Explicit-Symbolic Modelling for Formal Verification

Umberto Costa[1],[2]   Sérgio Campos[3]   Newton Vieira[5]

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil

David Déharbe[4]

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal, Brazil

Abstract

We propose a model that combines explicit and symbolic representations in an explicit-symbolic formal verification model. Both explicit and symbolic models have been successfully used in the verification of finite state concurrent systems, such as complex sequential circuits and communication protocols. The proposed model aims to use explicit and symbolic techniques simultaneously to verify the same model and to make it possible to employ the most efficient technique to each aspect of the model. First, we formalize the explicit-symbolic model and show how it can be generated from a labeled state-transition system. Then, we apply those ideas to systems described in the Verimag Intermediate Format and present the main algorithms for integrating the underlying models.

Keywords: model checking, explicit state, binary decision diagrams

1   Introduction

Due to the ever-growing complexity of computing applications, automatic tools have been used to help developers to find bugs in hardware and software systems. Two approaches that have been applied to achieve this goal are simulation and formal verification. The simulation approach consists of executing tests over given inputs and examining the results. This technique can easily evaluate both control and data aspects, but generally only part of the set of states is examined. In other words, simulation can state that specifications hold for some inputs, but it does not prove correctness over the whole set of states.

Formal Methods include mathematical based languages, techniques and tools for specifying and verifying hardware and software systems [12]. Complex systems and critical applications benefit from formal methods because they make systems more reliable. Different from simulations, formal verification techniques execute exhaustive searches over problem domains, proving conformity to specifications instead of only pointing errors for some input data.

Model checking is one well-known and successful formal technique for verifying finite state concurrent systems. It has been successfully used to verify complex sequential circuits designs and communication protocols [11]. Model checking consists of representing a given system by means of a finite model to be exhaustively analyzed in order to determine its conformance to some properties. A model can be represented as a graph where each vertex is a state of the system and edges are valid transitions between states. States can be represented either explicitly or by means of some symbolic representation.

Explicit models enumerate states individually, storing them in structures like hash tables. Explicit methods tend to present relatively more predictable efficiency and memory behaviors. On the other hand, symbolic models explore regularity in the state space aiming to produce more compact representations. Instead of exploring states individually as explicit model checking does, symbolic model checking uses efficient encoding of Boolean logical formulae to represent and to explore sets of states atomically. So, symbolic model checking usually allow us to verify systems with a much higher number of states when compared to explicit model checking [14]. In order to perform a symbolic model checking, sets of states and transitions are represented by characteristic functions. Computational representations of characteristic functions must be efficient and should provide essential operations, as conjunction, disjunction, equality tests, existential quantification and substitution. Binary Decision Diagrams [6], known as BDDs, are a very efficient symbolic representation of propositional logical functions. Unfortunately, sometimes it is

difficult to express satisfactorily data information, like constraints on integer or real values, by means of simple Boolean expressions [17]. Generally, integers and arithmetic operations are not efficiently represented by means of simple Boolean expressions, due to the size of produced representations [8]. Also, systems with continuous variables ranging over non-countable domains, like real-time systems, are not suitable for symbolic model checking [16]. Usually, explicit methods are more efficient than symbolic methods when the state space presents a less regular structure.

In this paper, we propose a model for combining explicit and symbolic representations in an explicit-symbolic formal verification model. We intend to use explicit and symbolic techniques together to verify the same model and to make it possible to employ the most efficient technique to each aspect of the model. In the next section, we present related works and our main goals. After that, we develop the explicit-symbolic model and show how generate it from a labeled state-transition system. Then, we discuss how formal verification tools can use our model to verify systems given in the Verimag Intermediate Format [4]. The algorithms developed for the integration of underlying models are also shown. Finally, we present our next steps and some remarks.

2 Related Work

Our explicit-symbolic model integrates the underlying explicit and symbolic models. So, it has to do with explicit and symbolic verifiers and their techniques. Currently, SPIN [13] and JPF [5] are well-known and largely used explicit-state model checkers. On the other hand, SMV [15], NuSMV [9] and Verus [7] are representative and successful symbolic-state model checking tools. Existing algorithm optimizations from such tools, like the on-the-fly technique used by SPIN, should be considered for the implementation of underlying models in order to improve the model checking.

2.1 SLAM

The approach adopted in the SLAM project [3] is the one that more closely relates to our project, as it employs both explicit and symbolic models inside a same environment. SLAM extracts abstract models from C code and statically checks temporal properties of software. Also, SLAM uses predicate abstractions, symbolic reasoning and iterative refinement. Bebop is the part of SLAM liable for performing reachability analysis of Boolean programs.

Boolean programs are abstraction of programs where the concrete states have been mapped to abstract states under evaluation of a finite set of predicates. The resulting program roughly corresponds to a C program with the

same control-flow constructs, but where all variables have Boolean type. Because all variables have Boolean type, the state space of the program is finite. Consequently, reachability and termination are decidable for Boolean programs. More information on the predicate abstraction algorithm and its corresponding implementation can be found on [1]. Given a Boolean program, Bebop performs an inter-procedural data-flow analysis in order to determine reachability information [2]. Bebop represents control flow explicitly and sets of states implicitly using BDDs. The explicit representation of control-flow features is justified due to the usage of compiler optimization techniques. On the other hand, BDDs are used to symbolically represent the input and output behavior of procedures. They are used to represent sets of reachable states at a program point. A state contains the program counter and values to all the variables visible at that point.

Comparing to our explicit-symbolic model, SLAM lacks flexibility because control-flow and data-flow information must have explicit and symbolic representations, respectively. The proposed model is more general because it allows us to move variables between explicit and symbolic spaces according to any policy. Thus, we can experiment with a variety of combinations and choose the one that best fits the system needs. See section 7 for more details about the flexibility of the proposed model.

3 The Explicit-Symbolic Modelling

Although an explicit-symbolic model can be directly generated from a labeled state-transition model, we decided to accomplish this task in two distinct steps. Initially, both explicit and symbolic models are generated as projections induced by the partitioning of variables of the original model. Next, the explicit and symbolic models are combined to compose the explicit-symbolic model. Such approach provides a more intuitive and clear procedure to understand the formalization of the model.

3.1   Decomposition of the Original Model

Let $M = (S, R, L)$ be the model for a given system, where S represents the set of states, $R \subseteq S \times S$ represents the transition relation between states, $L : S \to 2^{AP}$ represents the labeling function which assigns a set of atomic propositions to each state. Assume AP as the set of atomic propositions of M. Suppose that the set AP is decomposed into subsets of explicit and symbolic propositions AP_e and AP_s, respectively, where $AP = AP_e \cup AP_s$ and $AP_e \cap AP_s = \emptyset$. Using this proposition partitioning, two equivalence relations are generated, $\approx_e$ and $\approx_s$, such that

$$\forall\, s,\, s' \in S,\; s \approx_e s' \longleftrightarrow L(s)|_{AP_e} = L(s')|_{AP_e}$$

$$\forall\, s,\, s' \in S,\; s \approx_s s' \longleftrightarrow L(s)|_{AP_s} = L(s')|_{AP_s}$$

Each induced equivalence class $[s]_e$ is an explicit state and each equivalence class $[s]_s$ is a symbolic state. In other words, each state $s \in S$ corresponds to an explicit state $[s]_e$ and a symbolic state $[s]_s$. Consequently, the set of states S is projected into explicit and symbolic sets of states, S_e and S_s, as defined below:

$$S_e = \{[s]_e \,|\, s \in S\} \qquad S_s = \{[s]_s \,|\, s \in S\}$$

The $\approx_e$ and $\approx_s$ relations define two labeling functions L_e and L_s for explicit and symbolic states, respectively. Labels for an explicit state $[s]_e$ and a symbolic state $[s]_s$ are shown below:

$$L_e([s]_e) = L(s)|_{AP_e} \qquad L_s([s]_s) = L(s)|_{AP_s}$$

Therefore, $L(s) = L_e([s]_e) \cup L_s([s]_s)$ because $L(s) = L(s)|_{AP_e} \cup L(s)|_{AP_s}$.

The projection of the original system requires the original transition relation R to be split into a transition relation for the explicit and another for the symbolic part of the model, R_e and R_s, respectively. In order to maintain the correspondence between explicit and symbolic transitions, the original transitions must be labeled. One explicit transition and one symbolic transition will be assigned the same label $id \in \mathbb{N}$ only if they come from the same original transition (s, id, s'), where $s, s' \in S$. So, labels associate explicit and symbolic transitions, indicating which occur together on the models. Given one original transition relation R, the following two transition relations are obtained:

$$R_e = \{([s]_e, id, [s']_e) \,|\, (s, id, s') \in R\} \qquad R_s = \{([s]_s, id, [s']_s) \,|\, (s, id, s') \in R\}$$

that is, if $(s, id, s') \in R$ then $([s]_e, id, [s']_e) \in R_e$ and $([s]_s, id, [s']_s) \in R_s$.

The original model can be restored using the explicit and symbolic models together. Each two transitions $([s]_e, id_e, [s']_e) \in R_e$ and $([s]_s, id_s, [s']_s) \in R_s$, where $id_e = id_s$, defines one original transition $(s, id, s') \in R$ such that $id = id_e = id_s$, $L(s) = L_e([s]_e) \cup L_s([s]_s)$ and $L(s') = L_e([s']_e) \cup L_s([s']_s)$.

3.2 Composition of the Explicit-Symbolic Model

Given $M_e = (S_e, R_e, L_e)$ and $M_s = (S_s, R_s, L_s)$, we obtain the explicit-symbolic model $M_h = (S_h, R_h, L_h)$ by the composition of such explicit and symbolic models. Each explicit-symbolic state $s_h \in S_h$ is given by a pair (s_e, s_s), where $s_e \in S_e$, $s_s \in S_s$ and there exist explicit and symbolic transitions

$(s_e, id_e, s'_e) \in R_e$ and $(s_s, id_s, s'_s) \in R_s$ such that $id_e = id_s$. For each $s_h \in S_h$, we have $L_h(s_h) = L_h(s_h)|_{AP_e} \cup L_h(s_h)|_{AP_s}$, where $L_h(s_h)|_{AP_e} = L_e(s_e)$ and $L_h(s_h)|_{AP_s} = L_s(s_s)$. If $(s_e, id, s'_e) \in R_e$ and $(s_s, id, s'_s) \in R_s$ then $(s_h, id, s'_h) \in R_h$, where $s_h, s'_h \in S_h$. Due to the definition of R_h, models M_e and M_s must be explored in an interleaved and synchronized fashion, based on transition labels. Note that states of S_h are visited during the exploration of states in S_e and S_s.

Taking an explicit-symbolic model, we can restore the original explicit and symbolic models in a straightforward way. Given $M_h = (S_h, R_h, L_h)$, we generate $M_e = (S_e, R_e, L_e)$ such that $S_e = \{s_e | (s_e, s_s) \in S_h\}$, $R_e = \{(s_e, id, s'_e) | ((s_e, s_s), id, (s'_e, s'_s)) \in R_h\}$ and $L_e(s_e) = L_h((s_e, s_s))|_{AP_e}$. The model M_s can be analogously obtained.

4 Intermediate Format

The Intermediate Format [4], also called IF from now on in this paper, is a language that has been developed in order to model asynchronous communicating real-time systems. It has been used as interchange format between a set of validation tools known as IF validation environment. IF models are composed from process instances, running in parallel and interacting asynchronously through shared variables and message-passing. Message-passing is accomplished by means of signals, instances of signal routes and FIFO communication buffers.

In this section, we apply the explicit-symbolic modelling to IF models. First, we describe the main components in the language and present some examples of its constructions. Next, we show how those components are employed for generating both explicit and symbolic models. Finally, we combine explicit and symbolic models in order to emulate the behavior of the original model by means of the explicit-symbolic model.

4.1 The Modelling Language

System specifications consist of generic components, including dynamic ones such as processes, signal routes and signals, and static ones such as variables, data types, constant values and external procedures. Formally, a system is given as a tuple (D, S, P), where D is the set of data types including Boolean, integer, clock, pid and user-defined ones, S is the set of typed signals and P is the set of process types. In figure 1, we show some of those elements composing an IF specification for the alternate bit protocol.

```
system bitalt;                          process transmitter(1);
type data = range 0..1;                 var b boolean;
signal ack(boolean);                    ...
...                                     state start #start;
process transmitter(1);                    task b := false;
...                                           nextstate idle;
endprocess;                             endstate;
...                                     ...
endsystem;                              endprocess;

              (a)                                     (b)
```

Fig. 1. Intermediate Format system (a) and process (b) examples.

Processes are defined as extended finite-state machines. Each process instance has a unique identifier number and an input FIFO buffer storing all the incoming messages. Process specifications include the set of local variables, which correspond to the local memory, and the set of control states. Data types, constants and procedure definitions may also be included. Formally, each process is viewed as a tuple $(Q, X, T, q_0) \in P$, where Q is the set of control states, X is the set of typed variables, T is the set of transitions and $q_0 \in Q$ is the initial state. Note that X includes both local and global variables.

Control states define the behavior of a process by means of actions, transitions to other states and, possibly, substates. There is not theoretical limit for the number of nested substates. State specifications include temporal constraints and the set of deferred signals. The initial state is defined by the keyword $\#start$ during the state definition. Because model checkers need to keep track of the current state being visited, our explicit-symbolic model uses a variable for storing such information. From now on, we refer to such variable as the program counter.

Transitions between states are controlled by condition guards. So, transitions can be viewed as process reactions in response to stimuli. Transition can be triggered by the activation of some untimed guard, activation of some timed guard or the presence of some signal in the input buffer of the process instance. When a condition guard is satisfied, some actions may be required before passing the control to another state. Actions include variable assignments, clock setting, clock and variable resetting, signal sending, procedure calls, creation and destruction of processes and signal routes. Transitions can also be used with a stop action, which means the destruction of the process instance. In a generic way, a condition guard g is defined as an expression over system variables, that is, $g \in 2^X$. Untimed guards are implemented through *provided* clauses, composed of constants, variables and Boolean, arithmetic and relational operators. Timed guards are implemented through *when* clauses, which control time constraints on clock variables. Finally, the presence of signals in the input buffer is implemented through *input* clauses. All signals share one queue per process instance and, therefore, an *input* clause gets blocked when

```
state q8;                            state busy;
...                                     input ack(c);
   provided c = b;                         nextstate q8;
   ...                                  when t = 1;
      nextstate idle;                      ...
...                                        nextstate busy;
endstate;                            endstate;

            (a)                                      (b)
```

Fig. 2. Condition guards.

the expected message is in the buffer but it is not the message in the head of the queue. In order to model queues, our explicit-symbolic model adds other variables to X, include those for controlling reading and writing positions. Ultimately, reading and writing position are used to determine success or failure on *input* clauses. In figure 2, some guards and transitions are shown. The *provided* clause, figure 2(a), establishes the condition for the execution of the transition from state $q8$ to state *idle*. Similarly, the *input* and *when* clauses, figure 2(b), establish the conditions for the execution of transitions from state *busy* to states $q8$ and *busy*, respectively. Note that there can be some statements between the guard condition and the transition itself.

Because more than one transition can be enabled at some control state, and all the situations have to be considered at execution, process may have non-deterministic behavior. Execution of individual processes is interleaved.

4.2 Generation of Explicit and Symbolic Models

Let $M = (D, S, P)$ be a system specified in the Intermediate Format. Two independent models, $M_e = (D, S, P_e)$ and $M_s = (D, S, P_s)$, must be generated according to the partitioning of M variables into explicit and symbolic partitions, X_e and X_s, respectively. Considering each process $(Q, X, T, q_0) \in P$, the variable partitioning induces the explicit process $(Q_e, X_e, T_e, q_{0e}) \in P_e$ and the symbolic process $(Q_s, X_s, T_s, q_{0s}) \in P_s$. Given each process $(Q, X, T, q_0) \in P$, its control states and respective guards must be analyzed. Guards, actions and transitions performed over X_e variables are projected in the model M_e, while those over X_s variables are projected in the model M_s. Let $q \xrightarrow{g \ a} q' \in T$ be a transition of $(Q, X, T, q_0) \in P$, where $q \in Q$ is the source state, $g \in 2^X$ is a Boolean guard, a is an action, $q' \in Q$ is the target state. Such transition affects variables both in X_e and X_s, where those variables represent the program counter, condition guards and actions. Because both models are affected, the original transition induces one explicit transition and one symbolic transition:

$$q_e \xrightarrow{g_e \ a_e} q'_e \in T_e \quad \text{and} \quad q_s \xrightarrow{g_s \ a_s} q'_s \in T_s$$

such that $g_e = g|_{X_e}$, $a_e = a|_{X_e}$, q_e and $q'_e \in Q_e$, $g_s = g|_{X_s}$, $a_s = a|_{X_s}$, q_s e $q'_s \in Q_s$, where $Q_e = Q|_{X_e}$, $Q_s = Q|_{X_s}$. The initial states in M_e and M_s are those projected by the original initial state q_0, that is $q_{0e} = q_0|_{X_e}$ and $q_{0s} = q_0|_{X_s}$. Remember that such explicit and symbolic transitions must be associated to each other.

4.2.1   The Explicit Transition Relation

The explicit transition relation should be represented as an array of linked lists where one array entry corresponds to one state and the linked list defines to which states the current state can transition to:

$$T_e(Q_e, Q'_e) = (q_{e_1}, L_1), \ (q_{e_2}, L_2), ...$$

where $L_i = q'_{e_{i1}}, q'_{e_{i2}}, q'_{e_{i3}}, ...$ For example, this transition relation defines transitions between states $(q_{e_1} \rightarrow q'_{e_{11}})$, $(q_{e_1} \rightarrow q'_{e_{12}})$ and so on.

4.2.2   The Symbolic Transition Relation

The symbolic transition relation should be represented as a formula between propositions in the current and in the next state, $T_s(Q_s, Q'_s)$, to be implemented using BDDs.

4.3   Generation of the Combined Explicit-Symbolic Model

The partitioning of system variables induces the generation of distinct explicit and symbolic models. Each generated model covers just a subset of variables and investigates only part of the search space. Such models have to be executed in an interleaved fashion in order to allow us to explore the whole search space and emulate the behavior of the original system. Due to efficiency questions, we propose a new way of associating transitions. Instead of using the transition labels defined in the modelling, the explicit and symbolic models are interleaved by associating symbolic transitions to explicit transitions. Given $M_e = (D, S, P_e)$ and $M_s = (D, S, P_s)$, the explicit-symbolic model $M_h = (D, S, P_h)$ is obtained as shown below. For each $(Q_h, X_h, T_h, q_{0h}) \in P_h$, we combine explicit and symbolic representations by introducing a symbolic transition for each explicit transition:

$$T_h(Q_h, Q'_h) = (q_{e_1}, L_1), \ (q_{e_2}, L_2), ...$$

where $L_i = (q'_{e_{i1}}, T_{s_{i1}}), (q'_{e_{i2}}, T_{s_{i2}}), (q'_{e_{i3}}, T_{s_{i3}}), ...$ and $T_{s_{ij}} = T_{s_{ij}}(Q_s, Q'_s)$. Graphically each $(q_{e_i}, L_i) \in T_h(Q_h, Q'_h)$ can be represented as shown in figure 3. According to this explicit-symbolic model, first we explore states in the explicit model and then states in the symbolic model. Intuitively $T_{s_{ij}}$ represents the

symbolic transition from q_{s_i} to $q'_{s_{ij}}$ associated to the explicit transition from q_{e_i} to $q'_{e_{ij}}$. Such transitions define the explicit-symbolic transition from (q_{e_i}, q_{s_i}) to $(q'_{e_{ij}}, q'_{s_{ij}})$, both in Q_h. So, Q_h corresponds to the set of pairs (q_e, q_s) associated by the explicit-symbolic transition relation T_h, where $q_e \in Q_e$, $q_s \in Q_s$. Consequently, $X_h = X_e \cup X_s$ and $q_{0h} = (q_{0e}, q_{0s})$.
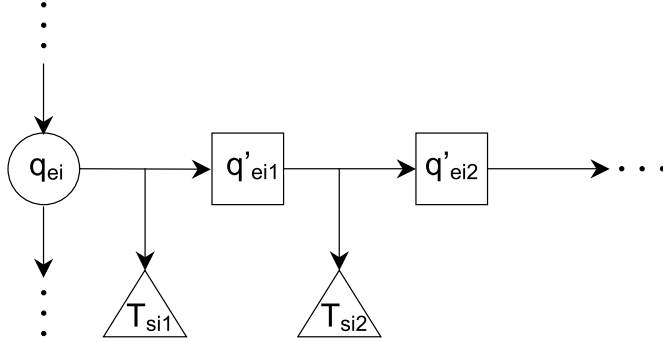

Fig. 3. Explicit-Symbolic Transitions

The explicit and symbolic models can be obtained from the explicit-symbolic model in a straightforward way. Because each $q_h \in Q_h$ corresponds to one pair (q_e, q_s), where $q_e \in Q_e$ and $q_s \in Q_s$, one explicit-symbolic state is associated to exactly one explicit and one symbolic state. Regarding the transition relation $T_h(Q_h, Q'_h) = (q_{e_1}, L_1), (q_{e_2}, L_2), ..., (q_{e_n}, L_n)$, each pair (q_{e_i}, L_i) corresponds to explicit transitions from q_{e_i} to every explicit state $q'_{e_{ij}}$ in the pairs of L_i, where $L_i = (q'_{e_{i1}}, T_{s_{i1}}), (q'_{e_{i2}}, T_{s_{i2}}), (q'_{e_{i3}}, T_{s_{i3}}), ...$ as above. Similarly, the symbolic transition relation can be obtained by $\cup T_{s_{ij}}$, where $(q'_{e_{ij}}, T_{s_{ij}}) \in L_i$.

5   Model-Checking Algorithms

Our algorithms assume the system properties are given in Computation Tree Logic, CTL for short. CTL is an important branching temporal logic with a discrete notion of time. CTL is sufficiently expressive for the formulation of an important set of system properties, allowing safety, liveness, fairness and deadlock freedom to be specified. For details regarding the semantics of CTL, please refer to [10].

Let Φ be a CTL specification over a model M. According to the definition, Φ is either given by an atomic proposition ap or it is composed of temporal-logical operators applied to CTL subformulae:

$$\Phi ::= False \mid True \mid ap \mid (\neg\Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid (\Phi \rightarrow \Phi) \mid$$
$$AX\Phi \mid EX\Phi \mid A[\Phi U\Phi] \mid E[\Phi U\Phi] \mid AG\Phi \mid EG\Phi \mid AF\Phi \mid EF\Phi$$

where the new connectives AX, EX, AU, EU, AG, EG, AF and EF are called temporal connectives. Assume that Φ is given in a structural representation by means of a parsing tree. In such a parsing tree, the leaves stand for atomic propositions, represented either in the explicit model or in the symbolic model, while internal nodes stand for explicit-symbolic expressions, represented in the explicit-symbolic model. First, algorithms convert both explicit and symbolic states into corresponding explicit-symbolic states. After that, algorithms handle sets of input and output explicit-symbolic states.

5.1   Atomic Propositions

During the parsing of atomic propositions, we must consider two different situations. First, the atomic proposition ap can be explicit, being represented in an explicitly coded state. Second, ap can be symbolic, being represented by a BDD. Independently of the situation, inputs must produce a set of explicit-symbolic states S.

5.1.1   Explicit Propositions

Consider the proposition $ap \in X_e$. First, the algorithm determines the set of explicit states E where ap holds. Next, for each $q_{e_i} \in E$ it determines symbolic states, represented by q_{s_i}, that compose a valid explicit-symbolic state (q_{e_i}, q_{s_i}).

```
01  S ← ∅
02  E ← {q_e ∈ Q_e|q_e ⊨ ap}
03  for each q_{e_i} ∈ E do
04   for each T_{s_{ij}} where (q_{e_i}, L_i) ∈ T_h and (q_{e_j}, T_{s_{ij}}) ∈ L_i do
05      add (q_{e_i}, q_{s_i}) to S where T_{s_{ij}} = q_{s_i} ∧ q'_{s_j}
```

5.1.2   Symbolic Propositions

Consider the proposition $ap \in X_s$. First, the algorithm determines the BDD q_{s_i} that represents the set of symbolic states where ap holds. After that, it visits the explicit transition relation looking for symbolic transitions $T_{s_{ij}} = q_{s_i} \wedge q'_{s_j}$, for some $q'_{s_j} \in Q_s$. For each $T_{s_{ij}}$ found, the algorithm creates an explicit-symbolic state (q_{e_i}, q_{s_i}), where (q_{e_i}, q_{e_j}) is the explicit transition associated to $T_{s_{ij}}$.

```
01  S ← ∅
02  q_{s_i} ←Q_s|ap
03  for each q_{e_i} ∈ Q_e do
04   for each T_{s_{ij}} where (q_{e_i}, L_i) ∈ T_h and (q_{e_j}, T_{s_{ij}}) ∈ L_i do
05      if (T_{s_{ij}} ∧ q_{s_i}) ≠ false then
06        begin
07           add (q_{e_i}, q_{s_i}) to S
```

```
08          break
09      end
```

Note that the algorithms above only include an explicit-symbolic state (q_{e_i}, q_{s_i}) into results if q_{e_i} and q_{s_i} hold simultaneously in the explicit and symbolic models, respectively. Specifically, the inner loops ensure that states q_{e_i} and q_{s_i} are associated to each other. Smaller sets of states require less memory and improve the performance of the search. Additionally, as many symbolic variables as possible must be explored before passing to the exploration of explicit variables in order to reduce the cost of combining the explicit and the symbolic models. In other words, we should explore each underlying model as much as possible before exploring the counterpart model.

5.2 Explicit-Symbolic Expressions

The following algorithms assume that initial expressions have been used to produce explicit-symbolic states. Each algorithm takes sets of input explicit-symbolic states I_1 and I_2 and produces another set of explicit-symbolic states S. Because temporal-logical operators $\neg$, $\vee$, EX, EU and EG can be used to define the remaining CTL operators, we restrict our discussion to such basic operators.

5.2.1 Negation

Given an expression φ represented by the set of input states I_1, the algorithm produces the set of output explicit-symbolic states S where $\neg\varphi$ holds. For each explicit state $q_{e_i} \in Q_e$, if (q_{e_i}, q_{s_i}) is in I_1 for some $q_{s_i} \in Q_s$, the algorithm produces the complement for the symbolic state, $\neg q_{s_i}$, and checks if it composes a valid pair with the explicit state q_{e_i}. If so, the pair $(q_{e_i}, \neg q_{s_i})$ is added into S. On the other hand, if (q_{e_i}, q_{s_i}) is not in I_1 for some $q_{s_i} \in Q_s$, the algorithm adds all the valid pairs (q_{e_i}, q_{s_i}) to S. In other words, the algorithm replaces the current set of valid explicit-symbolic states by its complement.

```
01 S ← ∅
02 for each q_eᵢ ∈ Q_e do
03   if (q_eᵢ, q_sᵢ) is in I₁ for some q_sᵢ ∈ Q_s then
04     begin
05       if (T_sᵢⱼ ∧ ¬q_sᵢ) ≠ false and (q_eᵢ, ¬q_sᵢ) ∉ I₁ then
06         add (q_eᵢ, ¬q_sᵢ) to S where (q_eᵢ, Lᵢ) ∈ T_h, (q_eⱼ, T_sᵢⱼ) ∈ Lᵢ
07     end
08   else
09     for each T_sᵢⱼ where (q_eᵢ, Lᵢ) ∈ T_h and (q_eⱼ, T_sᵢⱼ) ∈ Lᵢ do
10       add (q_eᵢ, q_sᵢ) to S where T_sᵢⱼ = q_sᵢ ∧ q'_sⱼ
```

5.2.2 Disjunction

Given that ψ and γ are represented by set of input explicit-symbolic states tables I_1 and I_2 respectively, the algorithm produces the set of output explicit-symbolic states S where $\psi \vee \gamma$ holds. First, the explicit-symbolic states in I_1 are stored in S. Next, explicit-symbolic states in I_2 are also copied to S. Explicit-symbolic states with shared explicit states are merged by performing the disjunction of their symbolic states.

```
01 S ← ∅
02 for each pair (q_e, q_s) of I_1 do
03    add (q_e, q_s) to S
04 for each pair (q_e, q_{s_i}) of I_2 do
05    if (q_e, q_{s_j}) ∈ S for some q_{s_j} ∈ Q_s then
06       replace (q_e, q_{s_j}) with (q_e, q_{s_j} ∨ q_{s_i}) in S
07    else
08       add (q_e, q_{s_i}) to S
```

5.2.3 $EX(\varphi)$

Given that φ is represented by the set of input explicit-symbolic states I_1, the algorithm produces the set of output explicit-symbolic states S where $EX(\varphi)$ holds. For each explicit-symbolic state (q_{e_j}, q_{s_j}) in I_1, first the algorithm computes the set of predecessors E for the explicit component q_{e_j}. After that, for each $q_{e_i} \in E$ it computes the predecessor q_{s_i} for the symbolic state q_{s_j}, restricted to $T_{s_{ij}}$, and adds (q_{e_i}, q_{s_i}) to S.

```
01 S ← ∅
02 for each pair (q_{e_j}, q_{s_j}) of I_1 do
03    begin
04       E ← {q_e | ∃(q_e, q_{e_j}) ∈ T_e}
05       for each q_{e_i} ∈ E do
06          begin
07             q_{s_i} ← EX_symb(T_{s_{ij}}, q_{s_j})
08             if (q_{s_i} ≠ false) then
09                begin
10                if (q_{e_i}, q_{s_k}) is in S for some q_{s_k} ∈ Q_s then
11                   replace (q_{e_i}, q_{s_k}) with (q_{e_i}, q_{s_k} ∨ q_{s_i}) in S
12                else
13                   add (q_{e_i}, q_{s_i}) to S
14                end
15          end
16    end
```

Note that $EX_{symb}(T_{s_{ij}}, q_{s_j})$ produces the predecessors of q_{s_j} over the symbolic transition $T_{s_{ij}}$, where $T_{s_{ij}}$ is the symbolic transition associated to the explicit transition from q_{e_i} to q_{e_j}. This restriction guarantees that the explicit transition (q_{e_i}, q_{e_j}) and the symbolic transition (q_{s_i}, q_{s_j}) occur simultaneously in the explicit and symbolic model, respectively.

5.2.4 $E(\psi\,U\,\gamma)$

Given that ψ and γ are represented by sets of input explicit-symbolic states I_1 and I_2 respectively, the algorithm produces the set of output explicit-symbolic states S where $E(\psi\,U\,\gamma)$ holds. The algorithm computes $E(\psi\,U\,\gamma)$ determining states where γ holds and looking backwards for states where ψ holds, until converging to the greatest set where $E(\psi\,U\,\gamma)$ holds.

```
01  S ← ∅
02  for each pair (qₑ,qₛ) of I₂ do
03     add (qₑ,qₛ) to S
04  do
05     continue ← false
06     Aux ← EX(S)
07     for each pair (qₑ,qₛ) of Aux do
08        if ((qₑ,qₛ) is not in S and (qₑ,qₛ) is in I₁) do
09           begin
10              add (qₑ,qₛ) to S
11              continue ← true
12           end
13  while (continue)
```

Because $E(\psi\,U\,\gamma)$ holds in states where γ holds, first the algorithm adds states of I_2 to S. After that, it computes the predecessors of states in S, named Aux, using the previously EX algorithm defined over explicit-symbolic states. States of Aux, included in I_1 and not yet in S, are added to S. The procedure finishes when no state is included in S in the loop spanning lines 06-11.

5.2.5 $EG(\varphi)$

The explicit-symbolic algorithm for $EG(\varphi)$ is adapted from the traditional algorithm for the explicit EG. The explicit $EG(\varphi)$ is computed over a modified state graph where all states at which φ does not hold are deleted and the transition relation is restricted accordingly. After that, the explicit version of the algorithm is based on the computation of strongly connected components (SCCs). In the explicit-symbolic version, the original state graph must be modified so that the relevant states are those where both the explicit and the symbolic components of the state hold. Because formulas are given in a structural representation being computed from elementary sub-formulas to the more complex ones in bottom-up fashion, the set of relevant states for the computation of EG over the explicit-symbolic model corresponds to the set I_1 of input explicit-states given by φ. So, initially we have to eliminate from the explicit-symbolic state graph all the states not found in I_1. In the explicit-symbolic state graph, the information about symbolic states is recorded by means of the symbolic transitions associated with explicit transitions, so the determination of the symbolic states require additional computation to existentially quantify out the next state variables from the symbolic transition.

The algorithm below is used for computing the set of output explicit-symbolic states S where $EG(\varphi)$ holds. Note that X'_s stands for the set of next state symbolic variables.

```
01 for each qe ∈ Qe do
02    if (qe,qs) ∉ I1 for all qs ∈ ∃X's(Ts), where (qe,L) ∈
Th,(q'e,Ts) ∈ L
03       eliminate qe from the state graph
04 SCC ← {C|C is a nontrivial SCC of T'h}
05 S ← ∪C∈SCC{(qe,qs)|(qe,L) ∈ T'h,(q'e,Ts) ∈ L,qs = ∃X's(Ts)}
06 S' ← ∅
07 while (S' ≠ S) do
08    begin
09       S' ← S
10       for each (qe,qs) where (qe,L) ∈ T'h,(q'e,Ts) ∈ L,qs =
∃X's(Ts) do
11          if (qe,qs) ∉ S then
12             add (qe,qs) to S
13    end
```

The loop spanning lines $01 - 03$ in the algorithm above is used to generate the modified state graph where φ holds, T'_h. For each explicit state q_e, the loop looks for symbolic transitions T_s associated with some explicit transition where q_e is the current state. Line 02 computes the set of current symbolic states where transitions of T_s come from, $\exists X'_s(T_s)$. If there exists some $q_s \in \exists X'_s(T_s)$ such that (q_e, q_s) belongs to I_1, q_e is maintained on the explicit-symbolic state graph. Otherwise, q_e is eliminated from the state graph. After that, line 04 computes the nontrivial strongly connected components considering only the explicit transitions of T'_h . Such a computation can be adapted from the algorithm of Tarjan [18], for example. Next, line 05 adds all the explicit-symbolic states belonging to the strongly connected components into the solution S. Finally, lines from 06 to 13 are used for finding all of those states that lead to states in S.

6 The Microwave Oven Model

In this section, we present an example in order to clarify the generation of the explicit-symbolic model and the application of related techniques and algorithms. Although we have mentioned the alternate bit protocol in our previous discussion about the modelling language, for didactic reasons we illustrate the model checking on a small example that describes the behavior of a microwave oven, taken from [11].

6.1   Decomposition of the Microwave Oven Model

Assume that the behavior of the microwave oven is modeled by the process (Q, X, T, q_0), given by the Kripke structure shown in figure 4, where the set of states Q is represented by ellipses, the set of variables X is represented by propositions shown within ellipses, that is $X = \{start, close, heat, error\}$, the transition relation T is represented by arcs and $q_o = S1$. For clarity, each state is labeled with both the atomic propositions that are true in the state and the negations of the propositions that are false in the state. Labels on the arcs indicate the actions that cause transitions but are not part of the Kripke structure.
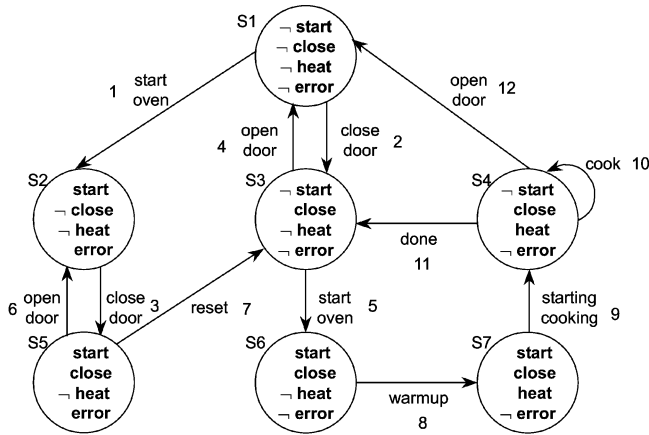

Fig. 4. The microwave oven model.

Suppose that the set X is decomposed into subsets of explicit and symbolic propositions $X_e = \{start, close\}$ and $X_s = \{heat, error\}$, respectively. Now, consider the set of states under an special point of view where only explicit propositions matters. Because states with the same set of explicit propositions are not distinct under such explicit point of view, such states are considered the same and are projected as one explicit state. Figure 5 shows the grouping of states considering explicit propositions (a) and the corresponding explicit process (Q_e, X_e, T_e, q_{0e}) generated (b). Note that we have another set of states, Q_e, and another set of propositions composed only of explicit propositions, X_e. The arcs on figure 5(b) correspond to the explicit transition relation T_e and have been labeled in order to reveal the original transitions they were projected from. Naturally, $q_{oe} = ES1$.

The symbolic process (Q_s, X_s, T_s, q_{0s}) is generated in a similar manner, by considering the original model under the symbolic point of view. Figure 6
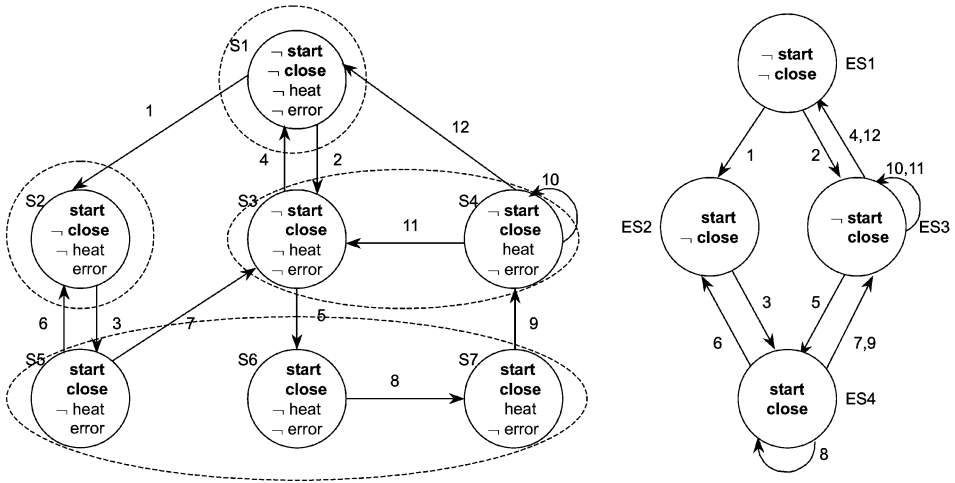
Fig. 5. Grouping of explicit states (a) and corresponding explicit model (b).

presents the grouping of symbolic states (a) and the corresponding symbolic process (Q_s, X_s, T_s, q_{0s})(b).
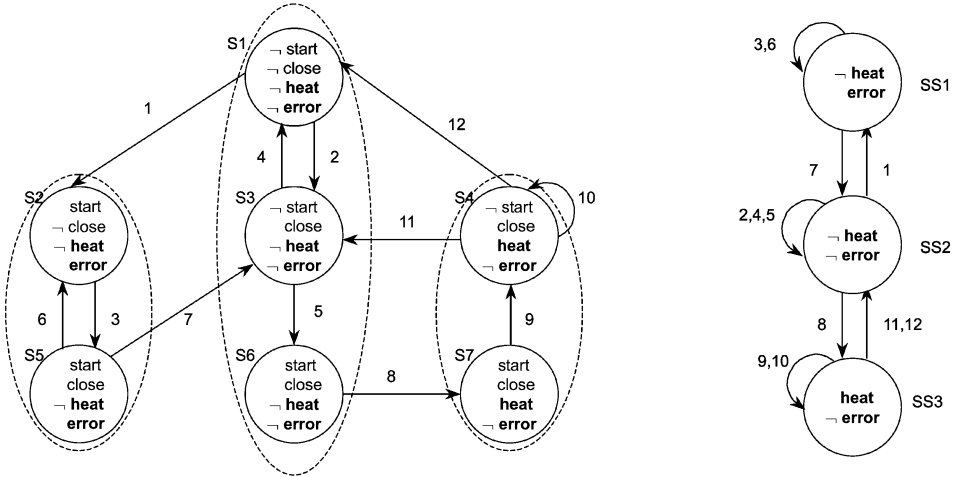
Fig. 6. Grouping of symbolic states (a) and corresponding symbolic model (b).

6.2 Verification on the Explicit-Symbolic Microwave Oven Model

Given the explicit and symbolic models shown in figures 5(b) and 6(b), respectively, explicit and symbolic transitions with labels in common should be linked together in order to establish their interdependence during the model checking. This subsection explores the operation of such combined model for

checking CTL expressions. The expression $EX(\neg close \vee error)$ is the expression of interest, because it allow us to explore some of the more important CTL operators. According to the previous partitioning, $close \in X_e$ and $error \in X_s$. Note that all the algorithms produce explicit-symbolic states as output.

Finding States Where *close* Holds

By using the algorithm for processing explicit atomic propositions, line 02, we generate the explicit states where *close* holds, $E = \{ES3, ES4\}$. The loop spanning lines 03-05 generates the set S of explicit-symbolic states where *close* holds, such that $S = \{(ES3, SS2), (ES3, SS3), (ES4, SS1), (ES4, SS2), (ES4, SS3)\}$.

Finding States Where $\neg close$ Holds

By using the negation algorithm over the set of explicit-symbolic states where *close* holds, specifically lines 04-07, we generate $S = \{(ES1, SS2), (ES2, SS1)\}$.

Finding States Where *error* Holds

By using the algorithm for processing symbolic atomic propositions, line 02, we have that q_{s_i} represents the set $\{SS1\}$. Lines 03-09 produce $S = \{(ES2, SS1), (ES4, SS1)\}$.

Finding States Where $\neg close \vee error$ Holds

By using the disjunction algorithm, lines 02-03, we add every state where $\neg close$ holds into the output set. After that, lines 04-08 include states where *error* holds into the output set, producing $S = \{(ES1, SS2), (ES2, SS1), (ES4, SS1)\}$.

Finding States Where $EX(\neg close \vee error)$ Holds

Given the input state set $I_1 = \{(ES1, SS2), (ES2, SS1), (ES4, SS1)\}$, consider that it is processed in the order presented. Each iteration of the outer loop of the EX algorithm considers a specific input explicit-symbolic state. For the first state, $(ES1, SS2)$, the algorithm initially computes the set of predecessors E_1 for its explicit component, $ES1$, by using line 04. After that, the loop spanning lines 05-15 computes predecessors of the symbolic component, $SS2$, by considering only symbolic transitions associated to relevant explicit transitions. The relevant explicit transitions are those from states of E_1 to

$ES1$. Each symbolic predecessor found and its corresponding explicit predecessor compose an explicit-symbolic state of S_1. Below, we show results from the first to the third and last iteration of the outer loop of the EX algorithm. The set of explicit-symbolic states of the third iteration, S_3, are those where $EX(\neg close \lor error)$ holds. Such explicit-symbolic states correspond to the original states $\{S4, S3, S1, S5, S2\}$, as expected.

$$E_1 = \{ES3\}$$
$$S_1 = \{(ES3, SS3), (ES3, SS2)\}.$$

$$E_2 = \{ES1, ES4\}$$
$$S_2 = \{(ES3, SS3), (ES3, SS2), (ES1, SS2), (ES4, SS1)\}.$$

$$E_3 = \{ES2, ES3, ES4\}$$
$$S_3 = \{(ES3, SS3), (ES3, SS2), (ES1, SS2), (ES4, SS1), (ES2, SS1)\}.$$

7 The Explicit-Symbolic Model Flexibility

Our explicit-symbolic model has important differences with regard to the SLAM project. Instead of dealing with a fixed approach, where control-flow and data-flow information must have explicit and symbolic representations, respectively, our approach is more general. The explicit-symbolic model allows us to move variables between explicit and symbolic spaces according to any policy. Thus, we can experiment with a variety of combinations and choose the one that best fits the system needs. Consider, for example, systems with data dependent control, that is, systems where the control-flow depends on evaluations of symbolically represented variables. In such cases, we have the opportunity of improving the overall performance by moving those symbolic variables to the explicit model, reducing interaction between explicit and symbolic representations. Naturally there are other questions involved, as the dependence between those symbolic variables and the remaining ones, but the proposed explicit-symbolic model can support such configuration whenever it is advantageous. As another example of the generality of the flexibility achieved, our explicit-symbolic model also supports the partitioning of variables accomplished by SLAM. Because the program counter and variables involved in control guards of IF systems determine control-flow information,

it suffices to move such variables to the explicit model in order to achieve a SLAM-like partitioning. Therefore, the explicit-symbolic model offers the possibility of using a more flexible environment to evaluate different representations and choose the one that improves the model-checking procedure.

8 Final Remarks

In this paper, we proposed a model that combines explicit and symbolic representations. The conceived explicit-symbolic model considers that explicit and symbolic techniques should be used in an integrated and synchronized fashion, allowing us to have a better exploration of the search space of the modeled systems. Thus, our main contribution is the proposal of a flexible environment for the formal verification of systems. Currently, our efforts are driven to the computational implementation of the explicit-symbolic model and its algorithms, considering systems specified in the Intermediate Format. Such implementation will help us to improve the conceptual model and will make it possible to measure the impact that different representations have over the verification of control and data-flow intensive systems. According to the literature, it seems to be a good choice to represent control-flow explicitly and data-flow symbolically, as done in SLAM. Future experiments will make it possible to have more information to answer this question.

References

[1] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[2] Thomas Ball and Sriram K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN*, pages 113–130, 2000.

[3] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[4] Marius Bozga, S. Graf, L. Ghirvu, L. Mounier, and J. Sifakis. The intermediate representation IF. VERIMAG Technical Report, 1998.

[5] G. Brat, K. Havelund, S. Park, and W. Visser. Java Pathfinder - A second generation of a Java model checker. In *Workshop on Advances in verification*, July 2000.

[6] R. E. Bryant. Graph-Based Algoritms for Boolean Function Manipulation. *IEEE Trans. on Computers*, pages 677–691, 1986.

[7] Sérgio Vale Aguiar Campos. *A Quantitative Approach to the Formal Verification of Real-Time Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, September 1996.

[8] William Chan, Richard Anderson, Paul Beame, and David Notkin. Combining constraint solving and symbolic model checking for a class of a systems with non-linear constraints. In *Computer Aided Verification*, pages 316–327, 1997.

 [9] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In *ACM Transactions on Programming Languages and Systems*, volume 8-2, pages 244–263, April 1986.

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[12] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[13] Gerard J. Holzmann. The Model Checker Spin. In *IEEE Transactions on Software Engineering*, volume 23, pages 279–295, May 1997.

[14] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Phd thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1992.

[15] K. L. McMillan. Symbolic Model Checking. Kluwer Academic Publ., 1993.

[16] David Notkin. Symbolic Model Checking for Large Software Specifications. Department of Computer Science and Engineering, University of Washington, 2001.

[17] R. Sebastiani. Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms, 2001.

[18] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.