Automated Composition of Nondeterministic and Partially Observable Stateful Services

Riccardo De Masellis and Francesco Fusco

December 9, 2008

1 Abstract

In this paper we study the issue of service composition, in the case that services are nondetermistic and partially observable (POTS). In particular, the target service is represented as a finite deterministic transition system, whereas available services are described as finite, nondeterministic and partially observable transition systems. Our aim is, in the first place, to expose a methodology to transform a POTS in a different, but fully observable TS, named KTS, then we show a particular polinomial encoding of this problem in SMV, that allows to solve the problem of composition using the known techinque of reducing the synthesis to the search a safety game winning strategy.

2 Introduction

A finite and potentially nondeterministic transition system is a tuple $TS = \langle A, S, s_0, \delta, F \rangle$ where:

- A is the finite set of action;
- S is the finite set of states;
- s_0 is the initial state;
- $\delta \subseteq S \times A \times S$ is the transition relation;
- F is the set of final states.

A finite, potentially nondeterministic and partially observable transition system differs from the previous one in the presence of an observability function. More formally, is a tuple $POTS = \langle A, S, s_0, \delta, F, \sigma \rangle$ where:

- $\sigma : S \rightarrow \mathcal{C}$ is a function that returns the observation of a state. We don't care about the codomain $\mathcal{C}$, it may be whatever, e.g. the set of natural numbers $\mathbb{N}$ or a set of propositional variables that represent some information of the given state. If this function is injective, we say that the TS is fully observable.

The partial observability raises no problems if the TS is deterministic, i.e. if $\forall s_1, s_2, s_3 \in S,\ a \in A\,(\delta(s_1, a, s_2) \wedge \delta(s_1, a, s_3) \rightarrow s_2 = s_3)$. In this case, the transition relation is, in fact, a function $\delta : S \times A \rightarrow S$, and even if all states have the same observation, we can know any moment in which state the TS is, simply by storing actions performed. To the contrary, when we have to deal with a nondeterministic and partially observable TS, we may don't know the future state after a transition. Here we show a methodology to transform a POTS in a fully observable TS. This formalization is based on the idea that in a sequence of action calls to the comunity services, not only we can use the observation of a state, but also the actions sequence itself (previously stored for the purpose) to understand in which state the TS actually is.

3 KTS

The new TS, is a tuple $KTS = \langle A_k, K, k_0, \delta_k, F_k \rangle$ built in this way:

- The set K is built inductively as follows:
  - $s_0 \in K$;
  - $\vec{s} \in K \ \wedge\ (\vec{s}, a, \vec{s'}) \in \delta_k \rightarrow \vec{s'} \in K$.
- The transition relation δ_k is built as follows:
  $\forall \vec{s} \in 2^S, \forall a \in A$ we define:

  $$img(\vec{s}, a) = \begin{cases} \emptyset & if\ (\exists s \in \vec{s} \mid (s, a, *) \notin \delta) \\ \{s' \in S \mid \exists s \in \vec{s} \wedge (s, a, s') \in \delta\} & otherwise \end{cases}$$

  $$obs(\vec{s}) = \{o \mid \exists s \in \vec{s} \wedge \sigma(s) = o\}$$

  Then, $(\vec{s}, a, \vec{s'}) \in \delta_k$ if and only if:

  $$\vec{s'} \neq \emptyset \ \wedge \vec{s'} = \{s \in img(\vec{s}, a) \mid \exists o \in obs(img(\vec{s}, a)) \wedge \sigma(s) = o \wedge$$
  $$\wedge \forall s', s'' \in \vec{s'} \rightarrow \sigma(s') = \sigma(s'')\}$$

- The finite set of actions A_k is composed by all actions contained in δ_k;
- $k_0 = s_0$;
- $F_k = \{s \in F \cap K\} \cup \{\vec{s} \in K \mid \forall s_i \in \vec{s} \rightarrow s_i \in F\}$;

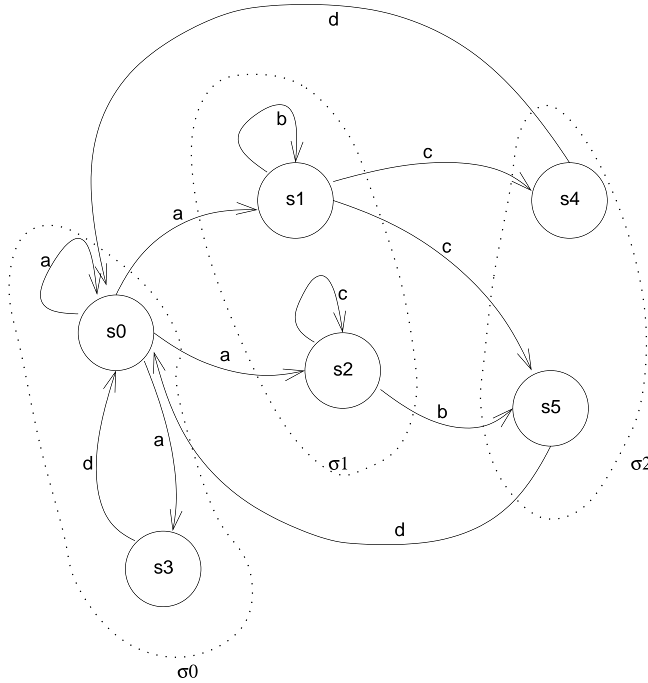


Figure 1: Example of POTS.

Note that, by constuction: $\forall \vec{s} \in K \rightarrow |obs(\vec{s})| = 1$ and, by consequence, every $\vec{s}$ such that $obs(\vec{s}) \neq 1$ cannot be reached by any transition.

In other words, the new KTS' states, are, in effect, macrostates, that represent our incomplete knowledge after a nondeterministic transition which arrival states have the same observation. Theoretically, everyone of 2^S states could be in the set K and every $2^s \cdot A \cdot 2^s$ transition could be in the set δ_k, so the procedure works as follows: it first calculates every possible transition relations that could be in δ_k, and then builds inductively the set of states K.

In Figure 1 is shown a POTS, and in fig.2, the corresponding KTS, build with the methodology described above.

From a computational point of view, this metodology is exponential in the number of POTS' states. This exponential cost cannot be avoided, because in the worst case the cardinality of KTS' macrostates is exactly 2^S, as shown in Figure 3 and Figure 4. In this example, transition relations of the associated KTS are not complete, but what is important is that the number of states is exponential in the cardinality of POTS' states.

Once we convert a POTS into a KTS, that is in fact, a standard nondeterministic TS, we can use one of the known nondeterministic services composition tecniques. But this solution is unacceptable for at least two reason: *(i)* building a KTS as presented above is very inefficient, because its cost is exponential even in the best case; *(ii)* conversions are made in any case, even if the converted

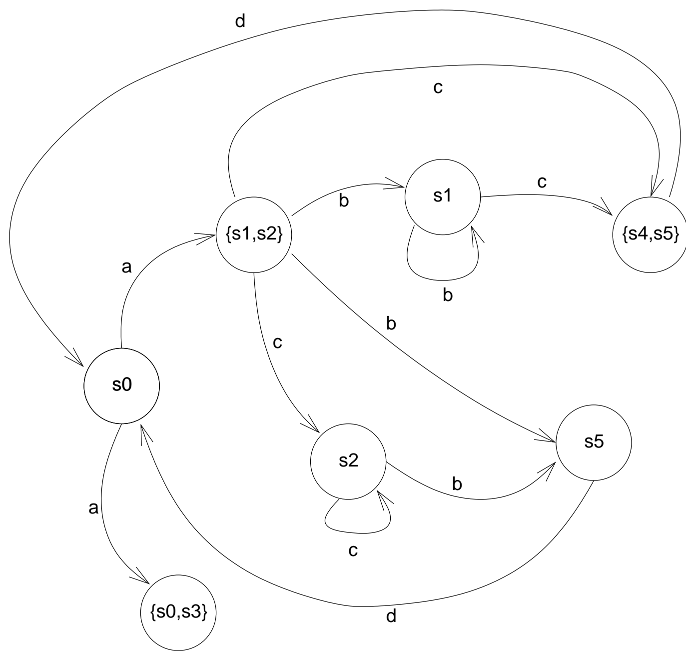


Figure 2: KTS corresponding to the POTS of Figure 1

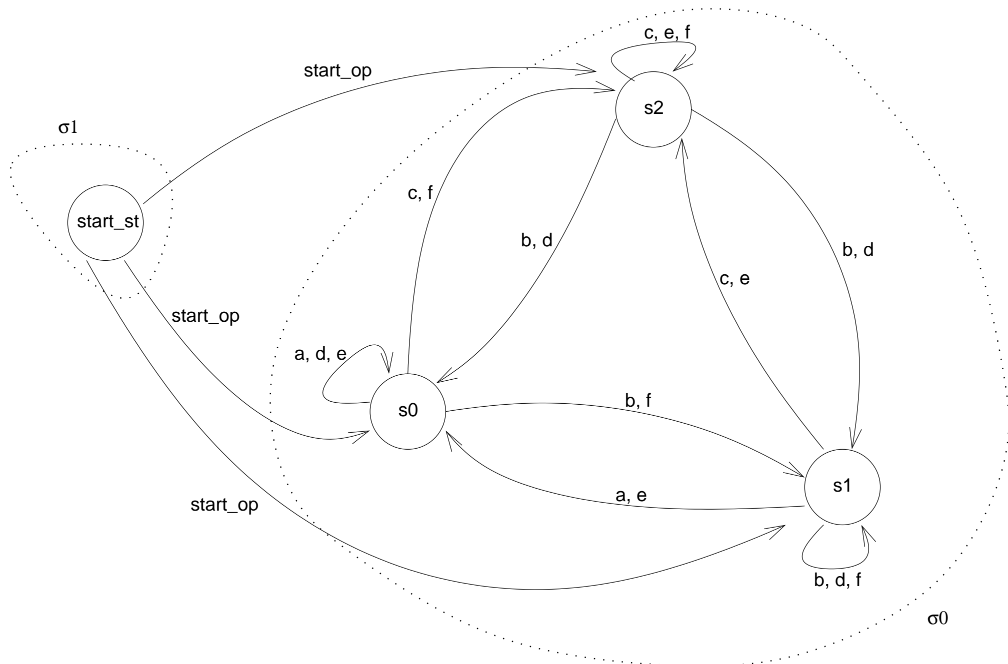


Figure 3: Example of worst case transformation: POTS.

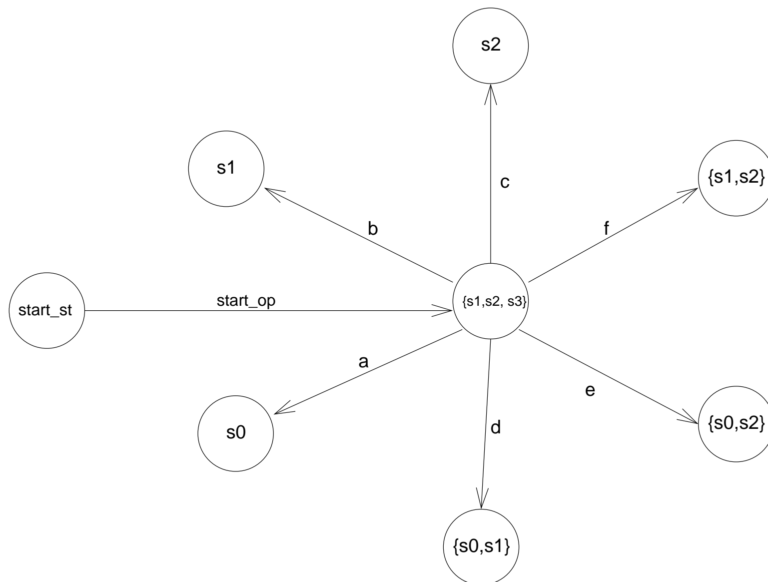


Figure 4: Example of worst case transformation: KTS (not complete).

TSs will never be called. For this reasons, we tried a different approach: an on-line algorithm, that makes the conversion step-by-step, only if the service is selected, that is to say, only if it is necessary. This algorithm is based on the idea that, during the composition, the only informations we need to know are: *(i)* the actual state of the TSs, i.e. if they are in a final, initial, or any other state; *(ii)* if they are able to perform a certain action. The algorithm is an extension of a pre-existing framework used to synthesizing compositions, that use the TLV system [1], it will be illustrated in Section 5, because first we need to introduce the reason why we can use TLV for computing compositions. Since in paper [1] is used the word operation instead of action, from now on we will use these two words as synonyms.

4 Composition via Safety Games

In this Section, we show how a service composition problem instance can be encoded into a game automaton. The main motivation behind this approach is the increasing availability of software systems, such as TLV, which provide *(i)* efficient procedures for strategy computation, *(ii)* convenient languages for representing the problem instance in a modular, intuitive and pretty straight-forward way and *(iii)* efficent data structures for representig boolean functions.

4.1 Safety Game structure

Throughout the rest of the paper, we assume to deal with infinite-run TSs, possibly obtained by introducing fake loops, as customary in LTL verification/synthesis.

Starting from [8], we define a safety-game structure (or $\Box$-game structure or $\Box$-GS, for short) as a tuple $G = \langle \mathcal{X}, \mathcal{V}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$, where:

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a finite set of *state* variables, ranging over *finite* domains *V1, . . . , Vn*, respectively. We assume that each state $s = \langle x, y \rangle$ i.e., a complete assignment of values to variables. Let V be the set of all possible valuations for the variables in $\mathcal{V}$;
- $\mathcal{X} \subseteq \mathcal{V}$ is the set of *environment* variables. Let X be the set of all possible valuations for variables in $\mathcal{X}$, $x \subseteq X$ is called *environment state*;
- $\mathcal{Y} = \mathcal{V} \setminus \mathcal{X}$ is the set of *system* variables. Let Y be the set of all possible valuations for variables in $\mathcal{Y}$, $y \subseteq Y$ is called a *system state*;
- Θ is a formula representing the initial states of the game. It is a boolean combination of expressions $(v_k = \bar{v_k})$ where $v_k \in \mathcal{V}$ and $\bar{v_k} \in V_k$ $(k \in \{1, \ldots, n\})$ (partial assignments are allowed). For such formulae, given a state $\langle x, y \rangle \in V$, we write $\langle x, y \rangle \models \Theta$ if state s satisfies the assignments specified by Θ;
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is the *environment transition relation* which relates a current (unprimed) game state to a possible next (primed) environment state;
- $\rho_s(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$ is the *system transition relation*, which relates a game state plus a next environment state to a next system state;
- $\Box\varphi$ is a formula that represents the invariant property to be guaranteed. In particular, φ has the same form as Θ.

A game state $\langle x', y' \rangle$ is a *successor* of $\langle x, y \rangle$ iff $\rho_e(x, y, x')$ and $\rho_s(x, y, x', y')$. A *play* of G is a maximal sequence of states $\eta : \langle x_0, y_0 \rangle \langle x_1, y_1 \rangle \ldots$ satisfying *(i)* $\langle x_0, y_0 \rangle \models \Theta$, and *(ii)* for each $j \geq 0$, $\langle x_{j+1}, y_{j+1} \rangle$ is a successor of $\langle x_j, y_j \rangle$.

Given a $\Box$-GS G, in a given state $\langle x, y \rangle$ of a game play, the environment chooses an assignment $x' \in X$ such that $\rho_e(x, y, x')$ holds and the system chooses assignment $y' \in Y$ such that $\rho_s(x, y, x', y')$ holds. A play is said to be *winning for the system* if it is infinite and satisfies the winning condition $\Box\varphi$. Otherwise, it is *winning for the environment.* A *strategy* for the system is a partial function $f : (X \times Y)^+ \times X \to Y$ such that for every $\lambda : \langle x_0, y_0 \rangle \ldots \langle x_n, y_n \rangle$ and for every $x' \in X$ such that $\rho_e(x_n, y_n, x')$, $\rho_s(x_n, y_n, x', f(\lambda, x'))$ holds. A play η : $\langle x_0, y_0 \rangle, \langle x_1, y_1 \rangle \ldots$ is said to be *compliant* with a strategy f iff for all $i \geq 0$, $f(\langle x_0, y_0 \rangle \ldots \langle x_i, y_i \rangle, x_{i+1}) = y_{i+1}$. A strategy f is winning for the system from a given state $\langle x, y \rangle$ iff all plays starting from $\langle x, y \rangle$ and compliant with f are so. When such a strategy exists, $\langle x, y \rangle$ is said to be a *winning state* for the

Algorithm 1 WIN. Computes system's maximal set of winning states in a $\Box$-GS

1: $W := \{\langle x, y\rangle \in V \mid \langle x, y\rangle \models \varphi\}$
2: **repeat**
3: $\quad W' := W$;
4: $\quad W := W \cap \pi(W)$;
5: **until** $(W' = W)$
6: **return** W

system. A $\Box$-GS is said to be *winning for the system* if all initial states are so. Otherwise, it is said to be *winning for the environment.*

Our objective is to encode a composition problem into a $\Box$-GS, compute the maximal set of states that are *winning* for the system, and from this, directly computing the orchestrator. Let us show how such *winning set* can be computed in general on a $\Box$-GS. The core of the algorithm is the following operator:

Definition 1 *Let $G = \langle \mathcal{X}, \mathcal{V}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi\rangle$ be a $\Box$-GS as above. Given a set $P \subseteq V$ of a game states $\langle x, y\rangle$ the set of P's controllable predecessor is:*

$$\pi(P) \doteq \{\langle x, y\rangle \in V \mid \forall x'.\rho_e(x, y, x') \to \exists y'.\rho_s(x, y, x', y') \wedge \langle x', y'\rangle \in P\}$$

Intuitively, $\pi(P)$ is the set of states from which the system can force the play to reach a state in P, no matter how the environment evolves. Based on this, Algorithm 1 computes the set of all system's winning states of a $\Box$-GS $G = \langle \mathcal{X}, \mathcal{V}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi\rangle$ as Thoerem 1 shows.

Theorem 1 *Let $G = \langle \mathcal{X}, \mathcal{V}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi\rangle$ be a $\Box$-GS as above and W be obtained as in Algorithm 1. Given a state $\langle \bar{x}, \bar{y}\rangle \in V$, a system's winning strategy f starting from $\langle \bar{x}, \bar{y}\rangle$ exists iff $\langle \bar{x}, \bar{y}\rangle \in W$.*

In fact, one can define a system's winnig strategy $f(\langle x_0, y_0\rangle, \ldots, \langle x_i, y_i\rangle, x) = y$, by picking up, for each x such that $\rho_e(x_i, y_i, x)$ holds, any $\langle x, y\rangle \in W$.

4.2 From composition problem to safety games

Before giving technical details about the reduction from composition problem instances to $\Box$-GS, let us introduce our approach from a high-level perspective. Recall that the service composition problem consists in finding a orchestrator able to coordinate a community in a way such that the obtained system can always satisfy the requests of a client compliant with a given deterministic target service. First, note that given the target service, all and only legal client behaviors can be obtained by executing the target service in every possible way. Second, observe that the target service is a virtual entity whose operations are to be actually executed by one available service, subject to its current state and capabilities. Therefore, target service and community can be soundly thought to evolve synchronously, the former issuing requests for operations, i.e. acting as a client, and the latter satisfying them, i.e. actually implementing the

target service. In addition, observe that community services are mutually asynchronous, i.e., *exactly* one moves at each step. Finally, we can isolate the only part that needs to be synthesized, i.e., the orchestrator. It is an automaton which, synchronously with both the target service and the community, outputs an identifier to delegate one available service to execute the operation currently requested.

In order to encode the composition problem as a safety-game structure, we need first to individuate which place each component, e.g. target and available services, will occupy in the game representation. The above remarks are intended for this purpose. Conceptually, our goal is to refine an *uncostrained* orchestrator, that is, an automaton capable of selecting, at each step, one among all the available services, in a way such that the community is always able to satisfy target service requests. This suggests immediately to represent the orchestrator as the system in the game structure, that is the entity we want to synthesize a winning strategy for. Consequently, and coherently with above observations, community services and target service, will be properly combined and encoded as environment. Finally, the winning condition needs to be defined. We remark that the goal of system's strategies is to guarantee the winning condition holds throughout all possible plays induced by the strategy itself, starting from an initial state. The winning condition will be formally encoded as conjunction of the following high-level properties:

- (game's) initial state is winning by default;
- if target service is in a final state, all community services do, as well;
- the servie selected by the orchestrator is able to perform the action currently requested by the target service.

Now, let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle$ be a community, where $\mathcal{S}_i = \langle A_i, S_i, s_{i0}, F_i, \delta_i, \sigma_i \rangle$ $(i = 1, \ldots, n)$ and $\mathcal{S}_t = \langle A_t, S_t, s_{t0}, F_t, \delta_i \rangle$ a target service. We assume, without loss of generality, that every service of the comunity was a POTS, because a TS is simply a POTS with an injective observability function. From $\mathcal{C}$ and $\mathcal{S}_t$ we derive a $\Box$-GS $G = \langle \mathcal{X}, \mathcal{V}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ as follows:

- $\mathcal{V} = \{s_t, \vec{s_1}, \ldots, \vec{s_n}, a, ind\}$, where:
  - $s_t \in S_t \cup \{init\}$;
  - $\vec{s_i} \in 2^{S_i} \cup \{init\}$ $(i = 1, \ldots, n)$;
  - $a \in A_t \cup \{init\}$;
  - $ind \in \{1, \ldots, n\} \cup \{init\}$;

  with an intuitive semantics: each complete valutation of $\mathcal{V}$ represents: *(i)* the current state of the target service and community (variables $s_t, \vec{s_1}, \ldots, \vec{s_n}$); *(ii)* the operation to be performed next, that belongs to the set of action of the target service; *(iii)* the available service selected to perform it (variable ind). Special value $init$ has been introduced for convenience, so as to have fixed initial state.

- $\mathcal{X} = \{s_t, \vec{s_1}, \ldots, \vec{s_n}, a\}$ is the set of environment variables;
- $\mathcal{Y} = \{ind\}$ is the (singleton) set of system variables;
- $\Theta = (s_t = init) \wedge (\bigwedge_{i=0,\ldots,n}(\vec{s_i} = init)) \wedge (a = init) \wedge (ind = init)$;
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is defined as follows:
  - $\langle\langle init, \ldots, init\rangle, init, \langle s_t, \vec{s}_1, \ldots, \vec{s}_n, a\rangle\rangle \in \rho_e$ iff $s_t = s_{t0}$, $\vec{s}_i = \vec{s}_{i0}$ for $i = (1, \ldots, n)$ and there exists a transition $\langle s_{t0}, a, s_{tj}\rangle \in \delta_t$ for some $j \in \{0, \ldots, |S_t|\}$;
  - if $s_t \neq init$, $\vec{s_i} \neq init$ with $i = (1, \ldots, n)$, $a \neq init$ and $ind \neq init$, then $\left\langle\langle s_t, \vec{s_1}, \ldots, \vec{s_n}, a\rangle, ind, \langle s'_t, \vec{s_1}', \ldots, \vec{s_n}', a'\rangle\right\rangle \in \rho_e$ iff the followings hold in conjunction:
    1. there exists a transition $\langle s_t, a, s'_t\rangle \in \delta_t$;
    2. Recall Section 3, if we suppose to have associated KTSs to all POTSs of the community, either there exists a transition $\langle \vec{s}_{ind}, a, \vec{s}'_{ind}\rangle \in \delta_{k\ ind}$ or $\vec{s}_{ind} = \vec{s}'_{ind}$ i.e., service wrongly makes no move and the error violates the safety condition φ, see below;
    3. $\vec{s_i} = \vec{s_i}'$ for all $i = 1, \ldots, n$ such that $i \neq ind$;
    4. there exist a transition $\langle s'_t, a, s''_t\rangle \in \delta_t$ for some $s''_t \in S_t$.
  - $\langle\langle init, \ldots, init\rangle, init, \langle s_t, \vec{s}_1, \ldots, \vec{s}_n, a\rangle, ind'\rangle \in \rho_s$ iff $ind' \in \{1, \ldots, n\}$;
  - Formula φ is defined depending on the current state, operation and service selection, as:

$$\varphi \doteq \Theta \vee ((\bigwedge_{i=1}^{n} \neg fail_i) \wedge (final_t \rightarrow \bigwedge_{i=1}^{n} final_i))$$

    where:

    * $fail_i \doteq (ind = i) \wedge (\bigwedge_{\langle \vec{s}, op, \vec{s}'\rangle \in \delta_{k\ i}}(op \neq a \vee \vec{s_i} \neq \vec{s}))$,encodes the fact that service i has been selected but, in tis current state, no transition can take place which executes the requested operation;
    * $final_i \doteq \forall s \in \vec{s_i} \rightarrow s \in F_i$, for $i = (1, \ldots, n)$ and $final_t \doteq \vee_{s \in F_t}(s_t = s)$, encodes the fact that the service its currently in one of its final states.

Once we have computed the set W, and encoded our composition problem in a $\Box$-GS, *synthesizing* an orchestrator becomes an easy task. There is a well-defined procedure, that given the set W and a $\Box$-GS, builds a finite state program that returns, at each point, the set of available behaviours capable of performing a target-conformant operation. We call such a program *orchestrator generator*, or simply *PG*. Formally:

Theorem 2 *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle$ be a community and $\mathcal{S}_t$ a target service, where, as usual, $\mathcal{S}_t = \langle A_t, S_t, s_{t0}, F_t \delta_i \rangle$ and $\mathcal{S}_i = \langle A_i, S_i, s_{i0}, F_i, \delta_i, \sigma_i \rangle$ for $i = (1, \ldots, n)$. From $\mathcal{C}$ and $\mathcal{S}_t$ we derive a $\Box$-GS $G = \langle \mathcal{X}, \mathcal{V}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ as shown above. Let W be the system's winning set for G computed as in Algorithm 1. If $\langle init, \ldots, init \rangle \in W$ then the orchestrator generator $PG = \langle A_t, \{1, \ldots, n\}, \Sigma, \partial, \omega \rangle$ of $\mathcal{C}$ for $\mathcal{S}_t$, can be built from W as follows:*

- *A_t is the usual set of operations;*
- *$\{1, \ldots, n\}$ is the set of available services' indexes;*
- *$\Sigma \in 2^{S_1} \times \cdots \times 2^{S_n} \times S_t$ is the set of states of PG, such that $\langle \vec{s_1}, \ldots, \vec{s_n}, s_t \rangle \in \Sigma$ iff there exists a game state $\langle \vec{s_1}, \ldots, \vec{s_n}, s_t, a, ind \rangle \in W$ for some $a \in A_t$ and $ind \in \{1, \ldots, n\}$;*
- *$\partial \in (\Sigma \times A_t \times \{1, \ldots, n\} \times \Sigma)$ is the transition relation, such that $\langle \langle \vec{s_1}, \ldots, \vec{s_n}, s_t \rangle, a, k, \langle \vec{s_1}', \ldots, \vec{s_n}', s_t' \rangle \rangle \in \partial$ iff $\langle \vec{s_1}, \ldots, \vec{s_n}, s_t, a, k \rangle \in W$ and there exists $a' \in A_t$ and $k' \in \{1, \ldots, n\}$ such that $\langle \langle \vec{s_1}, \ldots, \vec{s_n}, s_t, a, k \rangle, \langle \vec{s_1}', \ldots, \vec{s_n}', s_t', a', k' \rangle \rangle \in \rho_s$;*
- *$\omega : \Sigma \times A_t \to 2^{\{1,\ldots,n\}}$ is defined as $\omega(\langle \vec{s_1}, \ldots, \vec{s_n}, s_t \rangle, a) = \{i \in \{1, \ldots, n\} \mid \langle \vec{s_1}, \ldots, \vec{s_n}, s_t, a, i \rangle \in W\}$.*

Intuitively, PG is a finite state transducer, that, given an operation a (compliant with the target service), outputs, through ω, the set of *all* available services able to perform a next, according to the $\Box$-GS and W. The role of PG is very simple. In fact, given a current state $\langle \vec{s_1}, \ldots, \vec{s_n}, s_t \rangle$ and an operation $a \in A_t$ to be executed, a service selection s "good" (i.e the selected service can actually execute the operation and the whole community can still simulate the target service) if and only if W contains a tuple $\langle \vec{s_1}, \ldots, \vec{s_n}, s_t, a, ind \rangle$ for some $ind \in \{1, \ldots, n\}$. Consequently, at each step, on the basis of the current state s_t of the target service, the state $\langle \vec{s_1}, \ldots, \vec{s_n} \rangle$ of the community and the operation a, one can select a tuple from W, extract the *ind* component, and use it for next service selection.

5 Using TLV for computing compositions

In Section 4 is shown that the synthesis of a composition can be reduced to the search for a system's winning set and strategy in a safety-game structure. In order to solve such problems, we use TLV, that is a software for verification and synthesis of general LTL specifications, based on symbolic manipulation of states, through Binary Decision Diagrams (BDDs). It takes two inputs: a synthesis procedure, encoded in TLV-BASIC, and a LTL specification, encoded in SMV, to be manipulated by this procedure. In particular, we refer to a TLV-BASIC pre-existing procedure for safety games (aggiungere pubblicazione) which takes as input an LTL specification supposed to encode a safety-game structure and returns, if any, the system's maximal winning set. The purpose of

this section is to show a methodology for producing, starting from an abstract description of a composition problem, the SMV code that the procedure can take as input, to automatically generate, if any, the system's winning set. Here we describe only the metodology for describing a POTS in SMV, all other modules are identical to the ones described in [1].

5.1 The SMV encoding

The encoding presented in [1] is composed by several MODULEs:

- *System module* represents the *uncostrained orchestrator*, which, apart from initialization, at each step outputs an index representing the service that has to perform the actual operation;

- *Environment module* represents the *abstract environment*. As described in [1], when a composition problem is reformulated as a safety-game structure, the environment state is composed of: *(i)* the state of each available service; *(ii)* the state of the target service; and *(iii)* the operation to be performed next. State transitions are subject to the following rules. The target service's component moves to the (only) successor obtained by performing the requested operation in the current state, as prescribed by target service's transition function, whereas all available service's components but one, i.e., the selected one, move by looping in their current state, not failing; the selected one may either generate a failure and loop in its current state (requested operation operation cannot be performed by selected service) or correctly move according to its transition relation. Finally, the operation changes to one of those the target service can perform in its successor state. Summing up, in the game formulation, all components synchronously move at each step, even though some of them are forced to loop. It is important to remark this fact because, in our encoding, the environment is built up from several modules which are supposed, by smv semantics, to move synchronously. Recall that available services' modules are defined so that they loop on the current state both when they are not selected and when they are but the service they refer to is not able to perform the requested action in current state. In the latter case, however, the expression failure becomes true. This perfectly reflects the semantics associated to the environment when formulating a composition problem as a safety-game structure. The aim of the module `Env` is to combine all modules defined so far and to define when the goal property g is violated (and the purpose of the synthesis is to build up a orchestrator such that this never happens). In order to achieve this aim, the `DEFINE` section of the environment is build as follows:

```
DEFINE
  initial := s1.initial & s2.initial & ...
    & sn.initial & target.initial & operation=start_op;
```

```
failure := (s1.failure | s2.failure | ...  | sn.failure) |
(target.final & !(s1.final & s2.final & ...  & sn.final));
```

These definitions state that: *(i)* the environment is in the initial state if all services that belong to the community and the target service are in their initial states, and the operation is the fictitious initial start_op; *(ii)* the environment failure, i.e. the composition fails, if anyone of the services fails or the traget service is in a final state but there is at least one service that is not.

- *Main module* puts toghether the environment and the system to build the game and defines the g forumula as follows:

  ```
  DEFINE
    good := (sys.initial & env.initial) | !(env.failure);
  ```

  with immediate understanding. The synthesis algorithm interprets: *(i)* the first declared module as environment; *(ii)* the second one as system; and *(iii)* the expression good as the invariant property to be guaranteed by the synthesized orchestrator.

- *Target module* desribes in the SMV's usual way the TS that represents the service that we want to synthesize. At every iteration, it outputs the operation that has to be performed.

- *Community service module.* Every service of the community is represented as a module that takes as input two parameters: the index generated by the *system module* and the operation that has to be performed. Every service has a different index, so, at every iteration and for every service s, if the index corresponds to s, then either it performs the operation as descibed in its `TRANS` section or it fails, otherwise it loops in its current state. If the service is a standard nondeterministic TS, then the smv encoding is trivial. If it is a POTS, the encoding has to be extended to "implement", in a some way, an algorithm that verifies whether the operation can be performed or not.

5.1.1 SMV encoding of a POTS

As anticipated above, the main idea that suggested this encoding is that we don't need to have knowledge of the full observable associated KTS. In fact, we don't even need to build an associate KTS. The only information we require at every iteration is the current macrostate in which the selected service actually is. From the current macrostate, and, obviously, from the description of the POTS, we can derive all other information we need, for instance, if the service can perform the operation, or every future macrostate the service could reach by performing the operation. This encoding is made up in two main part: the first is a trivial description of the original POTS, the latter is the algorithm itself, that taking

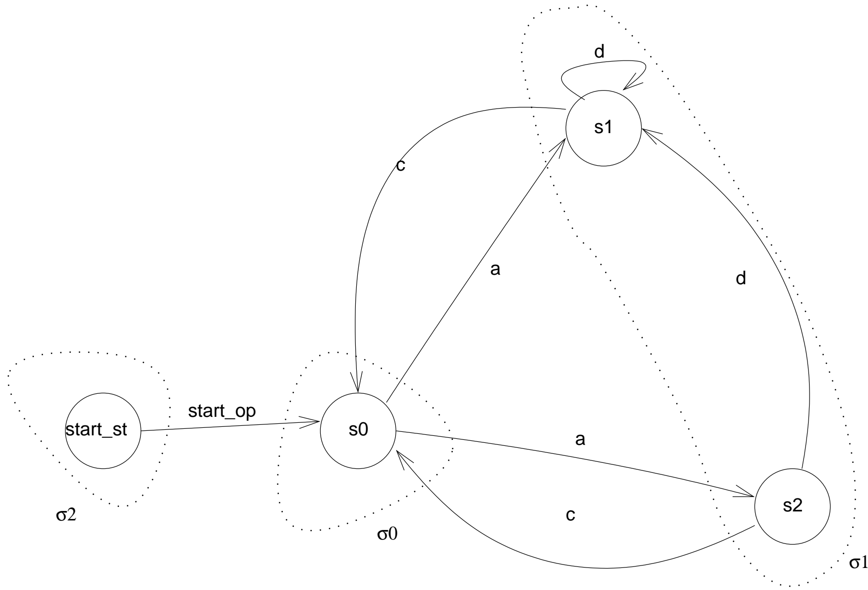


Figure 5: Example of simple POTS.

into account the actual macrostate, the operation, and the description of the POTS, computes, if it doesn't fail, all possible arrival macrostates by performing the operation, and then "moves" nondeterministically in one of these.

We analyze the encoding in detail by the means of example in Figure 5.

As shown before, every POTS is a module of the smv file, and takes two inputs, an index and an operation:

```
MODULE kts1(index, op)
```

In the `VAR` section, we define two variables: the first represents the current macrostate and the latter is `choice`, its meaning will be explained later on. As we have to deal with macrostate, i.e. set of states, we have decided to represent a set in smv by his characteristic function, ($cf_{curr_macrostate} : S_{POTS} \rightarrow \{0, 1\}$) so, the variable curr_macrostate is an array of boolean values of length number of POTS' states. The i-th entry is setted to 1 if and only if the i-th state belongs to current macrostate.

```
VAR
  curr_macrostate :  array 0..3 of {0, 1}; --0=s0, 1=s1,
                                           --2=s2, 3=start_st.
  choice :  {0, 1, 2};
```

In the `DEFINE` section, we store all informations about the POTS (with the exception of the set A, that is not the POTS' set of actions, but the set

of operations that the *Target Service* can perform) i.e. the tuple $POTS = \langle A, S, s_0, \delta, F, \sigma \rangle$, using once more characteristic functions. More precisely, with reference to the example of Figure 5, the initial state ($cf_{initial_state} : S_{POTS} \rightarrow \{0,1\}$) is an array of POTS' number of sates, and only the state that refers to start_st is setted to true:

```
DEFINE
  POTS_initial_state[0] := 0;
  POTS_initial_state[1] := 0;
  POTS_initial_state[2] := 0;
  POTS_initial_state[3] := 1;
```

Analogously, for the final states ($cf_{final_states} : S_{POTS} \rightarrow \{0,1\}$). In our example, only s_0 is a final state:

```
POTS_final_states[0] := 1;
POTS_final_states[1] := 0;
POTS_final_states[2] := 0;
POTS_final_states[3] := 0;
```

Since we ecode also transition relations with his characteristic function ($cf_{transitions} : S_{POTS} \times A_{Target} \times S_{POTS} \rightarrow \{0,1\}$), we need $|S_{POTS}| \cdot |A_{Target}| \cdot |S_{POTS}|$ definitions:

```
POTS_trans[0][0][0] := 0; -- Second index (operation):
POTS_trans[0][0][1] := 1; -- 0=a, 1=b, 2=c, 3=d, 4=start_op.
POTS_trans[0][0][2] := 1;
POTS_trans[0][0][3] := 0;
...
POTS_trans[3][4][3] := 0;
```

The first four lines mean that in the state s_0 by performing action a, i.e. action 0, the next POTS' state could be nondeterministically either s_1 or s_2. Is important to remark that we use numbers to identify operations, because since `op` is an input parameter, we can use it, in certain cases, as an arrays index.

Informations about observability function is encoded in the usual way by the characteristic function ($cf_{observations} : \mathcal{C}(\sigma) \times S_{POTS} \rightarrow \{0,1\}$), so we need $|\mathcal{C}(\sigma)| \cdot |S_{POTS}|$ definitions:

```
POTS_oss[0][0] := 1; -- First index:
POTS_oss[0][1] := 0; -- 0=sigma0, 1=sigma1, 2=sigma2.
POTS_oss[0][2] := 0;
POTS_oss[0][3] := 0;
...
POTS_oss[2][3] := 1;
```

By convention, we use a new observability value for the fictitious state *start_st*.

Here ends the description of the POTS, now we define some variables that are necessary for the algorithm.

As shown before we need to know if it is possible to perform an operation in a certain state. In a macrostate s is possible to perform an operation a if and only if in all states that belongs to s it is possible to perform a. So we define:

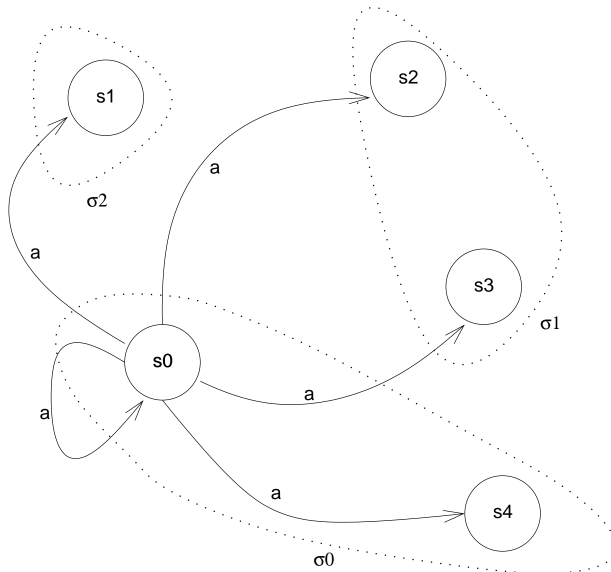
```
op_possible :=
  (curr_macrostate[0]->
     (POTS_trans[0][op][0] | POTS_trans[0][op][1] |
     POTS_trans[0][op][2] | POTS_trans[0][op][3])) &
  (curr_macrostate[1]->
     (POTS_trans[1][op][0] | POTS_trans[1][op][1] |
     POTS_trans[1][op][2] | POTS_trans[1][op][3])) &
  (curr_macrostate[2]->
     (POTS_trans[2][op][0] | POTS_trans[2][op][1] |
     POTS_trans[2][op][2] | POTS_trans[2][op][3])) &
  (curr_macrostate[3]->
     (POTS_trans[3][op][0] | POTS_trans[3][op][1] |
     POTS_trans[3][op][2] | POTS_trans[3][op][3]));
```

It is imporant to notice that, since we have only informations on POTS' transitions, to know if it is possible to perform an operation op, we have to check, for every s, if there is at least one state s' such that $POTS_trans[s][op][s'] = 1$.

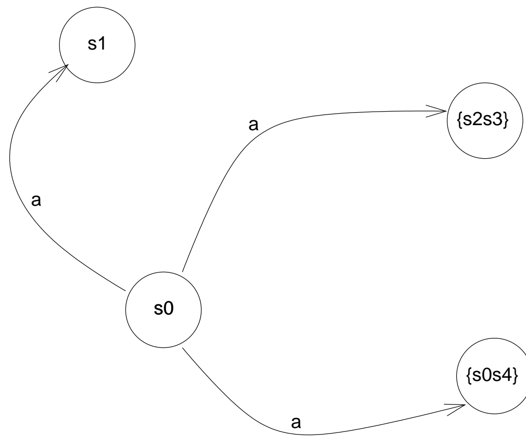
The most important part of the algorithm is how to compute the possible arrival states. Suppose that the TS was in a generic macrostate, then, if the operation is possible, we have to know all possible arrival states, grouped by observations, and then the next macrostate will be formed nondeterministically among all states that belongs to one of the observations. An example is shown in Figure 6.

We compute possible arrival states in this way: for every observation o, and for every state s, s is a possible arrival state if the operation is possible in the current macrostate $\vec{s}$ and there is at least one state $s' \in \vec{s}$ such that $(s', op, s) \in \delta_{POTS}$.

```
imm_oss[0][0] :=
  op_possible &
  POTS_oss[0][0] &
  ( (curr_macrostate[0] & POTS_trans[0][op][0]) |
  (curr_macrostate[1] & POTS_trans[1][op][0]) |
  (curr_macrostate[2] & POTS_trans[2][op][0]) |
  (curr_macrostate[3] & POTS_trans[3][op][0]) );
...
imm_oss[2][3] := ...
```



(a) Nondeterministic POTS' transition. S0 is the initial state.



(b) Associated KTS. Notice that the "new" transition is nondeterministic, and arrival states are grouped by observations.

Figure 6: Example of nondeterminitic transition.

So we define $|\mathcal{C}(\sigma)| \cdot |S_{POTS}|$ number of variables which, at every iterations, represent possible arrival states' characteristic function ($cf_{imm_oss} : \mathcal{C}(\sigma) \times S_{POTS} \rightarrow \{0, 1\}$).

As anticipated above, we have to know if the current macrostate is inital or final. A TS is in an initial state if the input operation is equal to start_op, input index is equal to 0, and for the (only) state s that belongs to the current macrostate $\vec{s}$ holds $s = s_0$.

```
initial :=
  op=4 &
  index=0 &
  (curr_macrostate[0]->POTS_initial_state[0]) &
  (curr_macrostate[1]->POTS_initial_state[1]) &
  (curr_macrostate[2]->POTS_initial_state[2]) &
  (curr_macrostate[3]->POTS_initial_state[3]);
```

A macrostate is final if all states that belongs to that macrostate are final.

```
final :=
  (curr_macrostate[0]->POTS_final_states[0]) &
  (curr_macrostate[1]->POTS_final_states[1]) &
  (curr_macrostate[2]->POTS_final_states[2]) &
  (curr_macrostate[3]->POTS_final_states[3]);
```

The `DEFINE` section ends with the definition of failure, i.e. the condition that has not to be veirfied in order for the goal propery g to hold. A TS fails if it is selected by the *System*, i.e. the orchestrator, but it cannot perform the requested operation.

```
failure :=
  index=1 & !(op_possible);
```

The `INIT` module is trivial, we only initialize the current macrostate with start_st and the `choice` with an arbitrary value.

```
INIT
  curr_macrostate[0]=0 &
  curr_macrostate[1]=0 &
  curr_macrostate[2]=0 &
  curr_macrostate[3]=1 &
  choice=2
```

In the `TRANS` section, if the service is selected, it moves nonderministically in one of the imm_oss, otherwise, it loops in its current macrostate. This can be done by choosing nondeterministically an integer, the variable `choice` that has as many values as the number of obervations, and then update the current macrostate with the states of respective, nondeterministically chosen observation. It is important to notice that we have to be careful that the arrival imm_oss was not empty.

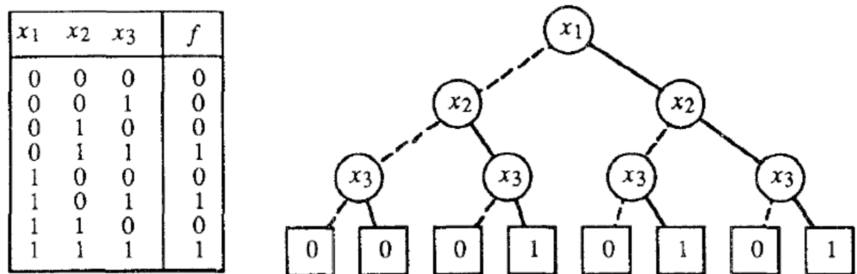
```
TRANS
  case (index=0 | index=1) :
    next(choice) in {0, 1, 2}
    &
    ( next(choice)=0 ->
    ( next(curr_macrostate[0])=imm_oss[0][0] &
    next(curr_macrostate[1])=imm_oss[0][1] &
    next(curr_macrostate[2])=imm_oss[0][2] &
    next(curr_macrostate[3])=imm_oss[0][3] &
    ( next(curr_macrostate[0]) | next(curr_macrostate[1]) |
    next(curr_macrostate[2]) | next(curr_macrostate[3])) ) )
    &
    ( next(choice)=1 ->
    ( next(curr_macrostate[0])=imm_oss[1][0] &
    next(curr_macrostate[1])=imm_oss[1][1] &
    next(curr_macrostate[2])=imm_oss[1][2] &
    next(curr_macrostate[3])=imm_oss[1][3] &
    ( next(curr_macrostate[0]) | next(curr_macrostate[1]) |
    next(curr_macrostate[2]) | next(curr_macrostate[3])) ) )
    &
    ( next(choice)=2 ->
    ( next(curr_macrostate[0])=imm_oss[2][0] &
    next(curr_macrostate[1])=imm_oss[2][1] &
    next(curr_macrostate[2])=imm_oss[2][2] &
    next(curr_macrostate[3])=imm_oss[2][3] &
    ( next(curr_macrostate[0]) | next(curr_macrostate[1]) |
    next(curr_macrostate[2]) | next(curr_macrostate[3]) ) ) );

  !(index=1 | index=0) :
    next(curr_macrostate[0])=curr_macrostate[0] &
    next(curr_macrostate[1])=curr_macrostate[1] &
    next(curr_macrostate[2])=curr_macrostate[2] &
    next(curr_macrostate[3])=curr_macrostate[3] &
    next(choice)=choice;
```

The variable choice is necessary, because as we have to update a set of state, if we had encoded:

```
next(curr_macrostate[0]) in
{imm_oss[0][0], imm_oss[1][0], imm_oss[2][0]} &
next(curr_macrostate[1]) in
{imm_oss[0][1], imm_oss[1][1], imm_oss[2][1]} &
...
```

then we would obtained:

```
next(curr_macrostate[0])=imm_oss[0][0] &
```



x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 7: Truth table and decision tree representations of a Boolean function. A dashed (solid) branch denotes the case where the decision variable is 0(1).

```
next(curr_macrostate[1])=imm_oss[2][1] &
...
```

It is important to underline that, even if this algorithm is intrinsically exponential, this encoding is polinomial w.r.t. the input, so the exponential cost is paied only if it necessary, and in any case is handled by TLV.

6 TLV and Ordered Binary Decision Diagrams

The key idea of OBDDs is that by restricting the representation, boolean manipulation becomes much simpler computationally. Consequently, they provide a suitable data structure for a symbolic boolean manipulator.

6.1 Introduction to BDDs

A binary decision diagram represents a boolean function as a rooted, directed acyclic graph. As an example, Figure 7 illustrates a representation of the function $f(x_1, x_2, x_3)$ defined by the truth table given on the left, for the special case where the graph is actually a tree. Each nonterminal vertex v is labeled by a variable *var(u)* and has arcs directed toward two children: *lo(v)* (shown as a dashed line) corresponding to the case where the variable is assigned 0 and *hi(v)* (shown as a solid line) corresponding to the case where the variable is assigned 1. Each terminal vertex is labeled 0 or 1. For a given assignment to the variables, the value yielded by the function is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label. Due to the way the branches are ordered in this figure, the values of the terminal vertices, read from left to right, match those in the truth table, read from top to bottom.

For an Ordered BDD (OBDD), we impose a total ordering $<$ over the set of variables and require that for any vertex u, and either nonterminal child v, their

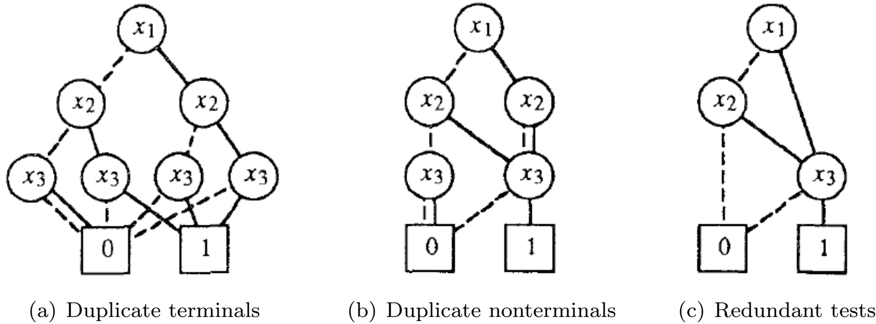
respective variables must be ordered $var(u) < var(v)$. In the decision tree of Figure 7, for example, the variables are ordered $x_1 < x_2 < x_3$. In principle, the variable ordering can be selected arbitrarily, the algorithms will operate correctly for any ordering. In practice, selecting a satisfactory ordering is critical for the efficient symbolic manipulation. We define three transformation rules over these graphs that do not alter the function represented:

- **Remove Duplicate Terminals.** Eliminate all but one terminal vertex with a given label and redirect all arcs into the eliminated vertices to the remaining one.
- **Remove Duplicate Nonterminals.** If nonterminal vertices u and v have $var(u) = var(v)$, $lo(u) = lo(u)$, and $hi(u) = hi(v)$, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.
- **Remove Redundant Tests.** If nonterminal vertex v has $lo(u) = hi(u)$, then eliminate u and redirect all incoming arcs to $lo(v)$.

Starting with any BDD satisfying the ordering property, we can reduce its size by repeatedly applying the transformation rules. We use the term "OBDD" to refer to a maximally reduced graph that obeys some ordering. For example, Figure 8 illustrates the reduction of the decision tree shown in Figure 7 to an OBDD. Applying the first transformation rule (a) reduces the eight terminal vertices to two. Applying the second transformation rule (b) eliminates two of the vertices having variable x_3 and arcs to terminal vertices with labels 0 (*lo*) and 1 (*hi*). Applying the third transformation rule (c) eliminates two vertices: one with variable x_3 and one with variable x_2. In general, the transformation rules must be applied repeatedly, since each transformation can expose new possibilities for further ones. The OBDD representation of a function is canonical, for a given ordering, two OBDDs for a function are isomorphic. This property has several important consequences. Functional equivalence can be easily tested. A function is satisfiable iff its OBDD representation does not correspond to the single terminal vertex labeled 0. Any tautological function must have the terminal vertex labeled 1 as its OBDD representation. If a function is independent of variable x, then its OBDD representation cannot contain any vertices labeled by x. Thus, once OBDD representations of functions have been generated, many functional properties become easily testable. As figg.7 and 8 illustrate, we can construct the OBDD representation of a function given its truth table by constructing and reducing a decision tree. This approach is practical, however, only for functions of a small number of variables, since both the truth table and the decision tree have size exponential in the number of variables. Instead, OBDDs are generally constructed by "symbolically evaluating" a logic expression or logic gate network using the *APPLY* operation described in the next section.

6.2 Construction of OBDDs

A number of symbolic operations on boolean functions can be implemented as graph algorithms applied to the OBDDs. These algorithms obey an important



(a) Duplicate terminals (b) Duplicate nonterminals (c) Redundant tests

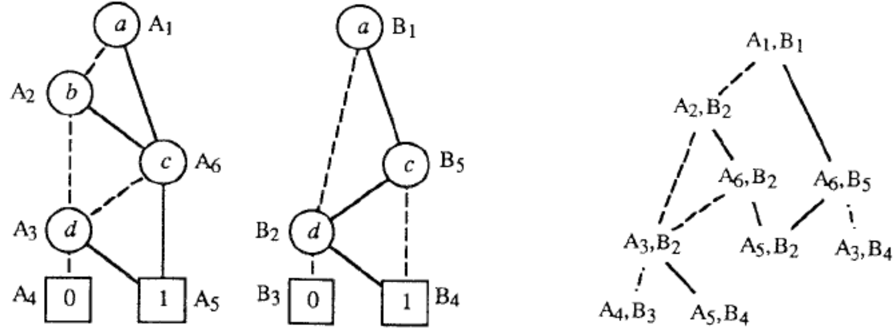
Figure 8: Reduction of decision tree to OBDD. Applying the three reduction rules to the tree of Figure 7 yields the canonical representation of the function as an OBDD.

closure property: given that the arguments are OBDDs obeying some ordering, the result will be an OBDD obeying the same ordering. Thus we can implement a complex manipulation with a sequence of simpler manipulations, always operating on OBDDs under a common ordering. Users can view a library of BDD manipulation routines as an implementation of a boolean function abstract data type, except for the selection of a variable ordering, all of the operations are implemented in a purely mechanical way. The user needs not to be concerned with the details of the representation or the implementation.

6.2.1 The *APPLY* operator

The *APPLY* operation generates boolean functions by applying algebraic operations to other functions. Given argument functions f and g, plus binary boolean operator $\langle op \rangle$, (e.g., *AND* or *OR*) *APPLY* returns the function $f \langle op \rangle g$. This operation is central to a symbolic boolean manipulator. With it we can complement a function f by computing $f \oplus 1$. Given functions f and g, and "don't care" conditions expressed by the function d (i.e., $d(\vec{x})$ yields 1 for those variable assignments $\vec{x}$ for which the function values are unimportant), we can test whether f and g are equivalent for all "care" conditions by computing $(f \bar{\oplus} g) + d$ and test whether the result is the function 1. The *APPLY* algorithm operates by traversing the argument graphs depth first, while maintaining two hash tables: one to improve the efficiency of the computation and one to assist in producing a maximally reduced graph. Note that whereas earlier presentations treated the reduction to canonical form as a separate step [Bryant 1986], the following algorithm produces a reduced form directly. To illustrate this operation, we will use the example of applying the + operation to the functions $f(a, b, c, d) = (a + b) \cdot c + d$ and $g(a, b, c, d) = (a \cdot \bar{c}) + d$, having the OBDD representations shown in Figure 9(a).

The implementation of the *APPLY* operation relies on the fact that algebraic



(a) Example arguments to APPLY operation. Vertices are labeled for identification during the execution trace.

(b) Execution trace for APPLY operation with operation $+$. Each evaluation step has as operands a vertex from each argument graph.

Figure 9: Example of *APPLY* operation.

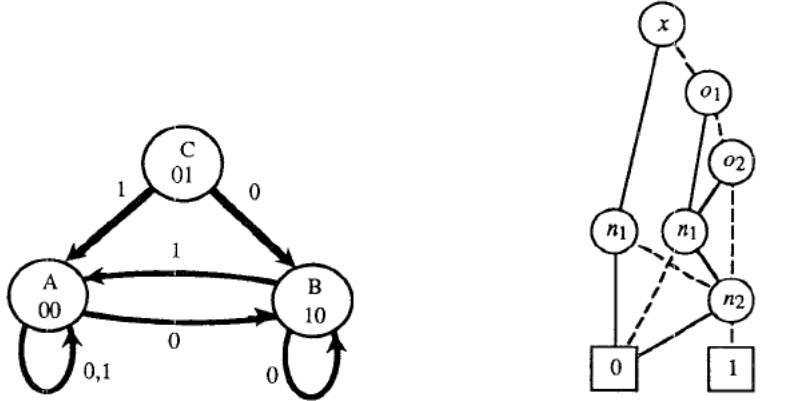
operations "commute" with the Shannon expansion for any variable x:

$$f \langle op \rangle g = x \cdot (f\ |_{x \leftarrow 1} \langle op \rangle\ g\ |_{x \leftarrow 1}) + \bar{x} \cdot (f\ |_{x \leftarrow o} \langle op \rangle\ g\ |_{x \leftarrow o})$$

This equation forms the basis of a recursive procedure for computing the OBDD representation of $f \langle op \rangle g$. For our example, the recursive evaluation structure is illustrated in Figure 9(b). Note that each evaluation step is identified by a vertex from each of the argument graphs. Suppose functions f and g are represented by OBDDs with root vertices r_f and r_g, respectively. For the case where both r_f and r_g are terminal vertices, the recursion terminates by returning an appropriately labeled terminal vertex. In our example, this occurs for the evaluations A_4, B_3 and A_5, B_4. Otherwise, let variable x be the splitting variable, defined as the minimum of variables $var(r_f)$ and $var(r_g)$. OBDDs for the functions $f\ |_{x \leftarrow 1} \langle op \rangle\ g\ |_{x \leftarrow 1}$ and $f\ |_{x \leftarrow o} \langle op \rangle\ g\ |_{x \leftarrow o}$, are computed by recursively evaluating the restrictions of f and g for value 1 (indicated in 9(b) by solid lines) and for value 0 (indicated by the dashed lines). For our example, the initial evaluation with vertices A_1, B_1 causes recursive evaluations with vertices A_2, B_2 and A_6, B_5.

6.3 Transitions represented as OBDDs

Classic algorithms that analize finite-state system construct an explicit representation of the state graph and then analyze its path and cycle structure [Clarke et al. 1986]. These techniques become impractical, however, as the number of states grows large. Unfortunately, even relatively small digital systems can have very large state spaces. More recently, researchers have developed "symbolic" state graph methods, in which the state transition structure is represented as a boolean function [Burch et al. 1990a; Coudert et al. 1990]. This involves



(a) Explicit representation of nondeterministic finite-state machine. The size of the representation grows linearly with the number of states.

(b) Symbolic representation of nondeterministic finite-state machine. The number of variables grows logarithmically with the number of states.

Figure 10: Explicit and symbolic representation of nondeterministic finite-state machine.

first selecting binary encodings of the system states and input alphabet. The next-state behavior is described as a relation given by a characteristic function $\delta(\vec{x}, \vec{o}, \vec{n})$ yielding 1 when input $\vec{x}$ can cause a transition from state $\vec{o}$ to state $\vec{n}$. As an example, Figure 10(a) illustrates an OBDD representation of the nondeterministic automaton having the state graph illustrated in Figure 10(a).

This example represents the three possible states using two binary values by the encoding $\sigma(A) = [0,0], \sigma(B) = [1,0]$, and $\sigma(C) = [0,1]$. Observe that the unused code value $[1,1]$ can be treated as a "don't care" value for the arguments $\vec{o}$ and $\vec{n}$ in the function δ. In the OBDD of Figure 10(b), this combination is treated as an alternate code for state C to simplify the OBDD representation. For such a small automaton, the OBDD representation does not improve on the explicit representation. For more complex systems, on the other hand, the OBDD representation can be considerably smaller. McMillan [1992] has characterized some conditions under which the OBDD representing the transition relation for a system grows only linearly with the number of system components, whereas the number of states grows exponentially. In particular, this property holds when both *(i)* the system components are connected in a linear or tree-structure and *(ii)* each component maintains only a bounded amount of information about the state of the other components. This bound holds for ringconnected systems, as well, since a ring can be "flattened" into a linear chain of bidirectional links. Given the OBDD representation, properties of a finite-state system can be expressed then by fixed-point equations over the transition

function, and these equations can be solved using iterative methods, similar to those described to compute the transitive closure of a relation. For example, consider the task of determining the set of states reachable from an initial state having binary coding $\vec{q}$ by some sequence of transitions. Define the relation S to indicate the conditions under which for some input $\vec{x}$, there can be a transition from state $\vec{o}$ to state $\vec{n}$. This relation has a characteristic function:

$$\chi_S(\vec{o}, \vec{n}) = \exists \vec{x}[\delta(\vec{x}, \vec{o}, \vec{n})].$$

$\chi_S(\vec{o}, \vec{n}) = \exists \vec{x}[\delta(\vec{x}, \vec{o}, \vec{n})]$. Then set of states reachable from state $\vec{q}$ has characteristic function:

$$\chi_R(\vec{s}) = \chi_S(\vec{q}, \vec{s}).$$

Unfortunately, the system characteristics that guarantee an efficient OBDD representation of the transition relation do not provide useful upper bounds on the results generated by symbolic state machine analysis. For example, we can devise a system having a linear interconnection structure for which the characteristic function of the set of reachable states requires an exponentially sized OBDD [McMillan 1992]. On the other hand, researchers have shown that a number of real-life systems can be analyzed by these methods.

6.4 How TLV uses OBDDs

TLV reads programs written in SMV or in SPL, translates them to OBDDs, and then enters an interactive mode where OBDDs can be manipulated. The interactive mode includes a high level programming language, called TLV-BASIC which also understand SMV expressions. The TLV system is constructed on the top of the CMU SMV system.

TLV uses Boolean Decision Diagrams (BDDs) to represent functions, transitions and state valutations.

The present version of the program has no built-in heuristics for selecting variable ordering. Instead, variable appear in the BDDs in the same order in which they are declared in the program. This means that variables declared in the same module are grouped together, which is generally a good practice, but this alone is not generally sufficient to obtain good performance in the evaluation.

Usually the variable ordering must be adjusted by hand, in an ad hoc way. A good heuristic is to arrange the ordering so that variables which often appear close to each other in formulas are close together in the order of declaration, and global variables should appear first in the program. The number of BDD nodes currently in use is printed on standard error each time the program performs garbage collection, if verbose level is greater than zero. An important number to look at is the number of BDD nodes representing the transition relation. It is very important to minimize this number.

TLV uses a simple automatic variable reordering algorithm (adopted from smv2.4.4) which can be used if you do not want to bother to reorder the variables by hand. It goes over all variables of the program, tries to put each variable in

all possible positions and nds the position which minimizes the total number of bdd nodes. The variable is then moved to that location.

7 Implementation

In this section we will walk through the implementation process of Automatic Composition of Partially Observable Services. Before getting in depth into the actual implementation process, we will make clear the supposed input and the output products of this phase together with the assumptions we made along the development process.

7.1 Input-output

The inputs to the system are Transition Systems describing the behavior of the services belonging to the community and of the target service, plus some further information concerning community properties (see section 7.2). As was made clear before a Service behavior can be completely specified through a Transition System. The informations we need from a TS are:

- the service name;
- the service states;
- the service actions;
- the service initial state;
- the service final states;
- the service transition function;
- the service observation function (in the case the service was partially observable).

In order to make simpler, for the user, to give these TSs as an input, we made possible for the system to accept a XML representation of these TSs. To be more precise, the representation we chose is not a strict XML file. As the information we needed, could be expressed in really simple way we did not specify a proper XML schema, but we just introduced the language tags we needed for our purpose. An example of a TS in its XML format, is the following:

```
<service>

    <name>kts1</name>

    <actions>
        <action>0</action>
        ...
```

```
    </actions>

    <states>
        <state>s0</state>
        ...
        <state>start_st</state>
    </states>

    <initial_state>
        <state>s0</state>
    </initial_state>

    <final_states>
        <state>s0</state>
    </final_states>

    <trans_funct>
        <trans>
            <state>s0</state>
            <action>0</action>
            <state>s1</state>
        </trans>
        ...
        <trans>
            <state>start_st</state>
            <action>4</action>
            <state>s0</state>
        </trans>
    </trans_funct>

    <obs_funct>
        <obs>
            <state>s0</state>
            <obs_value>0</obs_value>
        </obs>
        ...
        <obs>
            <state>start_st</state>
            <obs_value>2</obs_value>
        </obs>
    </obs_funct>

</service>
```

The output of the system is a .smv file, which can be directly executed by the TLV engine. This file contains all the modules needed to check whether the

composition, for the specific target service, exists in the community, or not.

7.2 Assumptions

The assumptions we made in the project are mainly about two aspects: the information contained in the XML files, and the information we need about some properties of the community.

The first assumption we made about the XML TS specification is about the names that can be used to define the states, the actions and the service names. The names that can be used for the states are character strings of the form *X-k-* where *X* is a single character, and -*k*- is an integer number.

The names that can be used for the actions are of the form *k* where *k* is an integer number.

The service names that should be used are *target* for the target service, and: *kts-k-* for the community services, where -*k*- is an integer number.

Another assumption is that the XML files describing the TSs contain a special state called *start_ st* and a special action *start_ op* representing, respectively, the service starting state and the service starting operation. Of course on the observation function section of the XML, there must be the observation value associated to *start_ st*. This is a sort of normalization we have to make among the community services. It's needed because of the particular model that was adopted to check for the composition existence [Pnueli 2006]. So we kept this normalization for the sake of consistency with that model.

Concerning the community properties, we assume that the user explicitly specifies, the number of services in the community, the number of distinct actions the services in the community can perform, and the that the user provides an unique identifier (an integer) to each service in the community.

7.3 The framework

The programming language we used for this project is java, this was because of the previous knowledge we had of this language, but the choice of any other high level programming language would have been absolutely equivalent.

The actual implementation of the Automatic Composition of Partially Observable Services system is made up, mainly, of two java classes, which provide, the methods needed to generate a directly executable .smv file. There's a class, which provides the utilities needed to read and parse the XML files, and another one that provides the utilities needed to generate the .smv files itself. These two classes are actually utilities classes provided with mainly static methods.

The supposed way to use the framework we implemented by defining a java Client class, which calls the methods of the above utilities classes in order to actually generate the .smv output file. In this Client class the user needs to specify the file system paths we the XML resources are located, the file system path where he wants the output to be saved, plus the community properties we mentioned in the section 7.2.

We won't go in depth into the APIs we provided in our framework, because these can be easily found in the javadoc documentation that comes along with framework itself.

7.4 Problems

The main problems we had to face during the implementation process are related to the mismatch that exists between the representations we have of the TSs and the data structures used in the SMV language. The natural interpretation of the information contained in the XML file, representing a TS, is a "list" of properties. For instance, if we consider the transition function, in the XML, it's encoded as a list of triples of the form:

```
<trans>
  <state>starting_state</state>
  <action>action</action>
  <state>arrival_state</state>
</trans>
```

In the SMV files, instead, we needed to encode this function in a slightly different way, because, what we needed was a characteristic function. For example, the above transition would be encoded as:

```
service_trans_funct[starting_state][action][arrival_state] = true;
```

that meaning: the transition that, starting from *starting_ state*, performing the action *action*, leads to *arrival_ state*, belongs to the transition function.

Moreover when we had to define the TRANS section in the Target service MODULE, we had to add, to the information provided by the transition function, the information about the action the service could perform, once arrived in the arrival_state. In order to do this we had to explicitly extract this information from the transition function, and this needed the use of some auxiliary data structures.

We had some problems also because, as we needed to use integer numbers as pointers to elements in data structures, we choose for the use of arrays, but due to the static length of these data structures, we had to pay much attention on the usage of those.

8 A simple example

Here we show a simple example of how our algorithm and implementation works. In this example the target service is represented in Figure 11 and the community is composed by two services shown in Figure 12. It is not hard to see that the target service can be sinthesized.

To start our program just launch the class `Client`, and the graphic interface (Figure 13) suggests to: *(i)* insert the number of service of the community; *(ii)*

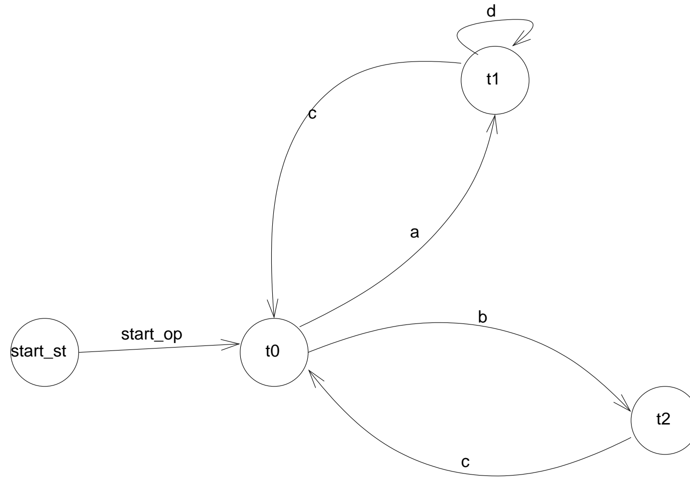
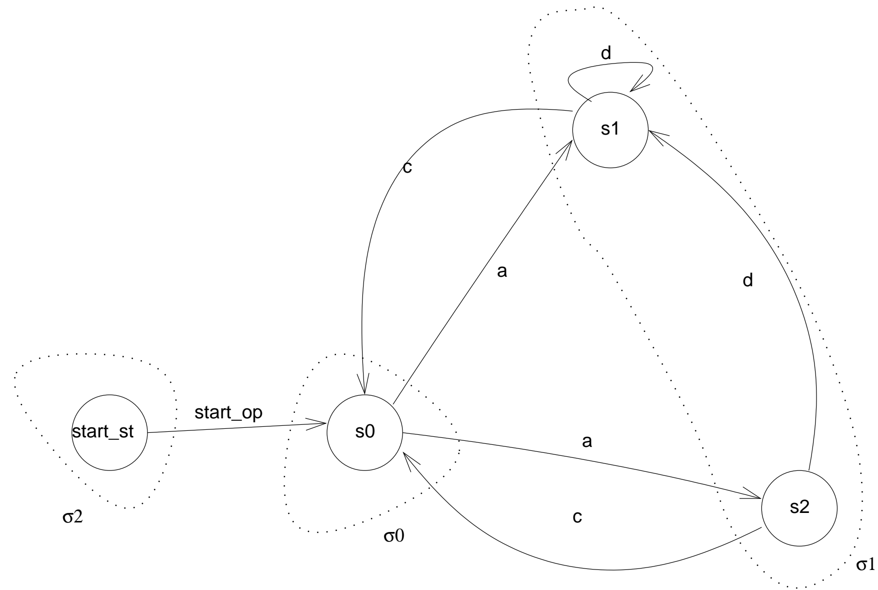


Figure 11: Target service.

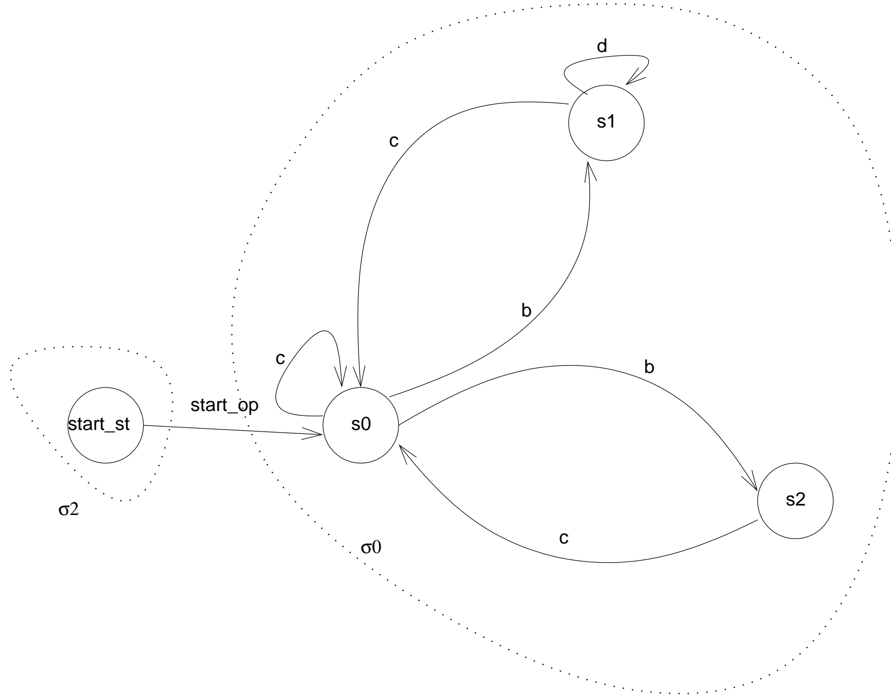
insert the number of actions; *(iii)* browse the HD to search for the .xml files that describe the behaviour of target and community services; *(vi)* click on the "Save .smv file" to select a location for the .smv that is automatically generated.

After that we use the just generated .smv file as input to the TLV procedure for checking the existence of a composition. The output is the following:

```
Mac-Mas:~/Studio/Seminari_Ing_SW/tlv-src-4.18.4 De_Mas$
./tlv comp-inv.pf /Users/De_Mas/Desktop/tesina_ver2.0/composition2.smv
TLV version 4.18.4
System #1:
System #2:
System #3:

Resources used:
user time:  0.01 s
BDD nodes allocated:  1959
max amount of BDD nodes allocated:  1959
Bytes allocated:  327744
Loading Util.tlv $Revision:  4.3 $
Loading MCTL.tlv $Revision:  4.3 $
Loading MCTLS.tlv $Revision:  4.1 $
Loading TESTER.tlv $Revision:  4.2 $
Loading MCsimple.tlv $Revision:  4.2 $
Loading SIMULATE $Revision:  4.2 $
Loading Floyd.tlv $Revision:  4.1 $
Loading Abstract.tlv $Revision:  4.2 $
Loading deductive.tlv $Revision:  4.2 $
```



(a) kts1



(b) kts2

Figure 12: Community services

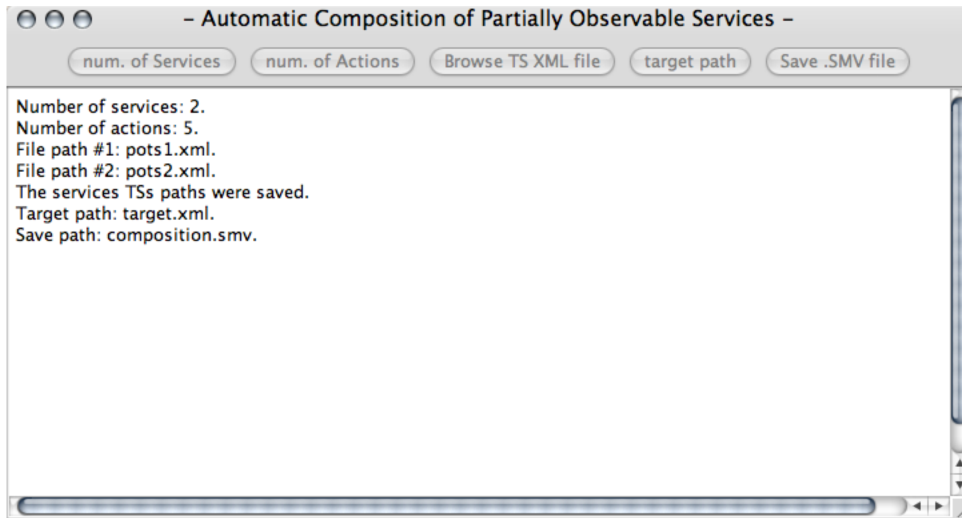


Figure 13: The Java program's graphic interface.

```
Loaded rules file Rules.tlv.

Check Realizability

Specification is realizable

Check that a symbolic strategy is correct

Transition relation is complete

All winning states satisfy invariant

Automaton States

State 1
env.operation = 4, env.target.state = start_st,
env.s1.curr_macrostate[0] = 0, env.s1.curr_macrostate[1] = 0,
env.s1.curr_macrostate[2] = 0, env.s1.curr_macrostate[3] = 1,
env.s1.choice = 1,
env.s2.curr_macrostate[0] = 0, env.s2.curr_macrostate[1] = 0,
env.s2.curr_macrostate[2] = 0, env.s2.curr_macrostate[3] = 1,
env.s2.choice = 1,
sys.index = 0,

State 2
env.operation = 0, env.target.state = t0,
```

```
env.s1.curr_macrostate[0] = 1, env.s1.curr_macrostate[1] = 0,
env.s1.curr_macrostate[2] = 0, env.s1.curr_macrostate[3] = 0,
env.s1.choice = 0,
env.s2.curr_macrostate[0] = 1, env.s2.curr_macrostate[1] = 0,
env.s2.curr_macrostate[2] = 0, env.s2.curr_macrostate[3] = 0,
env.s2.choice = 0,
sys.index = 1,

State 3
env.operation = 1, env.target.state = t0,
env.s1.curr_macrostate[0] = 1, env.s1.curr_macrostate[1] = 0,
env.s1.curr_macrostate[2] = 0, env.s1.curr_macrostate[3] = 0,
env.s1.choice = 0,
env.s2.curr_macrostate[0] = 1, env.s2.curr_macrostate[1] = 0,
env.s2.curr_macrostate[2] = 0, env.s2.curr_macrostate[3] = 0,
env.s2.choice = 0,
sys.index = 2,

State 4
env.operation = 2, env.target.state = t2,
env.s1.curr_macrostate[0] = 1, env.s1.curr_macrostate[1] = 0,
env.s1.curr_macrostate[2] = 0, env.s1.curr_macrostate[3] = 0,
env.s1.choice = 0,
env.s2.curr_macrostate[0] = 0, env.s2.curr_macrostate[1] = 1,
env.s2.curr_macrostate[2] = 1, env.s2.curr_macrostate[3] = 0,
env.s2.choice = 0,
sys.index = 2,

State 5
env.operation = 2, env.target.state = t1,
env.s1.curr_macrostate[0] = 0, env.s1.curr_macrostate[1] = 1,
env.s1.curr_macrostate[2] = 1, env.s1.curr_macrostate[3] = 0,
env.s1.choice = 1,
env.s2.curr_macrostate[0] = 1, env.s2.curr_macrostate[1] = 0,
env.s2.curr_macrostate[2] = 0, env.s2.curr_macrostate[3] = 0,
env.s2.choice = 0,
sys.index = 1,

State 6
env.operation = 3, env.target.state = t1,
env.s1.curr_macrostate[0] = 0, env.s1.curr_macrostate[1] = 1,
env.s1.curr_macrostate[2] = 1, env.s1.curr_macrostate[3] = 0,
env.s1.choice = 1,
env.s2.curr_macrostate[0] = 1, env.s2.curr_macrostate[1] = 0,
env.s2.curr_macrostate[2] = 0, env.s2.curr_macrostate[3] = 0,
env.s2.choice = 0,
```

```
sys.index = 1,

State 7
env.operation = 2, env.target.state = t1,
env.s1.curr_macrostate[0] = 0, env.s1.curr_macrostate[1] = 1,
env.s1.curr_macrostate[2] = 0, env.s1.curr_macrostate[3] = 0,
env.s1.choice = 1,
env.s2.curr_macrostate[0] = 1, env.s2.curr_macrostate[1] = 0,
env.s2.curr_macrostate[2] = 0, env.s2.curr_macrostate[3] = 0,
env.s2.choice = 0,
sys.index = 1,

State 8
env.operation = 3, env.target.state = t1,
env.s1.curr_macrostate[0] = 0, env.s1.curr_macrostate[1] = 1,
env.s1.curr_macrostate[2] = 0, env.s1.curr_macrostate[3] = 0,
env.s1.choice = 1,
env.s2.curr_macrostate[0] = 1, env.s2.curr_macrostate[1] = 0,
env.s2.curr_macrostate[2] = 0, env.s2.curr_macrostate[3] = 0,
env.s2.choice = 0,
sys.index = 1,

Automaton Transitions

From 1 to 2 3
From 2 to 5 6
From 3 to 4
From 4 to 2 3
From 5 to 2 3
From 6 to 7 8
From 7 to 2 3
From 8 to 7 8

Automaton has 8 states, and 15 transitions
```

For an easy reading, in Figure 14 is show the output of the computation, the automaton, i.e. the orchestrator generator.

8.1 Counterexample

In order to test our algorithm, here we show a counterexample. To make the composition of the previous example not realizable, we just modify a transition in the *kts1*. More precisely, instead of $\langle s_2, c, s_0 \rangle \in \delta_{kts1}$, we replace $\langle s_2, c, s_2 \rangle$. It is not hard to see that now, if the target service perform the action c while it is in the state t_1, it goes in a final state, t_0, whereas the service *kts1* remains in a not final state, s_2, and that violates the invariant property `good`. The output

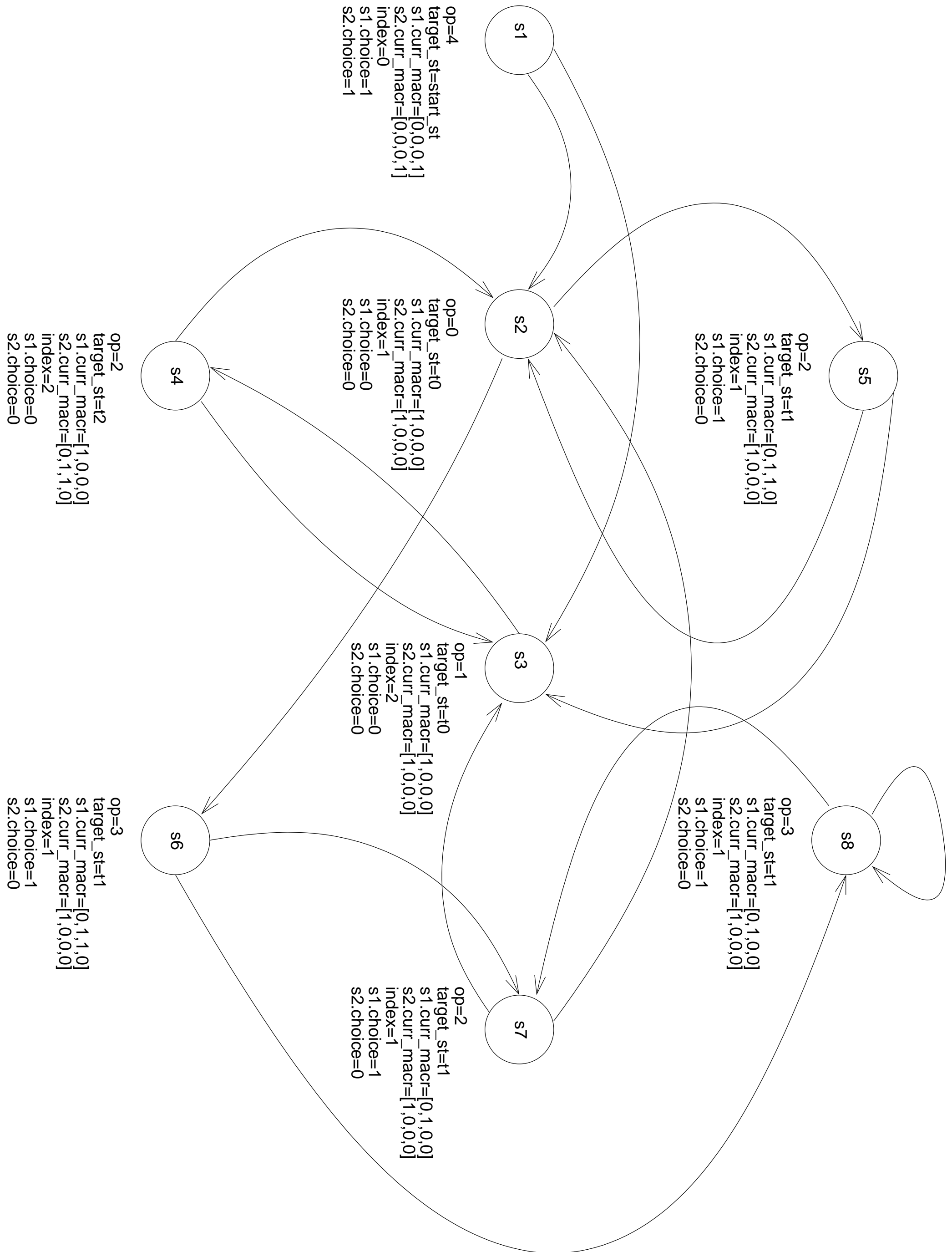


Figure 14: Orchestrator Generator.

of TLV engine, is, in fact:

```
Check Realizability

Specification is unrealizable
```

9 Limitations and future work

Concerning to the SMV encoding, even if the encoding is not exponential, but polinomial, some parts, for instance the part of the algorithm that computes the `imm_oss[][]`, have still a complexity of $\mathcal{O}(n^2)$ where $n = |S|$, even in the best case. It is possible that some refinement could optimize this bound.

Concering to the JAVA implementation, most of the limitations of our system are about the assumptions we make when we take the information as an input. Most of these could be overcome by removing these assumptions we mentioned before. That is to say, for instance, allowing for the automatic insertion of the *start_ st* and the *start_ op*, into the TSs representation.; allowing the user to choose arbitrary names for the states and for the actions; and allowing the user to describe the TSs via a GUI, without the need of actually write the TS XML file. Another nice thing to implement, would be a graphical visualization of the output composition if it exits. We also know that, with a better structured design phase, some of the limitations we underlined above would have been clear before, and so, maybe, also solved before. But, even if, of course, our approach has some obvious limitations, these are just implementation limitations, i.e., they are not restrictive, by any mean, on the theoretical results we showed above. All of the issues we have, can in fact, be solved with ease, and the only reason why, we didn't provide solutions to these issues here, is only for a lack of time.

References

[1] G. De Giacomo and F. Patrizi. Automated Composition of Nondeterministic Stateful Services. Technical report, 2008.

[2] D. Berardi, F. Cheikh, G. De Giacomo and F. Patrizi. Automated Service Composition Via Simulation. Int. J. Found. Comput. Sci. 19(2): 429-451 (2008).

[3] F. Patrizi. Using TLV as a Planning Tool for Non-deterministic or Against-enemies Domains. Technical report.

[4] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. http://portal.acm.org/ft_gateway.cfm?id=136043&type=pdf&coll=GUIDE&dl=GUIDE &CFID=14039920&CFTOKEN=36747860, Sept. 1992.

[5] E. Shahar. The TLV Manual. http://www.wisdom.weizmann.ac.il/~amir/Course02a/tlvmanual.ps, Dec. 2002.

[6] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri and Andrei Tchaltsev. NuSMV User Manual. http://nusmv.fbk.eu/NuSMV/userman/v24/nusmv.pdf, 2005.

[7] F. Patrizi. Simulation-based Techniques for Automated Service Composition. Technical report.

[8] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In VMCAI, pages 364–380, 2006.