in any call statements of the body of t.
Parametric transition. Defining such a transition t is equivalent to defining a set of transitions, containing one transition for each element in the cartesian product of the parameter ranges (i.e. each possible assignment of values to parameters). In each of these transitions which have no parameters, the guard, body and label of t are replaced by a version where each parameter reference is replaced by a constant (its assigned value). Occurrence of a parameter in a label (that of t itself or occurring in a call of the body of t) builds a new label where the parameter is substituted by a string representing its numeric value.

This mechanism is similar in many ways to the way colored Petri net transitions are defined with respect to their unfolded P/T net version. This construct makes specifications much more compact and readable in many cases. It also eases traceability when the GAL model is obtained by a model transformation. It also helps exhibit nice symmetry properties of the transition relation, depending on how the parameters are actually used in the guard and body of t. Lastly, reasoning on parameters before discarding them through instantiation can allow to significantly reduce the transition relation representation size.

Sequential iteration. Given these ranges, we also introduced a limited iteration $\langle for(p : r)\{b\}\rangle$ statement (for each p in r, do b), where p is a parameter with range r and b is a body statement. It is equivalent to a sequence of $|r|$ statements $\langle b_1; \ldots ; b_{|r|}\rangle$, where each b_i is the statement b where the parameter p is replaced by its value in r. This construct mostly eases modeling when manipulating arrays. It can be seen as a dual for the use of parameters in transitions (that builds a sum or union of $|r|$ effects), since it builds a composition of $|r|$ effects.

Instantiation. GAL models are structurally analyzed before model-checking, allowing to simplify away the parametric features. This analysis simplifies expressions that can be statically evaluated, removes structurally unreachable behaviors (e.g. transitions with false guards), and instantiates parameters with on the fly simplifications. Other simplifications and rewritings (described on the webpage) are also available, some of which are more involved such as attempting to rewrite transitions with several parameters as a sequence of calls to transitions with a single parameter each. When parameters are in fact independent (no statement uses them both), having a sequence of choices (rather than the explosion due to choosing all parameter values at once) leads to a transition relation in desirable composition of sums of effects form, with possibly an exponentially more compact GAL specification than plain instantiation of parameters.

One-hot. Any variable or array can be tagged with the "hotbit(r)" keyword, indicating the user wants a one-hot state encoding, where a variable with domain $0..n-1$ is encoded as n Boolean variables with only one "hot" bit set to 1. Apart from this keyword at declaration, the variable is manipulated normally in the GAL syntax. We then use a GAL to GAL transformation to instantiate such variables, translating accesses and assignments to the variable to reflect the one-hot encoding. The translation involves adding a parameter to represent the current value and testing that its corresponding bit is hot in transition guards. We further automatically identify and tag variables that could benefit from one-hot encoding: any variable that is only assigned constants (this allows to statically compute the range) and whose domain size is greater than a threshold (we use 8) is set by default to one-hot encoding. By increasing locality, one-hot encodings can be favorable to DD techniques, for instance this feature is often used to encode locations of automata in symbolic model-checkers.

At model-checking time, every statement is encoded as a symbolic operation. ITS-tools then fully exploits commutativity and on-the-fly simplifications at every level of the evaluation to adaptively exploit the structure of the decision diagram encoding the states (see [11, 5]).

4 Instantiable Transition Systems

This section recalls the Instantiable Transition Systems (ITS) framework and defines the Guarded Action Language (GAL). ITS has been designed for the description of component based systems, while GAL is a C-like description of the components. Both are connected with a verification library where the states of the resulting systems are encoded with various kinds of decision diagrams. More precisely, the hierarchical characteristics of systems use Hierarchical Set Decision Diagrams (SDD [**?**]), while the data content is encoded with Data Decision Diagrams (DDD [**?**]), using the recent efficient algorithms of [**?**] to encode GAL semantics. In the process, ITS definitions have been revised with respect to [**?**] and [**?**], where they were first introduced, to be simpler while having the same expressivity.

4.1 ITS type and instances

ITS describe a minimal Labeled Transition System (LTS) style formalism using notions of *type* and *instance* to emphasize locality of actions and to exploit the similarity of copies of a given type. The composition mechanism is based solely on transition *synchronizations* (no explicit shared memory or channel).
Notation: For a tuple $z = \langle X, Y, \cdots \rangle$, we denote by $z.X, z.Y \ldots$ the elements $X, Y, \ldots$.

The following definition sets an abstract contract or interface that must be implemented by concrete ITS types.

Definition 1 *An **ITS type** is a tuple* $\tau = \langle S, A, Locals, Succ \rangle$ *where:*

- *S is a set of states; A is a finite set of public action labels;*
- *$Locals : S \mapsto 2^S$ is a local successor function;*
- *$Succ : S \times A \mapsto 2^S$ is a transition function.*

An ITS type can be instantiated, possibly several times. With an instance i is associated its ITS type $type(i)$.

Reachability: Let i be an ITS instance and s, s' be two states in $type(i).S$. State s' is reachable from s if there exist states $s_0, \ldots s_n \in type(i).S$ such that $s = s_0$, $s' = s_n$ and for all j, $1 \leq j \leq n, s_j \in type(i).Locals(s_{j-1})$.

The two functions *Locals* and *Succ* are used for different purposes: *Locals* represents moves that may occur within an instance autonomously or independently from the rest of the system. Hence it returns states reachable through occurrences of local events. The function *Succ* produces successors by explicitly synchronizing actions via an action label from the alphabet. Note that *Succ* is the only way to control the behavior of a (sub)system from outside.

Remarks. This definition is enriched in practice by specifying an initial state, as well as state-based predicates giving a Kripke state labeling for model-checking purposes.

The transition relation of a full system can only be defined in terms of subsystem synchronizations using *Succ* and of independent local behaviors. Hence, a full system is defined by a single instance of a particular type in a specific initial state: the system is self-contained and thus reachability only depends on the definition of *Locals*.

4.2 Composite ITS types

We now define a *composite ITS type*, designed to offer support for the hierarchical composition of ITS instances. This new version, adapted from [**?**], is aligned with standard labeled synchronized product definitions (e.g [**?**,**?**]). An example of composition is given in Fig. **??** using our concrete syntax.

Notations: For a tuple $I = (i_1, \ldots, i_n)$ of ITS instances, $|I|$ denotes the size n of I, S_I is the set $type(i_1).S \times \ldots \times type(i_n).S$. For $s \in S_I$ and i an instance, we denote $s[i]$ the component of s that corresponds to i.

Given a tuple I of ITS instances and a set Lab of labels, we inductively define the set $Stat_C$ of composite statements by:

- $\langle call(i,\lambda)\rangle$ a call statement to a label λ of $type(i)$, that invokes a transition of i labelled by λ,
- $\langle call_{self}(\lambda)\rangle$ a call statement to a label λ of Lab, that invokes a transitiion of the current composite type, labelled by λ. This allows to structure the transition relation and chain behaviors. We syntactically forbid cycles of self-calls.
- $\langle \sigma_0; \ldots ; \sigma_k\rangle$ a sequence of semi-colon separated statements in $Stat_C$.

Definition 2 *A **composite** over alphabet Lab is a tuple $C = \langle I, Sync\rangle$ where:*

- *I is a tuple of ITS instances, said to be* contained *by C. We further require that the type of each ITS instance already exists when defining I, in order to prevent circular or recursive type definitions.*
- *$Sync \subseteq Lab \times Stat_C$ is the finite set of synchronizations, where for $t = \langle \lambda, \sigma\rangle \in Sync$, λ is the label of t and σ its body.*

Next state function by a statement: The function $Next_I : S_I \times Stat_C \mapsto 2^{S_I}$, used in definition 3 below, is defined for $s, s' \in S_I$ and $\sigma \in Stat_C$ by:

$s' \in Next_I(s,\sigma)$ *iff*

$$\begin{cases} \exists s_0 \ldots s_{k+1}, s_0 = s, s_{k+1} = s',\ \forall i \in [1 \ldots k], s_{i+1} \in Next_I(s_i,\sigma_i) & \text{if } \sigma = \langle \sigma_0; \ldots ; \sigma_k\rangle \\ s'[i] \in type(i).Succ(s[i],\lambda) \wedge \forall j \in I, j \neq i, s'[j] = s[j] & \text{if } \sigma = \langle call(i,\lambda)\rangle \\ \exists t = \langle \lambda, \sigma'\rangle \in Sync,\ \textit{such that } s' \in Next_I(s,\sigma') & \text{if } \sigma = \langle call_{self}(\lambda)\rangle \end{cases}$$

Definition 3 *Let $C = \langle I, Sync\rangle$ be a composite over alphabet Lab. The ITS type $\tau_C = \langle S, A, Locals, Succ\rangle$ corresponding to C, is defined by:*

- *$S = S_I$; $A = Lab \setminus \{\top\}$;*
- *$Locals : S \mapsto 2^S$ is defined for $s, s' \in S$ by: $s' \in Locals(s)$ iff*

$$\begin{cases} \quad \exists i \in I, s'[i] \in type(i).Locals(s[i]) \wedge \forall j \in I, j \neq i, s'[j] = s[j] \\ or\ \exists t = \langle \top, \sigma\rangle \in Sync, s' \in Next_I(s,\sigma) \end{cases}$$

- *$Succ : S \times A \mapsto 2^S$ is defined for $s, s' \in S, \lambda \in A$ by:*
  $s' \in Succ(s,\lambda)$ iff there exist $t = \langle \lambda, \sigma\rangle \in Sync, s' \in Next_I(s,\sigma)$.

Definition 3 thus describes an implementation of the generic ITS type contract. It contains either elementary instances (such as LTS, or the guarded action language introduced later in this paper), or inductively other instances of composite nature.

In this definition, $Locals(s)$ is defined as the set of states resulting from the action of $Locals$ in any nested instance (without affecting the other instances), or states reachable from s through the occurrence of any synchronization associated to the local label $\top$. The set of successors $Succ(s,\lambda)$ is obtained by applying the effect of label λ, which can trigger a sequence of successive calls. This sequence is then fired atomically.

5 Conclusion

The symbolic model-checker ITS-tools, its Eclipse based editor front-end, source code as well as user documentation are freely available from the webpage `http://ddd.lip6.fr`. It now offers easy access to symbolic model-checking for a wide range of formalisms thanks to its support for the general purpose Guarded Action Language.

References

1. Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`.
2. A. Ben Salem, A. Duret-Lutz, F. Kordon, and Y. Thierry-Mieg. Symbolic Model Checking of stutter invariant properties Using Generalized Testing Automata. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume ? of *Lecture Notes in Computer Science*, page to be published, Grenoble, France, April 2014. Springer.
3. R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96,*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, USA, July 1996. Springer.
4. M. Colange. *Symmetry Reduction and Symbolic Data Structures for Model Checking of Distributed Systems*. PhD thesis, Université Pierre et Marie Curie, Paris, France, December 2013.
5. M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. Towards Distributed Software Model-Checking using Decision Diagrams. In *25th International Conference on Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 830–845. Springer Verlag, July 2013.
6. M. Colange, F. Kordon, Y. Thierry-Mieg, and S. Baarir. State Space Analysis using Symmetries on Decision Diagrams. In *12th International Conference on Application of Concurrency to System Design (ACSD'2012)*, pages 164–172, Hamburg, Germany, June 2012. IEEE Computer Society.
7. J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. Data decision diagrams for Petri net analysis. *Application and Theory of Petri Nets 2002*, pages 129–158, 2002.
8. J.-M. Couvreur and Y. Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. *Formal Techniques for Networked and Distributed Systems-FORTE 2005*, pages 443–457, 2005.
9. A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. Self-loop aggregation product—a new hybrid approach to on-the-fly ltl model checking. In *Automated Technology for Verification and Analysis*, pages 336–350. Springer, 2011.
10. E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, June 1987.
11. A. Hamez, Y. Thierry-Mieg, and F. Kordon. Hierarchical Set Decision Diagrams and Automatic Saturation. In *Applications and Theory of Petri Nets 2008, ICATPN 2008, Xian, China*, volume 5062 of *LNCS*, 2008.
12. H. Iwashita, T. Nakata, and F. Hirose. Ctl model checking based on forward state traversal. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 82–87. IEEE, 1996.

13. LRDE. Spot: a library for LTL model-checking. `http://spot.lip6.fr/`.
14. K. Y. Rozier and M. Y. Vardi. Ltl satisfiability checking. In *Proceedings of the 14th International SPIN Conference on Model Checking Software*, pages 149–167, Berlin, Heidelberg, 2007. Springer-Verlag.
15. F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic SCC hull algorithms. In *Proc. of FMCAD'02 (FMCAD'02)*, volume 2517 of *LNCS*, pages 88–105. Springer.
16. Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical set decision diagrams and regular models. *Tools and Algorithms for the Construction and Analysis of Systems*, 5505:1–15, 2009.
17. M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.