Symbolic Refinement of Extended State Machines with Applications to the Automatic Derivation of Sub-Components and Controllers

Khaled El-Fakih and Gregor v. Bochmann, *Fellow IEEE Fellow ACM*

Abstract—Nowadays, extended state machines are prominent requirements specification techniques due to their capabilities of modeling complex systems in a compact way. These machines extend the standard state machines with variables and have transitions guarded by enabling predicates and may include variable update statements. Given a system modeled as an extended state machine, with possibly infinite state space and some non-controllable (parameterized) interactions, a pruning procedure is proposed to symbolically derive a maximal sub-machine of the original system that satisfies certain conditions; namely, some safeness and absence of undesirable deadlocks which could be produced during pruning. In addition, the user may specify, as predicates associated with states, some general goal assertions that should be preserved in the obtained sub-machine. Further, one may also specify some specific requirements such as the elimination of certain undesirable deadlocks at states, or fail states that should never be reached. Application examples are given considering deadlock avoidance and loops including infinite loops over non-controllable interactions showing that the procedure may not terminate. In addition, the procedure is applied for finding a controller of a system to be controlled. The approach generalizes existing work in respect to the considered extended machine model and the possibility of user defined control objectives written as assertions at states.

Index Terms— Requirements/Specifications, component design and refinement, discrete event control systems, extended state machines, submodule construction and automatic derivation of a component behavior.

1 Introduction

State machines are often used for modeling the dynamic behavior of reactive systems that interact with their environment. The purpose of such a model is two-fold: (1) to provide a blueprint for the implementation of the system, and (2) to define the set of possible interaction sequences, also called traces, that could occur when the system is executed. For the comparison of the behavior of an implementation with the behavior specified by the requirement model, one normally distinguishes two concerns: (a) *trace inclusion* – are all traces performed by the implementation included in the allowed set of traces specified by the model? - and (b) absence of deadlocks, and the question whether all specified traces can be realized by the implementation (trace equivalence). We note that trace equivalence is often not mandatory.

In this paper we do not deal with simple state machines, but with extended state machines which are state machine models extended with additional state variables of arbitrary type and interactions that may have parameters. In this context, a *major state* (or simply called a *state*) corresponds to the state of an extended state machine – it is associated with many (possibly infinitely many) concrete states. A *concrete state* consists of a major state and a particular valuation of the machine state variables. A transition not only defines a starting and a destination (major) state and the involved interaction, but also includes:

- A *predicate*, called *guard*, that must be True for the transition to be executable. It depends on the values of the state variables and parameters of the interaction.

- *K. El-Fakih is with the College of Engineering, American University of Sharjah, Sharjah, UAE, P.O. Box: 26666, Tel: (971) 6-51522492, Fax (971) 6 5152979. E-mail: kelfakih@ aus.edu.*
- *G. v. Bochmann is with the School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada, P.O. Box 450, Stn A, K1N 6N5 Tel: (613) 562 5800 ext. 6205; Fax. (613) 562-5664. E-mail: Bochmann@uottawa.ca.*

- An *action* to be performed when the transition is executed. It updates the values of the variables (depending on the previous values and the interaction parameters).

Through the presence of these extensions, the power of expression of these extended state machines is equivalent to the power of programming languages, and therefore many questions concerning their properties are undecidable (in contrast to simple finite state machines for which all interesting questions are decidable). Examples of commercially used specification techniques that employ extended state models include the UML state diagrams, Statecharts, and the Specification Description Language SDL.

In this paper, we make abstraction from input and output and assume that a state machine communicates with another machine (or the environment) through rendezvous communication with interaction interleaving. That is, both machines may perform a transition with a given common interaction when both machines have an enabled transition for that interaction from their current state. However, we consider the possibility that some interactions are *non-controllable*, that is, they are supposed to possibly occur at certain (or all) states of the machine and they must be accepted at these states, as for instance certain interactions coming from the environment. Having some interactions as non-controllable can be due to various reasons, including limited visibility of the environment.

One way for refining an extended state machine is the construction of a refined sub-machine that is obtained from the original machine by reducing the possible transitions. We assume that the additional constraints on the transitions should be such that some user-defined pruning requirements or goals should be satisfied. These requirements restrict the original machine by introducing stronger guards on transitions, and thus may lead to pruning certain major states and transitions as they become inaccessible from the initial state. In other words, for a major state of the original machine, the designer may specify certain goals that identify the corresponding concrete states that are undesirable at the state and thus should become inaccessible in the refined machine. In our case, these goals are introduced by the user as symbolic assertions at major states of the original machine and thus these assertions are not part nor affect the execution of this machine. In fact, a goal assertion for a particular major state represents a property of the variable assignment at this state that should hold at this state, but which will not necessarily hold during the execution of the machine. By construction, a refined submachine satisfies the trace inclusion property in comparison with the original machine. In fact, an assertion at a (major) state initially splits its corresponding concrete states into bad (or undesirable) and OK (or Required , i.e., not bad, but desirable) states depending whether the concrete state satisfies the given pruning requirement, specified as an assertion, at the state or not. However, one of the difficulties that often occurs during the refinement process is that some of the concrete states that are initially considered as OK states become undesirable because they may lead to other bad states, and thus pruning has to continue to make such states inaccessible too. This is not a trivial task as some interactions of the machine are non-controllable.

• **Main contribution:** The main contribution of this paper is the presentation of a symbolic pruning procedure for extended state machines which can be used to build, from a given (possibly infinite) extended state machine $\mathbf{M}$ (over both controllable and non-controllable parametrized interactions), a maximal sub-machine $\mathbf{M}'$ that satisfies certain given pruning requirements, also called goals. These requirements are defined by the goal assertions, depending on the local variables of the machine, associated with certain major states of $\mathbf{M}$. Whenever the machine reaches one of these states, the associated goal should be True. Thus, concrete states at major states that do not satisfy the associated goal can be regarded as undesirable and should become inaccessible from the initial concrete state through the pruning process. Besides general goals, one can define specific types of goals, such as defining a major state as **Fail** state, which means that it should never be reached (by defining the associated assertion to be False), or making the concrete states for which there is an undesired deadlock inaccessible. An application example demonstrating the usefulness of introducing user-defined assertions is provided in Section 3.4, and many other simple examples are provided in Section 4.5 to demonstrate certain situations related to loops; for instance show that the pruning procedure may not terminate due to loops over non-controllable interactions. Although we do not deal with liveness conditions in this paper, yet in Section 4.5.2, we informally discuss how certain ideas of the pruning procedure can be adapted for dealing with the problem of *liveness* conditions, especially loop termination.

We note that the pruning procedure ensures that the pruning process does not introduce any new undesirable deadlocks, although the system behavior may include some so-called final or accepting (major) states in which no further progress is expected.

One application area where such pruning procedures are used regularly is the *derivation of controllers* for discrete-event systems [1] and *submodule construction* [2], also called "*The unknown component problem*" [3]. The most simple system architecture for controller derivation is shown in Fig. 1(a). The behavior of the plant A is given in the form of a state machine, and the behavior of the controller (component B) is to be found such that certain states of the plant - which are undesirable - are never reached. These undesirable states, also called **Fail** states, are defined by the *control objective*. The controller communicates with the plant through rendezvous interactions through the interfaces I_{AB} and I_{ABS}. The interactions over the latter interface also involve the system environment named hereafter S. The interactions of the controller are classified into *controllable* and *non-controllable* interactions. The former can be prevented from happening by the initiative of the controller, while the latter must be accepted by the controller when they occur. The *control objective*, which should be satisfied by the design of the controller, contains two parts:

1. A subset of the states of the plant, called **Fail** states, that should never be reached.
2. Constraints on the order of the interactions visible by the environment. Note: This constraint is called *specification* in the case of submodule constructions (see below).

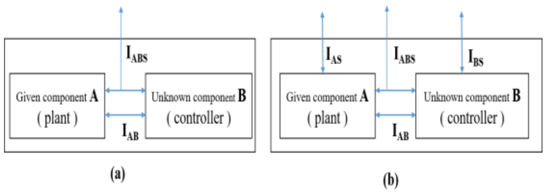
I_{ABS}
Given component A (plant)
Unknown component B (controller)
I_{AB}
(a)
I_{AS}
I_{ABS}
I_{BS}
Given component A (plant)
Unknown component B (controller)
I_{AB}
(b)

Fig.1. Architectures for controller derivation (submodule construction)

A more general architecture for controller derivation is shown in Fig. 1(b). Here there are additional interfaces I_{AS} and I_{BS} for direct interaction with the environment S for the plant and the controller, respectively. The most important difference with Fig. 1(a) is that here the interactions at the interface I_{AS} are not observable by the controller, and therefore also non-controllable. Therefore the controller must "play safe", that is, its behavior must ensure the control objectives for all possible cases of interactions that may occur at the I_{AS} interface. This problem is the same as submodule construction [2-3] where the behavior of (the known) component A is given, as well as the desired behavior S of the system (called *specification* in submodule construction and defined over the interactions at the interfaces I_{AS} , I_{ABS} and I_{BS}, which are visible by the environment), and a suitable behavior of component B (unknown component or desired solution) is to be found such that the combined behavior of components A and B satisfies the requirements defined by the desired system behavior S. According to Fig. 1(b), for component B, interactions I_{BS} , I_{AB}, and I_{ABS} are observable and some of them could be non-controllable.

Solutions to the submodule and supervisory control problems are given for systems modeled as Labeled Transition Systems (LTSs) [1-2], [4], CCS or CSP [5], Mealy FSMs [6], automata [7-10], etc. Application areas include in addition to control and protocol/logic synthesis and derivation of protocol converters [7], [11-12] modular connectors to solve interface mismatches [13], assumptions for software component verification [14], test cases for embedded FSM components [15-16], etc.

- **Second contribution:** We show how the algorithm for controller derivation or submodule construction for finite state machines can be extended to extended state machines with parameterized interactions. In this case, the goal assertions mentioned above may be useful to define the control objectives (e.g. that certain states of the plant should never be reached). We consider the general architecture of Fig. 1(b) with rendezvous communication and interaction interleaving where some interactions may not be controllable or visible by the controller. We present a formula for obtaining the behavior of the maximal solution to the submodule construction or the most general controller over (non-extended, possibly infinite) LTSs. This formula uses operators for constructing machine products, for complementing, and for determinization and elimination of certain transitions and states. We show how this formula can be used for extended state machines (ELTS) by defining these operators for ELTS, although the problems of determinization and pruning for extended machine model are in general non-decidable – which implies that the algorithm may not terminate.

The paper is structured as follows: In Section 2, we present a simple example which will be used throughout the paper. Section 3 defines a formal modeling framework for extended state machines. We use extended labelled transition systems which include an interaction in each transition. Abstract transition guards and state assertions are introduced. Section 4 contains the definition of the pruning procedure and a theorem stating its correctness. It also contains the discussion of several examples including an example showing that the procedure may not terminate. Section 5 discusses controller derivation (or submodule construction) based on extended state machine models. The required operators on extended state machine models, such as complement, product, and hiding of unobservable interactions is first discussed, and then an example is described in detail. Section 6 includes related work and Section 7 contains our conclusions and proposes extensions of our work.

2 Example of a Calculator

As a running application example, we consider a calculator which is modelled as an ELTS as shown in Fig. 2. The calculator has two internal integer variables *acc* (accumulator) and *reg* (register), and three (major) states **a** (initial), **b** and **c**. The machine has the (non-parameterized) interactions *st* (store), *out*, *load* and *add*, and the parameterized interactions *i*(*p*) and *o*(*p*). Hereafter, transition guards are written in […], and update statements in {…}. The transition *st* leads to state **b** from where the *i*(*p*) interaction is possible and which stores the value of the parameter *p* in the internal *reg* variable. Similarly, the *out* transition is followed by the transition *o*(*p*) which is only possible when the parameter value equals the value of the internal *acc* variable. In a sense, this is an "output" transition since the value of the interaction parameter is determined by the machine. The transition *load* loads the value of the *reg* into *acc*, and the transition *add* adds the value of *reg* to *acc*.

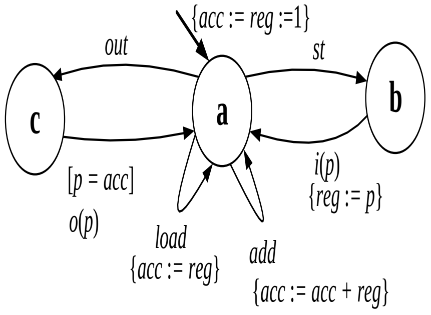

Fig. 2. ELTS model of the calculator

3 Modeling Formalisms

3.1 Labeled Transition Systems (LTSs)

A LTS M = (S, ∑, T, s^0) consists of a set S of states, a set ∑ of interactions (also called alphabet), a set T of transitions and an initial state s^0. A transition t ∈ T is usually written as t = (s – *b* -> s′) where s and s′ are states in S and *b* ∈ ∑ is an interaction. The meaning of a transition t is as follows: if the machine is in state s it may execute the transition t by performing the interaction *b* and changing its state to s′, which is sometimes called the *next state* (reached by transition t). Traditionally, a LTS is a finite state machine, that is, the sets S, ∑ and T are finite sets. However, in this paper we consider non-finite LTS where the sets S, ∑ and T may be infinite sets. The following definitions for finite LTS also apply to non-finite LTS.

Interaction Sequences: Given an alphabet $\sum$, an interaction sequence over $\sum$ is a sequence (b_1, b_2, … b_n) of interactions $b_i \in \sum$. We consider only finite interaction sequences in this paper.

Execution sequences: Given an LTS M = (S, $\sum$, T, s^0), an *execution sequence* from some state s^0 to state s_n of M is an interaction sequence (b_0, b_1, b_2, … b_{n-1}) such that there exist transitions $t_i \in$ T (i = 0, 1, 2, … (*n*-1)) with t_i = (s_i – b_i -> s_{i+1}). We say that an interaction sequence σ = (b_0, b_1, b_2, … b_{n-1}) is an *execution sequence* of a machine M = (S, $\sum$, T, $\mathbf{s^0}$) if there is a state s_n such that σ is an interaction sequence from $\mathbf{s^0}$ to s_n.

We say that a state s_n is *reachable* in M if there is an execution sequence that leads to s_n .

An LTS M = (S, $\sum$, T, s^0) is *deterministic* if there are no two transitions from the same state with the same interaction to two different next states. In a deterministic machine, a given execution sequence from some state $\mathbf{s^0}$ uniquely determines the state reached after the last interaction.

In this paper we consider trace semantics, that is, we are interested in the possible execution sequences of a given machine or system. This means that we consider that the meaning of an LTS model is the set of execution sequences that can be realized by the model. We note that the set of execution sequences of an LTS model is prefix-closed.

3.2. Extended LTS

An extended LTS (ELTS), or (*extended*) *machine*, $\mathbf{M}$ = ($\mathbf{S}$, V, $\sum$, $\mathbf{T}$, $\mathbf{s}^0$, $\mathcal{V}^0$) is a finite LTS which is extended by a finite sequence V of state variables and where interactions may have parameters. For simplicity, we assume in this paper that each interaction may have at most one parameter. Let us assume that there are n state variables, V = {x_1, x_2, … x_n}. We write ($\mathcal{V}_1$, $\mathcal{V}_2$, … $\mathcal{V}_n$) for an assignment of values to these state variables. For each state variable V_i in V, we let D_i denote its domain and then denote as the domain of value assignments D = $D_1 \times D_1 \times … \times D_n$.

More precisely, $\mathbf{S}$ is a finite set of (major) states, and $\mathbf{s}^0 \in \mathbf{S}$ is the initial (major) state of the machine and $\mathcal{V}^0$ is the sequence of initial values of the state variables. The transitions $\mathbf{t} \in \mathbf{T}$ have the form $\mathbf{t}$ = ($\mathbf{s}$ – *b*(*p*) [G] up -> $\mathbf{s}'$) where $\mathbf{s}$ and $\mathbf{s}'$ are the *current* and *next* (major) states of the transition, *b* is an interaction with the parameter *p*, G is the enabling *predicate* of $\mathbf{t}$ (also called *guard*) which depends on the values of the state variables and the parameter, and *up* is a concurrent update statement which defines new values for certain state variables as functions of the current values of all state variables and the parameter.

The meaning of a transition $\mathbf{t}$ = ($\mathbf{s}$ – *b*(*p*) [G] up -> $\mathbf{s}'$) is the following: If the machine is in (the major) state $\mathbf{s}$ then the machine may make a transition to (the major) state $\mathbf{s}'$ by performing the interaction *b* with parameter value $\not{p}$ if the guard G is True for the current values $\mathcal{V}_i$ of the state variables and the value $\not{p}$ of the interaction parameter *p*. Note that the presence of the parameter is optional. If the transition is executed the values of the state variables will be changed according to the concurrent update statement up.

It is clear that an extended LTS can be considered as an abbreviated notation for the definition of a (possibly) non-finite LTS, as defined above. Given an extended LTS $\mathbf{M}$ = ($\mathbf{S}$, V, $\sum$, $\mathbf{T}$, $\mathbf{s}^0$, $\mathcal{V}^0$), an equivalent (possibly) non-finite LTS M = (S, $\sum$, T, $\mathbf{s^0}$) is defined as follows. By equivalent we mean that both models define the same set of execution sequences. We use $\mathcal{V}^0$ = {$\mathcal{V}^0_1$, $\mathcal{V}^0_2$, … $\mathcal{V}^0_n$ } for the initial value assignments of the variables.

- S is the Cartesian product of $\mathbf{S}$ with D, the domains of all the variables. A given concrete state s ∈ S, therefore, can be written as s = ($\mathbf{s}$, $\mathcal{V}_1$, $\mathcal{V}_2$, … $\mathcal{V}_n$). This concrete state corresponds to the state of the extended LTS with major state $\mathbf{s}$ and $\mathcal{V}_1$, $\mathcal{V}_2$, … $\mathcal{V}_n$ as values of its state variables.
- The initial concrete state of M is $\mathbf{s^0}$ = ($\mathbf{s}^0$, $\mathcal{V}^0_1$, $\mathcal{V}^0_2$, … $\mathcal{V}^0_n$).
- $\sum$ is the set of interactions of the non-finite LTS. It is the union of $\sum$, the set of interactions of the extended LTS (for the case that these interactions occur without any parameter) with the Cartesian product of $\sum$ with the domain of the parameter values (for the case that the interactions occur with a parameter).
- We consider a transition $\mathbf{t}$ = ($\mathbf{s}$ – *b*(*p*) [G] up} -> $\mathbf{s}'$) ∈ $\mathbf{T}$ and assume that the guard has the form G = g(x_1, x_2, … x_n, *p*) and the concurrent update statement updates k variables and is of the form up = { (x_{i1} := $expr_{i1}$ (x_1, x_2, … x_n, *p*); … x_{ik} := *expr*$_{ik}$ (x_1, x_2, … x_n, *p*); }. Then the transition $\mathbf{t}$ stands for the set of transitions in M of the form ($\mathbf{s}$, $\mathcal{V}_1$, $\mathcal{V}_2$, … $\mathcal{V}_n$) - *b*($\not{p}$) -> ($\mathbf{s}'$, $\mathcal{V}_1'$, $\mathcal{V}_2'$, … $\mathcal{V}_n'$) for which the following conditions are satisfied:
  1. g($\mathcal{V}_1$, $\mathcal{V}_2$, … $\mathcal{V}_n$, $\not{p}$) and
  2. for all i = 1, …n we have
     - if the variable x_i is updated, that is, if i = ij for some j = 1, 2 … k, then, $\mathcal{V}_i'$= $expr_{ij}$ ($\mathcal{V}_1$, $\mathcal{V}_2$, ... $\mathcal{V}_n$, $\not{p}$)
     - else $\mathcal{V}_i'$= $\mathcal{V}_i$

We note that parameterized transitions may effectively represent input or output operations. For instance, a transition with interaction *a*(*p*) and the update statement *x* := *p*, where *x* is a variable represents an input action where the received parameter value is stored in the variable *x* for later processing. On the other hand, a transition with interaction *a*(*p*) and guard [*x* = *p*] represents an output operation where the value of the *x* is made available to the environment since the guard must hold.

Based on this correspondence between the modeling paradigm of extended LTS with variables and interaction parameters, on the one hand, and the modeling paradigm of non-finite (simple) LTS, on the other hand, we will in the following profit from both views: Extended LTS modeling allows us to write down relatively condensed requirements for system components, and the corresponding model of non-finite (simple) LTS allows us to use the theoretical tools that have been developed in this context which allows us to talk about execution sequences, requirements in terms of sets of execution sequences, interaction hiding, and submodule construction.

3.3 State Assertions

We use assertions in the same way as in documentation and verification of software where an assertion introduced at a certain place in the program means that the assertion should be satisfied by the values of the program variables each time that the execution of the program reaches this place. Similarly, given an extended ELTS $\mathbf{M}$ = ($\mathbf{S}$, V, $\sum$, $\mathbf{T}$, $\mathbf{s}^0$, $\mathcal{V}^0$), we may associate an assertion Assr$_{\mathbf{s}}$ with the major state $\mathbf{s} \in \mathbf{S}$. This means that each time that the major state $\mathbf{s}$ is reached during the execution of the ELTS, the values of the variables V should ensure that the assertion is True.

Definition 1: Given an extended ELTS $\mathbf{M} = (\mathbf{S}, V, \sum, \mathbf{T}, \mathbf{s}^0, \mathcal{V}^0)$ and an assertion Assr$_{\mathbf{s}}$ for the major state $\mathbf{s} \in \mathbf{S}$, we say that the assertion is *consistent* with the ELTS if for all execution sequences of $\mathbf{M}$ that reach the major state $\mathbf{s}$ the values of the variables V are such that Assr$_{\mathbf{s}}$ is True.

For verifying the consistency between assertions and a program, Dijkstra developed the calculus of weakest preconditions for program statements [17]. Given an assignment statement *Assign* which assigns an expression *expr* to a variable *x* in a program, and assuming that after the execution of this statement the assertion Assr should hold, Dijkstra defined the weakest precondition of Assr in respect to *Assign* given by WP_{Assign}(Assr) = Assr(*x/expr*) where Assr(*x/expr*) is the assertion Assr in which each occurrence of *x* is replaced by *expr*. He proved that this is the weakest condition which, when satisfied before the execution of the statement, assures that the assertion Assr is satisfied after the execution of the statement. For a concurrent update statement of the form up = { (x_1 := expr$_1$ ($x_1, x_2, \dots x_n, p$); … x_n := expr$_n$ ($x_1, x_2, \dots x_n, p$);)}, the weakest precondition **WP$_{\text{up}}$(Assr)** in respect to an assertion Assr is obtained from Assr by substituting each occurrence of x_j by expr$_j$ (for all j = 1, … n). In this paper, we write "**Assr o up**" for WP$_{\text{up}}$(Assr). This notation recognizes that the update statement *up* is a mapping from D to D, and it means that first the *up* mapping is applied to the current values of the variables, and then the assertion Assr is evaluated on the obtained variable values, that is, "o" means composition of functions.

We say that an assertion A is *stronger* than an assertion B if A implies B, written A => B, and A is not equivalent to B.

Definition 2: Given an extended machine $\mathbf{M} = (\mathbf{S}, V, \sum, \mathbf{T}, \mathbf{s}^0, \mathcal{V}^0)$, a transition $\mathbf{t} = (\mathbf{s}' - b(p)\ [\ G\ ]\ \text{up} \rightarrow \mathbf{s}) \in \mathbf{T}$ and two assertions Assr$_{\mathbf{s}'}$ and Assr$_{\mathbf{s}}$ associated with the starting and ending states of the transition, we say that Assr$_{\mathbf{s}'}$ is consistent with Assr$_{\mathbf{s}}$ in respect to transition $\mathbf{t}$ if the following condition is satisfied:

for all values of $x_1, x_2, \dots x_n$ and p: Assr$_{\mathbf{s}'} \wedge$ G => Assr$_{\mathbf{s}}$ o up **(1)**

Proposition 1: Given an extended machine $\mathbf{M} = (\mathbf{S}, V, \sum, \mathbf{T}, \mathbf{s}^0, \mathcal{V}^0)$ and a set of assertions {Assr$_{\mathbf{s}}$ | $\mathbf{s} \in \mathbf{S}$} associated with the (major) states of the machine. The assertions are consistent with $\mathbf{M}$ if the following conditions are satisfied:

1. Assr$_{\mathbf{s}0}$ holds for the initial concrete state $(\mathbf{s}^0, \mathcal{V}^0)$.
2. For all transitions $\mathbf{t} = (\mathbf{s}' - b(p)\ [\ G\ ]\ \text{up} \rightarrow \mathbf{s}) \in \mathbf{T}$: Assr$_{\mathbf{s}'}$ is consistent with Assr$_{\mathbf{s}}$ in respect to $\mathbf{t}$.

This proposition can be easily proved by induction: Assr$_{\mathbf{s}0}$ is satisfied for the initial (concrete) state $(\mathbf{s}^0, \mathcal{V}^0)$. When a transition $\mathbf{t} = (\mathbf{s}' - b(p)\ [\ G\ ]\ \text{up} \rightarrow \mathbf{s})$ is executed, we can assume that Assr$_{\mathbf{s}'}$ holds. The transition can only be executed when G holds. Therefore Assr$_{\mathbf{s}'} \wedge$ G holds before the execution, and because of condition (2) Assr$_{\mathbf{s}}$ o up also holds before the execution. Therefore Assr$_{\mathbf{s}}$ holds after the execution of the update statement of the transition. □

We note that consistent assertions are not uniquely determined by the given ELTS – one could, for instance, associate with each state the assertion True. We also note that the assertion associated with an ELTS state that is part of a loop is in fact an invariant for that loop. Some methods for deriving loop invariants are described in [18].

3.4 The Example of the Calculator

We consider the calculator of Fig. 2 as an example. It has two variables, *acc* and *reg*. After the transition (**b** - *i*(*p*) { *reg* := *p*} -> **a**) the variable *reg* assumes the value which was provided as interaction parameter. After the interaction *load*, the values of the two variables are equal (to the parameter) and subsequent interactions *add* lead to states where the value of *acc* is a multiple of the value of *reg*, that is, the assertions [*acc* = *reg*], [*acc* = 2 * *reg*], [*acc* = 3 * *reg*], etc. may hold.

In order to describe this situation in a more systematic manner, one may build a new model by adding an integer variable *n* which represents the number of successive executions of the *add* interactions since the last *load* interaction, plus one. Then we may consider the assertion [*acc* = *n* * *reg*] which holds when the load interaction has been executed. Since this assertion does not hold initially, one may split the (major) state **a** into two (major) states **a$_1$** and **a$_2$**, as shown in Fig. 3. Now it is easy to show that the set of associated assertions is consistent. For instance, the weakest precondition for the assertion of state **a$_2$**, namely [($acc = n * reg) \wedge (n > 0)$], in respect to the *load* transition is calculated by replacing *acc* by *reg* and *n* by the value 1. This gives [($reg = 1 * reg) \wedge (1 > 0)$] which is True. Similarly, we obtain the weakest precondition in respect to the *add* transition: [($acc + reg = (n + 1) * reg) \wedge ((n + 1) > 0)$] which is implied by [($acc = n * reg) \wedge (n > 0)$]. Therefore, this assertion is an invariant for both of the looping transitions *load* and *add*.

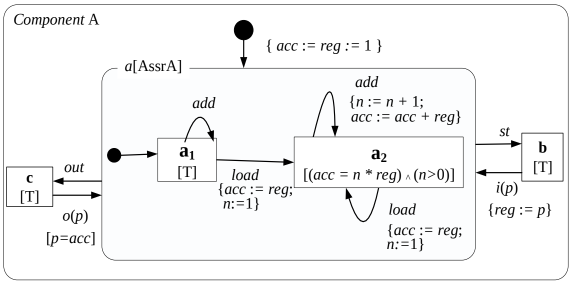

Fig. 3. A new model of the calculator written as a hierarchical UML state machine, including assertions.

3.5. Sub-machines

As mentioned in the Introduction, our pruning algorithm, applied to an ELTS $\mathbf{M}$, results in a sub-machine $\mathbf{M}'$ of $\mathbf{M}$. In the context of LTS, a sub-machine is defined as follows:

Definition 3(a) – Sub-machine of an LTS: An LTS M′ = (S′, ∑, T′, s^0) is a sub-machine of an LTS M = (S, ∑, T, s^0) if M′ is obtained from M by pruning (eliminating) certain transitions, that is, if T′ is a subset of T.

One usually also eliminates all those states of M (together with their outgoing transitions) which become inaccessible from the initial state, that is, S′ is a subset of S. For ELTSs, we use the following definition:

Definition 3(b) – Sub-machine of an ELTS: Given $\mathbf{M}' = (\mathbf{S}, V, \sum, \mathbf{T}', \mathbf{s}^0, \mathcal{V}^0)$ and $\mathbf{M} = (\mathbf{S}, V, \sum, \mathbf{T}, \mathbf{s}^0, \mathcal{V}^0)$, we say that $\mathbf{M}'$ is a sub-machine of $\mathbf{M}$ if the transitions in $\mathbf{T}'$ are the same transitions as in $\mathbf{T}$ except that their guards may be stronger or equivalent.

It is clear that in this case the equivalent (possibly) non-finite LTS M′ of $\mathbf{M}'$ is a sub-machine (LTS) of the LTS equivalent to $\mathbf{M}$. We note that the transitions of $\mathbf{M}'$ with a guard that is False may be eliminated without affecting the behavior of $\mathbf{M}'$, and as in the case of LTS, the major states of $\mathbf{M}'$ that are inaccessible from the initial state may also be eliminated together with their outgoing transitions.

In fact, in this paper, we are interested in sub-machines that satisfy certain properties such as assertions that should hold at major states of the sub-machine.

Definition 3(c) – Maximal sub-machine of an ELTS: $\mathbf{M}'$ is the maximal sub-machine of $\mathbf{M}$ that satisfies certain properties C if for each sub-machine $\mathbf{M}''$ of $\mathbf{M}$ that satisfies C, $\mathbf{M}''$ is a sub-machine of $\mathbf{M}'$.

4 Deriving a Sub-Machine that Satisfies Some Given Pruning Requirements

In this section we discuss how a given ELTS $\mathbf{M}$ can be modified such that certain properties, called *pruning requirements*, become satisfied. We assume that the modified ELTS $\mathbf{M}'$ should be a sub-machine of $\mathbf{M}$. In Section 4.1, we explain how we characterize the pruning requirements, and in Section 4.2 we present a pruning procedure which constructs $\mathbf{M}'$. Since the problem of finding $\mathbf{M}'$ is undecidable in general, the pruning procedure may not terminate for certain given ELTS. A theorem stating the properties of this procedure is proved in Section 4.4. Examples are given in Sections 4.3 and 4.5. In Section 4.5.2, we informally discuss how certain ideas of the pruning procedure can be adapted for dealing with the problem of *liveness* conditions, especially loop termination.

4.1. Pruning Requirements

We assume that an ELTS $\mathbf{M} = (\mathbf{S}, V, \Sigma, \mathbf{T}, \mathbf{s}^0, \mathcal{V}^0)$ is given and that the designer wants to obtain a submachine $\mathbf{M}'$ of $\mathbf{M}$ for which certain pruning requirements are satisfied. These requirements are given in the form of assertions that should hold at certain major states. For a major state $\mathbf{s} \in \mathbf{S}$, the following types of (Goal) assertions can be considered.

Types of Goal Assertions (or Pruning Requirements):

1. General Goal: The assertion $\text{Goal}_\mathbf{s}$ should be satisfied each time the major state $\mathbf{s}$ is reached.
2. A major **Fail** state : $\text{Goal}_\mathbf{s}$ = False. This means that the major state $\mathbf{s}$ should never be reached. This concept is important for the derivation of controllers for discrete event systems (see Section 5).
3. No undesired deadlock at $\mathbf{s}$: $\text{Goal}_\mathbf{s}$ = $\textit{enabled}_\mathbf{s}$ where $\textit{enabled}_\mathbf{s}$ denotes the logical **or** of the guards $G_\mathbf{t}$ of all outgoing transitions $\mathbf{t}$ from major state $\mathbf{s}$. This means that $\neg\textit{enabled}_\mathbf{s}$ should never be True when state $\mathbf{s}$ is reached – therefore at least one transition will be possible from this state. In fact, this goal should apply in all major states, except those that are designated as *final* or *accepting.* If the value of the guard $G_\mathbf{t}$ of a transition $\mathbf{t}$ depends on the value of an interaction parameter *p*, then the condition for this transition would be "there exists a value of parameter *p* such that $G_\mathbf{t}$ is True."

4.2. The Pruning Procedure

The pruning procedure prunes certain transitions of the ELTS such that the pruning requirements become consistent assertions in the resulting sub-machine. We say that a transition $\mathbf{t}$ of the given ELTS $\mathbf{M}$ is pruned if a stronger guard is introduced for the transition, or if the transition is completely eliminated. The latter happens if the guard of a transition becomes False, the transition is never executable and thus can be eliminated from the ELTS. Major states that become inaccessible can also be eliminated. Therefore, the resulting ELTS $\mathbf{M}'$ is a sub-machine of $\mathbf{M}$. We note that $\mathbf{M}'$ is the maximal sub-machine of $\mathbf{M}$ for which the pruning requirements are consistent.

When this pruning procedure is used for the derivation of a controller, it is important to note that certain transitions of the controller behavior cannot be pruned because their interactions cannot be controlled, such as input interactions from the environment (see Section 5). Therefore we assume here that we know which interactions of $\mathbf{M}$ are controllable. Any transition of $\mathbf{M}$ that has an interaction that is non-controllable cannot be pruned and must remain as is. We note that the pruning procedure avoids all undesired deadlocks that may be introduced when a transition is pruned.

Initialization Step: The initial inputs for the pruning procedure are the following:

1. An initial ELTS $\mathbf{M} = (\mathbf{S}, V, \Sigma, \mathbf{T}, \mathbf{s}^0, \mathcal{V}^0)$.
2. For each interaction of $\mathbf{M}$ whether it is controllable or not.
3. The pruning requirements in the form of goal assertions $\text{Goal}_\mathbf{s}$ for certain major states $\mathbf{s}$, as illustrated above.

The pruning procedure uses a variable called ***Problematic*** which represents the set of major states for which the required assertion has not yet been checked. Initially, this set contains each state $\mathbf{s}$ with an associated $\text{Goal}_\mathbf{s}$ assertion from point (3) above.

An initial set of assertions $\{ \text{Assr}_\mathbf{s} \mid \mathbf{s} \in \mathbf{S} \}$ associated with the (major) states of $\mathbf{M}$ is defined as follows. $\text{Assr}_\mathbf{s}$ is equal to $\text{Goal}_\mathbf{s}$ if the state $\mathbf{s}$ is associated with a Goal, or True otherwise.

Pruning Procedure:
While the set ***Problematic*** is not empty and the associated assertion of the initial state, $\text{Assr}_{\mathbf{s}0}$, is not False, take a state $\mathbf{s}$ from ***Problematic*** and call the procedure **prune** ($\mathbf{s}$) where this procedure is defined as follows:

```
Procedure prune (majorState $\mathbf{s}$)
    For each incoming transition $\mathbf{t} = (\mathbf{s}' - b(p)\ [\ G\ ]\ up \to \mathbf{s})$
do the following:
    Step 1: If the interaction $b$ of the transition is controllable:
            If G does not imply ($\text{Assr}_\mathbf{s}$ o up), replace $\mathbf{t}$ by the
            following transition (which has a stronger guard):
                $\mathbf{t}_{\mathbf{OK}} = (\mathbf{s}' - b(p)\ [\ G \wedge (\text{Assr}_\mathbf{s}\ o\ up)\ ]\ up \to \mathbf{s})$
                Define $\text{Assr}_{\mathbf{s}'}^{(new)} := \text{Assr}_{\mathbf{s}'} \wedge \textit{enabled}_{\mathbf{s}'}$ or if $\mathbf{s}$
                is a final or accepting state, $\text{Assr}_{\mathbf{s}'}^{(new)} :=$
                $\text{Assr}_{\mathbf{s}'}$ .
                    Note A: The introduction of the
                    stronger guard of $\mathbf{t_{OK}}$ may lead to a potential undesirable deadlock in state $\mathbf{s}'$.
                    Such deadlock is captured by the update $Assr_{\mathbf{s}'}{}^{(new)} := Assr_{\mathbf{s}'} \wedge enabled_{\mathbf{s}'}$
                    and later in Step (3) $\mathbf{s}'$ will be put into the Problematic set if such an update
                    produces a stronger assertion at $\mathbf{s}'$.
    Step 2: Otherwise (if the interaction $b$ is non-controllable):
        Define $Assr_{\mathbf{s}'}{}^{(new)} := Assr_{\mathbf{s}'} \wedge$ for all $p$:$( \neg G \vee Assr_{\mathbf{s}}$ o up)
                    Note B: This means that the reachable concrete states of $\mathbf{s}'$ should be such
                    that either the transition $\mathbf{t}$ is not executable or, after the transition $\mathbf{t}$ is
                    performed, the associated assertion $Assr_{\mathbf{s}}$ should be satisfied in the next state $\mathbf{s}$.
    Step 3:
        If $Assr_{\mathbf{s}'}{}^{(new)}$ is stronger than $Assr_{\mathbf{s}'}$, then
            Update the associated assertion $Assr_{\mathbf{s}'} := Assr_{\mathbf{s}'}{}^{(new)}$, and
            Put $\mathbf{s}'$ into the Problematic set.
                    Note C: We may want to prune more than necessary, that is, we use some
                    $Assr_{\mathbf{s}'}{}^{(stronger)}$ instead of $Assr_{\mathbf{s}'}{}^{(new)}$ where $Assr_{\mathbf{s}'}{}^{(stronger)}$ implies $Assr_{\mathbf{s}'}{}^{(new)}$.
                    We call this non-required pruning. This type of pruning is demonstrated
                    in the Factorial Example given in Section 4.5.
                    Note D: $\mathbf{s}'$ may be further constrained (that is, another "part" (subset of
                    states) of $\mathbf{s}'$ may be pruned) when another outgoing transition from $\mathbf{s}'$ leads
                    to another problematic state on which the prune procedure is later applied.
        Else, then there is effectively no constraining – nothing further to do.
        Note E: We note that the decision whether $Assr_{\mathbf{s}'}{}^{(new)}$ is stronger than $Assr_{\mathbf{s}'}$ is in
        general undecidable. This decision is important when there are loops in the transition graph
        in order to determine whether the pruning procedure will loop forever (see examples in Section 4.5.1).
  EndFor
End prune
```

Note F: The states $\mathbf{s}$ for which the associated assertion $Assr_{\mathbf{s}}$ is False are not accessible from the initial state in the ELTS obtained by the pruning procedure. In certain cases (see for instance Example A-1 in Section 4.5.1) only the initial state may remain accessible – clearly, that is not an interesting solution. If the associated assertion of the initial state, $Assr_{\mathbf{s}0}$, is False and no **non-required pruning** was used, then there is no solution.

4.3. A Simple Example: Avoiding Undesirable Deadlocks

The given ELTS is shown in Fig. 4(a). There are two major states with potential deadlocks: states **D** and **E**. To avoid these deadlocks, the designer may constrain, under point (3) of the Initialization Step, states **D** and **E** by defining the corresponding goal assertions $[x = 15 \vee w]$ and $[w]$, and obtain **D** and **E**, respectively, as shown in Fig. 4(b). Thus, the set ***Problematic*** initially includes states **D** and **E**. Let us assume that interactions *c*, *d*, and *g* are non-controllable, while *e*, *h*, and *b* are controllable.

Let us assume that we start the pruning procedure by calling the procedure **prune** (**D**) as shown in Fig. 4(b). Since the incoming transition *d* is non-controllable, in Step 2, state **C** is considered problematic with the required assertion $Assr_C{}^{(new)}$ equals $[x =14 \vee w]$, and the procedure **prune** will later be called for **C** while the associated assertion is $Assr_C{}^{(new)} := [x =14 \vee w]$.

We now continue the pruning procedure by applying **prune** to the second initially problematic state **E** which now has the associated assertion $[w]$ as described above and shown in Fig. 4(b). The incoming transition with the (controllable) interaction *h* is a self-loop without any update action. Therefore, the assertion $[w]$ (for starting and ending state) is consistent with this looping transition. The *g*-transition is another incoming transition to state **E**. Since we assume that this transition is non-controllable, the pruning process would have to continue. The constraint of the *g*-transition would be transferred to the state **C** which would be constrained by the additional stronger assertion $[w]$; this constrained state is called **C** in Fig. 4(c). When this state is processed, as the incoming *c*-transition is non-controllable, then state **B** must also be constrained to $[w]$, and corresponding state **B** becomes problematic with the additional constraint $[w]$ obtained in Steps 2 and 3, respectively.

Then, the (controllable) incoming *b*-transition to **B**, with the associated assertion $[w]$, is processed and according to Step 1, the required guard for this transition becomes a contradiction as it includes $[w]$ and $[\neg w]$. Therefore, this transition must be completely eliminated. Note, if interaction *b* was non-controllable, then the starting state of this transition must also be eliminated. However, since this state is the initial state, this means that there would be no solution to avoiding possible deadlocks.

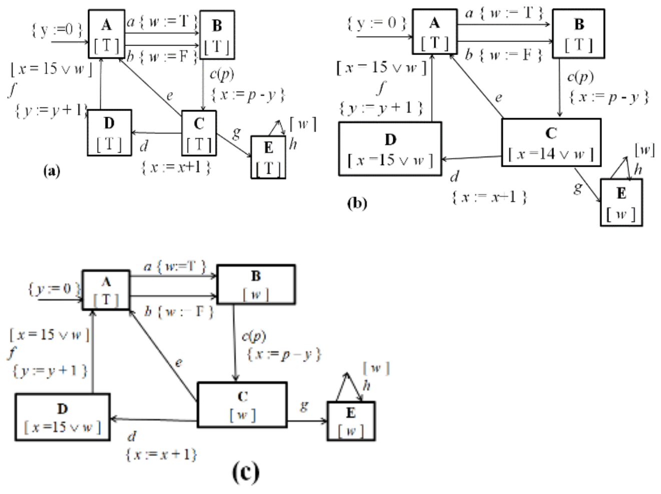

Fig. 4. Simple example, (a) The given ELTS, (b) After dealing with the problematic state **D**, (c) After dealing with the problematic states **E** and **C**.

4.4. Proof of Correctness

The following theorem states the properties of the pruning procedure of Section 4.2, and it is proved using two lemmas and a proposition.

Theorem 1: Given an ELTS $\mathbf{M} = (\mathbf{S}, V, \Sigma, \mathbf{T}, \mathbf{s}^0, \mathcal{V}^0)$ for which certain transitions may be uncontrollable, and with goal assertions for a subset of the major states. Let $\mathbf{M}' = (\mathbf{S}', V, \Sigma, \mathbf{T}', \mathbf{s}^0, \mathcal{V}^0)$ be the ELTS obtained by applying the pruning procedure of Section 4.2 to $\mathbf{M}$, and let $\{ \text{Assr}_{\mathbf{s}}^{\text{final}} \mid \mathbf{s} \in \mathbf{S}' \}$ be the set of associated assertions obtained when the procedure ends. If the procedure terminates and the associated assertion of the initial state, $\text{Assr}_{\mathbf{s}0}$, is not False, we have the following properties:

1. $\mathbf{M}'$ is a sub-machine of $\mathbf{M}$.
2. $\{ \text{Assr}_{\mathbf{s}}^{\text{final}} \mid \mathbf{s} \in \mathbf{S}' \}$ is a consistent set of assertions for $\mathbf{M}'$.
3. For all major states $\mathbf{s}$ for which a goal assertion $\text{Goal}_{\mathbf{s}}$ was given in the Initialization Step, $\text{Assr}_{\mathbf{s}}^{\text{final}}$ implies $\text{Goal}_{\mathbf{s}}$.
4. The pruning algorithm does not introduce any new deadlocks at major states.
5. $\mathbf{M}'$ is the maximal sub-machine of $\mathbf{M}$ that satisfies properties (3) and (4) above, i.e., for every sub-machine $\mathbf{M}''$ of $\mathbf{M}$ that satisfies properties (3) and (4), $\mathbf{M}''$ is a sub-machine of $\mathbf{M}'$. □

Lemma 1: During the execution of the pruning procedure, each time the **prune ($\mathbf{s}$)** procedure terminates after processing a major state $\mathbf{s}$, we have for all transitions $\mathbf{t} = (\mathbf{s}' - b(p)\ [\ G\ ]\ up \rightarrow \mathbf{s})$ that are incoming to $\mathbf{s}$ that the associated assertions $\text{Assr}_{\mathbf{s}'}$ and $\text{Assr}_{\mathbf{s}}$ are consistent with transition $\mathbf{t}$.

Proof: If the interaction b is controllable, the additional guard ($\text{Assr}_{\mathbf{s}}$ o up) introduced in Step 1 ensures that $\text{Assr}_{\mathbf{s}}$ is satisfied after the execution of the transition. Therefore (according to Definition (2) in Section 3.3.1) $\text{Assr}_{\mathbf{s}'}$ and $\text{Assr}_{\mathbf{s}}$ are consistent with transition $\mathbf{t}$, even if $\text{Assr}_{\mathbf{s}'}$ is strengthened in Step 3 according to the assertion defined in Step 1. – If Step 2 is executed, i.e., b is non-controllable, $\text{Assr}_{\mathbf{s}'}$ (updated in Step 3 according to the assertion defined in Step 2) implies ($\neg$ G $\vee$ $\text{Assr}_{\mathbf{s}}$ o up), which means that – if $\mathbf{t}$ is executable – then G is True; and thus the assertion $\text{Assr}_{\mathbf{s}}$ should be satisfied after the execution of the transition in the next state $\mathbf{s}$.

Lemma 2: During the execution of the pruning procedure, let us assume that the associated assertions $\text{Assr}_{\mathbf{s}'}$ and $\text{Assr}_{\mathbf{s}}$ are consistent with the transition transitions $\mathbf{t} = (\mathbf{s}' - b(p)\ [\ G\ ]\ up \rightarrow \mathbf{s})$. If the **prune ($\mathbf{s}''$)** procedure is executed for another state $\mathbf{s}''$ that also has an incoming transition $\mathbf{t}_1 = (\mathbf{s}' - b'(p)\ [\ G'\ ]\ up' \rightarrow \mathbf{s}'')$ from $\mathbf{s}'$, then the update of the associated assertion of $\mathbf{s}'$ (when handling $\mathbf{t}_1$ in Step 3 according to the assertions defined in Steps 1 and 2 of the procedure) will never weaken the assertion. Therefore the assertions $\text{Assr}_{\mathbf{s}'}$ and $\text{Assr}_{\mathbf{s}'}$ remain consistent with the transition $\mathbf{t}$.

Proof: This is evident from the form of Step3 (and the related assertions defined in Steps 1 and 2), and Definition 2. □

Proposition 2: During the execution of the pruning procedure, between the executions of the procedure **prune** for different major states in *Problematic*, we have for all transitions $\mathbf{t} = (\mathbf{s}' - b(p)\ [\ G\ ]\ up \rightarrow \mathbf{s})$ in $\mathbf{T}'$ the following: If $\mathbf{s}$ is not in *Problematic*, then the assertions $\text{Assr}_{\mathbf{s}}$ and $\text{Assr}_{\mathbf{s}'}$ are consistent with transition $\mathbf{t}$.

Proof: By induction over the sequence of **prune** executions (using Lemma 1 and Lemma 2). □

Proof of Theorem 1:

1. This follows directly from the definition of the pruning algorithm.
2. This follows directly from Proposition 2, since *Problematic* is empty when the procedure stops.
3. This follows directly from Lemma 2.
4. A new deadlock may be introduced in major state $\mathbf{s}'$ by the **prune** procedure in Step 1 due to the stronger guard of the transition. However, the definition of the *enabled*$_{\mathbf{s}'}$ assertion ensures that no deadlock is possible in state $\mathbf{s}'$ if the new $\text{Assr}_{\mathbf{s}'}$ (updated in Step 3) is satisfied.
5. This follows from the fact that the additional guard in Step 1 and the assertions *enabled*$_{\mathbf{s}'}$ and (for all p: ($\neg$ G $\vee$ $\text{Assr}_{\mathbf{s}}$ o up)) , defined in Steps 1 and 2, respectively, which are additional constraints to be satisfied by the new version of $\text{Assr}_{\mathbf{s}'}$ are the weakest constraints that are sufficient to ensure consistency with the processed transition. □

4.5. Examples

4.5.1. Looping Examples

We consider an ELTS as shown in Fig 5 where the a-transition is controllable, but the other transitions are non-controllable. The d-transition has no guard nor update statement, while the b and c transitions have various guards and assignments, depending on the cases discussed below. And State $\mathbf{s}_4$ has various associated goals. The initial associated assertions are True for all states.

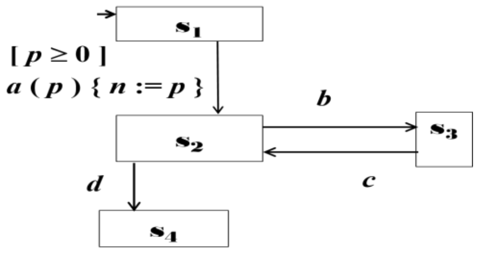

Fig. 5. ELTS of the looping examples

(A): Examples where n is an Integer variable

Example A-1. Goal of $\mathbf{s}_4$ = [$n = 0$]; c has assignment**{ $n := n - 1$ }** (**Note:** unless stated otherwise, the transitions b and c have no guard and no assignment). In the following, we indicate the different steps of the pruning process, where in each step the **prune** procedure is applied to one of the states and one of the incoming transitions is handled.

(Step 1) The **prune** procedure is called for state $\mathbf{s}_4$ which has the assertion [$n = 0$]; handling the incoming transition $d : \mathbf{s}_2$ obtains the associated assertion $\text{Assr}_{\mathbf{s}2}$ = [$n = 0$]

(Step 2) **Prune** is called for state $\mathbf{s}_2$; incoming transition is $c : \mathbf{s}_3$ obtains the associated assertion $\text{Assr}_{\mathbf{s}3}$ [$n = 1$]

(Step 3) **Prune** is called for state $\mathbf{s}_3$; incoming transition is *b* : the new assertion for $\mathbf{s}_2$ becomes $\text{Assr}_{\mathbf{s}2}^{(new)} = [\ n = 1 \ \wedge n = 0\ ]$ which is False, which means that this state should become inaccessible.

(Step 4) **Prune** is called for state $\mathbf{s}_2$; incoming transition is *a* (which is controllable): The transition *a* obtains the guard $[(p \geq 0) \wedge \text{False}]$, which means that this transition is eliminated. The resulting ELTS has an initial state without any transitions.

Example A-2. Goal of $\mathbf{s}_4 = [\ n \geq\ 0\ ]$; *c* has assignment **$\{ n := n - 1 \}$**

(Step 1) **prune** is called for state $\mathbf{s}_4$ with the associated assertion $[\ n \geq\ 0\ ]$; incoming transition is *d* : a new assertion is assigned to state $\mathbf{s}_2$ - $\text{Assr}_{\mathbf{s}2} := [\ n \geq 0\ ]$

(Step 2) $\mathbf{s}_2$; *c* : $\text{Assr}_{\mathbf{s}3} := [\ n \geq 1\ ]$

(Step 3) $\mathbf{s}_3$; *b* : $\text{Assr}_{\mathbf{s}2} := [\ n \geq 1\ ]$ – this is a stronger assertion than at (Step 1). Therefore the pruning process continues.

(Step 4) $\mathbf{s}_2$; *c* : $\text{Assr}_{\mathbf{s}3} := [\ n \geq 2]$

(Step 5) $\mathbf{s}_3$; *b* : $\text{Assr}_{\mathbf{s}2} := [\ n \geq 2\ ]$ – this is stronger again. **The pruning process will never terminate.**

Example A-3. Goal of $\mathbf{s}_4 = [\ n \geq 0\ ]$; *c* has assignment **$\{ n := n - 1 \}$; *b*** has guard **$[n > 0]$**

(Step 1) **prune** is called for state $\mathbf{s}_4$ with the associated assertion $[\ n \geq 0\ ]$; *d* : $\text{Assr}_{\mathbf{s}2} := [\ n \geq 0\ ]$

(Step 2) $\mathbf{s}_2$; *c* : $\text{Assr}_{\mathbf{s}3} := [\ n \geq 1\ ]$

(Step 3) $\mathbf{s}_3$; *b* : $\text{Assr}_{\mathbf{s}2}. := [\ n \leq 0 \ \vee\ n \geq 1]$ – this is not stronger than at (Step 1). Pruning stops

Example A-4. Goal of $\mathbf{s}_4 = [\ n\ \geq 0\ ]$; *c* has assignment **$\{ n := n - 1 \}$; *b*** has guard **$[\ n < 2\ ]$**

(Step 1) **prune** is called for state $\mathbf{s}_4$ with the assertion $[\ n \geq 0\ ]$; *d* : $\text{Assr}_{\mathbf{s}2} := [\ n \geq 0\ ]$

(Step 2) $\mathbf{s}_2$; *c* : $\text{Assr}_{\mathbf{s}3} := [\ n \geq 1\ ]$

(Step 3) $\mathbf{s}_3$; *b* : $\text{Assr}_{\mathbf{s}2} := [\ (n \geq 2) \vee (n \geq 1)\ ] = [\ n \geq 1\ ]$ – pruning process continues.

(Step 4) $\mathbf{s}_2$; *c* : $\text{Assr}_{\mathbf{s}3} := [\ n \geq 2\ ]$

(Step 5) $\mathbf{s}_3$; *b* : $\text{Assr}_{\mathbf{s}2} := [\ n \geq 2\ ]$ – pruning process continues.

(Step 6) $\mathbf{s}_2$; *c* : $\text{Assr}_{\mathbf{s}3} := [\ n \geq 3\ ]$

(Step 7) $\mathbf{s}_3$; *b* : $\text{Assr}_{\mathbf{s}2} := [\ (n \geq 2) \vee (n \geq 3)\ ] = [\ n \geq 2\ ]$ – This is the same constraint as at (Step 5) and this repeated pruning process stops. When $[\ n \geq 2\ ]$ holds, the *b*-transition is not enabled and there is no loop.

(Step 8) The pruning process continues for the *a*-transition (which is controllable). This leads to the introduction to a stronger guard $[\ p \geq 2\ ]$.

(B): Examples where *n* is a Real variable (always positive or zero)

Example B-1. Goal of $\mathbf{s}_4 = [\ n < 1\ ]$; *c* has $\{ n := n / 2 \}$

(Step 1) **prune** is called for state $\mathbf{s}_4$ with the assertion $[\ n < 1\ ]$; *d* : $\text{Assr}_{\mathbf{s}2} := [\ n < 1\ ]$

(Step 2) $\mathbf{s}_2$; *c* : $\text{Assr}_{\mathbf{s}3} := [\ n < 2\ ]$

(Step 3) $\mathbf{s}_3$; *b* : $\text{Assr}_{\mathbf{s}2} := [\ n < 2]$ – this condition is not stronger than at (Step 1). Therefore the pruning process stops here, and (Step 2) is the solution. That is, the *a*-transition will get a guard ensuring $p < 1$.

Example B-2. Goal of $\mathbf{s}_4 = [\ n < 1]$; *c* has $\{ n := n*2 \}$

(Step 1) **prune** is called for state $\mathbf{s}_4$ with associated assertion $[n < 1]$; *d* : $\text{Assr}_{\mathbf{s}2} := [\ n < 1]$

(Step 2) $\mathbf{s}_2$; *c* : $\text{Assr}_{\mathbf{s}3} := [\ n < 0.5\ ]$

(Step 3) $\mathbf{s}_3$; *b* : $\text{Assr}_{\mathbf{s}2} := [\ n < 0.5\ ]$ – this is a stronger condition than at (Step 1). Therefore the pruning process continues.

(Step 4) Next time the condition will be $[\ n < 0.25]$, etc. This means that the pruning process will never stop. However, we know that when the parameter *p* has the value zero (and therefore $n = 0.0$), there is no problem with the goal (although the pruning process does not show this).

4.5.2. Examples with Considerations of Termination

(C) Calculator Example: We consider the application of the pruning procedure to the ELTS shown in Fig. 2. There are no fail states nor deadlocks in this ELTS. However, we assume that the goal assertion $\text{Goal}_{\mathbf{c}} = [\ acc = reg\ ]$ is introduced in State **c**. Assuming that all interactions are controllable, the application of the **prune** procedure to State **c**, will add an additional guards $[\ \text{Goal}_{\mathbf{c}}\ ]$ to the transition *out*. The pruning procedure then terminates.

The resulting behavior of the ELTS $\mathbf{M}'$ does not reach any bad state, however, it includes many execution sequences that are not very useful because of the looping behavior of the transitions *load* and *add*. Therefore the question comes up whether it is possible to prune the behavior of $\mathbf{M}'$ further in such a way that loops without progress are eliminated and possibly also other execution sequences that are not useful for attaining the state with the specified goal. Similarly, the ELTS of $\mathbf{M}$ may include inaccessible goal states which, however, should be reachable by definition. We call such questions *termination considerations*.

We note that the constraining process defined by the pruning procedure can be further applied to the ELTS $\mathbf{M}'$ in order to determine which transitions could be pruned further in order to include only execution sequences that are useful for the termination properties. The basic idea is to split a state $\mathbf{s}$ which is involved in an infinite loop **and** has a transition leading out of this loop, into two sub-states, which we call $\mathbf{s}_{\text{req}}$ and $\mathbf{s}_{\text{looping}}$. The additional assertion associated with $\mathbf{s}_{\text{req}}$ is the guard of the transition leading out of the loop, and the additional assertion of the other sub-state is the opposite. Then we split each incoming transition into two transitions leading into the two states s_{req} and $s_{looping}$, respectively, and apply the pruning algorithm to both of them by adding an additional guard (if the transition is controllable) or splitting the starting state of the transition into two sub-states according to the additional constraints which must be applied according to the pruning algorithm. This is a recursive process, as defined in the pruning procedure. Finally, one uses one's intuition to decide the pruning of those transitions that are not useful.

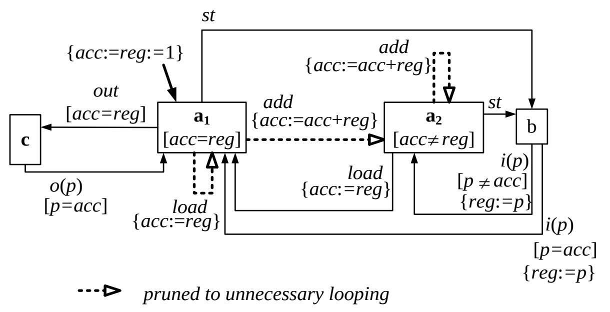

Fig. 6. Result of pruning the calculator of Fig. 2 with the goal [*acc* = *reg*] in state **c**.

If we apply this approach to the calculator example considered above (where all transitions are controllable), we obtain the machine in Fig. 6. We would propose to prune all *add* transitions, and the *load* transition from the major state "$\mathbf{a_1}$ [*acc* = *reg*]". This additional pruning is indicated in Fig. 6 by the dashed transitions.

(D): Factorial Example

Like the examples from Section 4.5.1, this example also uses the state diagram of Fig. 5. It has the following additional features: Goal of s_4 **= [*f* = *n*!]; *b*** has the update **{ *i* := *i* + 1 }; *c*** has **{ *f* := *f* * *i* }; *d*** has the guard **[*n*= *i*]**. This ELTS represents the classical Factorial function which contains two variables: a counter *i* and the variable *f* which contains the intermediate result of the calculation. The variable *n* represents the input, and the final result should be *n*! .

(Step 1) **prune** is called for state s_4 with the associated assertion [$f = n!$]; handling *d* with the guard [$n= i$], one obtains $Assr_{s2}$:= [$n \neq i \vee f = n!$] which is equivalent to [($n = i$) implies ($f = n!$)].

(Step 2) Based on some intuition, we introduce instead a stronger constraint for s_2 (called **non-required pruning** in Note **C** of the **prune** procedure), namely $Assr_{s2}$ = [$f = i!$]

(Step 3) s_2 ; *c* : $Assr_{s3}$:= [$f * i = i!$]

(Step 4) s_3 ; *b* : $Assr_{s2}$:= [$f * (i+1) = (i+1)!$] = [$f = i!$] – we see that this is a loop invariant, and the pruning process stops for the loop.

(Step 5) We now have to introduce an update statement for the *a*-transition that ensures that this invariant is satisfied when this transition leads to s_2. The simplest way is to add { $i := 1$; $f := 1$; }. ***But this is not part of the pruning process.***

If we do not use non-required pruning and instead use $Assr_{s2}$ from Step 1 directly, we get $Assr_{s3}$ = [$n \neq i \vee f*i = n!$] and a new assertion for s_2, $Assr_{s2}^{new}$ = [$n \neq i+1 \vee f*(i+1) = n!$] $\wedge$ $Assr_{s2}$ which is equivalent to [(($n=i+1$) implies ($f*(i+1) = n!$)) $\wedge$ (($n=i$) implies ($f = n!$))]. This is stronger than $Assr_{s2}$ and the pruning will not terminate, therefore not providing a solution.

5 Application to the Control of Discrete Event Systems and Submodule Construction

As mentioned in the Introduction, the problem of deriving a controller for discrete-event systems is the same as the problem of submodule construction. In both cases the system has a structure as shown in Fig. 1(b) and we have the problem, called "The unknown component problem" in [3], which is the following: The behavior of component A is given, also the desired behavior S of the system as seen at the external interfaces (while the interactions at the internal interface I_{AB} are hidden). The problem consists of finding a suitable behavior for the component B such that the composition of A and B exhibits a behavior that conforms to the requirements of S. The conformance requirements are normally safeness requirements, either expressed in the form that certain states of S should never be reached, or that all interaction sequences produced by the composition of A and B should be among the set of valid sequences defined by S. Depending on the behavior of A, the following three cases could occur: (1) all sequences of S can be realized by A and B, (2) only a subset of the valid sequences of S can be realized by A and B, or (3) there is no solution to the problem, that is, there is no B such that the composition of A and B would only produce valid sequences.

The problem of deriving controllers for discrete-event systems can be considered a special case of submodule construction, where component A represents the plant to be controlled, and component B is the controller. The main difference with submodule construction is the fact that some of the interactions of the controller are controllable while others are non-controllable. The occurrence of non-controllable interactions is solely determined by the environment of the controller, either component A or the environment E; the controller has no way to avoid the occurrence of these interactions. Controllable interactions can be prevented by the controller; they can be interpreted as rendezvous between the controller and its environments, that is, if the controller is in a state without an outgoing transition with the controllable interaction *c* then *c* cannot occur when the controller is in this state.

5.1 Algorithm for Submodule Construction

In [19], Bochmann showed that algorithms for solving the submodule construction problem in the context of different communication paradigms (including rendezvous with synchronous communication, interleaving rendezvous semantics, and input-output interactions) can be derived from a single general problem formulated in first-order logic. Equations are given for the maximal set of execution sequences of B (called *maximal solution*) and for the so-called *reduced maximal* solution which includes in general less execution sequences, but produces – in collaboration with component A – the same externally visible execution sequences as the maximal solution.

For the case of rendezvous communication with interleaving

– the communication paradigm of LTSs assumed in this paper – the following equation is derived for C_B^{red}, the reduced maximal solution for the component B (see Equation 5^{LTS} in [19], Proposition 4.1):

$$[\ C_B^{red}(x_{BS}, x_{AB})\ ] \ = \ hide^{(LTS)}{}_{AS}\,[\ C_A(x_{AS}, x_{AB}) \ \wedge \ C_S(x_{AS}, x_{BS})\ ] \setminus hide^{(LTS)}{}_{AS}\,[\ C_A(x_{AS}, x_{AB}) \ \wedge \ \neg C_S(x_{AS}, x_{BS})\ ] \qquad \textbf{(2)}$$

where the names of the interfaces have been adapted to the names of Fig.1(b). This formula uses a modeling paradigm with synchronous (simultaneous) interactions at all interfaces including null-interactions (stuttering). In order to enforce interleaving semantics, there is the constraint that at each instant, there is at most one non-null interaction at any of these interfaces. The term x_{BS}, for instance, represents an interaction sequence (including null interactions) at the interface I_{BS}; $C_S(x_{AS}, x_{BS})$, for instance, is the condition (predicate) that characterizes the pairs of sequences at the interfaces I_{AS} and I_{BS} that satisfy the behavior of the system requirement S; and $[\ C_S(x_{AS}, x_{BS})\ ]$ represents the set of pairs of sequences that satisfy this predicate. The $\wedge$ operator (logical **and**) corresponds to component composition, the operator $hide^{(LTS)}{}_{AS}$ hides the interactions at the I_{AS} interface, and $\neg$ and $\setminus$ are logical negation and set difference, respectively. Note that it is assumed here that there are no interactions at the I_{ABS} interface.

The above formula can be rewritten in a more intuitive form as follows. We write $\sum SA$, $\sum SB$, and $\sum AB$ for the set of interactions that may occur at the interfaces I_{SA}, I_{SB}, and I_{AB}, respectively. Then the alphabets of the components A, B, and of S are $\sum A = \sum SA \cup \sum AB$, $\sum B = \sum SB \cup \sum AB$, and $\sum S = \sum SA \cup \sum SB$, respectively. We write σS, σA, σB for interaction sequences over $\sum S$, $\sum A$, $\sum B$, respectively, and σ_{global} for an interaction sequence over the global system alphabet $\sum global = \sum SA \cup \sum SB \cup \sum AB$. Finally, we write Seq_X for the set of execution sequences of an X (X = A, B or S). Then the above formula for the behavior of the reduced maximal solution to the submodule construction problem can be rewritten as follows:

$$Seq_B = \{hide_{SA}(\sigma_{global}) \mid hide_{SB}(\sigma_{global}) \in Seq_A \ \wedge \ hide_{AB}(\sigma_{global}) \in Seq_S\} \setminus \{hide_{SA}(\sigma_{global}) \mid hide_{SB}(\sigma_{global}) \in Seq_A \wedge hide_{AB}(\sigma_{global}) \notin Seq_S\} \qquad \textbf{(3)}$$

where the operator $hide_Y$ eliminates the interactions in the alphabet $\sum_Y$ from the given sequence.

For the case that the behavior of S and A is defined by finite-state LTS, [19] gives an algorithm for constructing an LTS that is the solution to the submodule construction problem. This algorithm follows the evaluation of the above formula and consists of the following steps:

1. **Complementing S:** Built a completed model S′ of S by introducing a **fail** state. This completed model realizes the execution sequences accepted by the behavior of S and also the sequences in the complement which will lead to the **fail** state.
2. **Product:** Construct the product machine A x S′ which has execution sequences over the global alphabet.
3. If there are interactions at the I_{SA} interface do the following:

   3.1. **Hiding:** Hide the interactions at the I_{SA} interface, that is, replace them by spontaneous transitions that are unobservable. This results in general in a non-deterministic LTS because the hidden interactions are replaced in the machine model by spontaneous transitions.

   3.2. **Determinization:** Build the equivalent deterministic state machine. An exponentially complex algorithm exists for finite state machines [20].
4. **Pruning:** Eliminate certain transitions (of the obtained deterministic machine) such that the **fail** states of S′ become unreachable, and such that the resulting behavior, which will be the behavior of component B, has no undesirable deadlock. Note that in this process, some useful states may also be pruned. If the initial state must be pruned, then there is no solution to the submodule construction problem. It is important to note that transitions with non-controllable interactions cannot be pruned. This is the only step of the algorithm where the controllability of interactions comes into play.
5. **Checking for deadlocks:** While undesirable deadlocks in the behavior of component B are avoided by Step 4, it is still possible that the interactions between the components A and B may lead to an undesired deadlock if the Steps 3.1 and 3.2 were performed. This is already well-known for simple state machine models [19]. Such deadlocks may be eliminated by constructing the product of the components A and B and applying the pruning algorithm to this product, where – again – only the transitions in which B is involved with controllable interactions may be pruned. Since this is similar to the case of simple state machines [19], we do not discuss this further here.

5.2 Submodule Construction for Extended LTS Models: Overview

We note that Equation (**3**) above is quite general in nature and applies to arbitrary models, including models with an infinite number of states. Although the solution procedure in five steps, given above, was designed for finite-state models, it can also be used for state machine models with an infinite number of states – if each of the steps of the procedure can be performed. We explain in Sections 5.3 and 5.4 how the first two steps can be performed for ELTS models. Step 3.1 (hiding) is not a problem, but Step 3.2 (determinization) is in general undecidable. This is further discussed in Section 5.5. Finally, for Step 4 (pruning) – also an undecidable problem in general – we propose to use the pruning procedure described in Section 4.

5.3 Completing an Extended State Machine

The approach of completing a state machine by adding a (major) **Fail** state makes sense if the given state machine is deterministic. Then any execution sequence that ends up in the **Fail** state is not accepted by the original state machine. The same approach can be used for an extended state machine if it is deterministic. A sufficient condition for an ELTS to be deterministic is the following: If there is a major state with several outgoing transitions with the same interaction, then the guards of these transitions are mutually exclusive.

The **completion procedure** for a deterministic ELTS is as follows:

1. Add a new (major) state, called **Fail**.
2. For each (major) state $\mathbf{s}$ (including **Fail**), and for each interaction *b* in the alphabet, add a transition from $\mathbf{s}$ to **Fail** with interaction *b*, an empty update operation, and a guard which is the complement of the logical **and** of the guards of all transitions from $\mathbf{s}$ with interaction *b* in the original ELTS.

As an example, we consider the desired system behavior S shown in Fig. 7(a). It has three states named **1**, **2**, and **3**, and one integer variable *w*, two parameterized actions *i*(*p*) , *o*(*p*) with the integer parameter *p* , and one non-parameterized action called *two*. At the initial state **1** under *i*(*p*) the machine executes the update statement $w := p$ (*w* stores the value of the parameter *p* of *i*(*p*)) and then moves to state **2**. At state **3**, under *o*(*p*) if the value of parameter *p* equals 2 times the value of variable *w*, i.e., $p = 2 * w$, the machine returns back to state **1**. In fact, this means that the value of this parameter can be selected by the machine in such a way as to satisfy this condition. The completed version of this ELTS is shown in Fig. 7(b).

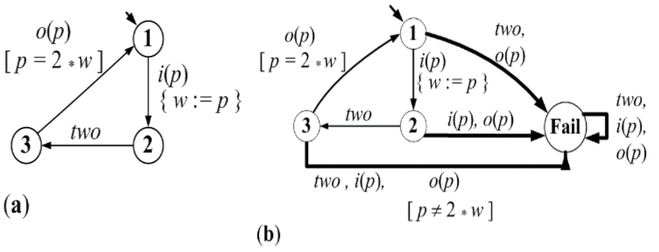

Fig. 7. **(a)** Service specification S, **(b)** Specification of (a) completed

5.4 Constructing the Product of Two Extended State Machines

The construction of an ELTS $\mathbf{M} = (\mathbf{S}, V, \Sigma, \mathbf{T}, \mathbf{s}^0, \mathcal{V}^0)$ that is equivalent to the product of two given ELTSs $\mathbf{M}_1 = (\mathbf{S}_1, V_1, \Sigma_1, \mathbf{T}_1, \mathbf{s}^0_1, \mathcal{V}^0_1)$ and $\mathbf{M}_2 = (\mathbf{S}_2, V_2, \Sigma_2, \mathbf{T}_2, \mathbf{s}^0_2, \mathcal{V}^0_2)$ is very similar to the construction of the product of two simple LTS. We assume here that the variables of these two machines are disjoint. The product $\mathbf{M}$ is constructed as follows:

- $\mathbf{S}$ is the Cartesian product of $\mathbf{S}_1$ and $\mathbf{S}_2$: $\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2$. A (major) state of $\mathbf{M}$ is written as $(\mathbf{s}_1 , \mathbf{s}_2)$, and $\mathbf{s}^0 = (\mathbf{s}^0_1 , \mathbf{s}^0_2)$.
- The variables V of $\mathbf{M}$ correspond to the variables of $\mathbf{M}_1$ and $\mathbf{M}_2$, but they are different (disjoint) from the variables of $\mathbf{M}_1$ and $\mathbf{M}_2$. For each variable of $\mathbf{M}_1$ (and of $\mathbf{M}_2$), there is a corresponding variable of $\mathbf{M}$ with the same domain. For simplicity, we assume that the variables of $\mathbf{M}$ have the same names as the variables of $\mathbf{M}_1$ and $\mathbf{M}_2$. Therefore we can write $V = V_1.V_2$ (concatenation of variable sequences), and the initial values are $\mathcal{V}^0 = \mathcal{V}^0_1.\mathcal{V}^0_2$.
- $\Sigma = \Sigma_1 \cup \Sigma_2$. Note, there is rendezvous for the interactions that are in both alphabets.
- $\mathbf{T}$ contains for each (major) state $(\mathbf{s}_1 , \mathbf{s}_2)$ the following type of transitions:
    1. Joint (rendezvous) transitions: If $\mathbf{T}_1$ contains the transition $\mathbf{t}_1 = (\mathbf{s}_1 - b(p)\ [\ G_1\ ]\ up_1 \text{ -> } \mathbf{s}_1')$ and $\mathbf{T}_2$ contains the transition $\mathbf{t}_2 = (\mathbf{s}_2 - b(p)\ [\ G_2\ ]\ up_2 \text{ -> } \mathbf{s}_2')$ then $\mathbf{T}$ contains the transition $\mathbf{t} = (\ (\mathbf{s}_1, \mathbf{s}_2) - b(p)\ \ [G_1 \wedge G_2\ ]\ up \text{ -> } (\mathbf{s}_1', \mathbf{s}_2')\ )$ where *up* is the concurrent execution of up_1 and up_2 .

2. Separate transition of $\mathbf{M}_1$ involving an interaction *b* that is not in Σ_2: If $\mathbf{T}_1$ contains the transition $\mathbf{t}_1 = (\mathbf{s}_1 - b(p)\ [\ G_1\ ]\ up_1 \text{ -> } \mathbf{s}_1')$ and $b(p)$ is not in Σ_2, then $\mathbf{T}$ contains the transition $\mathbf{t} = (\ (\mathbf{s}_1, \mathbf{s}_2) - b(p)\ [\ G_1\ ]\ up_1 \text{ -> } (\mathbf{s}_1', \mathbf{s}_2)\ )$.
3. Separate transition of $\mathbf{M}_2$ (similarly).

In Section 3.3.1 we considered that for a given ELTS, a (major) state $\mathbf{s}$ may be associated with an assertion $\text{Assr}_{\mathbf{s}}$ that is True whenever this (major) state is reached. Given the above definition of the product of two state machines $\mathbf{M}_1$ and $\mathbf{M}_2$ the following is clear:

> If two states $\mathbf{s}_1$ and $\mathbf{s}_2$ of the respective machines are associated with the assertions $\text{Assr}_{\mathbf{s1}}$ and $\text{Assr}_{\mathbf{s2}}$, respectively, then the state $(\mathbf{s}_1, \mathbf{s}_2)$ of the product machine $\mathbf{M}$ would be associated with the assertion $\text{Assr}_{(\mathbf{s1},\mathbf{s2})} = (\text{Assr}_{\mathbf{s1}}\ \wedge\ \text{Assr}_{\mathbf{s2}}\ )$.

However, additional assertions may become true. If a state of the product machine is only reached through a rendezvous transition involving an interaction with a parameter, then the value of this parameter will be visible to both components and, through the respective update statements, may give rise to an assertion relating variables of the two different machines. Let us consider as an example the transition $\mathbf{t} = (\ (\mathbf{s}_1, \mathbf{s}_2) - b(p)\ [\ G_1 \wedge G_2\ ]\ \text{up} \text{ -> } (\mathbf{s}_1', \mathbf{s}_2'))$ considered under point (1) above. If the update statement up_1 of $\mathbf{M}_1$ assigns the value of the parameter *p* to the local variable v_1 and the update statement up_2 of $\mathbf{M}_2$ assigns the value of *p* to the local variable v_2, then we can add the assertion $\text{Assr}_{\mathbf{new}} = [v_1 = v_2]$ to the state $(\mathbf{s}_1', \mathbf{s}_2')$, assuming that it cannot be reached by other transitions. We call such assertions ***parameter-implied assertions***.

We note that [21] also introduces a composition operator that uses rendezvous communication with interaction interleaving. This operator allows that a variable is shared among different components, however, it imposes the constraint that a shared variable must be updated the same way by the sharing components. The shared variables in [21] are used for exchanging values between the sharing components. The operator employed in this paper assumes disjoint variables of components; however, it allows the exchange of variables values via the parameterized interactions. Thus, the consistent sharing constraint of [21] corresponds to the parameter-implied assertions discussed in the paragraph above. Our formalism has the advantage that the variables and updates of a component can be defined independently of any assumption about the other component variables and updates. Thus, when the proposed operator is used in the context of submodule construction (supervisor control), the desired behavior of the system and that of the known component (plant) have different variables; and the desired solution (controller) obtains local copies of these variables.

5.5 Dealing with Non-Observable Interactions

If non-observable interactions are identified in Step 3.1 of the controller derivation algorithm of Section 5.1, then Step 3.2 must be performed to obtain a determinized version of the product machine A x S′. For the case of non-extended state machine, there is a well-known algorithm of exponential complexity to obtain an equivalent deterministic state machine for a given non-deterministic one [20]. For extended state machines, the determinization of non-deterministic state machines without spontaneous (non-observable) interactions is described in [22]. Nondeterminism occurs, if in a given major state, there are several outgoing transitions with the same interaction and overlapping guards. This problem is in general undecidable - the paper also explains under which conditions the given algorithm terminates.

We note that Step 3.1 leads to a state machine model of the product that contains spontaneous transitions, which in general introduces nondeterminism. In the case that the extended machine contains no loop of (only) spontaneous transitions, then the product machine can be transformed into a (in general) non-deterministic machine that contains no spontaneous transition, as explained in [22]. The idea is to replace a sequence of spontaneous transitions $\mathbf{t}_i$ (i=1, 2, … (n-1)) followed by an observable transition $\mathbf{t}_n$, by a modified transition $\mathbf{t}_n'$, where all guards and update operations of the spontaneous transitions are deferred to $\mathbf{t}'_n$. Assuming that the given transitions have the form $\mathbf{t}_i = (\mathbf{s}_i - b_i\ [\ G_i\ ]\ up_i \text{ -> } \mathbf{s}_{i+1})$, where b_i (i=1, … (n-1)) = ε (spontaneous), then transitions $\mathbf{t}'_n$ has the form $\mathbf{t}'_n = (\mathbf{s}_n - b_n\ [\ G_1 \wedge (G_2 \text{ o } up_1) \wedge \ldots \wedge (G_{n-1}\ up_{n-1} \text{ o } \ldots \text{ o } up_1)]\ (up_n \text{ o } up_{n-1} \text{ o } \ldots \text{ o } up_1) \text{ -> } \mathbf{s}_{i+1})$. This means, the update operations are performed by $\mathbf{t}'_n$ in the order of the spontaneous transitions, and the guard of the transition $\mathbf{t}'_n$ ensures that the guard conditions of the spontaneous transitions are satisfied. Note that a spontaneous transition with a parameter will introduce a FORALL quantification in the resulting guard for all values of this parameter.

We note that in the case that the elimination of the spontaneous transitions does not introduce any nondeterminism and the state $\mathbf{s}_n$ has no other incoming transitions, we do not need to eliminate these spontaneous transitions, and we can apply the pruning procedure of Section 4 directly on the product machine including the spontaneous transitions.

5.6 Example: The Controller for the Calculator

As an example, we consider the completed desired system behavior of S (Fig. 7(b)) and Component A (the calculator, see Fig. 3). The architecture of this example is shown in Fig. 8, where the solution (*controller*) will obtain local copies of the variables of Component A and S. The product machine of S and A is in Fig. 9. In this example, interactions *i*(*p*) and *o*(*p*) are non-controllable by the controller as they are not visible, while all other interactions are controllable.

We note that for simplicity of presentations, we write in the following figures A1 for the assertion [$acc = n{*}reg \wedge n > 0$] of state $\mathbf{a}_2$ of component A, and Ap for the *parameter-implied* assertion [$reg = w$].

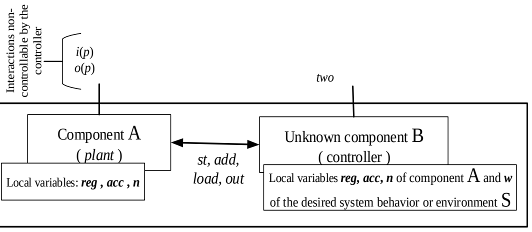
Interactions non-controllable by the controller
i(p)
o(p)
two
Component A
(plant)
Local variables: reg , acc , n
st, add, load, out
Unknown component B
(controller)
Local variables reg, acc, n of component A and w of the desired system behavior or environment S

Fig. 8. Architecture of the Calculaor example

In the following, we go through the four steps of controller construction, as discussed in Section 5.1. The first step, the completion of the desired model S, was already discussed in Section 5, and leads to the completed S shown in Figure 7(b).

The second step of the controller construction (the product construction) leads to the product machine shown in Figure 9. The update statement of the *i*(*p*) transition from state **(1, b)** leads to the parameter-implied assertion Ap = [$reg = w$] which remains True in all succeeding states.

In the third step (hiding and determinization), the transitions *i*(*p*) and *o*(*p*) are recognized to be hidden (unobservable) and therefore uncontrollable. In particular, the parameter of these transitions is also hidden from the controller. Therefore, the controller cannot know the values of its own variables *reg* and *w*, since they are assigned by the hidden *i*(*p*) transition, and also the value of the variable *acc* is unknown, since it depends on *reg*. Only the value of the variable *n* is known since it is assigned by visible (controllable) transitions. However, the controller knows certain assertions that are associated with its major states.

During the subsequent determinization sub-step, the hiding of *i*(*p*) leads to the combination of the states **(1, b)** and **(2, $\mathbf{a}_1$)** into a single state, which we call **{(1, b), (2, $\mathbf{a}_1$)}**, and similarly, the hiding of *o*(*p*) leads to the combined state **{ (3, c) , (1, $\mathbf{a}_1$) }.**

In the following we describe the fourth step of the controller construction: Pruning. Each major state of the product that contains the **Fail** state of S, denoted as • in Figure 9, is a problematic state with assertion False and thus, such states should not be reached. In the following, we explain the pruning process in several steps.

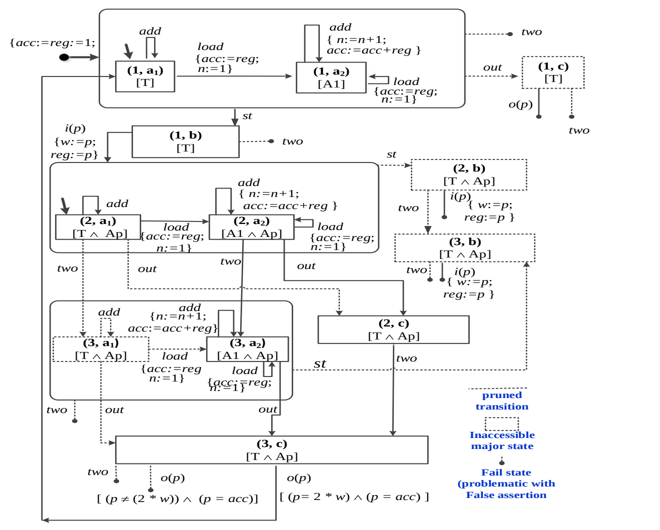
Fig. 9. The product completed S (Fig. 7(b)) and Component A (Fig. 3)

• **Pruning the controllable *two*-transitions leading to the Fail state:** Let us first consider the transition *two* from state **{ (3, c) , (1, a$_1$) }** in Fig. 9. The next state of this transition is a **Fail** state, which has the associated assertion False. Applying the prune procedure to this state and handling the incoming *two* transition, this transition obtains the guard False, which means that it cannot be executed and therefore is eliminated. In the same way, all other *two* transitions leading to the **Fail** states can be pruned in Fig. 9. We note that also transition *two* from state **(1, b)** must be pruned because it leads to the **Fail** state. However, this state is combined with state **(2, a$_1$)** through the determination step. Therefore the *two* transition must also be pruned from state **(2, a$_1$)**. As a result, the state **(3, a1)** becomes inaccessible from **{ (1, b) , (2, a$_1$) }** and is eliminated with all its outgoing transitions.

• **Pruning non-controllable (spontaneous) transitions leading to the Fail state:** The transition *o*(*p*) from state **(1, c)** leads to the **Fail** state. Since the transition cannot be pruned, its outgoing state **(1, c)** becomes a problematic state with a False assertion which means it is a failing state and should not be accessible. Since *out* is controllable, we can prune the *out* transition leading to this state. Similarly, the *i*(*p*) transitions from states **(2, b)** and **(3, b)** lead to **Fail** states. Therefore we prune the *st* transitions that lead to these states. The states become inaccessible and can be removed.

• **Dealing with the o(p) transitions from the state { (3, c) , (1, a1) }:** The major states **(3, c)** in Figure 9, which is part of **{ (3, c) , (1, a1) }**, has two hidden outgoing *o*(*p*) transitions with guards [$p = 2{*}w \wedge p = acc$] and [$p \neq 2{*}w \wedge p = acc$], respectively. As these transitions are non-controllable, to make the bad transition leading to the **Fail** state **(**written as **•)** impossible, we need the required Assr$_{s'}^{(new)}$ for this state to be $\exists\, p$ such that $p = (2 * w \wedge p = acc)$. This means $acc = 2 * w$. Since Ap = [$reg = w$] holds, we have the assertion [$acc = 2 * reg$]. Then pruning continues by considering the two incoming transitions under the actions *two* and *out*. We note that the *out* transition from the state **(3, a1)** must be eliminated since the assertion [$acc = 2 * reg$] cannot be ensured. For the *out* transition from state **(3, a2)**, [$acc = 2 * reg$] is ensured if we add the guard [$n = 2$], since the starting state of that transition has the assertion [$acc = n{*}reg \wedge n > 0$] which together with the guard ensures [$acc = 2 * reg$]. For the *two* transition from state **(2, c)** the required assertion for that state will become the same as for state **{(3, c), (1, a$_1$)}**. Then the pruning of the incoming *out* transitions from the states **(2, a1)** and **(2, a2)** would be handled as the *out* transition to state **{ (3, c) , (1, a$_1$) }**.

The resulting sub-machine of the product of Fig. 9 is shown in Fig. 10. The reader may observe that the obtained sub-machine includes only the paths that do not lead to any Fail (•) or a deadlock. We note that the resulting controller is a maximal submachine that ensures that during an *o*(*p*) transition the parameter *p* satisfies the requirement [$p = acc = 2 * w$]. The pruning of transitions is based on the visibility of the controller component (as shown in Figure 8), and in particular the fact that the controller cannot know the value of its own *reg* variable which is assigned during the hidden *i*(*p*) transition. If the controller could know the value of its *reg* variable, it could for instance allow for a weaker guard for the *out* transition from state **(3, a2)**, namely [($reg = 0 \wedge n > 0) \vee [n = 2]$ instead of [$n = 2$].

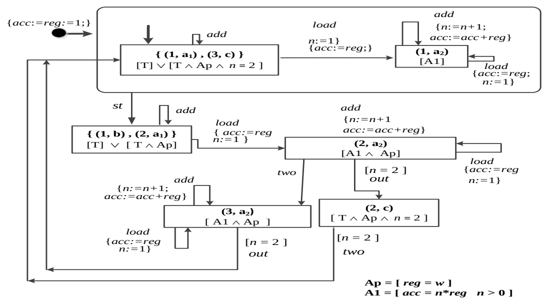
Fig. 10. The machine obtained after applying the pruning algorithm

6 Related Work

According to our knowledge, the procedure we proposed for symbolically pruning a possibly infinite ELTS that may have parameterized and non-controllable interactions, with respect to some user defined requirements (given as assertions at states) is novel. The way our procedure Prune uses assertions is very similar to the use of assertions and invariants for proving properties of programs. In other words, Prune utilizes for the considered ELTS the calculus for weakest preconditions developed for programs by Dijkstra [17]. The computation of weakest preconditions for many types of programs under various restrictions is investigated in many papers. The reader may refer to Barnett et al. [23] for more related information. In addition, Prune uses a backward propagation strategy which is well studied in verification for solving the reachability problem over various types of state models. For example, closely related to the considered model, Delzanno [24] studied symbolic reachability using backward propagation for Cache Coherent Protocols modeled as infinite extended state machines over integer variables where guards can be represented as integer constraints. We note that efficient solutions for backward propagation, which usually use some combinations of model checking tools, abstraction, binary decision diagrams, and constraint satisfaction techniques, are well studied in numerous papers. The reader may refer to [24] for some related work. In this paper, we also use backward propagation as a strategy; however, our procedure is rather different from the previous backward procedures as we do not determine reachability but rather derive a sub-machine of a given extended machine taking into consideration certain general and specific goals or requirements, written as predicates at states, such that the concrete states that do not satisfy these requirements become inaccessible and no undesirable deadlocks are introduced while deriving the sub-machine. In the Conclusion section, we state how this previous work can be used to extend our work.

To the best of our knowledge, the only earlier work on submodule construction for extended state machines (with possibly infinite state space) was presented by Daou and Bochmann in [25]. This work introduced the idea of state splitting for handling the pruning of the submodule behavior in order to avoid global interaction sequences that violate the given desired behavior of the system. This idea inspired us for the pruning procedure presented in this paper. We note, however, that [25] considered a very restricted form of ELTS where values of interaction parameters or variables are only copied, but never used for calculating new values through assignment expressions. As a consequence, state assertions were limited to equalities between variable values, and the procedure itself does not consider and handle user defined assertions at states.

A method for the derivation of a controller from an extended state model is proposed by Ouedraogo et al. in [21]. However, the work in [21] considers the simplified architecture of Figure 1(a) where all interactions are visible by the controller; in addition, the considered extended machine model is assumed to be finite (in terms of state space and variables domains) and the approach does not consider exactly the same control objectives (pruning requirements) discussed in this paper. Accordingly, the controller construction approach is simpler and always terminates. Here we show that when considering the general architecture (Figure 2(a)) and the assumed extended machine model (with possibly infinite state space and assertions at states), a solution might not be possible in general. In fact, in this paper, a formula for obtaining the behavior of the controller is provided for possibly infinite LTSs and an algorithm is proposed, for the considered ELTS model and control objectives, and appropriate operators are utilized in the algorithm for constructing machine products, for complementing, and for determinization and pruning. In addition, the work in [21] refers to individual concrete states while our pruning procedure is formulated at a higher level of abstraction using transition guards and state assertions. Again a major difference between of our approach and the work in [21] is that our approach provides the intended solution (controller) considering general user defined goals introduced as assertions at states. Thus, our construction approach of a supervisor is much involved as it has to satisfy certain user defined assertions taking into account that some interactions are non-controllable. In this case, restricting the guards of incoming transitions with non-controllable interactions to certain major states with undesirable behavior is not possible; and there is a need to appropriately propagate this bad behavior to the starting states of these transitions, and then do further propagation as needed. Also, as some interactions are non-controllable, determinization is usually used while deriving the controller.

We note that there exists other approaches for the synthesis of controllers for simple and extended state machines with partial observability where it is assumed that the controller observes states of the plant. The reader may refer to Kalyon et al. [26] for a related summary. These approaches to partial observability are rather different from what is considered in this paper and in [25] for extended machines and in [2-3], [27-28] for simple state machines, where partial observability of actions (not states) is assumed.

7 Conclusions

This paper presents a procedure that symbolically prunes a given extended state machine leading to a maximal sub-machine that satisfies certain user-defined general goals, specified as assertions at states, and other specific goals, such as the elimination of undesired deadlocks at certain states and fail states which should never be reached. The procedure also removes all undesirable deadlocks which could be produced during pruning. Application examples demonstrating several aspects of the proposed procedure including its usefulness and limitations are provided. It is also worth mentioning that the presented pruning procedure can also be used for pruning non-deterministic extended state machines.

As topics for further study, we mention the following. While the pruning requirements described in Section 4.1 are safeness properties and deadlocks, it would be interesting to further explore how the pruning algorithm could be adapted to take into account progress properties (liveness), and possibly eliminate execution sequences that are not useful for attaining a specific goal state. This aspect is only discussed informally in this paper (see Section 4.5.2).

The second contribution of the paper is the application of the pruning algorithm to the control of discrete event systems and submodule construction. In this regard, we also show how the general submodule (controller) construction algorithm for finite state machines can be adapted to extended (possibly infinite) state machine models. Here we consider certain traditionally considered specific control objectives in addition to certain general user defined objectives. In this paper, we make abstraction from input and output; however, the work can also be applied when a distinction between inputs and outputs is necessary, i.e. to systems modeled as extended I/O automata; for example.

As another topic of further research, it would be interesting to implement and experimentally assess the proposed work and also consider efficient decidable implementations of both the pruning and the submodule construction problems. This can be done by either considering various restricted versions (in terms of types of variables, assertions, and guards) of the considered extended model or by using combinations of model checking tools, abstract interpretation, binary decision diagrams, and constraint satisfaction techniques [24], [26], [29-31].

References

[1] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," in *Proc. of the IEEE,* 77(1), 1989.

[2] P. Merlin and G. v. Bochmann, "On the construction of submodule specifications and communication protocols," *ACM Trans. On Programming Languages and Systems*, 5(1), 1983, pp. 1-25.

[3] T. Villa, N. Yevtushenko, R. Brayton, A. Mishchenko, A. Petrenko, and A. Sangiovanni-Vincentelli, *The Unknown Component Problem: Theory and Applications*, Springer, 2012.

[4] B. A. Brandin and W. M. Wonham, "Supervisory control of timed discrete-event systems," *IEEE Tran. on Automatic Control,* 39(2), pp 329-342, 1994.

[5] J. Parrow, "Submodule construction as equation solving in CCS," *Theoretical Computer Science*, 68, 1989.

[6] N. Yevtushenko, T. Villa. R. K. Brayton, A. Petrenko, A. Sangiovanni-Vincentelli, “Solution of parallel language equations for logic synthesis,” In *Proc. of the International Conference on Computer-Aided Design*, 2001, pp. 103-110.

[7] R. Kumar, S. Nelvagal, and S. I. Marcus, “A discrete event systems approach for protocol conversion,” *Discrete Event Dynamical Systems: Theory and Applications*, 7(3), 1997, pp. 295-315.

[8] J. Drissi and G. v. Bochmann, “Submodule construction for systems of I/O automata,” Technical Report #1133, DIRO, Université de Montréal, Canada, 1999.

[9] H. Qin and P. Lewis, “Factorisation of finite state machines under strong and observational equivalences,” *Formal Aspects of Computing*, 3, 1991, pp. 284-307.

[10] L. de Alfaro and T. Henzinger, “Interface automata,” In *Proc. of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[11] S. G. H. Kelekar, “Synthesis of protocols and protocol converters using the submodule construction approach,” In. A. Danthine et al, editors, *Protocol Specification, Testing, and Verification-* PSTV XIII, 1994.

[12] Z. Tao, G. v. Bochmann and R. Dssouli, “A formal method for synthesizing optimized protocol converters and its application to mobile data networks,” *Mobile Networks & Applications*, 2(3), 1997, pp. 259-69.

[13] P. Inverardi and M. Tivoli, “Automatic synthesis of modular connectors via composition of protocol mediation patterns,” *2013 35th International Conference on Software Engineering (ICSE),* San Francisco, CA, 2013, pp. 3-12.

[14] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer, “Assumption generation for software component verification,” In *Proceedings of the 17th IEEE international conference on Automated software engineering* (ASE '02). IEEE Computer Society, Washington, DC, USA, 3-, 2002.

[15] A. Petrenko, N. Yevtushenko, G. v. Bochmann and R. Dssouli, “Testing in context: framework and test derivation,” *Computer Communications Journal, Special issue on Protocol Engineering,* 19, 1996, pp. 1236-1249.

[16] K. El-Fakih and N. Yevtushenko, “Test translation for embedded finite state machine components,” *The Computer Journal*, 59(12), 2016, pp. 1805-1816.

[17] E. W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[18] C. A. Furia, B. Meyer, and S. Velder **“**Loop invariants: Analysis, classification, and examples,” *ACM Computing Surveys*, 46(3), Article 34, January, 2014.

[19] G. v. Bochmann, “Using logic to solve the submodule construction problem,” *Discrete Event Dyn. Syst.*, 23, 2012, pp. 27–59.

[20] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986.

[21] L. Ouedraogo, R. Kumar, R. Malik, K. Åkesson, “Nonblocking and safe control of discrete-event systems modeled as extended finite automata,” *IEEE Trans. Automation Science and Engineering* 8(3), 2011, pp. 560-569.

[22] T. Jeron, H. Marchand, and V. Rusu, “Symbolic determinisation of extended automata,” *Proc. of the 4th IFIP Int. Conf. on Theoretical Computer Science*, 2006, pp.197-212.

[23] M. Barnett and K. Rustan M. Leino, “Weakest-precondition of unstructured programs,” In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (PASTE '05), Michael Ernst and Thomas Jensen (Eds.). ACM, New York, NY, USA, 2005, pp. 82-87.

[24] G. Delzanno, “Automatic verification of parameterized cache coherence protocols,” *International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2000.

[25] B. Daou, G. Bochmann, “Submodule construction for extended state machine models,” *Proc. of the 25th Int. Conf. on Formal Techniques for Networked and Distributed Systems- FORTE* ‘05, Lecture Notes in Computer Science 3731, Springer, 2005, pp. 396-410.

[26] G. Kalyon, T. Le Gall, H. Marchand, Thierry Massart, “Symbolic supervisory control of infinite transition systems under partial observation using abstract interpretation”, *Discrete Event Dynamic Systems*, Springer Verlag, 2012, 22 (2), pp.121-161.

[27] F. Lin, W. Wonham W, “On observability of discrete-event systems,” *Inf Sci.*, 44(3), 1988, pp. 173–198.

[28] D. Ciolek, V. Braberman, N. D’Ippolito, N. Piterman and S. Uchitel, “Interaction models and automated control under partial observable environments,” in *IEEE Transactions on Software Engineering*, 43(1), 2017, pp. 19-33.

[29] Z. Fei, S. Miremadi, K., Akenson, B. Lennartson, “Efficient symbolic supervisor synthesis for extended finite automata”, *IEEE Trans. On Control Systems Technology*, 22(6), 2014.

[30] T. Le Gall, B. Jeannet, H. Marchand, “Supervisory control of infinite symbolic systems using abstract interpretation,” In: *Decision and Control, 2005 and 2005 European Control Conference*, CDC-ECC ‘05, 2005, pp 30–35.

[31] P. Cousot P and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” In: POPL ‘77, 1977, pp 238–252.