Observation Graph implementation for TINA toolbox

Rodrigo T. Saad, François Vernadat, Bernard Berthomieu, Silvano Dal Zilio
CNRS; LAAS;
7, avenue du Colonel Roche, F-31077 Toulouse – France
Université de Toulouse;
UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
{rsaad, francois, bernard, dalzilio}@laas.fr

Abstract

Model Checking is a formal technique for the verification of finite systems. However, it is well known that this technique suffers from the state explosion problem. We describe work in progress to implement in the TINA *toolbox an enumerative variant of a state based observation graph algorithm defined by Klai and Poitrenaud.*

I. Introduction

Model Checking is a formal technique for the verification of finite systems. However, it is well known that this technique suffers the drawback of the state explosion problem. Two major strategies are known to deal with this problem, they are: the state explosion management and its reduction. The first one concerns techniques interested in bypass the difficulty of dealing with complex systems. The second group is the ones where the complexity of the system is reduced by taking advantage of regularities, redundancies or existing similarities in the exhaustive representation.

This paper describes a work in progress to implement in the TINA toolbox an enumerative variant of a state based observation graph algorithm defined by [4]. The state based symbolic observation graph presented by Klai and Poitrenaud is a hybrid approach for checking linear time temporal logic properties (LTL) of finite systems combining on-the-fly and symbolic approaches for Petri nets. In our case, we are interested at the moment on a management strategy for the construction of a partial enumerative state based graph, which is in fact an abstraction of the original reachability graph. This abstraction is based on LTL formula supplied by the user, namely as atomic proposition set. Our motivation for this experiment is to be able to construct partial graphs when the size of the complete state graph is too big. In addition, this technique is easy to be used because different abstractions may be achieved only by providing different sets of atomic propositions.

This work has been supported by the French AESE project Topcased and by region Midi-Pyrénées

This paper is structured as follows: In section II, we introduce the TINA toolbox. In section III, the state based observation graph is formally presented. Then, section IV discuss about its implementation into the TINA toolbox. Some experiments and results are illustrated in section V. Finally, section VI concludes this work and gives some perspectives.

II. Tina ToolBox

TINA is a software environment to edit and analyze Petri nets, Time Petri nets, and some extensions of these nets. In addition to the usual editing and analysis facilities of similar environments, `tina` - name of the exploration engine - offers various abstract state space constructions that preserve specific classes of properties of the state spaces of nets, like absence of deadlocks, linear time temporal properties, or bi-similarity. Another tool that is part of the toolbox is the *selt* tool, which implements a model-checker for an extension of linear time temporal logic known as State/Event LTL, a logic supporting both state and transition properties. More details about the different tools constituting the TINA toolbox can be found in [1].

III. Observation Graph

Considering a set AP of atomic propositions, an observation graph can be informally presented as a graph of state sets (*aggregates*), which comprehends only states with the same atomic propositions, interconnected by edges that modify these atomic propositions. From [4], we extracted the following definitions:

Definition *Aggregate*: Let $T = \langle \Gamma, L, \rightarrow, s_0 \rangle$ be a *KS* over an atomic proposition set *AP*. An aggregate a of T is a non empty subset of Γ satisfying $\forall s, s' \in a, L(s) = L(s')$.

Definition *Observation Graph*: Let $T = \langle \Gamma, L, \rightarrow, s_0 \rangle$ be a *Kripke structure*(*KS*) over an atomic proposition set *AP*. An observation graph of T is a 4-tuple $G = \langle \Gamma', L', \rightarrow', a_0 \rangle$ where:

- $a_0 \in \Gamma'$ is the initial aggregate.
- $\Gamma' \subseteq 2^{\Gamma}$ is a finite set o aggregates.
- $L' : \Gamma' \rightarrow 2^{AP}$ is a labeling function satisfying $\forall a \in \Gamma'$, let $s \in a, L'(a) = L(a)$.
- $\rightarrow \subseteq \Gamma' \times \Gamma'$ is a transition relation.

A. Observation Graph Model Checking

In [4], the equivalence between checking a given $LTL\backslash X$ - LTL minus the next operator because of the abstraction of the immediate successors - property over the observation graph or over the original labeled transition system is ensured by the preservation of maximal paths. Then, to capture all the maximal paths of an observation graph under the form of infinite sequences, it is enough to transform its finite maximal paths into infinite ones. This transformation happened when the aggregates(a_i) have the presence of a dead state ($Dead(a_i) := \{\exists s \in a \mid \nexists s' \mid s \rightarrow s'\}$) or a circuit ($Live(a_i) := \{\exists \pi \mid \pi = s_1 \rightarrow ... \rightarrow s_m \rightarrow ... \rightarrow s_n$ and $s_m \rightarrow ... \rightarrow s_n$ a circuit of $a\}$).

As can be noticed, the *AP* set is extracted from the $LTL\backslash X$ supplied by the user. In this work, for simplicity reason, we refer the *AP* set directly as the set of places instead of the original $LTL\backslash X$.

IV. TINA impelementation

The Observation Graph algorithm has been implemented using the TINA toolbox *API*. The main characteristic of our implementation is to handle two reachability algorithms: one for the observation graph itself; another one for the local search. In addition, we take advantage of the TINA functional architecture to allow different input formats to make use of this abstraction without modifying the model or the toolbox. The following subsections present the algorithm implementation.

A. TINA toolbox Architecture

The TINA three layer functional architecture was conceived to be easily integrated with other tools. The first layer, the *Front-end*, comprehends the translators (*compilers*) used to translate the input models into one of the two internal abstract format (*Time Petri Nets*[2] or *Time Transition Systems*[3]). The second layer is a parameterized state exploration engine responsible for constructing the reachability graph of the system model. The third and final layer is the *Back-end* and it is responsible for printing the results in different formats to facilitate the toolbox integration with third-part tools.

As we mentioned before, we expect this implementation to be as transparent as possible for all the supported input formats (*Front-ends*). To achieve this objective, it becomes clear that our implementation will take place inside the parameterized engine layer as a new reachability algorithm. Particularly, in this case, the engine parameters will be the atomic proposition set supplied by the user.

A new reachability algorithm for TINA is achieved by defining a new reachability structure, which comprehends: a state data time, a transition firing function and a state comparison function. The observation structure handles two reachability searches, one for the observation and one for the local search. The main reachability search is the observation one and it holds a state type which is composed of a set of local state types - which is in this case Petri Nets Markings - ordered lexicographically and stored into a data list structure, and two Boolean values to indicate the presence of circuits and deadlocks. Regarding the firing transition function itself, this functions calls the local search, which is in this case an ordinary Petri Net reachability algorithm, in order to construct the local set of states, that is to say, the set of states with the same atomic propositions. With respect to the state comparison function, when applied over two aggregates, it carries out a lexicographically comparison over the states lists and analysis if they have the same Boolean values. Obviously, the comparison performed over each state from the list is the same comparison function used by the local search, which is in this case the marking comparison defined for ordinary Petri Nets. The following section clarifies the observation and local search.

B. Observable Search and Local Search

The graph construction is driven by the atomic proposition set (AP) supplied by the user. These atomic propositions are in fact the name of places in the Petri Net and they are going to be used to divide the transitions into two groups: observations (*ob* for short) and not observations (*nob*). The first group consists of the transitions that affect the atomic propositions (*AP*) and the second one the transition that do not affect the *AP* set.

The local search comprehends the complete set of markings that corresponds to an aggregate(AG_i), in another words, all states reachable by firing until saturation all the *nob* transitions. During this local search, all the states firable by an *ob* transition are separated into a set called $Out(AG_i)$. In addition, the local graph is tested to determine the existence of circuits ($Live(AG_i)$) and dead-states ($Dead(AG_i)$) to transform finite maximal paths into infinite ones (section III-A). Circuit detection is performed analyzing the local graph strongly connected components in order to detect nodes with more then one marking. About the dead-state detection, a state is defined as dead if it can not be fired by an *ob* or a *nob* transition. In the end of a local search for a given aggregate, the local graph is discarded and only the set $Out(AG_i)$ in conjunction with the two Boolean values are stored.

The observation search constructs the aggregates from the *AP* set. The initial aggregate (AG_0) is computed

from the initial marking(S_0) by calling the local search algorithm. Then, the local search algorithm returns all the exit states ($Out(AG_0)$) in conjunction with the circuit and dead properties associated for this aggregate. Next, for each firable *ob* transition from the exit states ($Out(AG_0)$), a new aggregate is computed by calling the local search algorithm from the set of reachable states ($IN(AG_1)$). This procedure is repeated over and over again until there is no more unexpanded set of reachable states ($IN(AG_i)$). Furthermore, two aggregates are seen as equivalent if they have the same exit states ($Out(AG_i) = Out(AG_j)$) and if both have or not the dead ($Dead(AG_i) = Dead(AG_j)$) and the live ($Live(AG_i) = Live(AG_j)$) properties defined for their local set of markings. Figure 1 shows the observation and local search.

Besides, in order to provide more freedom for further extensions, i.e. the addition of time intervals and data structure, we decided that both the *nob* transitions and the *ob* transitions are fired by the local search algorithm. In the case of the *ob* transitions, the observation search gives the exit state and the *ob* transition to be fired as the initial state for the local search.

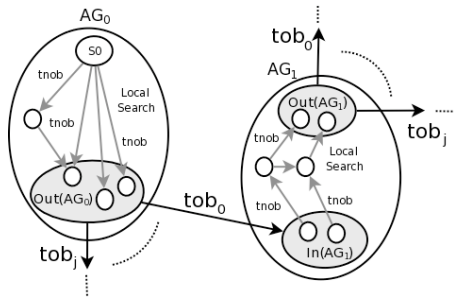


Figure 1. Observable and Local Search.

C. Example

In order to clarify the observation graph algorithm presented before, this section presents a small Petri Net example, which is a Token Ring model with only one site for simplicity. Figure 2 shows the Petri Net and the places highlighted ($AP = \{idle_1, wait_1, cs_1\}$) in gray are used as the abstraction to construct the partial graph.

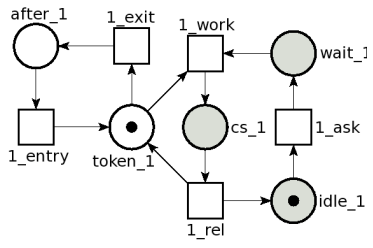


Figure 2. Token Ring Petri Net example.

First, based on the supplied AP set, the transitions from the model are separated into observation (t_{ob} = $\{1\_rel,\ 1\_work, 1\_ask\}$) and non-observation($t_{nob}$ = $\{1\_exit,\ 1\_entry\}$). Second, the first aggregate (AG_0) is obtained by applying the local search from the initial marking($S_0 = \{idle_1,\ token_1\}$). The local search constructs the local graph by firing until saturation all t_{nob} transitions. Third, the input states for aggregate AG_1 ($IN(AG_1) = \{(token_1, wait_1)$, $(after_1, wait_1)\}$) is obtained by firing the t_{ob} transition 1_ask from the AG_0 exit states $OUT(AG_0) = \{(idle_1, token_1), (after_1, idle_1)\}$. Once again, from the input set ($IN(AG_1)$), the local search is used to construct the local graph and the exit states are returned ($OUT(AG_1) = \{(token_1, wait_1)\}$). Next, the AG_2 input states ($IN(AG_1) = \{(cs_1)\}$) are obtained by firing the t_{ob} transition 1_work from the $OUT(AG_1)$. Finally, the last edge connecting AG_2 and AG_0 is achieved by applying the t_{ob} transition 1_rel from the $OUT(AG_2)$. Figure 3 illustrates this example.

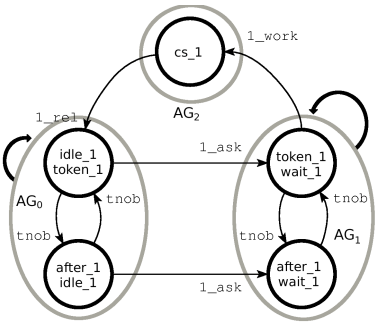


Figure 3. Token Ring Observation Graph.

A final remark about this example is the loop edges inserted at aggregates AG_0 and AG_1 to ensure the infinite maximal paths (Figure 3).

V. Experiments and Results

In this section we present two experiments performed with our preliminary implementation of the observation graph. The first one was to generate an observation graph when the complete graph is not feasible. For this experiment we chose the Token Ring example (Figure 2) increasing the number of sites. Figure 4 illustrates the results table, where the first column presents the complete graph and the following two columns the resulting graphs considering an abstraction of 1 ($AP = \{cs_1, wait_1, idle_1\}$) and 8 sites ($AP = \{..., cs_8, wait_8, idle_8\}$), respectively. These results were achieved using a 2.00GHz Intel Core 2 Duo with 2GB of memory and 2048 KB of cache memory.

N. Sites	Complete. Graph	Observable. Graph	
16	Nodes = 2621440 Time = 296.944s	1 sites is Ob. Nodes = 3 Time = 308.196s	8 sites is Ob. Nodes = 1280 Time = 285.455s
17	Not Treated	1 site is Ob. Nodes = 3 Time = 2204.056s	8 sites is Ob. Graph Nodes = 1280 Time = 9887.150s

Figure 4. Token Ring Results.

From Figure 4, TINA was not capable to generate the complete graph for a model with 17 sites. However, a

partial graph was possible considering an abstraction of up to 8 sites. In addition, this table show the difference between the time consumed for 1 and 8 sites on the last row. This difference can be explained because, depending on the selected *AP* set, the number of equivalent aggregates may increase and therefore a longer time will be expanded generating local graphs that belongs to equivalent aggregates.

The second experiment was performed to extend this abstraction for models with time constraints. Previously, on the implementation section, we presented its major particularity, which is its double search reachability algorithm. This double reachability algorithm holds one search for the observation graph itself and another one for its local graph. In order to allow the observation abstraction for TPN models, it is sufficient in this case to consider a local time search, in other words, to instruct TINA to perform a local search using the TPN reachability search instead of the ordinary Petri Net search. As a consequence, the time constraints are taken into account by the local state type, the firing and the comparison function. We state here that it is enough because our observation search uses the local state type and comparison function to define its state (list of exit states) and the comparison of two aggregates (comparison of their list of exit states). Figure 5 presents a simple Time Petri Net example where the atomic propositions are highlighted in gray ($AP = \{P_1, P_3\}$).

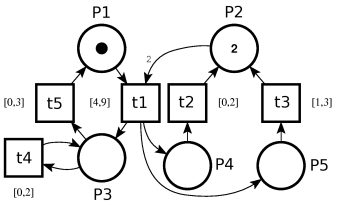
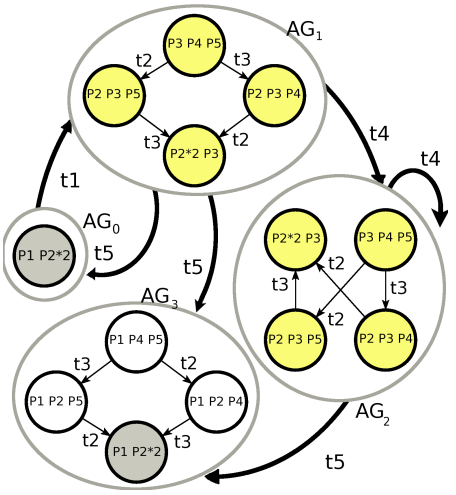


Figure 5. Time Petri Net Example.

Figure 6a) shows a partial observation graph exploration where the exit states are highlighted in yellow and gray. First of all, it is partial because this figure presents two separated aggregates (AG_0 and AG_3) that are in fact equivalent and consequently, they have the same exit states (highlighted in gray) and the same properties ($Live = false$ and $dead = false$) on their local graphs. Second, despite of the fact that aggregates AG_1 and AG_2 have exit states with the same marking (highlighted in yellow), these exit states differs from AG_1 and AG_2 because they have different time constraints on their local graphs, thus they are not equivalent. The aggregate comparison, performed over the list of exit states, could take into account the time constraints because it reuses the local search state comparison function, which is in this case the Time Petri Net state comparison. Figure 6b) presents the complete observation graph for this example.

VI. Conclusion and Future Works

We described the implementation of an abstraction approach to generate partial reachability graphs using



a) Partial Observation Graph.

b) Complete Observation Graph.

Figure 6. Time Observable Graph

TINA toolbox. From experimentation 1 (section V), we were able to build partial graphs when the complete one were not feasible. In contrast with [4], in our case we do not use BDD (Binary Decision Diagrams) to compress the list of exit states mainly because we do not want to restrict the freedom provided by TINA *API*, i.e. data structure and time intervals. Nevertheless we do agree that both strategies are complementary and may result in combined benefits. Thus, we are going to investigate, for further implementations, compression techniques not so restrictive like BDD.

About our implementation, it already takes into account some degrees of freedom to allow further experiments extensions. For example, our next step it to extend the Observable Graph to accept data structures (FIFO queues, record, union, etc) as part of the local search states. This extension will follow the same idea described to add time intervals V.

References

[1] F. Vernadat B. Berthomieu, P.-O. Ribet. The tool tina – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14), 2004.

[2] B. Berthomieu, M. Menasche, and REA Mason. An enumerative approach for analyzing time Petri nets. *Information Processing: proceedings of the IFIPcongress 1983*, 9:41–46, 1983.

[3] T.A. Henzinger, Z. Manna, and A. Pnueli. Timed Transition Systems. In *REX Workshop*, pages 226–251, 1991.

[4] K. Klai and D. Poitrenaud. MC-SOG: An LTL Model Checker Based on Symbolic Observation Graphs. In *Applications and Theory of Petri Nets 2008: 29th International Conference, Petri Nets 2008, Xi'an, China, June 23-27, 2008, Proceedings*. Springer, 2008.