Synthesis from multi-paradigm specifications

Ioannis Filippidis   Richard M. Murray   Gerard J. Holzmann

May 11, 2015

Abstract

This work proposes a language for describing reactive synthesis problems that integrates imperative and declarative elements. The semantics is defined in terms of two-player turn-based infinite games with full information. Currently, synthesis tools accept linear temporal logic (LTL) as input, but this description is less structured and does not facilitate the expression of sequential constraints. This motivates the use of a structured programming language to specify synthesis problems. Transition systems and guarded commands serve as imperative constructs, expressed in a syntax based on that of the modeling language PROMELA. The syntax allows defining which player controls data and control flow, and separating a program into assumptions and guarantees. These notions are necessary for input to game solvers. The integration of imperative and declarative paradigms allows using the paradigm that is most appropriate for expressing each requirement. The declarative part is expressed in the LTL fragment of generalized reactivity(1), which admits efficient synthesis algorithms. The implementation translates PROMELA to input for the SLUGS synthesizer and is written in PYTHON.

Keywords: Reactive synthesis, Generalized Reactivity(1), Linear temporal logic, Infinite games, PROMELA

Contents

Chapter 1

Introduction

Over the past three decades system verification has aided design and become practical for industrial application. In the past decade, synthesis of systems from specifications has seen significant development [1, 2], partially owing to the discovery of temporal logic fragments that admit efficient synthesis algorithms [3, 4, 5, 6]. Applications range from protocol synthesis for hardware circuits [4], to correct-by-construction controllers for hybrid systems [7, 8, 9].

Model checking has polynomial complexity in the size of a given Kripke structure, which has motivated the development of a large number of tools and languages for modeling systems. Unlike verification using model checking, the tools for synthesis have been developed much more recently. One reason is that centralized synthesis from linear temporal logic (LTL) has doubly exponential complexity in the length of the specification formula [10], a result that did not encourage further development initially.

As a result, currently LTL is the language used for describing specifications as input to synthesis tools. There are many benefits in using a logic for synthesis tasks, its declarative nature being a major one, because it allows expressing separately individual requirements in a precise way. It also makes explicit the implicit conventions present in programming languages [11]. Another aspect of synthesis problems that makes declarative descriptions appropriate is that we want to describe as large a set of possible designs as possible, in order to avoid overconstraining the search space.

Paradigms However, not all specifications are best described declaratively. There exist synthesis problems whose description involves graph-like structures that are cumbersome for humans to write in logic. For example, robotics problems typically involve graph constraints that originate from possible physical configurations. Properties that specify sequential behavior also lead to graph-like structures, and require use of auxiliary variables that serve as memory. Expressing sequential composition in logic leads to long, unstructured formulas that deemphasize the specifier's intent. The resulting specifications are difficult to maintain, and writing them is error-prone. In addition, the specifier may need to explicitly write clauses that constrain variables to remain unchanged, in order to maintain imperative state. This leads to longer formulas in which the intent behind individual clauses is less readable.

Borel hierarchy Another motivation stems from the temporal logic, or Borel, hierarchy [12, 13]. The more expressive a syntactic fragment of logic is, the more computationally expensive it is to construct a model, or prove that none exists.

Currently, algorithms of time complexity polynomial in the size of the state space are known for the fragment of generalized reactivity of rank one (one Streett pair), known as GR(1). In the automata hierarchy, the GR(1) fragment corresponds to deterministic Büchi automata [3, 13, 14, 15]. From a game theoretic perspective, deterministic automata describe the winning condition in such a way, that existential branching during synthesis commits to a future strategy, without the ability to recall it.

A large subset of properties that are of practical interest in industrial applications [16, 17, 4] can be expressed in the GR(1) fragment. There do exist properties that cannot be represented by deterministic Büchi automata, e.g., persistence $\Diamond\Box p$ (eventually p holds uninterruptedly). Of these properties, those with Rabin rank equal to one are still amenable to polynomial time algorithms, by solving parity games with a limited number of colors [5]. Higher Rabin ranks are not expected to admit solution by algorithms of polynomial complexity, unless P = NP [5]. This motivates formulating the required properties in GR(1).

Translating properties to deterministic automata can be done automatically, but may lead to more states than manually written properties [14]. So the ability to write deterministic automata directly in a structured and readable language avoids the need for conversions, either automatic, or manual.

Synthesis tools have been developed for the GR(1) fragment, so temporal logic can be viewed as a lower level language above the representation used inside a solver, e.g., binary decision diagrams [18]. In this context, one can think of temporal logic as an analog of assembly language. It follows that by translating a problem description to logic, it can then be given as input to a suitable synthesis tool.

Modularization Another reason why specifications are not always purely declarative is that in many cases we want to synthesize a system using *existing* components. In other words, we already have a *partial* model of reality, and we *declare* to our synthesis tool what properties the controller under design should satisfy with respect to this model. So the partial model is best described imperatively, whereas the goal declaratively, using temporal logic.

Other Educationally, the transition for engineering students from a general purpose programming language, like Python or `C`, directly to temporal logic constitutes a significant leap. Using a multiparadigm language can make this transition smoother.

The preceding arguments on the need for a more structured specification language that spans paradigms do not allude as to whether `Promela` or some other syntactic variant is more appropriate for this purpose. The selection of syntax will be discussed throughout, and the results can be adapted to any suitable syntax.

Contents This work proposes a language that can describe synthesis problems for open systems that react to an adversarial environment. The syntax is derived from that of Promela, whereas the semantics interprets it as a two-player turn-based game of infinite duration. Both synchronous and asynchronously scheduled centralized systems with full information can be synthesized.

In Section 1.1, we review temporal logic and relevant notions about two-player games. The semantics of the language is defined in Chapter 2. The presence of two players requires declaring who controls each variable (Section 2.1), as well as the data flow, and control flow in transition systems (Section 2.2). In addition, the specification needs to be partitioned into assumptions about the environment, and guarantees that the system must satisfy (Section 1.1, Section 2.2). The integration of declarative and imperative semantics is obtained by defining imperative variables (Section 2.1), deconstraining, and executability of actions (Section 2.3.4). In order to be synthesized, the program is translated to temporal logic, as described in Chapter 3. Relevant work is collected in Appendix A.2.

Acknowledgments This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA. The first author was partially supported by a graduate research fellowship from the Jet Propulsion Laboratory, over the summer of 2014. The authors would like to thank Scott Livingston for providing helpful comments on an early draft of this document.

1.1 Preliminaries

1.1.1 Linear Temporal Logic

Linear temporal logic with past is an extension of Boolean logic used to reason about temporal modalities over sequences. The temporal operators “next” $\bigcirc$ and “until” $\mathcal{U}$ suffice to define the other operators [19, 20]. Let AP be a set of variable symbols p that can take values over $\mathbb{B} \triangleq \{\bot, \top\}$. A model of an LTL formula is a sequence of variable valuations called a *word* $w : \mathbb{N} \to \mathbb{B}^{AP}$. A well-formed formula is inductively defined by $\varphi ::= p|\neg\varphi|p \wedge p|\bigcirc\varphi|\varphi\mathcal{U}\varphi|\ominus\varphi|\varphi\mathcal{S}\varphi$. A formula φ is evaluated over a word w at a time $i \geq 0$, and $w, i \models \varphi$ denotes that φ holds at position i of word w. Formula $\bigcirc\varphi$ holds at position i if φ holds at position $j + 1$, $\varphi\mathcal{U}\psi$ holds at i if there exists a time $j \geq 0$ such that $w, j \models \psi$ and for all $0 \leq k < j$, it is $w, k \models \varphi$. The operator $\Diamond p \triangleq \text{true}\,\mathcal{U}p$ requires that p be "eventually" true, and the operator $\square p \triangleq \neg\Diamond\neg p$ requires that p be true over the whole word. The past fragment of LTL extends it with the "previous" and "since" operators, $\ominus, \mathcal{S}$ respectively [21, 22, 23]. Formula $\ominus\varphi$ holds at i iff $i > 0$ and $w, i-1 \models \varphi$, and formula $\varphi\mathcal{S}\psi$ holds at i iff there exists a time j with $0 \leq j \leq i$ such that $w, j \models \psi$, and for all k such that $j < k \leq i$ it is $w, k \models \varphi$. The *weak previous* operator $\circledtilde$ is defined as $\circledtilde\varphi \triangleq \neg\ominus\neg\varphi$, "once" $\diamondminus$ as $\diamondminus\varphi \triangleq \top\mathcal{S}\varphi$, and "historically" $\boxminus$ as $\boxminus\varphi \triangleq \neg\diamondminus\neg\varphi$.

1.1.2 Turn-based games

In many applications, we are interested in designing a system that does not have full control over its reality. Some problem variables represent the behavior of another entity, usually referred to as the "environment". The system reads these *input* variables and reacts by writing to *output* variables that it controls, continuing indefinitely. Such a system is called *open* [24, 25], to distinguish it from closed systems that have no inputs, and so full control.

The synthesis of an open system can be formulated as a two-player adversarial game of infinite duration [26, 27, 28]. The two players in the game are usually called the protagonist (system) and antagonist (environment). We control the protagonist, but not the antagonist. If the players move in turns, then the game is called *alternating*. Each pair of consecutive moves by the two players is called a *turn* of the game. In each turn, player 0 moves first, without knowing how player 1 will choose to move in that turn of the game. Then player 1 moves, knowing how player 0 moved in that turn.

Depending on which player we control, there are two types of game. If the protagonist is player 1, then the game is called *Mealy*, otherwise *Moore* [29, 30]. Due to the difference in knowledge about the opponent's next move between the two flavors of game, more specifications are realizable in a Mealy game, than in a Moore game. There exist solvers for both Moore and Mealy games. Here we will consider Mealy games only, but the results can be adapted to Moore games as well.

1.1.3 Games in logic

Temporal logic can be used to describe both the possible moves in a game (the *arena* or *game graph*), as well as the winning condition (e.g., fairness). Let $\mathcal{X}$ and $\mathcal{Y}$ be two sets of propositional variables, controlled by the environment and system, respectively. Let $\mathcal{X}'$ and $\mathcal{Y}'$ denote primed variables, where x' represents the next value $\bigcirc x$ of variable x. We abuse notation by using primed variables inside temporal formulae.

Synthesis from LTL specifications is in 2ExpTime [31, 10], motivating the search for fragments that admit more efficient synthesis algorithms. Generalized reactivity of index one, abbreviated as GR(1), is a fragment of LTL that admits synthesis algorithms of time complexity polynomial in the size of the state space [4]. We use GR(1) in the following [32, 33, 34, 35, 36], but the results can be adapted to larger fragments of LTL, provided that another synthesizer be used [37, 38, 39, 40, 41].

The possible moves in a Mealy game can be specified by initial and transition conditions that constrain the environment and system. Initial conditions are described by propositional formulæover $\mathcal{X} \cup \mathcal{Y}$. Transition conditions are described by safety formulae of the form $\square\varphi_i$ where, for the environment $i = e$ and φ is a formula over $\mathcal{X} \cup \mathcal{X}' \cup \mathcal{Y}$, and for the system $i = s$ and φ is a formula over $\mathcal{X} \cup \mathcal{X}' \cup \mathcal{Y} \cup \mathcal{Y}'$. Note that the system plays second in each turn, so it can see $\mathcal{X}'$, whereas the environment cannot see $\mathcal{Y}'$, because it represents future values. The winning condition in a GR(1) game is described using progress formulae of the form $\square\Diamond\psi_i, i \in \{e, s\}$, where ϕ_i is a propositional formulae over $\mathcal{X} \cup \mathcal{Y}$.

The overall specification of a GR(1) game is of the form

$$\underbrace{\Big(\bigwedge_{i=0}^{n_0} \theta_{e,i} \wedge \bigwedge_{i=0}^{n_1} \square\varphi_{e,i} \wedge \bigwedge_{i=0}^{n_2} \square\Diamond\psi_{e,i}\Big)}_{assumption} \rightarrow \underbrace{\Big(\bigwedge_{i=0}^{m_0} \theta_{s,i} \wedge \bigwedge_{i=0}^{m_1} \square\varphi_{s,i} \wedge \bigwedge_{i=0}^{m_2} \square\Diamond\psi_{s,i}\Big)}_{assertion} \tag{1.1}$$

Note that requirements that constraint the environment are called *assumptions* and guarantees that the system must satisfy are called *assertions*. Assumptions limit the set of admissible environments, because, in practice, it is impossible to satisfy the design requirements in arbitrarily adversarial environments [24]. The implication above is interpreted by prioritizing between safety and liveness, to prevent the system from violating its safety assertion in case this would allow it to prevent the environment from satisfying its liveness assumption.

The algorithm for GR(1) synthesis proposed in [3] has complexity $O\left((nm\left|\Sigma\right|)^3\right)$ and the one later proposed in [4] has complexity $O(nm\left|\Sigma\right|^2)$, where n is the size of the assumption formula, m the size of the assertion formula, and Σ the state space (all possible valuations of variables), with size $|\Sigma|$ is exponential in the number of variables.

Chapter 2

Language definition

The language we are about to define is syntactically an extension of PROMELA [42], but its semantics is defined by a translation to turn-based infinite games with full information. PROMELA is a guarded command language that can represent transition systems, non-deterministic execution, and guard conditions for determining whether statements are executable [42, 43]. Its syntax can be found in the language reference manual [42, 44]. Here we will introduce syntactic elements only as needed for the presentation. Briefly, we mention that a program comprises of transition systems and automata, whose control flow can be described with sequential composition, selection and iteration statements, `goto`, as well as blocks that group statements for atomic execution.

2.1 Variables

2.1.1 Ownership

In a modeling language for closed systems, one player controls all the variables. In contrast, in a game there are two players. Let $\mathcal{X}$ denote the variables controlled by the environment, and $\mathcal{Y}$ those controlled by the system. We call *owner* of a variable the player that controls it. We use the keywords `env` and `sys` to signify the owner of a variable. Variables can be of Boolean, bit, byte, (bounded) integer, or bitfield type. These types are discussed more in Sections 2.1.3 and 4.2.1.

So there are two flavors of variables, depending on who decides their values:

1. controlled variables (system variables, also known as outputs, or output ports)
2. uncontrolled variables (environment variables, also known as inputs, or input ports)

In the global context (i.e., outside `proctype` declarations), the default owner is the system. Let V_p denote the set of program variables declared as controlled by player $p \in \{e, s\}$, for environment and system, respectively. The set V_e is a subset of $\mathcal{X}$, because it does not include auxiliary variables that may be introduced by the compiler as described in Chapter 3. From a first-order logic viewpoint, ownership defines quantification [25]. System variables are existentially quantified, whereas environment variables universally.

2.1.2 Declarative and imperative semantics

In imperative languages, variables remain unchanged, unless explicitly assigned new values. In declarative languages, variables are free to change, unless explicitly constrained [45]. In verification, both declarative languages like TLA [46] and SMV [47] have been used, as well as imperative languages like PROMELA and DVE [48].

In a synthesis problem, there are variables whose behavior can more succinctly be described declaratively, whereas others imperatively. For this reason, we combine the two paradigms, by introducing a new keyword `free` to distinguish between imperative and declarative variables. Variables whose declaration includes the keyword `free` are by default allowed to be assigned any value in their domain, unless explicitly constrained otherwise. Variables without the keyword `free` have imperative semantics, so their value remains unchanged, unless otherwise explicitly stated. Let V^{free} denote free, and V^{imp} the imperative program variables, respectively.

2.1.3 Ranged integer data type

Symbolic methods for synthesis use binary decision diagrams (BDDs) [18, 20]. Reordering variables can be expensive, so reducing the number of bits is a primary objective, because each extra propositional variable of the problem introduces two BDD variables (a primed and an unprimed one). This effect has a milder impact on enumerated methods, because it increases the memory required to store each state, but leaves unaffected the number of reachable states. But for BDDs, the number of BDD nodes is sensitive to the number, and ordering, of variables.

In addition, the complexity of GR(1) synthesis is polynomial in the number $|\Sigma|$ of variable valuations (Section 1.1.3), which grows exponentially with each additional bit. We can reduce the number of bits by using bitfields whose width is tailored to the problem at hand, together with safety constraints. For example, a variable x that can take values in the set $\{0, 1, 2\}$ is represented in logic by two propositional variables $x_0, x_1 \in \mathbb{B}$ and the constraint

$$\Box(0 \leq x \leq 2) \iff \Box\left((x_0 \neq 1) \vee (x_1 \neq 1)\right),$$

which prevents the valuation $x_0 = 1, x_1 = 1$, i.e., $x = 3$.

For convenience, the *ranged* integer type `int(MIN, MAX)` is introduced to define a variable

$$x \in \{\text{MIN}, \text{MIN} + 1, \ldots, \text{MAX}\}.$$

This syntax is a slight extension of the `C`-like syntax of pure `Promela`. Ranged integer variables have saturating semantics [49]. Safety constraints are automatically imposed on the bitfield representing a ranged integer. Other numerical data types (bits, bytes, bitfields) have mod wrap semantics (modular arithmetic).

2.1.4 Notation

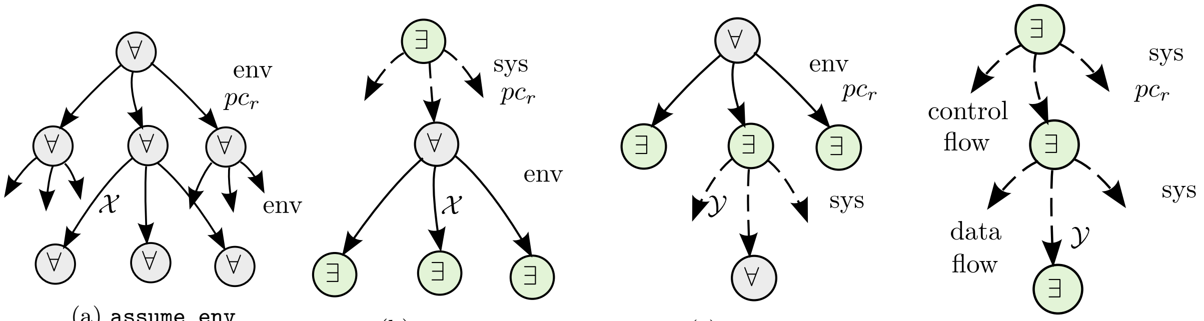
Summarizing the notation defined earlier:

- $\mathcal{X}$ are all the variables controlled by the system,
- $\mathcal{Y}$ are all the variables controlled by the environment,
- V is the set of all variables declared in a program source listing,
- $V_s \subseteq V$ are the program variables controlled by the system,
- $V_e \triangleq V \setminus V_s$ are the program variables controlled by the environment,
- V^{free} are the declarative program variables, and
- $V^{\text{imp}} \triangleq V \setminus V^{\text{free}}$ are the imperative program variables.

For a set of variables V, V' denotes the set of primed variables u', for each variable $u \in V$. The set $\text{Var}(\varphi)$ denotes the variables that appear in formula φ. Combinations of subscripts denote intersection, for example $V_s^{\text{imp}} = V^{\text{imp}} \cap V_s$ is the set of imperative system variables.

2.2 Game graphs as programs

In many synthesis problems, the specification includes graph-like constraints. These may originate from physical configurations in robotics problems, deterministic automata to express a formula in GR(1), or describe abstractions of existing components that are to be controlled. Program graphs can be used to describe graph-like constraints. A *program graph* is a rooted directed multi-graph $P_r \triangleq (V_r, E_r, u)$ whose



(a) `assume env`

(b) `assume sys`

(c) `assert env`

(d) `assert sys`

Figure 2.1: An assumption (assertion) process constrains the environment (system) variables, and `env` (`sys`) declares who chooses the next statement to be executed (when there are multiple).

edges $E_r \subset V_r \times V_r \times \mathbb{N}$ are labeled with program statements, and nodes V_r abstract states of the system [50, 20]. Execution starts from the graph's root $u \in V_r$.

The notions of *control flow* and *data flow* are relevant to defining the behavior of a program graph. Control flow is the traversal of edges, whereas data flow is the behavior of program variables along this traversal. A *program counter* variable pc_r that stores the current node in V_r suffices to define control flow (together with a key variable to distinguish edges with identical source and target nodes). The environment (system) controls the flow of variables in $\mathcal{X}$ ($\mathcal{Y}$).

A natural question to ask is who controls the program counter. Another question is whose data flow is constrained by the program graph P_r. If the same player controls the program counter and is constrained by P_r, then the program graph describes an automaton for the behavior of that player only. If one player controls the program counter, and the opponent reacts by choosing a compliant data flow, then the program graph describes a game.

As an example, suppose that the environment controls the program counter, and the system the data flow. At each node, the environment can pick any successor node, and the system must react by selecting a data flow compatible with the program statement that labels the edge that the environment picked. So *paths* in this program graph are *universally* quantified, whereas data flow is *existentially* quantified. This case is schematically depicted in Fig. 2.1c, and can be used to represent a verification problem Section 2.7.

The notion of path quantification corresponds to universal and existential nodes in alternating tree automata [51, 52, 53, 54, 55], although the program graphs presented here differ in how edges are labeled. If paths are universally quantified, then control flow nondeterminism is known as *demonic*, p.85 [56], [57], otherwise as *angelic* [58, 59, 60] The execution of such a program graph induces a *game graph* [61, 28].

2.2.1 Syntax

Program graphs are declared with the `proctype` keyword of `Promela` followed by statements enclosed in braces. We use the keyword `assume` (`assert`) to declare that a program graph constrains the environment's (system's) data flow. These keywords are common in theorem proving and program verification languages [62]. We use the keyword `env` (`sys`) to declare that the environment (system) controls the program counter pc_r. If the keywords are omitted, the control and data flow context are controlled by the system. We will call program graphs *processes*, noting that these processes have full information about each other, so they correspond to centralized synthesis, not distributed. The program counter *owner* is the player that controls variable pc_r. The *process player* is the player constrained by the program graph.

The example of Fig. 2.1c corresponds to

```
assert env proctype foo(){
    bit x;
    env bit y;
    ...
}
```

The default owner of variables declared in a `proctype` context is the player that controls the data flow. So in the previous code example, the variable `x` is controlled by the system, because of the keyword `assert`. The owner of local environment variables in an assertion needs to be explicitly declared, as shown for variable y.

2.2.2 Example

For example, the specification in Listing 2.1 defines a game between two players: the Bunny, and Taz, that move in turns. Each logic time step includes a move by Taz from (x_t, y_t) to (x'_t, y'_t), followed by a move by the Bunny from (x, y) to (x', y'). The Bunny must reach the carrot, without moving through a cell that Taz is in (`assert ltl`). Taz can only move between $x_t \in \{1, 2\}$, and has to keep visiting the lower row. Taz can move diagonally, but the Bunny only vertically or horizontally. Both players have an option to stay still (`skip`). Note that x_t is a declarative variable, so it can change unless constrained.

The process `taz` constrains the environment variables x_t, y_t (`assume`) and the environment controls its program counter (`env`). Similarly for `bunny` that is controlled by, and constrains the system. The `do` loops define alternatives that each player must choose from to continue playing the game.

Note how nondeterminism in process `taz` is demonic (universally quantified), whereas in `bunny` angelic (existentially quantified), i.e., the design freedom given to the synthesis tool. Each player has full information about all variables in the game, both local, as well as global. The solution is a strategy represented as a Mealy transducer [29] that the Bunny can use to win the game.

Note that the specification corresponds to an alternating time interpretation, i.e., that the two players move in turns (like chess). The subformula `(x == xt) && (y == yt)` ensures that the Bunny and Taz are not in the same cell at the same turn of the game. It does *not* specify that the Bunny should avoid Taz moving to its cell in the *next* turn of the game.

Instead, that requirement is ensured by the conjuct

$$\varphi \triangleq \Box \circledsim \neg((x = x'_t) \wedge (y = y'_t)),$$

which prevents the Bunny from moving next to Taz, from where Taz can catch it in the next turn. Note that `-X` and `--X` are the weak and strong "previous" operators, respectively. Using the strong "previous", this is equivalent to

$$\varphi = \Box \neg \ominus ((x = x'_t) \wedge (y = y'_t))$$

(commented in code).

A naive first attempt could have been $\Box \neg((x = x'_t) \wedge (y = y'_t))$. However, during solution of the Mealy game, this leads to a controllable predecessor operation [4, 28] of the form $\forall x'_t, y'_t, \ldots \exists x', y', \ldots .(x = x'_t) \wedge (y = y'_t)$. This is false, because variables x, y are not quantified (fixed already in the previous time step). Instead, we apply the axiom

$$\Box p = \Box \circledsim p \tag{2.1}$$

P4, p.58 [22]. This shifts the expression $\neg((x = x'_t) \wedge (y = y'_t))$ to the past, and yields the equivalent formula φ, which is suitable for attractor computations. Past LTL is implemented by a translation using temporal testers [23].

2.3 Expressions

In declarative languages like TLA, Smv, and the input syntax of GR(1) synthesis tools, the temporal operator "next" can be used to refer to values that variables will take in the next time step. Similarly, open `Promela` expressions can contain variable values at the next time step, extending the expressions of pure `Promela`. In temporal logic, the value of a variable x at the next time step is denoted by[1] $\bigcirc x$. In open `Promela` syntax, the value $\bigcirc x$ is denoted by priming the variable as `x'`. The prime is used as syntax for the "next" operator in `TLA` and for input to several synthesis tools. For readability, in the following we use primed variables also inside equations, so x' denotes $\bigcirc x$.

---

[1] There are two "next" operators, a function and a connective. By abusing notation, the symbol $\bigcirc$ denotes the unary temporal connective "next" $\bigcirc : \mathrm{wff} \to \mathrm{wff}$ if x is a propositional variable, or the function "next" $\bigcirc : \mathcal{D} \to \mathcal{D}$ if x is a function variable, where "wff" is the set of well-formed formulae. These are operators of unquantified first-order temporal logic [63].

Listing 2.1: The code that describes the example shown in Fig. 2.2.

```
#define H 3

free env int(1, 2) xt;
env int(0, H) yt;

assume env proctype taz(){
    do
    :: yt = yt - 1
    :: yt = yt + 1
    :: skip
    od
}

assume ltl { []<>(yt == 0) }

sys int(0, 3) x;
sys int(0, H) y;

assert sys proctype bunny(){
    do
    :: x = x - 1
    :: x = x + 1
    :: y = y - 1
    :: y = y + 1
    :: skip
    od
}

assert ltl {
  [] ! ((x == xt) && (y == yt)) &&
  /* [] ! --X ((xt' == x ) && (yt' == y )) && */
  [] -X ! ((xt' == x) && (yt' == y)) &&
  []<>((x == 3) && (y == 2)) }
```

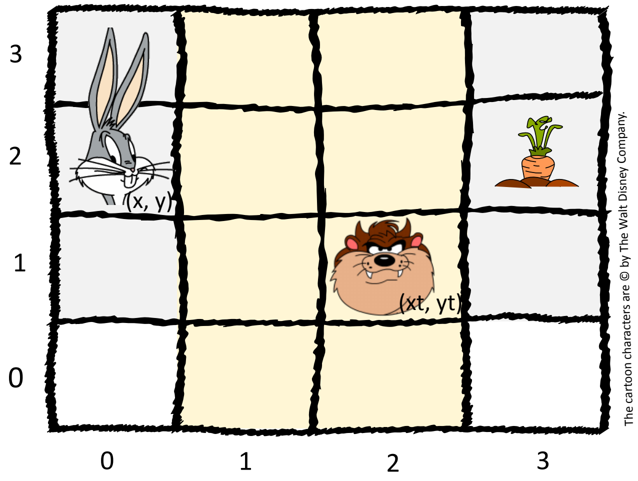


Figure 2.2: Turn-based game between two adversaries.

Table 2.1: Variables.

arity	Function	Predicate	Connective
0	`bit`, `byte`, `int`, `short`	`bool`	

Table 2.2: Constants.

arity	Function	Predicate	Connective
0	`#define`	`#define`	`false`, `true`
1	`-` (uminus)		`!`, `'`, `X`, `[]`, `<>` `-X`, `--X`, `-[]`, `-<>`
2	`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`	`<`, `<=`, `==`, `>=`, `>`, `!=`	`&&`, `||`, `->`, `<->` `U`, `W`, `R`, `S`
3	ite(`bool`, `num`, `num`)		ite(`bool`, `bool`, `bool`),

2.3.1 Functions, predicates, and connectives

Expressions are built from [64]:

- nullary function variables (integer-valued variables)
- unary function constants: `-`
- binary function constants: `+`, `-, *, /, %, &, |, ^`
- nullary predicate variables (Boolean-valued variables)
- binary predicate constants: `<, <=, ==, !=, >=, >` $(<, \leq, =, \neq, \geq, >)$
- nullary formula connectives `false, true` $(\bot, \top)$
- unary formula connectives `!` $(\neg)$
- unary temporal formula connective `', X, [], <>, -X, --X, -[], -<>` $(\bigcirc, \Box, \Diamond, \circledsim, \ominus, \boxminus, \diamondminus)$
- binary formula connectives `&&, ||, ->, <->, U, W, R, S` $(\wedge, \vee, \rightarrow, \leftrightarrow, \mathcal{U}, \mathcal{W}, \mathcal{R}, \mathcal{S})$

The prime is used as postfix syntax for the temporal operator “next”. For example, `x'` is the value of variable `x` at the next time step. There are also temporal logic formulae (with “always”, “eventually”, and “until”) that can appear in `ltl` blocks, but here we focus on expressions that can appear as statements in a process.

2.3.2 State predicates and actions

Following TLA, we call *state predicate* an expression that does not contain primed variables [11]. Note that a state predicate does not contain temporal operators, so it has no side effects. In pure `Promela`, all expressions are state predicates.

An *action* is an expression that contains primed variables [11]. If controlled variables appear primed in an action, then the action can be regarded as generalization of an assignment, as described next.

2.3.3 Deconstraining

By default, imperative variables are constrained to remain unchanged. If any assumption (assertion) process executes a statement that contains a primed environment (system) variable, then that variable is *not* constrained to remain unchanged in that time step. Such a variable is called *deconstrained*.

For example, suppose that the following statements appear in an assertion:

```
sys bit x = 0;
x == 0;
x' == 1 - y
```

When the statement `x == 0` is executed, the variable `x` is constrained by $x' = x$. But when the statement `x' == 1 - y` is executed, the synthesizer is allowed to pick the next value of `x` as needed, in order to satisfy the equality $x' = 1 - y$.

Note that if a statement in an assumption (assertion) process contains primed imperative system (environment) variables, these are *not* deconstrained. The reason is that assumptions (assertions) are relevant only to the environment's (system's) data flow.

Moreover, primed environment variables *can* appear in system processes, because those are *past* values that the environment just selected. In a Mealy game, primed system variables cannot appear in an assumption process, because they are *future* values that the system has not yet selected (if the game is interpreted using an alternating time refinement).

For example, suppose that the environment controls variable x and the system controls y, and the following code appears in an assertion:

```
env int(0, 10) x;
sys int(0, 10) y;
(x' == 5) && (y' == 3)
```

The above expression is interpreted as the formula

$$(x' = 5) \wedge (y' = 3). \tag{2.2}$$

Note that the default constraint $y' = y$ has not been applied, otherwise the action would be satisfiable only if variable y already equals 3. Also, no constraint resulted for variable x, because an assertion does not constraint the environment's data flow.

In implementation, the imperative variables are constrained separately, by collecting all the edges that deconstrain them, as discussed in Chapter 3.

Primed variables in a Mealy game

In this section, we describe the meaning of primed variables for a Mealy game. The game is alternating. If interpreted with a time refinement to alternating time, then a two-player alternating (turn-based) game is played between an output/input Moore machine and an input/output Mealy machine.

The system (Mealy transducer) looks at the current environment input that reacts to primed environment variables from $\mathcal{X}'$, and to the *previous* environment input (to which it has reacted in the previous turn of the game) by unprimed environment variables from $\mathcal{X}$. The environment refers to the current system output to which it reacts by using unprimed system variables from $\mathcal{Y}$. In order to refer to the last system output, the environment must use the unary temporal operator "previous" of past LTL. The past fragment of LTL can be used in GR(1), as proved in [4].

Nested "next"

Using a GR(1) synthesizer as back-end, nest "next" operators cannot be nested within a single statement, because nested next operators cannot appear in safety clauses of a GR(1) formula. If a synthesis tool for full LTL is used [37, 39, 40, 41], or bounded synthesis [65, 38], or the LTL formula that the program is translated to is converted to GR(1) [14], then multiply primed variables can be used, but the complexity is less favorable.

2.3.4 Executability

A condition called *guard* is associated to each statement [43]. The process can execute a statement only if the guard evaluates to true. If a process currently has no executable statement, then it *blocks*. If no process has an executable statement, then the system will *deadlock*, and lose the game.

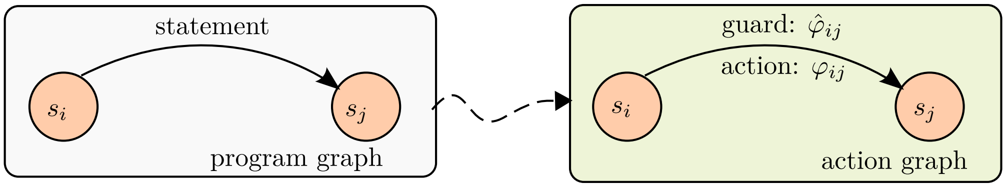


Figure 2.3: Transformation from a graph whose edges are labeled by program statements, to a graph whose edges are labeled by action formulae φ_{ij} (representing each statement) and guards $\hat{\varphi}_{ij}$, derived by existential quantification, as described in this section.

For each statement, its guard is defined by existential quantification of the primed variables of the data flow player. The quantification is applied after the statement is translated to a logic formula. So the guard of a statement is the realizability condition for that statement. It means that, from the local viewpoint of that statement only, given the current values of variables in the game, the constrained player can choose a next move. So the scheduler cannot pick as next process to execute a process that has blocked. Clearly, if all processes block, then that player deadlocks.

Using this definition, the guard of a state predicate is itself, as in PROMELA. In contrast to a state predicate, an action includes primed variables, so it can constrain the values assigned next to variables. As noted earlier, actions can be considered as generalized assignments. Unlike assignments in pure PROMELA, actions are not always executable.

In implementation, variables are quantified using the PYTHON binary decision diagram package `dd`[2]. If an unsatisfiable guard is found, then the implementation raises a warning.

For example, if we inserted the statement `xt && xt' && y'` in the process `bunny` (in Fig. 2.2), then its guard would be $\exists y'.x_t \wedge x'_t \wedge y' = x_t \wedge x'_t$. Similarly, the guard of an expression `xt && xt'` in the process `taz` is $\exists x'_t.x_t \wedge x'_t = x_t$.

Alternative An `atomic` block can also be used to combine a Boolean expression that acts as guard, with an action. This block forms a compound statement that becomes executable, if, and only if, the guard evaluates to true. Atomic statements are discussed in Section 2.6.

2.4 Assignments

Each assignment statement is translated to a formula that has the same effect. In an assumption (assertion), only environment (system) variables can be assigned. The assigned variable is not constrained to remain unchanged, if it happens to be an imperative variable. All other non-free variables (that do not appear inside the left operand) are subject to their default constraints. For example, the expression `x = (expr)` is interpreted (roughly – see Chapter 3 for details) as the formula

$$(x' = (expr)) \wedge \bigwedge_{y \in V^{\mathrm{imp}} \setminus \{x\}} (y' = y)$$

So the assignment operator overrides the default constraints that apply to imperative variables. The statement `x = x'` is equivalent to the formula $x' = x'$. which is a tautology, so true irrespective of the value that variable x takes next.

As remarked in Section 2.3.3, primed system variables cannot appear in assumptions, so the following assumption statements

```
env bit y;
sys bit x;
x = y';
```

---

[2] `dd` is a pure PYTHON BDD package by the first author https://github.com/johnyf/dd

are not well-defined. In particular, y' is a value that the system will select, after the environment selects x'. Therefore, it can always observe what value the environment assigned to y', and choose $x' = \neg y'$ to force the environment to lose the game.

Note that in an assumption (assertion), an imperative local variable of the system (environment) cannot be assigned to, because it is outside the scope of assertion (assumption) statements. Therefore, its default constraints imply that it will remain invariant. So such a local variable is of little use. Local opponent variables are useful only if declarative.

2.4.1 Truncation semantics

An assignment states that the next value of the assignee is equal to the value of the expression on the right hand side of the assignment symbol. If the assigned variable has mod-wrap semantics, then the value of the expression is truncated to the bitfield width of the assigned variable. As a result, an assignment to a variable with mod-wrap semantics cannot block.

If the assigned variable has saturation semantics, then the value of the expression is *not* truncated. So, an assignment to a variable with saturating semantics can block.

To formalize the previous, let $\mathrm{trunc}(y, w)$ denote a function that truncates the value of expression y to bitwidth w. An assignment `x = expr` is translated to the logic formula

$$x' = \mathrm{trunc}(\mathit{expr},\ \mathrm{width}(x)), \tag{2.3}$$

if variable x has mod wrap semantics, and to

$$x' = \mathit{expr} \tag{2.4}$$

otherwise. If variable x is imperative, then it is deconstrained.

2.4.2 Equality operator in logic

There is no assignment in logic. So only the operator "=" appears in logic formulae. In the input syntax of a solver like `gr1c`, the code fragment `x = x + 1` is interpreted as the formula $(x = x + 1)$, which is false, because the current value of variable x is different than the current value of x plus one. In contrast, in open `Promela` the token "`=`" signifies assignment, so the code fragment `x = x + 1` is interpreted as the formula $(x' = x + 1)$, which is satisfiable.

2.5 Automata products

We introduce the keywords `async` and `sync` to define *asynchronous* and *synchronous*, respectively, products of program graphs. A product is defined by one of the keywords `async`, `sync`, followed by `proctype` declarations enclosed in braces `{...}`. The product is taken over the processes enclosed in braces. In general, each such block can contain `proctype` declarations, as well as other products of processes. We will refer to each encapsulated block or `proctype` contained inside a block as an *element* of the container block. So the syntax is of the form `async{ proctypes or sync products }`, and `sync{ proctypes or async products }`.

A synchronous product blocks, if any program graph in it blocks. Otherwise, the scheduler can select the synchronous product for execution, so all program graphs must execute simultaneously. All program graphs in a synchronous product must constrain the same player.

An asynchronous product is the default top context, as in PROMELA. The scheduler can select any unblocked program graph (or `sync` product) inside an asynchronous product as the one that should execute next. The environment controls the scheduler, so if it picks a process whose guard evaluates to true, but its action is not satisfiable (Section 2.3), then the system fails to react and loses the game. If all program graphs of a player are blocked, then that player deadlocks and loses the game. In order to describe the process products discussed above, the `Promela` extended Backus-Naur form (EBNF) is augmented as described in Appendix A.1. An example of how asynchronous products can be nested is shown in Fig. 2.4.

```
sys proctype foo(){
    ...
}

env proctype hoo(){
    ...
}

sync{
    sys proctype boo(){
        ...
    }

    sys proctype woo(){
        ...
    }
}

sync{
    env proctype qoo(){
        ...
    }

    env proctype moo(){
        ...
    }
}
```

Figure 2.4: Products of program graphs.

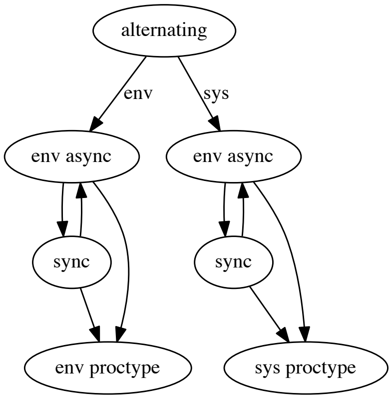


Figure 2.5: Nesting of synchronous and asynchronous process and player products. The prefix "env" signifies that the asynchronous product is universally quantified.

Note that nesting asynchronous (synchronous) products is equivalent to a single (flat) asynchronous (synchronous) product. Also, note that the asynchronous products between program graphs are in the context of full information, so the system is *not* asynchronous in the sense of [66, 67].

The asynchronous products defined above have universal quantification, because the environment schedules processes to execute. A possible extension is to define controlled asynchronous products, where the system selects the process that will execute. This is existential quantification of an asynchronous product. A prefix by `sys` can be used as syntax, i.e., `sys async{...}`. Currently, controlled asynchronous products are not implemented.

2.6 Atomic statements

The presence of two players and the target language (temporal logic) allow for a number of possible alternative interpretations for atomicity. We discuss them in this section.

Pure `Promela` includes two block statements that allow a process to request exclusive execution (of itself) by the process scheduler:

1. `atomic{...}`: the statements inside the block[3] that follows the keyword `atomic` are executed uninterruptedly, *unless* any one of them blocks. If a statement blocks, then atomicity is lost, allowing the scheduler to select the next process from among all the executable processes. Non-determinism is admissible inside an `atomic` statement[4].
2. `d_step{...}`: the statements inside the block that follows the `d_step` keyword are executed strictly uninterruptedly. No statement is allowed to block. In other words, a `d_step` cannot lose atomicity. If it blocks, then an error is raised. Non-determinism is not interpreted as such, but resolved arbitrarily once, and that choice remains fixed over the entire state space (as obtained by unfolding by the semantics engine).

2.6.1 Controllability of process execution

First, we observe that in pure `Promela` an atomic statement prevents *all* other (executable) processes from executing. In open `Promela` though, there are two players that in turn execute steps of the processes that they control. So an `atomic` statement can preempt either:

- all other processes of the player that controls the process that requests atomic execution, or
- all processes, of both players.

The first alternative models time consumed by the system to execute atomic execution, by allowing the environment to keep playing against it in turns. This models the situation where the system can request atomic execution among its own processes, but not from the environment. In other words, the environment acts in a strictly adversarial way, and we assume that the atomic block cannot be ensured to execute in alternation with the environment.

The second alternative models system actions that can either

- be ensured to execute in negligible time[5], within one turn of the system player, or
- a not totally adversarial environment, that cooperates slightly, by respecting preemption by system processes.

A unifying viewpoint is to consider these two cases of preemption power as another question of controllability:

- Preemption of system processes is controlled by the system in the first variant, but preemption of environment processes is uncontrolled by the system.
- Preemption of both environment and system processes is controlled by the system in the second variant.

The syntax `atomic(sys)...` can be used to define that only system processes are to be preempted. An `atomic...` block results in preemption of all processes, reducing the likelihood of confusion with pure `Promela`. Currently, preemption of both system and environment processes is implemented, but the syntax `atomic(sys)...` is a direct extension.

2.6.2 Expressing atomicity in temporal logic

An `atomic` statement can be represented in temporal logic by introducing auxiliary variables. Formally, atomicity is represented by Eqs. (3.7), (3.38), (3.41), (3.45) and (3.46) in Chapter 3, with some details discussed in Section 3.5. Here, we give an informal overview of how atomicity is represented in logic.

---

[3] In `Promela`, a *block* is defined as a set of statements contained in a pair of braces `{...}`.

[4] In verification, the system state is tracked during execution of an `atomic` block. The reason is to be "prepared" for losing atomicity, because, at that instance, the Kripke model's state will need to be stored in the state space. Note that in `SPIN`, a `never` claim is *not* evaluated during the execution of an `atomic` statement. By modeling claims with a guarantee on a process controlled by the environment that observes the system execution, we can say that `never` claims in `SPIN` have the same semantics as global preemption, which is discussed later in this section.

[5] For example, due to time-scale separation between the system and its environment, similarly to the synchronous hypothesis [68].

Consider a set of processes that execute asynchronously, and suppose that the scheduler selects a single process to execute next. The process executes its next statement, and can request to execute its next statement without interleaving with any other process, i.e., *exclusively*. We will call this process a *requestor*. Such a request is made when, in the program graph, the target node of the executed statement is in *atomic* context (Eq. (3.7)).

If the next statement is executable, then the scheduler grants the request for exclusive execution by selecting the requestor as the process that will execute next (Eq. (3.38)). Otherwise (i.e., when the next statement blocks), the scheduler selects another process to execute, one that is not blocked. The second case is called *loss of atomicity*.

In syntax, an `atomic` block defines nodes of the program graph that are in atomic context. If none of these statements blocks, then the block is executed uninterruptedly, without interleaving with other other processes (provided no statement is a `goto` with target outside the `atomic` block). Note that an `atomic` block cannot appear in a process that is part of a synchronous product.

Note that a request for exclusive execution is made when the target node is in atomic context. This can be the case when either the process enters atomic context, or when the process is already inside atomic context. A process can enter atomic context by either executing a statement that is syntactically the first in an `atomic` block, or by traversing a `goto` that jumps from outside, to inside an `atomic` block, or by resuming execution after losing atomicity.

It is interesting to note that an `atomic` context does *not* not necessarily comprise of a *single* syntactic `atomic` context, because `goto` statements can jump to another syntactic `atomic` context, from within the current one, without interruption of atomic execution, as long as the target node is in the interior of an `atomic` block.

The requestor sets an auxiliary integer variable ex_s to notify the scheduler of its identity. Upon entering atomic context, the requestor sets $ex_s = j$, where j is its identity (pid). If the requestor wants to preempt processes of both players, then it sets the propositional variable pm_s, in addition to $ex_s = j$. The scheduler reacts by selecting j as the next pid to execute, or not, in case j is blocked.

A process sets ex_s to its last value if it does not request exclusive execution. The last value is selected to not correspond to an existing process (so if there are n processes, then ex_s can take $n + 1$ values). The last value is used in order to avoid introducing additional auxiliary Boolean variables, thus economizing on bits for representing the problem. Extending the domain of ex_s by one increases the number of bits required to represent it *only* if the number n is already a power of two. Otherwise, no new BDD variables are introduced. For example, if $n = 6$, then the domains:

- $\{0, 1, \ldots, n - 1\} = \{0, 1, \ldots, 5\}$ and
- $\{0, 1, \ldots, n - 1, n\} = \{0, 1, \ldots, 6\}$

both require 3 bits to represent them.

Notice that the environment controls the scheduler, so it can serve requests only in the next turn of the game. So if the system's process j requests atomic execution at turn k, then the scheduler responds in turn $k+1$, by pausing the environment and selecting process j to execute in turn $k+1$. The environment executes as normal in turn k, when the request was issued. This conforms to the order of play, i.e., the environment in turn k executed *before* the system issued its request. So indeed, this does not interleave with the statements in the atomic block.

Defining a translation of `atomic` blocks from `Promela` to temporal logic extends previous work that did not handle iteration constructs, loss of atomicity, nor non-determinism inside atomic blocks [69, 70, 71].

2.6.3 Visibility to properties

If atomic statements execute over multiple time steps in temporal logic, then the intermediate states become visible in the game. A primary purpose of atomic transitions is to model changes of state that are ensured to happen indivisibly in implementation. Therefore, the intermediate states are a modeling artifact, so not relevant to LTL properties or other processes. For this reason, specification properties are not intended to take into account the states produced *during* atomic execution. In SPIN, these states are ignored in an atomic context, unless atomicity is lost (the last state produced becomes visible to LTL properties and other processes).

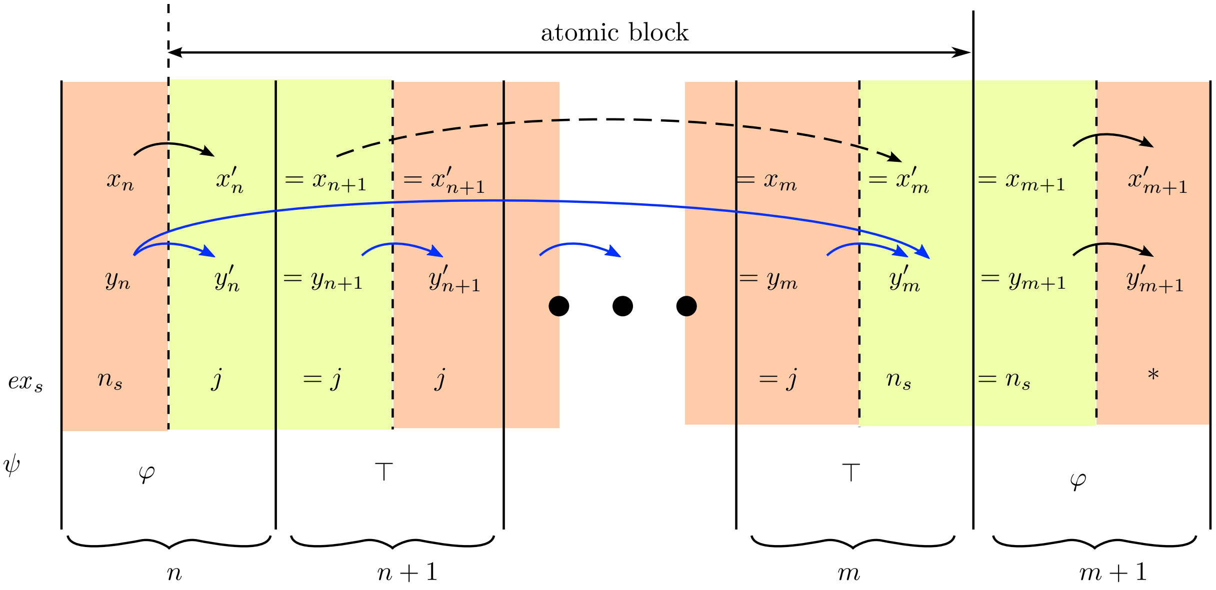


Figure 2.6: Timing diagram of atomic execution. The variables x are controlled by other processes and y by the atomically executing system process. The variables x "freeze" ($x'_{n+1} = x_{n+1}$) after atomic execution starts at time n. In addition, the formula $\psi \triangleq \varphi \vee (ps'_s = ex_s < n_s)$ deactivates the property φ, so that it won't observe any intermediate states produced during atomic execution.

Consider an LTL safety property φ defined by an `ltl` block. For the reasons just described, the property needs to be "deactivated" if two conditions hold: some process just requested atomic execution, *and* the scheduler grants the request, i.e., the requestor did not blocked and lost atomicity. A property that does not contain "next" operators can be deactivated by disjoining it with these two conditions, as in Eqs. (3.45) and (3.46) that are introduced in Section 3.5. Fig. 2.6 shows how other processes "freeze", and the property ψ is deactivated during atomic execution. Deactivation is necessary for safety properties, but not relevant to recurrence properties, because they are pure liveness properties.

Note that a variation is possible, by allowing a property to observe the intermediate states during atomic execution. Such a property represents a specification for the *internal* operation of that process, but not with respect to other processes. The implementation exposes an option to make atomic execution visible to safety properties.

2.6.4 Stutter invariance

The temporal logic representation of atomicity outlined in Section 2.6.2 remains valid under conditions that are discussed in this section. There are three cases to consider:

1. safety GR(1) properties that the user has defined as `ltl` blocks,
2. asynchronously executing other processes,
3. synchronously[6] executing other processes that are preempted, e.g., Büchi automata.

As discussed next, in order to ensure correctness of the representation of atomicity in logic, some of the above cannot contain the "next" operator. We need to consider two categories of GR(1) safety formulae: those that contain the "next" operator, and those that do not. Note that formulae that contain only primed variables are almost[7] equivalent to their unprimed counterpart.

As shown in Fig. 2.6, there are two different time scales:

---

[6] Nesting of asynchronous products inside synchronous products and inclusion of atomic statements are not implemented yet, but nevertheless discussed in the analysis.

[7] The primed version excludes the initial time.

- asynchronous time, which includes the time steps: $\dots n-1, n, m+1, \dots$ (macro steps)
- game time, which includes all time steps: $\dots n-1, n, n+1, \dots, m, m+1, \dots$ (micro steps).

Let us consider how these different time scales appear to the properties and processes listed earlier.

Asynchronous processes Suppose that system process j executes in turn n, and requests exclusive execution. Any other system process $j+1$ that executes asynchronously with process j has executed its last statement in turn $n-1$. It is currently paused. If the request is granted, then the earliest it may resume is at turn $m+1$. Therefore, expressions in process $j+1$ can refer to primed global variables without being affected by how atomic execution is represented in logic. In other words, asynchronous processes are insensitive to external time refinement, as that introduced by simulating atomic transitions in game time.

Safety GR(1) properties In contrast, a safety assertion must refer only to unprimed global variables, i.e., it cannot contain the operator "next". Referring to primed global variables can lead to incorrect results. For example, in Fig. 2.6 a property φ that refers to x' and y' will apply to valuation (x_n, x'_n, y_n, y'_n). However, this valuation does not correspond to two consecutive time steps of asynchronous time, because the value y'_n corresponds to an intermediate state during atomic execution. In asynchronous time, the successor state of (x_n, y_n) is (x_{m+1}, y_{m+1}), i.e., after the atomic transition has been completed (or exited atomic context due to loss of atomicity). So the correct value is $y'_m = y_{m+1}$, upon completion of the atomic transition of process j (note that $x_n = x'_m = x_{m+1}$). Therefore, the correct valuation to apply property φ to is (x_n, x'_n, y_n, y'_m). Alternatives that are compatible with properties that include "next" are described in Section 2.6.5.

Not all properties that contain "next" are stutter-sensitive [72]. In other words, other formulae must be in a syntactic subset of stutter-invariant properties, namely those that do not include the operator "next" for non-local variables.

Preempted synchronous processes Similar observations apply to a preempted process $j+2$ that executes synchronously with process j. A deterministic Büchi automaton may be such a synchronous process (so the "program graph" counterpart of a GR(1) safety property). Processes that execute synchronously with process j cannot include primed global variables in their statements. This ensures that intermediate states of atomic execution are not visible to guards of statements in those other processes. It also avoids interference of actions in these processes with the atomically executing process (e.g., if y'_n in Fig. 2.6 appeared in an action of a synchronously executing process that executes its last statement at time n). Only local variables can appear primed in expressions, or assigned to. Local variables are valid, because they are visible only to process $j+2$, so they do not introduce coupling between different processes. For example, the program counter is a local variable in this sense (with respect to other processes, excluding the scheduler that is aware of all data flow).

Statements in atomic context If a statement that is executed atomically refers to environment variables, then these are equal to x'_n, as shown in Fig. 2.6. Therefore, such references are to be understood as the last environment reaction. In other words, only primed environment variables in atomically executing statements correspond to the primed environment variables in asynchronous time. Unprimed environment variables correspond to unprimed environment variables in asynchronous time only for the statement that enters atomic atomic context.

Remarks In SPIN, the operator "next" is discouraged, because it reduces the applicability of partial-order reduction. Another reason to avoid stutter-sensitivity in component specifications is to ensure that the specifications remain invariant under time refinement [11].

Some limitations to the use of atomic blocks have been described in this section. Note that these are not fundamental limitations of open `Promela` itself, but rather of the translation to temporal logic formulae. Instead, an enumerative synthesizer or model checker can interpret open `Promela` directly, without the need of an intermediate translation to formulae. The limitations about `atomic` statements that are described above do not apply to an enumerative approach. Also, they do not apply to a less efficient symbolic execution approach of Section 2.6.5 that would employ copies of variables as solver memory, in order to simulate correctly more use cases for atomic block usage.

2.6.5 Alternatives that accommodate stutter-sensitivity

There are two other options for representing atomic blocks in logic:

- Use auxiliary variables (like ex_s) to request from the process scheduler to preempt other processes.
- Use symbolic execution to derive a single action formula for the entire atomic block in temporal logic.

Duplicate variables All program variables can be duplicated and additional scheduling introduced to "forward" the values x_n, y_n to time $m+1$ and use them as unprimed values for evaluating a property that contains both primed and unprimed variables, deactivating and activating the property accordingly. This approach increases the size of the state space exponentially in the number of variables. For this reason, it is not discussed further here.

Symbolic execution In the presence of stutter-sensitive properties, symbolic execution has to be used. Two limitations of symbolic execution are that:

- Non-determinism inside an `atomic` block can cause the formulae resulting from symbolic execution to grow large.
- An iteration construct inside an `atomic` block cannot be represented with symbolic execution if the number of iterations is not a priori bounded.

For example, the code fragment `atomic{ stmt0; stmt1; stmt2 }` would be translated to

$$\mathrm{ite}\Big(\hat{\varphi}_0, \mathrm{ite}\big(\hat{\varphi}_1, \mathrm{ite}(\hat{\varphi}_2, s[\varphi_0][\varphi_1][\varphi_2], s[\varphi_0][\varphi_1]), s[\varphi_0]\big), \text{no change}\Big),$$

where $s[\varphi_0][\varphi_1]$ is postfix notation for the application of action φ_0 on state s, then action φ_1 on the result [11].

Instead of symbolic execution, the operator `&&` can be used to conjoin state predicates with expressions containing primed variables, creating arbitrary guarded commands. This avoids the need to write `atomic` blocks for expressing guarded reactions that involve assignments (a common use case). See Section 2.3.4 for the definition of executability for actions.

For atomic blocks that do not contain iteration statements, a possible compiler optimization is to check if any statement after the entry can block. If none of the interior statements can block, then symbolic execution can be applied without the need for `ite` operators, because atomicity cannot be lost in this special case.

2.7 Verification in open `Promela`

Verification can be approached by either checking that a desired ("positive") property holds for all possible computations, or that there does not exist a satisfying computation for an undesired ("negative") property.

Validity If checking whether a property holds, we desire to check all possible control flow paths. So paths are universally quantified. In verification, expressions in a model do not include primed variables (primed variables result only indirectly from assignments). Therefore, each path corresponds to a unique data flow.

The data flow needs to be existentially quantified. The reason is that when no feasible data flow exists, then a deadlock has been found. In this case, existential quantification over an empty set of successors yields the desired result (false), whereas universal does not. In other words, if the environment controlled the data flow, then deadlock would result in trivial realizability, instead of unrealizability (which is the correct result). The combination `assert env proctype` can express this use case.

When using a GR(1) synthesizer, the desired property must be in GR(1), so a recurrence property.

Satisfiability If checking whether a property does not hold, then a counterexample can be found by quantifying control and data flow existentially (closed system synthesis). In this case, unrealizability implies that the (undesired) property does not hold. When using a GR(1) synthesizer, the undesired property must be in GR(1), thus of the form $\psi \triangleq \varphi_i \wedge \Box\varphi_s \wedge \Box\Diamond\varphi_r$, since there no environment variables or assumptions are defined. In other words, realizability implies that there exists a system that satisfies ψ. This is a counterexample that violates the desired property $\neg\psi = \neg\varphi_i \vee \Diamond\neg\varphi_s \vee \Diamond\Box\neg\varphi_r$.

Otherwise, a either a generalized Rabin(1) or full LTL synthesizer have to be used, in order to check persistence as negative properties, thus recurrence as positive properties.

Chapter 3

Translation to logic

3.1 Notation

A program comprises of processes and products of processes, which may be nested. For each process is identified by a unique integer identifier $r \in \mathbb{N}$. Each product is identified by a unique integer identifier $r \in \mathbb{N}$. These identifiers don't overlap, so there is no process and product that are identified by the same integer. Where appropriate, k is used, instead of r, to emphasize that a product identifier is meant. By $p \in \{e, s\}$ we denote a player (environment and system, respectively). The set $\text{pids}(p) \subset \mathbb{N}$ contains the identifiers of processes that constrain the dataflow of player p.

A synchronous product can contain processes and asynchronous products. An asynchronous product can contain processes and synchronous products. In syntax, a synchronous (asynchronous) product inside a synchronous (asynchronous) product is equivalent to placing the contents of the inner product directly under the outer product. So there is no loss of generality in assuming that the same type of product does not appear consecutively in the nesting. The processes and products inside a product are referred to as *elements* of that product.

In a synchronous product, either all the elements execute, or the don't. In contrast, in an asynchronous product, a choice has to be made among the elements, for selecting which element will execute next. For an asynchronous product with index k, this choice is made by an environment variable $ps_k \in \mathcal{X}$. The abbreviation "ps" stands for *process scheduler*. The variable ps_k takes values in $\{0, 1, \ldots, n_k\}$, where n_k is the number of elements in the product k. Each value of ps_k in $\{0, 1, \ldots, n_k - 1\}$ selects an element to execute next. If r is the identifier of an element, then $m(r)$ is the corresponding index in the asynchronous product that contains that element. In other words, if element r is inside product k, then setting $ps'_k = m(r)$ selects element r to execute. Note that the element may be a process, or another asynchronous product (as an element of a synchronous product inside a synchronous product, as shown in Fig. 3.1).

The last value n_k is reserved to signify that no process in the asynchronous product k will execute next. This may be the case if the product is inside a synchronous product that wasn't selected to execute next, or the dataflow player reached a deadlock, or was preempted by a request for atomic execution by the other player. Let ps_e, ps_s denote the the variables that correspond to the top asynchronous product of the environment and system, respectively, and n_e, n_s their maximal values.

A *program counter* variable $pc_r \in N_r \subset \mathbb{N}$ stores the current node of the program multi-digraph (N_r, E_r). If the program graph is an assumption (assertion), then the variable pc_r is controlled by the environment (system)

$$pc_r \in \begin{cases} \mathcal{X}, & \text{if } \texttt{assume env} \text{ or } \texttt{assert env} \\ \mathcal{Y}, & \text{if } \texttt{assume sys} \text{ or } \texttt{assert sys} \end{cases} \tag{3.1}$$

The set N_r of nodes is a contiguous subset of the natural numbers. Define

$$\text{player}(r) \triangleq \begin{cases} e, & \text{if process } r \text{ constrains environment dataflow } (\texttt{assume}) \\ s, & \text{if process } r \text{ constrains system dataflow } (\texttt{assert}) \end{cases} \tag{3.2}$$

For `assume sys` processes, an additional auxiliary variable $\widehat{pc}_r$ is needed for representing their control flow.

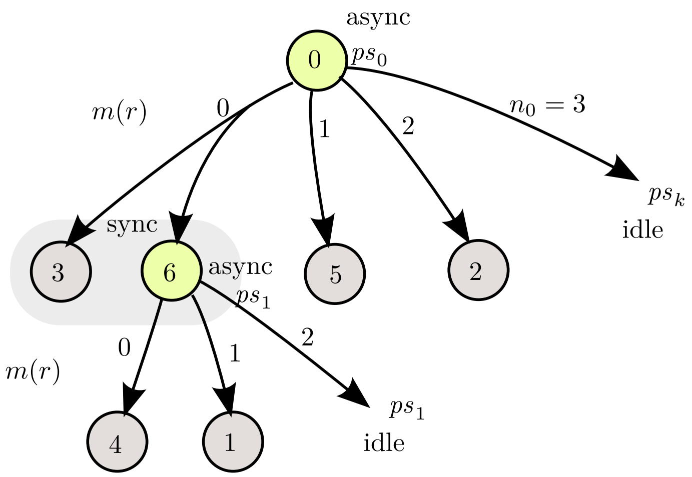


Figure 3.1: Product nesting of a synchronous product in an asynchronous product, and another asynchronous product inside the synchronous product. Each element in the asynchronous product ps_0 is selected by a value of variable ps_0. This value is given by $m(r)$, where r the (unique) integer that identifies the element. In this example, $m(3) = 0, m(6) = 0, m(5) = 1, m(2) = 2, m(4) = 0, m(1) = 1$, where $r \in \{3, 5, 2, 4, 1\}$ identifies single processes. The values $n_0 = 3, n_1 = 2$ serve for keeping an asynchronous product idle. For example, if $ps_0 = 1$, then process 5 must execute, and $ps_0 \neq 0 \rightarrow ps_1 = 2$. Note that process 3 is in the same synchronous product with the asynchronous product 6. For this reason, they share selection index, i.e., $m(3) = 0 = m(6)$.

For each program graph r, the function $\mathrm{key}(r)$ returns an auxiliary variable controlled by the player that controls pc_r. The key distinguishes multi-edges with same source and target nodes. The domain of variable $\mathrm{key}(r)$ is sufficiently large to accommodate for the maximal multi-edge multiplicity of a multi-edge in any set of edges E_r. This number is typically small, and bounded by the maximal branching factor among program graphs.

The same key variable is reused for program graphs that execute asynchronously. Inside a synchronous product, there exist multiple program graphs that execute at the same time, so multiple key variables are needed in that case. For example, a synchronous product that contains three program graphs requires three key variables.

In addition, in order to avoid conflicts, there are three disjoint collections of variables that serve as keys: environment key variables for `assume env`, system key variables for `assert sys`, and environment key variables for `assert env`. Note that the domain of $\mathrm{key}(r)$ can be larger than the multiplicity of any outgoing edge at a given node i, in order to accommodate for the multiplicity of outgoing edges at other nodes.

The system variable $ex_s \in \mathbb{N}_{\leq n_p}$ is used by a system process to request that it execute exclusively.and the system variable $pm_s \in \mathbb{B}$ to preempt the environment.

In the following, we use "=", with the understanding that "↔" must be used instead, if the variables involved are propositional. The indices i and j denote (in most cases) the source and target nodes, respectively, of an edge $(i, j, k) \in E_r$. Let $\mathrm{owner}(x)$ denote the player that controls variable x.

Define the formula for invariance of a variable x as

$$\mathrm{inv}(x) \triangleq \begin{cases} \bigwedge_{j=0}^{j=\mathrm{len}(x)-1} \mathrm{inv}(x_j), & \text{if } x \text{ is an array.} \\ x' = x, & \text{otherwise} \end{cases} \tag{3.3}$$

where $\mathrm{len}(x)$ is the length of the array variable x. Let $\mathrm{ite}(x, y, z)$ denote the ternary conditional operator "if x, then y, else z".

Each reference `a[i]` to an element in an array `a` is encoded using the (appropriate) ternary conditional operator

$$\text{ite}\Big(i = \text{len}(a) - 1,\, a_{\text{len}(a)-1},\qquad\qquad \text{ite}\Big(i = \text{len}(a) - 2,\, a_{\text{len}(a)-2},\qquad \text{ite}\big(\cdots,\qquad \text{ite}(i = 1, a_1, a_0)\big)\Big)\Big) \tag{3.4}$$

Array indexing errors are prevented by adding an additional safety assumption or assertion of the form

$$\Box(0 \leq i \leq \text{len}(a) - 1) \tag{3.5}$$

Unless noted otherwise, the formulae that follow are safety assumptions or assertions.

3.2 Control and data flow

Each process is parsed and converted to a program graph as outlined in Section 4.1, according to the definitions of control flow constructs in Promela [42]. To each edge $(i, j, k) \in E_r$ of process r corresponds an action formula $\varphi_{r,i,j,k}$. The statements in each program graph are translated to logic formulae and guards. Each graph, now labeled with formulae, is flattened to an equivalent formula of transitions (Eq. (3.6)), similarly to symbolic model checking [73]. This results in a data flow constraint (Eq. (3.8)).

$$\begin{aligned}
\text{trans}(r) &\triangleq \bigwedge_{i \in N_r} \left((pc_r = i) \rightarrow \bigvee_{(i,j,k) \in E_r} \big(\varphi_{r,i,j,k} \wedge (\widetilde{pc}_r = j) \wedge (\widetilde{\text{key}}_r = k) \wedge \text{exclusive}(r, i, j, k)\big) \right) \\
\widetilde{pc}_r &\triangleq \begin{cases} \hat{pc}_r, & \text{if player}(r) = e \text{ and } pc_r \in \mathcal{Y} \text{ (\texttt{assume sys})} \\ pc'_r, & \text{otherwise} \end{cases} \\
\widetilde{\text{key}}_r &\triangleq \begin{cases} \text{key}(r), & \text{if player}(r) = e \text{ and } pc_r \in \mathcal{Y} \text{ (\texttt{assume sys})} \\ \text{key}(r)', & \text{otherwise} \end{cases}
\end{aligned} \tag{3.6}$$

$$\text{exclusive}(r, i, j, k) \triangleq \begin{cases} (ex'_s = m(r)) \wedge pm'_s, & \text{if player}(r) = s \text{ and } j \text{ in atomic context} \\ (ex'_s = n_s) \wedge \neg pm'_s, & \text{if player}(r) = s \text{ and } j \text{ not in atomic context, and} \\ \qquad \text{there exists a system process with atomic blocks} & \\ \top, & \text{otherwise} \end{cases} \tag{3.7}$$

We consider the case of static process instantiation. Dynamic process instantiation is currently not supported, because it requires adding new BDD variables during computations.

The following formula constrains the data flow of player p to comply to program graph r

$$\text{dataflow}(r) \triangleq (ps(r)' = m(r)) \rightarrow \text{trans}(r). \tag{3.8}$$

So the scheduler selects element $m(r)$ to execute by setting $ps(r)'$ to $m(r)$. Those program graphs that are not selected to execute must remain idle, i.e., $\text{inv}(pc_r)$. Note that $\text{inv}(pc_r)$ does not suffice to distinguish self-loops from an idle program graph. In addition, the variables in the scope of an idle program graph must remain unchanged, as ensured by Eqs. (3.18) and (3.22).

The data flow constraint of Eq. (3.8) suffices for graphs with same control flow and data flow quantification (`assume env` or `assert sys`), because it ensures that the player that controls pc_r and $\text{key}(r)$, and is constrained by action $\varphi_{r,i,j,k}$, will select appropriate values for them.

However, if the control flow has different quantification than the data flow (`assume sys` or `assert env`), then a separate control flow assumption is needed, Eq. (3.12). This formula prevents the environment from selecting edges that have false guards.

The guard of each statement is obtained by existential quantification of the future values of the variables in the formula $\varphi_{r,i,j,k}$ that corresponds to that statement

$$\text{guard}(r, i, j, k) \triangleq \exists x \in Q'.\ \varphi_{r,i,j,k}, \tag{3.9}$$

where

$$Q \triangleq \begin{cases} \mathcal{X}, & \text{if player}(r) = e \\ \mathcal{Y}, & \text{if player}(r) = s \end{cases} \tag{3.10}$$

The resulting guards are used to define the behavior of the program counter variables pc_r, as follows

$$\text{guards}(r) \triangleq \begin{cases} \bigwedge\limits_{i \in N_r} \left((pc'_r = i) \rightarrow \begin{pmatrix} \big(\bigcirc \bigvee\limits_{(i,j,k) \in E_r} \text{guard}(r, i, j, k) \big) \rightarrow \\ \bigvee\limits_{(i,j,k) \in E_r} \begin{pmatrix} (\widehat{pc}'_r = j) \wedge \\ (\text{key}(r)' = k) \wedge \\ \bigcirc \text{guard}(r, i, j, k) \end{pmatrix} \end{pmatrix} \right), \\ \text{if player}(r) = e \text{ and } pc_r \in \mathcal{Y}. \\ \bigwedge\limits_{i \in N_r} \left((pc_r = i) \rightarrow \bigvee\limits_{(i,j,k) \in E_r} \begin{pmatrix} (pc'_r = j) \wedge \\ (\text{key}(r)' = k) \wedge \\ \text{guard}(r, i, j, k) \end{pmatrix} \right), \\ \text{otherwise} \end{cases} \tag{3.11}$$

In Eq. (3.11), the guards of an `assume sys` process r are primed, because the system has to select the candidate transition $(\hat{pc}'_r, \text{key}(r)')$ one time step before the environment scheduler decides whether process r will execute. This is also the reason why an additional variable $\hat{pc}_r$ has to be used, instead of the program counter pc_r. The program counter will change only if, in the next time step, the scheduler selects process r for execution.

$$\text{control\_flow}(r) \triangleq \text{ite}\big(ps(r)' = m(r),\ \text{pc\_trans}(r),\ \text{inv}(pc_r)\big) \tag{3.12}$$

$$\text{pc\_trans}(r) \triangleq \begin{cases} \text{guards}(r), & \text{if player}(r) = s \text{ and } pc_r \in \mathcal{X} \text{ (}\texttt{assert env}\text{)} \\ pc'_r = \widehat{pc}_r, & \text{if player}(r) = e \text{ and } pc_r \in \mathcal{Y} \text{ (}\texttt{assume sys}\text{)} \end{cases} \tag{3.13}$$

$$\widehat{\text{pc}}\text{\_trans} \triangleq \bigwedge_{r | \text{player}(r) = e \text{ and } pc_r \in \mathcal{Y}} \text{guards}(r). \tag{3.14}$$

This control flow constraint applies to processes where the control flow is controlled by a different player than the data flow (`assume sys` and `assert env` processes). It ensures that the player who controls the program counter selects an existing edge in the program graph, by selecting suitable values for the variables pc_r and key.

For example, consider a system process r that is currently at $pc_r = 0$ and has two outgoing edges leading to $pc_r = 1$ and $pc_r = 2$. Suppose that the edge $(i = 0, j = 1, k = 0)$ is currently executable, but the other edge $(i = 0, j = 2, k = 1)$ is not. If the environment selected $ps(r) = m(r)$ and $pc'_r = 2$, then the system would be blocked. This environment behavior is prevented by Eq. (3.12).

3.2.1 Initial conditions

Each variable has a default initial value, depending on its semantics. If a value is assigned to the variable upon declaration, then that value overrides the default initial value. For example, `free bit x = 1`. By default,

- a free variable is not constrained initially,
- a Boolean (numerical) imperative variable is initially `false` (zero),
- an imperative ranged integer is initially equal to the minimal value in its range.

Each element of an array variable is initialized separately.

Let assert_sys_pids, assume_env_pids, asser_env_pids denote the process identities declared with the corresponding path and data flow quantification. The system should initially satisfy

$$(ex_s = n_s) \wedge \neg pm_s \wedge \big(\forall r \in \text{assert\_sys\_pids}.\ (pc_r = P_r.root)\big). \tag{3.15}$$

The environment is assumed to initially satisfy

$$\begin{aligned}&\big(\forall r \in \text{assume\_env\_pids}.\ (pc_r = P_r.root)\big)\wedge\\ &\big(\forall r \in \text{assert\_env\_pids}.\ (pc_r = P_r.root)\big).\end{aligned} \tag{3.16}$$

In an `assume sys` process, the program counter is controlled by the system, and the variables pc_r and $\widehat{pc}_r$ are used to represent its behavior. These need to be initialized according to the initial nodes of process r, because the initial value of pc_r is its roots node, and if the scheduler selects process r in the first turn, then the initial value of $\widehat{pc}_r$ will be used for the first transition of process r. The initial condition of variable $\widehat{pc}_r$ should take into account the initial valuation of guards on edges outgoing from the root node, as follows

$$\text{init}(\widehat{pc}_r) \triangleq \left(\begin{array}{c}\Big(\bigvee_{(P_r.root,\ j,\ k)\in E_r} \text{guard}(r, P_r.root, j, k)\Big) \rightarrow \\ \bigvee_{(P_r.root,\ j,\ k)\in E_r} \big((\widehat{pc}_r = j) \wedge (\text{key}(r) = k) \wedge \text{guard}(r, P_r.root, j, k)\big)\end{array}\right) \tag{3.17}$$

Note that $\text{Var}(\text{guard}(r, P_r.root, j, k)) \subseteq \mathcal{X} \cup \mathcal{Y}$, because environment statements do not refer to primed system variables (future values), and primed environment variables have been quantified in Eq. (3.11). Therefore, the guard can be a conjunct in the initial condition of the system.

3.3 Invariance of variables

Eq. (3.18) ensures that local declarative variables remain invariant if their scope doesn't execute.

$$\text{local\_free}(p, r) \triangleq (ps(r)' \neq m(r)) \rightarrow \bigwedge_{x \in V^{\text{free}}_{p,r}} \text{inv}(x), \tag{3.18}$$

where $V^{\text{free}}_{p,r}$ is the set of free variables that are controlled by player $p \in e, s$ and declared in the scope of program graph r.

A related formula constrains the free environment variables when the environment does not execute (only if the system is granted exclusive execution, Section 3.5)

$$\text{freeze\_env\_free} \triangleq \begin{cases} \bigwedge_{x \in V^{\text{free}}_{e,\text{globals}} \cup \bigcup_{r \in \text{pids}(s)} V^{\text{free}}_{e,r}} \text{inv}(x), & \text{if there exists a} \\ \qquad \text{system process with atomic blocks} & \\ \top, \quad \text{otherwise} & \end{cases} \tag{3.19}$$

An imperative variable x is constrained to remain unchanged *only* if none of the currently executed statements deconstrains x. Statements that deconstrain an imperative variable are

- an assignment to the variable, or
- an action that contains the primed value of the variable, i.e., an action statement with $x' \in \text{Var}(\varphi_{r,i,j,k}) \cap (V^{\text{imp}}_{p,r})'$

If any of these two conditions is true for a given edge $(i, j, k) \in E_r$, it is denoted by $\text{deconstrained}(x, r, i, j, k)$.

Note that primed imperative variables that appear in an expression that comprises the right hand side of an assignment are not deconstrained (an assignment is understood as unidirectional, not as a balance equation).

It is possible that multiple statements are executed at the same time, by program graphs that are in the same synchronous product. So the constraints should be formulated at the scope that the variable x is defined, to account for all the program graphs that can refer to variable x, and so also change it. This is ensured by Eqs. (3.21) and (3.22). For brevity, in the following we use the auxiliary definition

$$\mathrm{edge}(r,i,j,k) \triangleq \begin{cases} (ps(r)' = m(r)) \land (pc_r = i) \land (\widehat{pc}_r = j) \land (\mathrm{key}(r) = k), \\ \quad \text{if } \mathrm{player}(r) = e \text{ and } pc_r \in \mathcal{Y} \ (\texttt{assume sys}) \\ (ps(r)' = m(r)) \land (pc_r = i) \land (pc'_r = j) \land (\mathrm{key}(r)' = k), \\ \quad \text{otherwise} \end{cases} \tag{3.20}$$

Eq. (3.21) ensures that primed references to elements in imperative arrays deconstrain only the referenced array element(s).

$$\mathrm{array\_inv}(r) \triangleq \bigwedge_{a \in \mathrm{arrays}} \bigwedge_{(i,j,k) \in E_r} \left(\mathrm{edge}(r,i,j,k) \to \bigwedge_{l=0}^{\mathrm{len}(a)-1} \left(\Big(\bigwedge_{e \in \mathrm{idx}(a,i,j,k)} (e \neq l) \Big) \to \mathrm{inv}(a_l) \right) \right), \tag{3.21}$$

where $\mathrm{idx}(a,i,j,k)$ is the set of expressions that appear as indices of primed references to elements of array a in action $\varphi_{r,i,j,k}$. This ensures that all array elements a_l other than those that appear primed in $\varphi_{r,i,j,k}$ remain invariant ($e \neq l$). The index expression e can be a variable, so its value is not fixed, thus conjunction over all array elements ($l \in \mathbb{N}_{<\mathrm{len}(a)}$) is necessary.

Eq. (3.22) ensures that imperative variables remain invariant, unless a statement that deconstrains them is executed (Section 2.3).

$$\mathrm{imperative\_inv}(p,r) \triangleq \mathrm{array\_inv}(r) \land \bigwedge_{x \in V^{\mathrm{imp}}_{p,r}} \Big(\mathrm{inv}(x) \lor \bigvee_{(i,j,k) \in E_r \mid \mathrm{deconstrained}(x,r,i,j,k)} \mathrm{edge}(r,i,j,k) \Big) \tag{3.22}$$

Eq. (3.22) applies also to global variables (by considering the global scope, instead of scope r).

Finally, a safety constraint is added for each ranged integer variable

$$\mathrm{ranged\_int}(p) \triangleq \bigwedge_{x \in \mathrm{ranged\_ints}(V_p)} \Big(\min(\mathrm{dom}(x)) \leq x \leq \max(\mathrm{dom}(x)) \Big), \tag{3.23}$$

where $\mathrm{dom}(x)$ denotes the domain of variable x.

3.4 Process scheduler

In this section, we describe how processes are selected for execution. Note that all processes inside a synchronous product constrain the same player, and that each `proctype` inside a synchronous product is limited to 1 active instance.

3.4.1 Nested products

Let k be the identifier of an asynchronous product that is not the top one. By definition of the product nesting, it follows that product k is the element of some synchronous product. Any synchronous product appears as an element of some asynchronous product. This "parent" asynchronous product has selector variable $ps(k)$. To avoid confusion, recall that ps_k is the selector variable of product k, so ps_k is a different variable from $ps(k)$. We will now impose two conditions that represent the nesting of asynchronous products.

If the asynchronous product k is not selected for execution ($ps(k)' \neq m(k)$), then no element in product k executes

$$(ps(k)' \neq m(k)) \to (ps'_k = n_k). \tag{3.24}$$

If the asynchronous product $ps(k)$ selects the synchronous product that contains product ps_k, then some element of product ps_k should execute

$$\begin{aligned} (ps(k)' = m(k)) \to (ps'_k \neq n_k) &= \neg(ps'_k \neq n_k) \to \neg(ps(k)' = m(k)) \\ &= (ps'_k = n_k) \to (ps(k)' \neq m(k)). \end{aligned} \tag{3.25}$$

Conjoining Eqs. (3.24) and (3.25) yields

$$\text{product\_selected}(k) \triangleq (ps(k)' \neq m(k)) \leftrightarrow (ps'_k = n_k). \tag{3.26}$$

The scheduler cannot select a blocked process to execute. A process blocks if at a given node i, all outgoing edges $(i, j, k) \in E_r$ have false guard $\text{guard}(r, i, j, k)$

$$\begin{aligned} \text{blocked}(r) &\triangleq \bigvee_{i \in N_r} \Big((pc_r = i) \wedge \bigwedge_{(i,j,k) \in E_r} \neg \text{guard}(r, i, j, k) \Big), \qquad (3.27) \\ \text{selectable}(r) &\triangleq \text{blocked}(r) \rightarrow (ps(r)' \neq m(r)). \qquad (3.28) \end{aligned}$$

The condition $\text{blocked}(r)$ tests whether a process has blocked, and $\text{selectable}(r)$ prevents the scheduler from selecting it if it is blocked.

Analogously, we can define when a product blocks.

- A synchronous product is blocked iff any element inside it is blocked
- An asynchronous product is blocked iff all elements inside it are blocked.

To formalize these, let R_k denote the identifiers of all elements in product k. Redefine Eq. (3.28) to address both elements that are processes, and elements that are products

$$\text{selectable\_element}(r) \triangleq \text{element\_blocked}(r) \rightarrow (ps(r)' \neq m(r)). \tag{3.29}$$

The blocking conditions are defined recursively as

$$\begin{aligned} \text{element\_blocked}(r) &\triangleq \begin{cases} \text{blocked}(r), \text{ if } r \text{ is a process} \\ \text{sync\_blocked}(r), \text{ if } r \text{ is a synchronous product} \\ \text{async\_blocked}(r), \text{ if } r \text{ is an asynchronous product.} \end{cases} \qquad (3.30) \\ \text{sync\_blocked}(r) &\triangleq \bigvee_{r \in R_k} \text{element\_blocked}(r) \qquad (3.31) \\ \text{async\_blocked}(r) &\triangleq \bigwedge_{r \in R_k} \text{element\_blocked}(r) \qquad (3.32) \end{aligned}$$

The recursion in this definition terminates, because a program comprises of a finite number of elements, and so the bottom products must contain only processes.

By substituting these definitions in Eq. (3.29), we obtain three constraints

$$\begin{aligned} \text{selectable\_element}(r) &= \text{blocked}(r) \rightarrow (ps(r)' \neq m(r)) \qquad (3.33) \\ \text{selectable\_element}(r) &= \text{sync\_blocked}(r) \rightarrow (ps(r)' \neq m(r)) \qquad (3.34) \\ \text{selectable\_element}(r) &= \text{async\_blocked}(r) \rightarrow (ps(r)' \neq m(r)). \qquad (3.35) \end{aligned}$$

Eq. (3.33) is Eq. (3.28). It suffices to require Eqs. (3.26) and (3.28). Next, we prove that Eqs. (3.26) and (3.28) imply both Eqs. (3.34) and (3.35).

This claim is proved for a synchronous product as follows

$$\begin{aligned} \bigwedge_{r \in R_k} \text{selectable\_element}(r) &= \bigwedge_{r \in R_k} \Big(\text{element\_blocked}(r) \rightarrow \big(ps(r)' \neq m(r) \big) \Big) \\ &= \Big(\bigvee_{r \in R_k} \text{element\_blocked}(r) \Big) \rightarrow (ps(k)' \neq m(k)) \qquad (3.36) \\ &= \text{sync\_blocked}(k) \rightarrow (ps(k)' \neq m(k)), \end{aligned}$$

because all elements in the synchronous product k are selected by the same value $m(k)$ (for all $r \in R_k$, $m(r) = m(k)$) of the same variable $ps(r)$ (for all $r \in R_k$, $ps(r) = ps(k)$). For an asynchronous product, the claim is proved as follows

$$\begin{aligned}
\bigwedge_{r \in R_k} \text{selectable\_element}(r) &= \bigwedge_{r \in R_k} \big(\text{element\_blocked}(r) \rightarrow (ps(r)' \neq m(r))\big) \\
&= \bigwedge_{m(r)=0}^{nk-1} \big(\text{element\_blocked}(r) \rightarrow (ps(r)' \neq m(r))\big) \\
&= \big(\bigwedge_{r \in R_k} \text{element\_blocked}(r)\big) \rightarrow \bigwedge_{i=0}^{n_k - 1} (ps_k' \neq i) \\
&\overset{0 \leq ps_k \leq n_k}{=} \text{async\_blocked}(k) \rightarrow (ps_k' = n_k) \\
&\overset{Eq.\ (3.26)}{=} \text{async\_blocked}(k) \rightarrow (ps(k)' \neq m(k)).
\end{aligned} \tag{3.37}$$

Note how Eq. (3.26) was used to recurse one level above in the nesting of products, from the product ps_k to its "parent" product $ps(k)'$.

This completes our proof of Eqs. (3.33) to (3.35) (equiv. Eq. (3.29) under the definitions Eqs. (3.30) to (3.32)), using as premises only Eq. (3.28) and Eq. (3.26).

The nesting of products forms a tree, with a finite number of nodes. The leaves are processes, as mentioned earlier (otherwise the nesting would never terminate). This provides the base case, by ensuring that the asserted Eq. (3.28) holds for all elements of the bottom products. The induction step reduces the number of nodes in the tree for which it has been proved that Eq. (3.29) holds. Assume that for all nodes below a product node, Eq. (3.29) holds. By the preceding proofs, this implies that Eq. (3.29) holds also for that node. The finiteness of the syntax tree implies termination of the induction. The only products for which Eq. (3.29) was not proved are the top ones. The top products are not nested in any other product, so the condition is undefined. Instead, other conditions are imposed on the top products, as discussed in the next section.

3.4.2 Top products (asynchronous)

A system process can request to execute *atomically*, meaning that it will be granted uninterrupted execution, until it exits atomic context, because it either blocked (Eq. (3.38)) or reached non-atomic context (Eq. (3.7)).

$$\text{grant}_s \triangleq \bigwedge_{\substack{r \in \text{pids}(s) \\ \text{that have an atomic block}}} \left(\begin{pmatrix} (ex_s = m(r)) \wedge \\ \text{frozen\_unblocked}(r) \end{pmatrix} \rightarrow (ps(r)' = m(r)) \right) \tag{3.38}$$

$$\text{frozen\_unblocked}(r) \triangleq \begin{cases} \bigvee_{i \in N_r} \Big((pc_r = i) \wedge \bigvee_{(i,j,k) \in E_r} \text{guard\_test}(r, i, j, k) \Big), & \text{if player}(r) = s \\ \neg \text{blocked}(r), & \text{otherwise} \end{cases} \tag{3.39}$$

$$\text{guard\_test}(r, i, j, k) \triangleq \begin{cases} \text{guard}(r, i, j, k)|_{x/x' \text{ for } x \in \mathcal{X}}, & \text{if } i \text{ in atomic context} \\ \text{guard}(r, i, j, k), & \text{otherwise} \end{cases} \tag{3.40}$$

If a system process requests atomic execution, and its next transition is not blocked, then the environment must grant the request Eq. (3.38). For this test only, the environment must check for loss of atomicity in the case that nothing changed until the requestor attempts its next transition. What could change is the valuation of $\mathcal{X}'$, because the environment decides for loss of atomicity at the same time that it could change $\mathcal{X}'$. In other words, leaving primed environment variables in the guard expressions would allow the environment to force loss of atomicity by choosing a suitable valuation of $\mathcal{X}'$. By substituting unprimed for primed environment variables in guard(r, i, j, k), Eq. (3.40), the check of Eq. (3.39) corresponds to the case that the environment did not change anything in the meantime.

If the check fails, then the requestor is blocked, and atomicity is lost. In this case, the environment can play freely. Note that if atomicity is lost, then the next environment move could select values for $\mathcal{X}'$ that result in the requesting system process becoming unblocked, as decided by Eq. (3.11) (because the guard expressions in Eq. (3.11) have no renames). The scheduler could then select that same process (that requested atomic execution) to execute next, but is *not* required to do so by Eq. (3.38).

If all elements in the top product (asynchronous) of player p have blocked, then the scheduler sets variable ps_p to its maximal value n_p, as a consequence of the proof in Eq. (3.37). This holds for both the environment and system. It does not restrict what happens when the some element remains unblocked.

Unless we restrict the scheduler's behavior in that case, the environment could choose to remain always inactive, and also keep the system inactive forever. One way to avoid this is to impose fairness assumptions. Fairness assumptions are more expensive than safety assumptions. For this reason, we decide to impose safety constraints that ensure that no player stops executing, unless it either blocks, or the other player is executing atomically. Replacing these safety assumptions with weak fairness (using a GR(1) synthesis algorithm), or strong fairness (using a full LTL synthesis algorithm) is an interesting direction of future research, with a higher computational cost.

The safety constraints are as follows. If no environment process is executing, then the environment has deadlocked and loses the game, unless the reason is that some system process currently paused the environment processes, in order to execute atomically (Eq. (3.41)).

$$\text{pause\_env\_if\_req} \triangleq \begin{cases} (ps'_e = n_e) \leftrightarrow \big(pm_s \wedge (ps'_s = ex_s < n_s)\big), & \text{if sys has any atomic blocks} \\ ps'_e \neq n_e, & \text{otherwise} \end{cases} \tag{3.41}$$

The environment must execute some element in the top asynchronous product of the system, unless all elements have blocked

$$\text{sys\_deadlock} \triangleq (ps'_s = n_s) \rightarrow \bigwedge_{i \in \text{async\_top\_sys}} \text{element\_blocked}(i). \tag{3.42}$$

Finally, if no system process executes, then the system loses the game. By Eq. (3.42), this can only happen if the system has deadlocked.

$$\text{sys\_never\_deadlock} \triangleq ps'_s \neq n_s. \tag{3.43}$$

Labeled program statements induce labeled nodes in the program graph. If the label string contains the word "progress", then a recurrence constraint is added to the liveness assumption or assertion, as in Eq. (3.44).

$$\text{progress}(p) \bigwedge_{r \in \text{pids}(p)} \bigwedge_{i \in N_r^{\text{progress}}} \Box\Diamond\big((ps(r) = m(r)) \wedge (pc_r = i)\big). \tag{3.44}$$

A summary of the translation is given in Table 3.1, including the separation into assumption and assertion formulae.

3.5 Exclusive execution

If `atomic` blocks are included in system processes, then the above translation to remains valid provided that `ltl` properties do not contain primed variables, and `atomic` blocks cannot appear inside synchronous products. The issues with priming and atomic blocks are discussed in Section 2.6.4. Note that Section 2.6.4 discusses also the case of a preempted process that is in a synchronous product with the atomically executing process. In this section, the restriction of atomic blocks to processes in the top asynchronous product implies that such cases do not arise.

3.5.1 Stuttering and visibility

Safety LTL formulae in `ltl` blocks are deactivated during atomic execution (Eqs. (3.45) and (3.46)), as for Promela. In other words, a safety requirement does not apply to atomic execution (the implementation exposes an option to make atomic execution visible to safety properties). Note that they are re-activated as

Table 3.1: Summary of translation to temporal logic, separated into assumptions and assertions.

Equation	Assumption	Assertion	Purpose
Eq. (3.5)	if a is sys variable	if a is env variable	array index bound
Eq. (3.8)	if player$(r) = e$	if player$(r) = s$	data flow
Eq. (3.12)	if $pc_r \in \mathcal{X}$	if $pc_r \in \mathcal{Y}$	control flow
Eq. (3.14)		if player$(r) = e$ and $pc_r \in \mathcal{Y}$	control flow
Eq. (3.15)		yes	sys init
Eq. (3.16)	yes		env init
Eq. (3.17)		yes	$\hat{pc}_r$ init
Eq. (3.18)	for $p = e$	for $p = s$	local free var inv
Eq. (3.22)	for $p = e$	for $p = s$	imperative var default inv
Eq. (3.23)	for $p = e$	for $p = s$	ranged integer safety
Eq. (3.26)	yes		(async) product nesting
Eq. (3.28)	yes		unblocked processes selectable
Eq. (3.38)	yes		grant if unblocked, else loses atomicity
Eq. (3.41)	yes		pause env only if preempted by sys
Eq. (3.42)	yes		pause sys only if blocked
Eq. (3.43)		yes	sys loses if paused
Eq. (3.44)	for $p = e$	for $p = s$	progress labels
Eq. (3.45)	yes		mask env LTL
Eq. (3.46)		yes	mask sys LTL

soon as atomicity is lost.

$$\text{mask\_env\_ltl} \triangleq \text{ite}\big(pm_s \wedge (ps'_s = ex_s < n_s),\ \text{freeze\_env\_free},\ \bigwedge_{\psi \in \mathsf{safety}(\mathtt{ltl}, e)} \psi\big) \tag{3.45}$$

$$\text{mask\_sys\_ltl} \triangleq (ps'_s = ex_s < n_s) \vee \bigwedge_{\psi \in \mathsf{safety}(\mathtt{ltl}, s)} \psi \tag{3.46}$$

If $i = e$ ($i = s$), then the formula safety(`ltl`, e) denotes the safety assumptions (assertions) that are declared inside `ltl` blocks.

3.5.2 Unbounded loops

If unbounded loops appear inside atomic context, then there can be no liveness assumptions. The reason is that the system can "hide" in atomic execution forever, preventing the environment from satisfying its liveness assumptions, thus winning trivially. Requiring that the system infinitely often exit from its atomic execution is insufficient, because the system is discharged of the recurrence assertion if it can force the environment to violate the recurrence assumption. In order to avoid this, the environment recurrence goals must be disjoined with a persistence property to obtain $\Box\Diamond$(env goal) $\vee$ $\Diamond\Box$(sys in atomic execution), a strong fairness formula. Persistence ($\Diamond\Box$) is outside GR(1).

Chapter 4

Implementation

The implementation is written in PYTHON and available [74, 75] under a BSD license. It comprises of a translator from open PROMELA to linear temporal logic with both (bounded) integer and Boolean variables, and an encoder of bounded integer arithmetic into propositional temporal logic. The result is used as input to the GR(1) synthesis tool `slugs` [36].

The front-end uses a lexer and parser written with `PLY` (Python lex-yacc) [76] that takes a `Promela` program source and returns an abstract syntax tree. The abstract syntax tree is then translated to a program graph. The program graph is a directed graph that has nodes that represent program states and edges labeled with program statements.

The program graph is flattened to logic in two passes by the compiler architecture as shown in Fig. 4.1. The first pass converts each statement that labels an edge to a guard formula and an action formula, producing a semi-symbolic representation. The second pass flattens the graph into a transition relation over primed and unprimed variables, introducing program counters and keys to distinguish multiedges. This flattening is applied to the program graph of each process to yield the formulae defined in Section 3.2.

These formulae are used to define the process scheduler, as described in Sections 3.4 and 3.5. Note that (orthogonally) the previous formulae are separated into an assumption clause and an assertion (guarantee) clause, as described in Table 3.1. The environment is obliged to satisfy the assumption and the system the assertion.

Up to this stage, the formulae are over both Boolean and (bounded) integer variables with modular arithmetic. In a second stage, the formulae are parsed and modular arithmetic is encoded into propositional logic, discussed in Section 4.2. The result is a propositional temporal logic formula that can be given as input to the `slugs` synthesizer. The synthesizer takes the formula and checks realizability using binary decision diagrams (BDDs) [18]. If the problem described by the program is realizable, then the synthesizer produces a winning strategy in the form of a Mealy machine. The Mealy machine can be represented either as a BDD, or as an enumerated graph (the latter is currently used). Note that the algorithm solves a two-player game of perfect information, in other words, every process can observe what other processes are doing internally and each player controls all its variables in a centralized way (even though these variables may be local to different system processes).

An example is shown in Figs. 4.5a and 4.5b for the process

```
assert active sys proctype foo(){
    do
    :: (x > 0); x = x - 1
    :: (x < 3); x = x + 1
    :: (y > 0); y = y - 1
    :: (y < 2); y = y + 1
    :: skip
    od
}
```

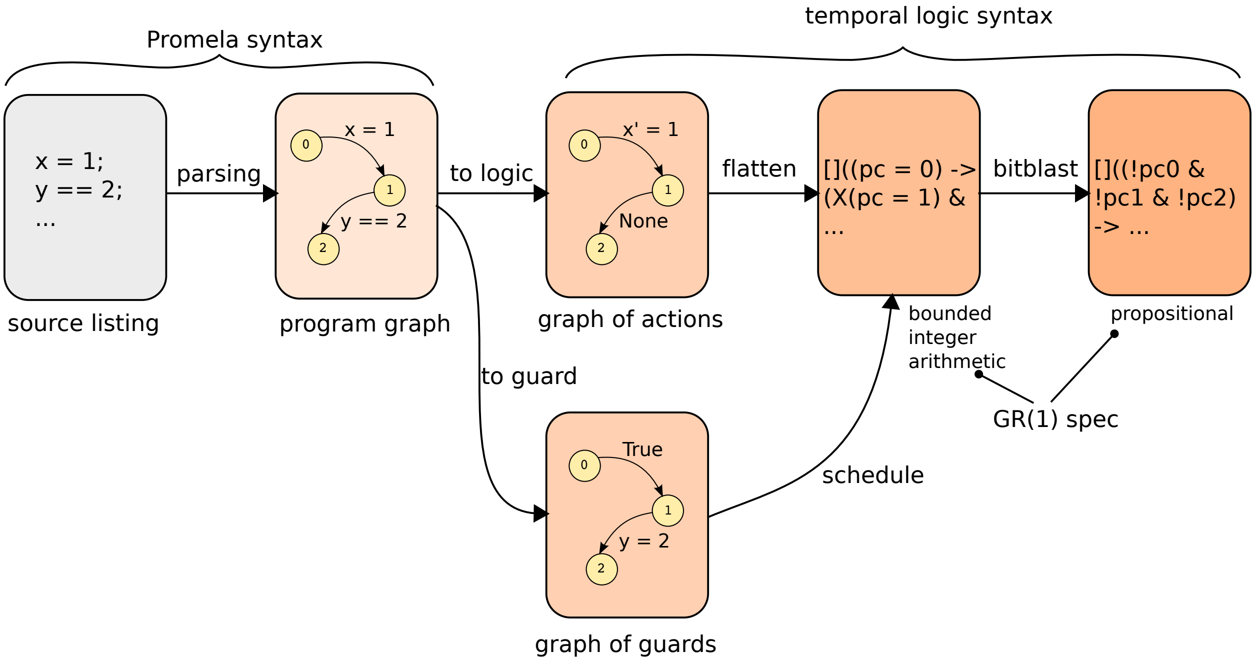


Figure 4.1: The compiler architecture that translates Promela source code to a propositional linear temporal logic specification that belongs to the fragment of generalized reactivity of rank 1.

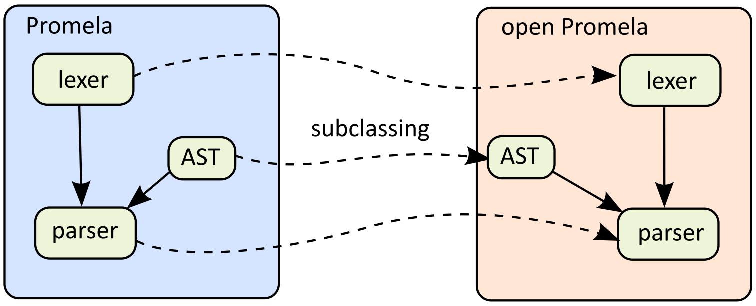


Figure 4.2: Parser front-end: the open `Promela` lexer and parser are subclasses of the pure `Promela` lexer and parser.

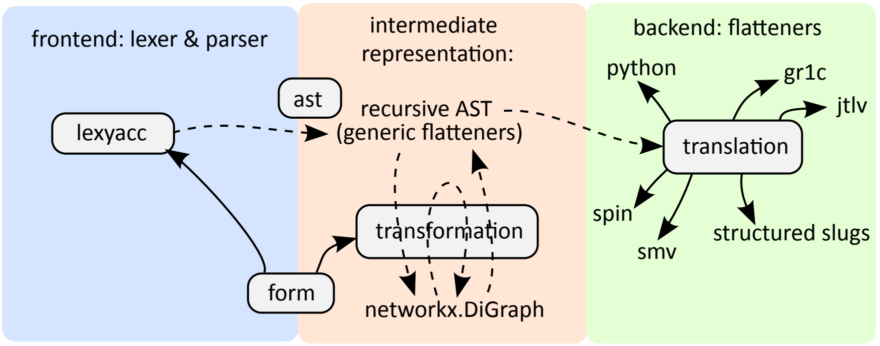


Figure 4.3: Architecture of subpackage `tulip.spec`.

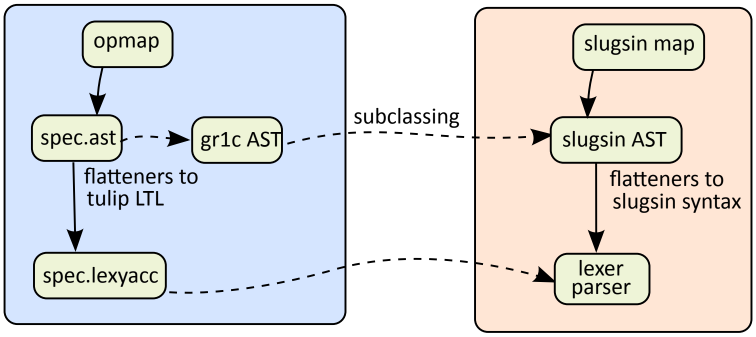
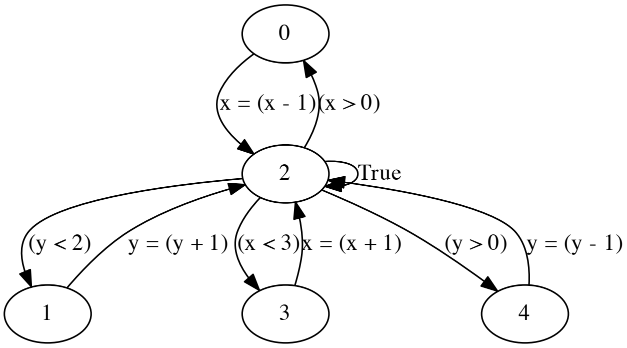
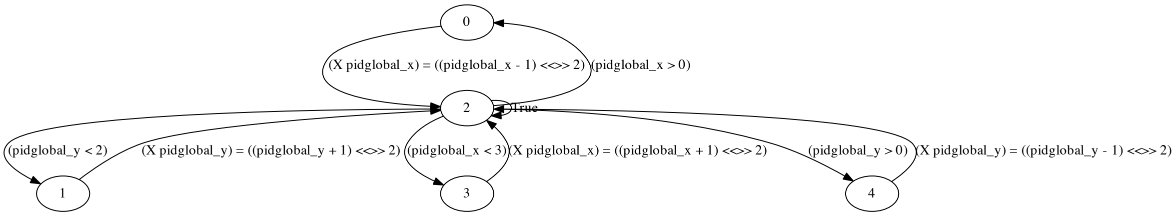


Figure 4.4: The logic AST in (sub)package `spec` is produced by a factory function. Passing a different operator map to this function results in AST nodes with flatteners to a different target syntax. In addition, node classes can be subclassed after being produced by the factory. The main reason for subclassing an AST node class is to override its `flatten` method.



(a) Program graph for an example process.



(b) Graph annotated with action formulae. The symbol "`<<>>`" denotes truncation of the left operand to the bitwidth defined by the right operand.

Figure 4.5: Program graph and graph of logic formulae.

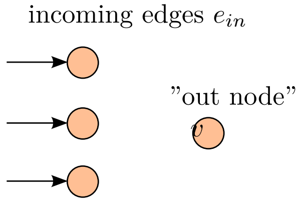


Figure 4.6: During recursive conversion to a program graph, each statement returns a set of incoming edges, together with an out-node.

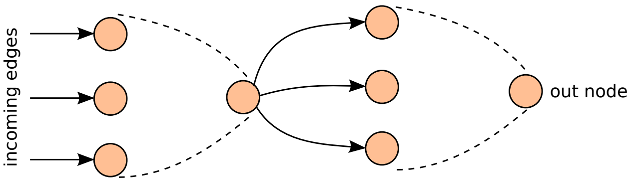


Figure 4.7: A sequence statement is the connection of its elements.

4.1 Program graph construction

The program graph is constructed by recursively converting the syntax tree to nodes and edges. There are two categories of AST nodes: those that define nodes and edges in the program graph, and those that represent expressions. Expressions comprise of operators (with arity 1 or 2) and terminals (variable references, integers, Boolean nullary connectives). Most other AST nodes represent part of the program graph.

We can distinguish roughly four types of AST nodes that define control flow:

1. edge: assignment, expression statement[1], assertion, `else` statement, `printf`, receive, send
2. simple paths: sequence
3. branching and merging: options (`do`, `if`)
4. jumps: `break`, `goto`, labels

The lists above are not exhaustive of all `Promela` constructs. Note that jumps can be regarded also as a special form of branching: two edges with common source node, and one edge guarded by `false`.

In order to convert an AST to a program graph, we need to define what nodes and edges result from each of the above four cases. During the recursion, each of the above returns a pair (e_{in}, v) of incoming edges $e_{in} \subseteq E_r$ and out-node v.

A statement that corresponds to an edge is converted to a fresh target node v, together with its incoming edge $(*, v, \mathrm{stmt})$ that has a yet unspecified source node and is labeled with the statement, Fig. 4.6.

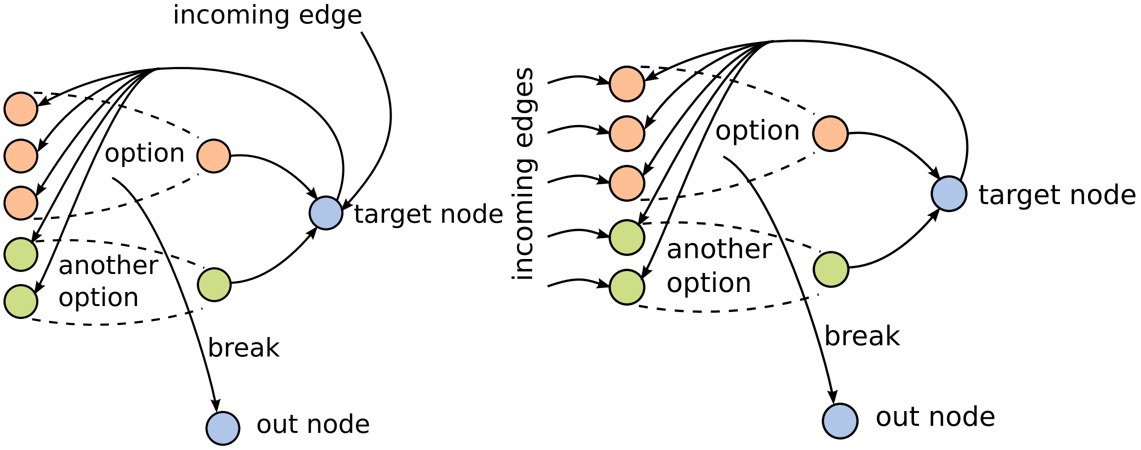
A sequence $\mathrm{stmt}_0, \mathrm{stmt}_1, \ldots, \mathrm{stmt}_n$ assembles the edges that it contains, by connecting each out-node v with the incoming edges of its successor. The result is again a pair $(e_{in,0}, v_n)$, where $e_{in,0}$ are the incoming edges of the first element stmt_0 and v_n the out-node of the last element stmt_n, Fig. 4.7. Sequences are used to define a context (the body of an `atomic` or `d_step` block) and to mark option guards as such[2].

Selection and repetition statements have similarity in their conversion. A fresh target node v is created first. For each option sequence, its incoming edges $e_{in,j}$ are collected in e_{in} and the out-node v_j is connected to the target node v, by a new `goto` edge (v_j, v). For a repetition construct, backward edges are also added, by connecting the target node v as source for the incoming edges $e_{in,j}$.

---

[1] In pure `Promela`, expression statements are known as *conditions*, implying that they are guards. However, in open `Promela` an expression statement can contain primed variables, so it is an action. Actions are not pure guards, so they are called *expression statements*. See Section 2.3.4 for details of how guards are defined for actions.

[2] If a sequence is an option in a selection or repetition construct, then its first element is an *option guard*. Note that the first element may comprise of options, and in that case repetition and selection constructs yield slightly different nodes and edges.



(a) Repetition construct that is not an option guard.

(b) Repetition construct that is an option guard.

Figure 4.8: Possible cases for a repetition construct.

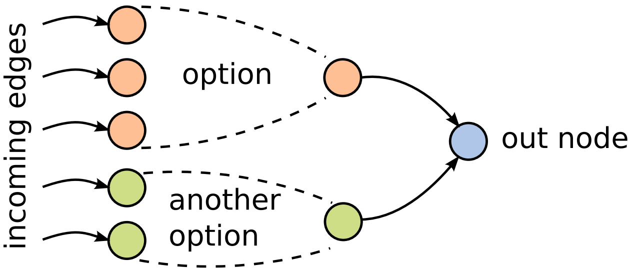


Figure 4.9: Selection construct.

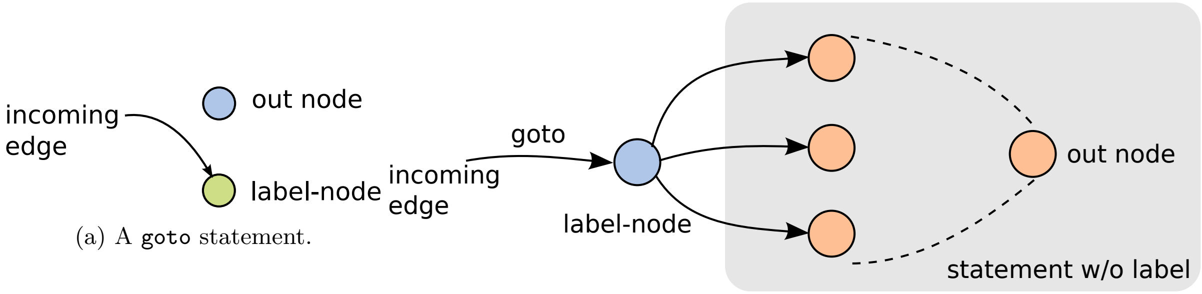
The result is a pair of incoming edges with an out-node. For an `if` statement, the out-node is the target node v and the incoming edges are those collected in e_{in} for the options, Fig. 4.9. For a `do` statement, the out-node is a fresh node w different than the target node v, Fig. 4.8. Any `break` statements inside a `do` statement are equivalent to `goto` statements that lead to w (of the innermost `do` statement, in case of nested repetitions), Fig. 4.8. If a `do` statement is *not* itself an option guard, then a `goto` to the target node v is returned as incoming edge (instead of the collected e_{in}), Fig. 4.8a. Otherwise, the edge set e_{in} is returned, as for an `if` statement, Fig. 4.8b.

It remains to describe how jumps contribute to the program graph, Fig. 4.10. As already remarked, a `break` statement is equivalent to a `goto` to the out-node of the (immediate) enclosing repetition structure. A `goto S0` statement results in a pair (e_{in}, v) of fresh out-node v and a single incoming edge labeled with "goto" and the label-node u_{S0} as target, corresponding to the label `S0`, Fig. 4.10a.

Note that in the special case that a `break` or `goto` statement appears as option guard, then a condition statement `true` is used to annotate the incoming edge, instead of "goto". This ensures that the edge will not be contracted later.

A labeled statement, as for example `S0:  x = 1`, is treated very similarly to a `goto` statement. Let (e_{in}, v) be the incoming edges and out-node of the statement that is labeled, here $x = 1$. The label-node u_{S0} is created and connected as source node to the edges e_{in}. A single edge labeled with "goto" and with target node u_{S0} is returned as incoming edge for the labeled statement. The node v is returned as out-node for the labeled statement. The result is shown in Fig. 4.10b.

Some of the conversions described so far produce edges labeled with "goto". In order to obtain the final program graph, these edges are contracted. A necessary condition for contraction is that the source node have a unique outgoing edge, so that after the contraction, no new outgoing edges be added to the target node (unless the target node also has a unique incoming edge – the edge that will be contracted). This



(a) A `goto` statement.

(b) A labeled statement.

Figure 4.10: Jump statements.

condition is satisfied by the edges annotated with `goto`, because the only case that violates this condition is a `goto` statement that appears immediately after branching. Such a `goto` statement must be an option guard, and as noted earlier, it is replaced by a `true` condition statement.

Besides control flow, context can be defined by enclosing blocks of statements in braces preceded by the keywords `atomic` or `d_step`. Parsing represents a block of statements as a sequence, so the context is stored in the corresponding AST node. All program graph nodes produced by statements inside the sequence are marked with the enclosing context. Note that `goto` statements add edges to special label-nodes (u_{S0} above), but do not create the label-nodes. The context of a label-node is determined by the context of the corresponding labeled statement.

This completes the conversions of statements to program graph edges and nodes. In addition to control flow, variable definitions are collected and used to populate the table of symbols.

4.2 Bitwise encoding

The `Promela` program is translated to logic formulae that contain (bounded) integer arithmetic. They are encoded in logic formulae over bitfields by a translation commonly known as *bitblasting*. Both signed and unsigned integers are supported and represented in two's complement. The operations currently supported are:

- addition
- subtraction
- multiplication
- quotient and remainder
- truncation to given constant width
- (dynamic) shift of a bitfield by a bitfield
- (static) shift of a bitfield by a given constant

The implementation uses the abstract syntax tree (AST) for first-order temporal logic available in the `omega.logic` subpackage. For each AST node class, a subclass is defined that overrides the `flatten` method. These flattening methods translate temporal logic (temporal because it involves primed variables) with signed arithmetic of bounded integers to the input syntax of the synthesizer `slugs`.

The truncation operator is necessary for representing the effect of assignment to variables with mod-wrap semantics, as in pure `Promela`. The distinction between assignment and equality results from the language definition. In `Promela`, each expression is evaluated by:

1. represent all values as signed integers in `C`,

2. apply all operations in the expression to the precision allowed by the machine's arithmetic-logic unit (ALU) and the conventions defined by the `C` standard.

After the above steps for evaluating an expression have been completed, then there are two cases:

1. the expression is a condition statement, so use its truth value (defined as equality to zero)
2. a variable is assigned the value defined by this expression.

In the first case, it follows from the evaluation rules that full machine precision applies. So truncation occurs when the bitfield width required for operations reaches either the value 32 or 64, depending on the hardware and operating system.

In synthesis applications, current capabilities are limited to domains much below 32 bit integers. This suggests that the ALU width is not expected to be reached frequently. However, the result of a multiplication requires a bitfield with width equal to the sum of widths of its operands. This implies that nested multiplications increase the required ALU bitwidth exponentially. For example, a 5-bit unsigned integer variable can lead to overflow if it appears inside three nested multiplications. In any case, the truncation operator can be applied as normal when this occurs.

The second case, involving assignment, is more interesting. It also constitutes the essential difference between the assignment:

```
x = y + 1
```

and the equality condition obtained by replacing the assigned variable with the corresponding primed variable, and the assignment statement with the equality operator:

```
x' == y + 1
```

In pure `Promela`, assignment *truncates* the right-hand side to the width of the datatype representing the variable x on the left-hand side. Let $\mathrm{width}(x)$ denote the width of the bitfield representing variable x. For a given variable declaration, the value $\mathrm{width}(x)$ is fixed. For a bitfield $a \in \mathbb{B}^l$ and an integer constant k with $0 < k \leq l$, the truncation operator trunc defined as

$$\mathrm{trunc}(a, k) \in \mathbb{B}^k \quad \text{and} \quad \forall j \in \mathbb{N}. j \leq k \implies \mathrm{trunc}(a, k)_j = a_j. \tag{4.1}$$

This says that $\mathrm{trunc}(a, k)$ is a bitfield of width $k \leq l$, and each bit $\mathrm{trunc}(a, k)_j$ is equal to the corresponding bit a_j of the given bitfield a.

Using these definitions, the previous assignment statement is equivalent to the logic formula

$$x' = \mathrm{trunc}\left(y + 1,\ \mathrm{width}(x)\right)$$

It becomes evident that the assignment `x = y + 1` differs from the action `x' == y + 1`, because, in general, the formula $\mathrm{trunc}\left(y + 1, \mathrm{width}(x)\right)$ is different than the formula $y + 1$. From a synthesis perspective, the assignment is always realizable, because the result is truncated according to the width of x. In contrast, if the value of $y + 1$ does not fit the width of the bitfield representing x, then the formula $x' = y + 1$ is not realizable. The reason is that the system fails to match the value of $y + 1$ by x, because x cannot take sufficiently large values.

4.2.1 Modwrap and saturating semantics

Assignment in C and PROMELA truncates the right hand side expression, before setting the value of the left hand side variable to be equal to the right hand side expression. This is called mod-wrap semantics. For variables whose domain corresponds to a bitfield, modwrap semantics are computationally viable.

We defined a ranged integer data type. Except for ranged integers with minimal value 0 and maximal value that is a power of two, the rest cannot take all the values that the underlying bitfield can. This raises the question of how we should interpret an assignment to a ranged integer.

We choose to not apply any truncation to a value assigned to a variable with saturating semantics. The motivation is computational efficiency of the induced BDD operations. Next, we discuss the reasons that led to this decision.

Suppose that we decided to truncate, using as base the interval range $\max(\mathrm{dom}(x))$ (suppose for now that $\min(\mathrm{dom}(x)) = 0$). If the range is a power of two, then modulo a power of two corresponds to truncation. Truncation does not increase the size of the bitvector formula. Unlike base two, modulo an arbitrary base requires BDDs of exponential size [18]. Therefore, this natural choice for truncation semantics is not computationally viable.

The remaining option is to define the semantics using modulo operations with base two. One option is to truncate to the width of the underlying bitfield, and if the result is outside the range, then that causes a safety violation. This semantics is confusing, because it is an indirect definition with little relation to the variable's actual domain. The other option is to first truncate modulo the binary representation, then subtract the range (the result is ensured to belong to the variable's domain, so assignment is always possible in this case). The problem with this approach is, again, its confusing semantics.

In summary, ranged integers serve as a shorthand for a longer bitfield, together with a safety formula, and deactivation of truncation during assignments.

4.2.2 Bitfield representation of singed ranged integers

A ranged integer declared as `int(a, b) x` is represented as a bitvector as follows. Let $a, b \in \mathbb{Z}$ be the endpoints of the interval $[a,\ b]$ over which the integer can take values. Define the magnitude $m \triangleq \max\{|a|\ ,\ |b|\}$. At least $\xi \triangleq \lceil \log_2 m \rceil$ bits are required to represent m with an unsigned bitfield.

If $a < 0$ and $0 < b$, then x has varying sign, so a bitfield of width $\xi + 1$ is used. This bitfield contains the representation of an integer in two's complement. This integer takes values over $\mathbb{N} \cap [-2^{\xi},\ 2^{\xi} - 1]$. To restrict this interval to the subset $[a,\ b]$, the safety constraint $\square(a \leq x \leq b)$ is imposed.

Otherwise, $a \leq 0$ and $b \leq 0$, or $0 \leq a$ and $0 \leq b$, so x has fixed sign. In this case, a bitfield of width ξ is used. This bitfield contains ξ bits of the two's complement representation. It does not contain the sign bit, because it is constant. In this case, before using the bitfield in bitvector formulae, the (fixed) sign bit is appended to it. To restrict the resulting integer to the subset $\mathbb{N} \cap [a,\ b]$, again the safety constraint $\square(a \leq x \leq b)$ is imposed.

4.2.3 Arithmetic as bitvector logic

Addition and subtraction All arithmetic operations can be computed using as main element an adder-subtractor, Fig. 4.13, that comprises of half-adders, Fig. 4.12. An adder-subtractor can do both addition and subtraction. A half-adder computes the functions

$$\begin{aligned}
\mathrm{sum}(a, b, c_{in}) &\triangleq a \oplus b \oplus c_{in} \\
\mathrm{carry}(a, b, c_{in}) &\triangleq (a \wedge b) \vee \big((a \oplus b) \wedge c_{in}\big),
\end{aligned} \tag{4.2}$$

where $\oplus$ denotes exclusive disjunction (XOR), i.e., $x \oplus y \triangleq (x \wedge \neg y) \vee (\neg x \wedge y)$. An adder-subtractor can perform both addition and subtraction (by setting the input signal c_{in}). The result is

$$\mathrm{addsub}(a, b, c_{in}) \triangleq a + (1 - 2c_{in})b. \tag{4.3}$$

Relational operators An adder-subtractor can be used to compute equality and strict inequality as follows

$$\begin{aligned}
a = b &\iff \bigwedge_{i=0}^{l-1} \neg(a_i \oplus b_i) \\
a < b &\iff \mathrm{addsub}(a, b, 1).c_{out} \oplus \neg(a_{l-1} \oplus b_{l-1}).
\end{aligned} \tag{4.4}$$

Negated equality and non-strict inequality are obtained by negating the previous

$$\begin{aligned}
a \neq b &\iff \neg(a = b) \\
a \geq b &\iff \neg(a < b).
\end{aligned} \tag{4.5}$$

By symmetry, these suffice to compute $a > b$ and $a \leq b$. This is a brief overview (including errata) of the encoding described in [77], where more can be found, e.g., about sign extension.

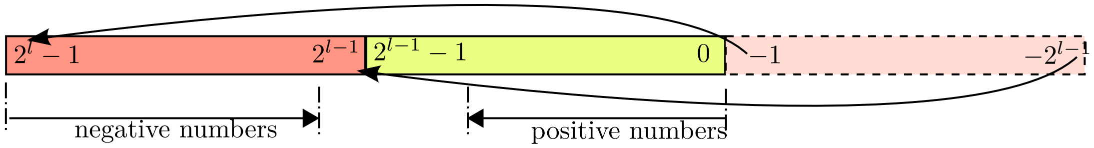


Figure 4.11: Two's complement on the line of integers.

Two's complement The circuit of Fig. 4.13 becomes a subtractor if the carry input bit c_{in} is 1. This value of c_{in} triggers the computation of two's complement for the second input b. Computing two's complement is equivalent to bitwise negation plus one. Therefore, the carry-in c_{in} plus the bitwise negation of the representation in two's complement of b is the representation in two's complement of $-b$. This turns addition of a and b to addition of a and $-b$, thus subtraction.

Note that 2^l requires $l+1$ bits to be represented. It holds that

$$2^l = 0b\underbrace{10\dots0}_{l+1} = (0b\underbrace{011\dots1}_{l+1}) + 1 = (0b\underbrace{11\dots1}_{l}) + 1, \tag{4.6}$$

where the prefix $0b$ signifies base two notation. A positive number $0 < x < 2^{l-1}$ in two's complement representation is

$$0b\,\underbrace{x_{l-1}x_{l-2}\dots x_0}_{l} = 0b\,\underbrace{0x_{l-2}\dots x_0}_{l}, \tag{4.7}$$

because $0 < x < 2^{l-1} \implies x_{l-1} = 0$. A negative number x with $-2^{l-1} \leq x < 0$ in two's complement representation is (Fig. 4.11)

$$\begin{aligned} 2^l - |x| &= \big(0b\underbrace{11\dots1}_{l}\big) - \big(0b\underbrace{|x|_{l-1}\dots|x|_0}_{l}\big) + 1 \\ &= \big(0b(1-|x|_{l-1})\dots(1-|x|_0)\big) + 1 \\ &= \big(0b(\neg\,|x|_{l-1})\dots(\neg\,|x|_0)\big) + 1. \end{aligned} \tag{4.8}$$

By the bounds on x, it is

$$\begin{aligned} &-2^{l-1} \leq x < 0 \iff 0 \leq 2^{l-1} + x < 2^{l-1} \iff 2^{l-1} \leq 2^{l-1} + 2^{l-1} - (-x) < 2^{l-1} + 2^{l-1} \iff \\ &2^{l-1} \leq 2^l - |x| < 2^l \implies (2^l - |x|)_{l-1} = 1 \end{aligned} \tag{4.9}$$

Up to this point we have noted the conversion from binary magnitude and separate bit sign representation to two's complement representation.

Next, it is shown that two's complement negates a number x. In other words, the two's complement of the representation y of x in two's complement yields the two's complement z of the negated $-x$. In addition, it is shown that it corresponds to the elementwise negation and carry-in $c_{in} = 1$ that turns a ripple-carry adder into a subtractor.

$$\begin{aligned} 2^l - (0by_{l-1}\dots y_0) &= \big(0b\underbrace{1\dots1}_{l}\big) + 1 - \big(0by_{l-1}\dots y_0\big) \\ &= \big(0b(1-y_{l-1})\dots(1-y_0)\big) + 1 \\ &= \big(0b(\neg y_{l-1})\dots(\neg y_0)\big) + c_{in}. \end{aligned} \tag{4.10}$$

This is exactly the operation performed when $c_{in} = 1$. If x is positive, then its two's complement y has $y_{l-1} = 0$ and $|x| = 0by_{l-2}\dots y_0$. As already shown, the above (z) is the two's complement of $-|x|$. If x is negative, then its two's complement y has $y_{l-1} = 1 \implies \neg y_{l-1} = 0$, and $0b(\neg y_{l-2}\dots(\neg y_0)) + 1 = 2^{l-1} - (2^l - |x| - 2^{l-1}) = 2^{l-1} + 2^{l-1} - 2^l - |x| = -|x|$.

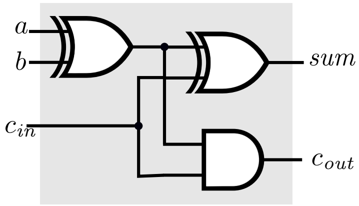


Figure 4.12: Full adder.

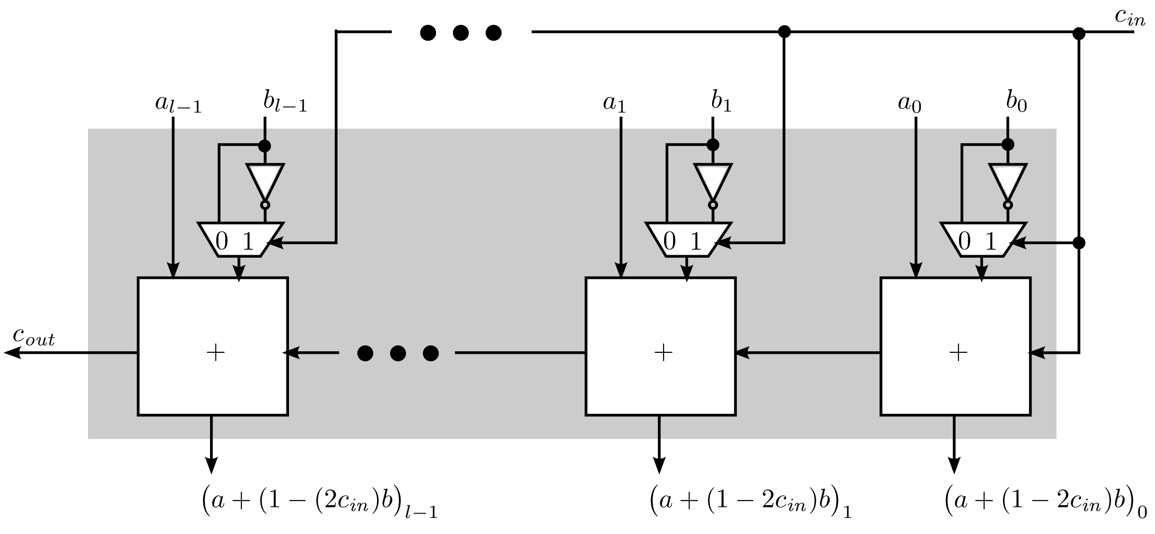


Figure 4.13: Ripple-carry adder-subtractor.

Multiplication A shift-and-add circuit as described[3] in [77] is used. Given signed bitfields u, v, the stages of the multiplier are defined recursively as follows

$$\begin{aligned}
n &\triangleq \mathrm{width}(u) + \mathrm{width}(v) \\
a &\triangleq \mathrm{extend}(u, n) \\
b &\triangleq \mathrm{extend}(v, n) \\
\mathrm{multiplier}(a, b, s) &\triangleq \begin{cases} 0_n, \text{ if } s = -1 \\ \mathrm{multiplier}(a,\ b,\ s-1) + \lambda j.\ (b_s \wedge (a \ll s)_j), \text{ if } s \in \{0, 1, \ldots, \mathrm{width}(b) - 1\} \end{cases}
\end{aligned} \tag{4.11}$$

where $\mathrm{extend}(x, w)$ denotes the sign extension of bitfield x from l to w bits, by appending bits $x_{l+1}, \ldots, x_{w-1}$, each one equal to the sign bit x_{l-1}. The index s identifies the stage in the multiplier. By 0_n we denote a bitfield of width n with all bits equal to 0. Note that $\mathrm{width}(b)$ must be used in $\mathrm{multiplier}(a, b, s-1)$ to address the case of negative v (whose sign extension to width n contains 1 in the $\mathrm{width}(u)$ most significant bits $b_{n-1}, b_{n-2}, \ldots, b_{n-\mathrm{width}(u)}$).

Division Division of integers is implemented using non-restoring division of positive integers. Non-restoring division of positive integers is implemented using memory buffers as follows. Assume that the integers u and v are given in two's complement representation. The absolute value can be computed as

$$|x| \triangleq \mathrm{ite}(\mathrm{sgn}\,(x)\,,\ 0 - x,\ x) \tag{4.12}$$

where an extension by 1 bit (to $\xi + 1$ bits) ensures that $x = -2^{\xi}$ with $|x| = 2^{\xi}$ does not overflow. Define the rectified operands

$$a \triangleq |u| \quad d \triangleq |v|\,. \tag{4.13}$$

So a is the dividend and d the divisor for the unsigned division. Let q denote the quotient, and r the remainder. Let $n \triangleq \mathrm{width}(a)$ and $\hat{d} \triangleq d \ll n$. The multiplier comprises of stages that are defined recursively as follows

$$\begin{aligned}
p_{-1} &\triangleq \mathrm{pad}(a,\ 2\,\mathrm{len}(a)) \\
r_s &\triangleq (p_{s-1} \ll 1) - \hat{d} \\
q_s &\triangleq \neg\mathrm{sgn}\,(r_s) \\
p_s &\triangleq \mathrm{ite}(q_s,\ r_s,\ p_{s-1} \ll 1)
\end{aligned} \tag{4.14}$$

where stage $s \in \{-1, 0, \ldots, n-1\}$, q_s is the s^{th} quotient bit, and p_s the partial remainder (at stage s). This circuit yields both the quotient and remainder, used to implement division and modulo. The remainder and divisor have the same sign, as in C99 semantics, §6.5.5 [78].

The signs are restored after the division stage

$$\begin{aligned}
\mathrm{quotient}(u, v) &\triangleq \mathrm{ite}(\mathrm{sgn}\,(u) \oplus \mathrm{sgn}\,(v)\,,\ -q,\ q) \\
\hat{p} &\triangleq \lambda j \in \{0, 1, \ldots, n-1\}.\ p_{n+j} \\
\mathrm{remainder}(u, v) &\triangleq \mathrm{ite}(\mathrm{sgn}\,(u)\,,\ 0 - \hat{p},\ \hat{p}).
\end{aligned} \tag{4.15}$$

Note that $\hat{p} = \mathrm{trunc}(p \gg_{\mathrm{arithmetic}} n,\ n)$.

---

[3] Eqs.6.50–6.51 in [77], with 0 in 6.50, and explicitly noting the initial sign extension.

Appendix A

Appendices

A.1 Open `Promela` syntax

A.1.1 Lexemes

The following new keywords have been introduced:

- `env`, `sys` for declaring variables and control flow controlled by the environment and system, respectively,
- `free` for declaring symbolic variables,
- `assume`, `assert` for declaring data flow constrains on the environment and system, respectively,
- `sync`, `async` for declaring synchronous and asynchronous products.

The new operators are:

- PRIME “`'`” as postfix “next” operator for variables ($\bigcirc$),
- `-X` “weak previous”,
- `--X` “strong previous”,
- `-[]` “historically”,
- `-<>` “once”,
- `S` “since”.

A.1.2 Grammar

The following new parser production rules have been introduced (in union with existing production rules for the same nonterminals). Upper case denotes new keywords, or tokens like PRIME. Nonterminals are to be found in the language reference [42] and in the `promela.yacc` module of the parser in PYTHON.

⟨*ltl*⟩ ::= [ASSUME | ASSERT] LTL ‘`{`’ ⟨*expr*⟩ ‘`}`’

⟨*module*⟩ ::= ⟨*async*⟩ | ⟨*sync*⟩

⟨*async*⟩ ::= [ENV | SYS] ASYNC ‘`{`’ ⟨*async-body*⟩ ‘`}`’

⟨*async-body*⟩ ::= [⟨*async-body*⟩] ⟨*async-unit*⟩

⟨*async-unit*⟩ ::= ⟨*sync*⟩ | ⟨*proc*⟩ | ⟨*one-decl*⟩ | ⟨*semi*⟩

Table A.1: The use case of a keyword `step` as a non-deterministic uninterruptible block.

		blocking	
		admissible	raises error
det/ic	No	`atomic`	`step`
	Yes	`atomic`	`d_step`

⟨*sync*⟩ ::= [ENV | SYS] SYNC ‘`{`’ ⟨*sync-body*⟩ ‘`}`’

⟨*sync-body*⟩ ::= [⟨*sync-body*⟩] ⟨*sync-unit*⟩

⟨*sync-unit*⟩ ::= ⟨*async*⟩ | ⟨*proc*⟩ | ⟨*once-decl*⟩ | ⟨*semi*⟩

⟨*proctype*⟩ ::= [ENV | SYS] PROCTYPE

⟨*prefix-proctype*⟩ ::= [ASSUME | ASSERT] ⟨*prefix-proctype*⟩

⟨*one-decl*⟩ ::= [FREE] [ENV | SYS] [⟨*typename*⟩ | ⟨*name*⟩] ⟨*var-list*⟩

⟨*typename*⟩ ::= ‘`int`’ ‘`(`’ ⟨*number*⟩ ‘`,`’ ⟨*number*⟩ ‘`)`’

⟨*expr*⟩ ::= ⟨*expr*⟩ PRIME

A.1.3 The keyword `step`

The two statements `atomic` and `d_step` serve three of the possible use cases, as depicted in Table A.1. The `d_step` without the prefix `d_` is a natural candidate for denoting strictly uninterrupted execution that can exhibit non-determinism, i.e., blocks of the form `verb{...}`. Such a block allows conditioning the selection of a block of statements on the a priori executability of all statements in the block. An atomic block cannot used for this purpose, because it can be interrupted. A `d_step` may be undesirable, to leave design freedom to the synthesizer. The `step` keyword is currently not implemented, but may be introduced in the future.

A.2 Relevant work

Our approach has common elements with program repair [79], program sketching [80], and syntax-guided synthesis [81]. Program repair aims at modifying an existing program in a conventional programming language. Syntax-guided synthesis uses a grammar to "slice" the admissible search space of terminating programs. Here, we are interested in reactive programs. Similarly, program sketching uses templates to restrict the search space and give hints to the synthesizer for obtaining a complete program.

The translation from PROMELA to declarative formalisms has been considered in [70, 71, 69] and decision diagrams in [82]. These translations aim at verification, do not have LTL as target language, and either have limited support for atomicity [69], no details [70], or do not handle program graphs correctly [71].

TLA [11, 46] subsumes the language proposed here, since it includes quantification, but is intended as a theorem proving activity, is declarative, and is aimed at verification. Nonetheless, one can view the proposed translation as from (open) PROMELA to TLA. SMV is a declarative language with "next" operator, as well as synchronous and asynchronous processes, but no notion of assumptions/assertions, controllability, or imperative variables [73, 47]. JTLV [33] takes as input an SMV-like language for synthesis specifications, but with no imperative constructs. ASPECTLTL is a further declarative extension for aspect-oriented programming [83].

RPROMELA is an extension of PROMELA that adds synchronous-reactive constructs (not in the sense of reactive synthesis) that include synchronous products and channels called ports [84, 85]. Its semantics are defined in terms of *stable states*, where the synchronous product blocks, waiting for message reception from its global ports. RPROMELA does not address modeling of the environment, nor declarative elements.

Synchronous-reactive languages can be considered as potential candidates for expressing synthesis specifications. These include imperative textual languages like ESTEREL, ESTEREL-C, JAVA-ESTEREL, QUARTZ, REACTIVE-C, imperative graphical languages like STATECHARTS, ARGOS, and SYNCCHARTS, and declarative textual ones like LUSTRE and LUCID SYNCHRONE, and declarative graphical ones like SIGNAL [86, 68]. These languages are by definition *deterministic*, intended for direct design of transducers [86, 68, 87]. In synthesis, non-determinism is an essential feature of the specification. Extending ESTEREL with non-determinism has been considered, but in the context of transducers and verification [88], and using guarded command languages like PROMELA as example.

The approach proposed here has common elements with *constraint imperative programming* (CIP), introduced with the experimental language KALEIDOSCOPE [89, 90, 91, 92], one of the first attempts to integrate the imperative and declarative constraint programming paradigms. An observation from [89], which applies also here, is that specifiers need to express two types of relations: long-lived (best described declaratively), and sequencing relations (more naturally expressed in an imperative style). However, CIP does *not* ensure correct reactivity, because the constraints are solved online. Constraints are a related approach that uses constraints for indirect assignment to imperative variables is [93].

In the following sections, we discuss some of the above in more detail.

A.2.1 RPromela and RSPIN

An earlier extension of `Promela` that adds synchronous-reactive constructs has been proposed under the name `RPromela`, which stands for "reactive `Promela`" [84, 85]. The term "reactive" here refers to the synchronous-reactive semantics of languages like ESTEREL and *not* to reactiveness as defined by Manna and Pnueli [63]. The seven new keywords it adds are

```
automaton, in, proctype, inport, outport, link, external
```

`RSPIN` is reported as a preprocessor to `SPIN` that translates `RPromela` models to `Promela` models. An example code fragment is

```
rproctype foo(
 inport a, b, c;
 outprt x, y, z
)(bit r){
    automaton subproc(
        external inport a;
        inport x={bool};
        outport g={byte}
    )(){...}
    ...
}
```

A reactive process type, denoted by the keyword `rproctype`, encapsulates a number of `automata`. The automata inside a single `rproctype` are composed synchronously. The semantics of synchronous composition are based on the concept of "stable states". A state is "stable" if it is a receive statement from channels external to the `rproctype`, and so can block. When a reactive process reacts, it receives from its `inports` (representing a transducer), computes until it blocks at the next stable state, and so produces its outputs by writing to its `outports`. The automata inside a single reactive process communicate exclusively by means of channels local to that `rproctype`. Not all automata need to execute during a synchronous reaction, i.e., at least one should be unblocked, but each of the remaining ones can be blocked or not. The synchronous product is taken statically by `RSPIN`, producing `Promela` code as output. The semantics of synchronous product are different than here (implicit "stable states", linking via channels), as well as its syntax. `RPromela` is designed to use channels heavily, which quickly leads to state space explosion in explicit model checking. The semantics are based on transitions (explicit) and not "stable states" as in [85, 84] (implicit). Stable states associate implicit semantics to the synchronous products that can cause confusion more easily. Overall, `RPromela` is an interesting approach, that suggests a syntax for defining synchronous products in `Promela` and a useful resource for ideas that discusses the issues involved.

`RPromela` is not suitable for synthesis specifications, for the limitation of synchronous-reactive languages to be deterministic, with the intent of representing transducer implementations. Besides, there are several points that are treated differently here, because of differences in semantics, e.g., the synchronous product, additional semantics for open systems that are not considered in `RPromela`, (notions of imperative/declarative, primed variables and their owners/quantification, assertions/assumptions, and pro), as well as practical considerations for syntax, e.g., the keyword `sync` instead of `rproctype` that has small lexical distance from `proctype`.

A.2.2 Kaleidoscope

`Kaleidoscope` is an object-oriented constraint imperative programming language that was introduced by Bjorn Freeman-Benson in 1990 [89]. It was a research language and although the authors discuss an implementation, currently an implementation cannot be found online. Nonetheless, it constitutes one of the first attempts to integrate the imperative and the declarative constraint programming paradigm. It introduces the term *Constraint Imperative Programming* (CIP). An observation from [89] that applies to some extent also here is that programmers (and specifiers alike) need to specify two types of relations:

1. Long-lived relations (more suitably described with a declarative paradigm)
2. Sequencing relations between program states (more naturally expressed in an imperative paradigm).

The integration of two paradigms serves this purpose in a single syntax.

Assignment in `Kaleidoscope` has some similarity to variable semantics as defined here. In particular, the `very_weak stay` constraint is very similar to the default invariance of imperative variables here.

An innovative feature of `Kaleidoscope` is the distinction between statements and the advancement of time. The statement *separator* is "`;`", whereas the variable valuation *streams* are advanced by "#". All constraints between two consecutive hash marks hold simultaneously, much like the semantics of synchronous-reactive languages. Note the difference with synchronous-reactiveness though, in that the advancement of time is explicit, instead of implicit. Instead of explicit time advancement, synchronous constrains in open PROMELA can be expressed by conjoining them into a single expression.

An interesting observation is that constraints in CIP languages are expressed using keywords like `always` ("durable relations"), `once` ("transitory relations"), and `while ... assert`, which effectively express temporal logic specifications.

However, the approach in `Kaleidoscope` is based on solving constraints at runtime [90, 91, 94, 92] using a constraint solver and a constrain hypergraph. This solution approach does not ensure satisfaction of temporal logic specifications, as is the case here. In other words, it does *not* produce designs that are correct by construction.

In addition, there is no notion of reactiveness in `Kaleidoscope` – no adversarial environment. Reactiveness is a significant motivation in this work, and distinguishes it from previous work in CIP languages.

The constraint imperative languages are full programming languages, with a lot of emphasis placed on type systems, constraints as objects, and the design of compilers and run-time constraint solvers interfaced to the virtual machine executing the program. They have been developed with the design of user interfaces in mind, as for example the graphical interface of an operating system.

This leads to another difference: `Kaleidoscope` is fundamentally object-oriented, whereas modeling Kripke structures in `Promela` is procedural (both are imperative) paradigms. Constraints in `Kaleidoscope` are intended to be applied to objects, so that object relations can be maintained. Note that this viewpoint is a direct consequence of aiming at the specification of behavior for geometric elements comprising a graphical user interface.

Another concept of `Kaleidoscope` and other CIP languages is that of a *constraint hierarchy*, that distinguishes between *required* constraints, and *optional* constraints that are ranked based on a subjective prioritization. Such constraint hierarchies cannot be expressed in a qualitative synthesis context and require quantitative games [95].

A recently proposed descendant of `Kaleidoscope` is `Babelsberg` [96], a new language that improves on those ideas (supported by the work on constraint satisfaction solvers, as for example `Cassowary`, that is used as the layout engine in `OS X Lion`).

Bibliography

[1] O. Kupferman, "Recent challenges and ideas in temporal synthesis," in *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science*, ser. SOFSEM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 88–98. [Online]. Available: dx.org/10.1007/978-3-642-27660-6_8

[2] I. Walukiewicz, "A landscape with games in the background," *Logic in Computer Science, Symposium on*, vol. 0, pp. 356–366, 2004. [Online]. Available: doi.org/10.1109/LICS.2004.1319630

[3] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive (1) designs," in *Verification, Model Checking, and Abstract Interpretation (VMCAI).* Charleston, SC, USA: Springer, January 2006, pp. 364–380. [Online]. Available: doi.org/10.1007/11609773_24

[4] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911 – 938, 2012, in Commemoration of Amir Pnueli. [Online]. Available: doi.org/10.1016/j.jcss.2011.08.007

[5] R. Ehlers, "Generalized Rabin(1) synthesis with applications to robust system synthesis," in *NASA Formal Methods*, ser. LNCS, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Springer, 2011, vol. 6617, pp. 101–115. [Online]. Available: doi.org/10.1007/978-3-642-20398-5_9

[6] R. Alur and S. La Torre, "Deterministic generators and games for ltl fragments," *ACM Trans. Comput. Logic*, vol. 5, no. 1, pp. 1–25, Jan. 2004. [Online]. Available: doi.org/10.1145/963927.963928

[7] H. Kress-Gazit, G. Fainekos, and G. Pappas, "Temporal-logic-based reactive mission and motion planning," *Robotics, IEEE Transactions on*, vol. 25, no. 6, pp. 1370–1381, Dec 2009. [Online]. Available: doi.org/10.1109/TRO.2009.2030225

[8] M. Kloetzer and C. Belta, "A fully automated framework for control of linear systems from temporal logic specifications," *Automatic Control, IEEE Transactions on*, vol. 53, no. 1, pp. 287–297, Feb 2008. [Online]. Available: doi.org/10.1109/TAC.2007.914952

[9] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Synthesis of control protocols for autonomous systems," *Unmanned Systems*, vol. 1, no. 01, pp. 21–39, 2013. [Online]. Available: dx.org/10.1142/S2301385013500027

[10] R. Rosner, "Modular synthesis of reactive systems," Ph.D. dissertation, Weizmann Institute of Science, Rehovot, Israel, 1992. [Online]. Available: http://www.researchgate.net/publication/238759536_Modular_synthesis_of_reactive_systems/file/50463527f8b648c3ba.pdf

[11] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, May 1994. [Online]. Available: doi.org/10.1145/177492.177726

[12] Z. Manna and A. Pnueli, "A hierarchy of temporal properties (invited paper, 1989)," in *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '90. New York, NY, USA: ACM, 1990, pp. 377–410. [Online]. Available: doi.org/10.1145/93385.93442

[13] K. Schneider, *Verification of reactive systems: formal methods and algorithms.* Springer, 2004.

[14] A. Morgenstern and K. Schneider, “A LTL fragment for GR(1)-synthesis,” *EPTCS*, vol. 50, pp. 33–45, Feb. 2011. [Online]. Available: doi.org/10.4204/EPTCS.50.3

[15] S. Sohail, F. Somenzi, and K. Ravi, “A hybrid algorithm for ltl games,” in *Verification, Model Checking, and Abstract Interpretation.* Springer, 2008, pp. 309–323. [Online]. Available: doi.org/10.1007/978-3-540-78163-9_26

[16] M. Dwyer, G. Avrunin, and J. Corbett, “Patterns in property specifications for finite-state verification,” in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, May 1999, pp. 411–420. [Online]. Available: doi.org/10.1145/302405.302672

[17] Z. Manna and A. Pnueli, “Tools and rules for the practicing verifier,” Stanford University, Stanford, CA, USA, Tech. Rep., 1990. [Online]. Available: ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/90/1321/CS-TR-90-1321.pdf

[18] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986. [Online]. Available: doi.org/10.1109/TC.1986.1676819

[19] A. Pnueli, “The temporal logic of programs,” in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS ’77, 31 1977-nov. 2 1977, pp. 46 –57. [Online]. Available: doi.org/10.1109/SFCS.1977.32

[20] C. Baier and J.-P. Katoen, *Principles of Model Checking.* The MIT Press, May 2008.

[21] O. Lichtenstein, A. Pnueli, and L. Zuck, “The glory of the past,” in *Conference on logics of programs*, ser. Lecture Notes in Computer Science. Springer, 1985, vol. 193, pp. 196–218.

[22] Z. Manna and A. Pnueli, “The anchored version of the temporal framework,” in *Linear time, branching time and partial order in Logics and models for concurrency*, ser. Lecture Notes in Computer Science. Springer, 1989, vol. 354, pp. 201–284.

[23] Y. Kesten, A. Pnueli, and L.-o. Raviv, “Algorithmic verification of linear temporal logic specifications,” in *Int. Colloquium on Automata, Languages and Programming (ICALP)*, ser. Lecture Notes in Computer Science. Springer, 1998, vol. 1443, pp. 1–16.

[24] M. Abadi and L. Lamport, “Open systems in tla,” in *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’94. New York, NY, USA: ACM, 1994, pp. 81–90. [Online]. Available: doi.org/10.1145/197917.197960

[25] A. Pnueli and R. Rosner, “On the synthesis of a reactive module,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’89. New York, NY, USA: ACM, 1989, pp. 179–190. [Online]. Available: doi.org/10.1145/75277.75293

[26] A. Church, “Application of recursive arithmetic to the problem of circuit synthesis,” in *Summaries of talks presented at the summer institute for symbolic logic*, 2nd ed. Cornell University: Communications Research Division, Institute for Defense Analyses, Princeton, N.J., (the summer institute was held in 1957, Church’s paper is a -significantly- extended version of the talk) 1960.

[27] W. Thomas, “On the synthesis of strategies in infinite games,” in *STACS 95.* Springer, 1995, pp. 1–13. [Online]. Available: doi.org/10.1007/3-540-59042-0_57

[28] ——, “Solution of church’s problem: A tutorial,” *New Perspectives on Games and interaction*, vol. 5, 2008. [Online]. Available: http://www.automata.rwth-aachen.de/download/papers/thomas/tho08c.pdf

[29] G. H. Mealy, “A method for synthesizing sequential circuits,” *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955. [Online]. Available: doi.org/10.1002/j.1538-7305.1955.tb03788.x

[30] E. F. Moore, “Gedanken-experiments on sequential machines,” in *Automata Studies*, ser. Annalos of mathematics studies, C. E. Ashby, W. R.; Shannon, Ed. Princeton University Press, 1956, vol. 34, pp. 129–153.

[31] A. Pnueli and R. Rosner, "On the synthesis of an asynchronous reactive module," in *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, ser. ICALP '89. London, UK, UK: Springer-Verlag, 1989, pp. 652–671. [Online]. Available: doi.org/10.1007/BFb0035790

[32] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "Anzu: A tool for property synthesis," in *Computer Aided Verification*. Springer, 2007, pp. 258–262. [Online]. Available: doi.org/10.1007/978-3-540-73368-3_29

[33] A. Pnueli, Y. Sa'ar, and L. D. Zuck, "Jtlv: A framework for developing verification algorithms," in *Computer Aided Verification*. Springer, 2010, pp. 171–174. [Online]. Available: doi.org/10.1007/978-3-642-14295-6_18

[34] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber, "Ratsy–A new requirements analysis tool with synthesis," in *Computer Aided Verification*. Springer, 2010, pp. 425–429. [Online]. Available: doi.org/10.1007/978-3-642-14295-6_37

[35] S. Livingston, R. Murray, and J. Burdick, "Backtracking temporal logic synthesis for uncertain environments," in *ICRA*, 2012, pp. 5163–5170. [Online]. Available: dx.org/10.1109/ICRA.2012.6225208

[36] R. Ehlers and V. Raman, "Low-effort specification debugging and analysis," *EPTCS*, vol. 157, pp. 117–133, 07 2014. [Online]. Available: doi.org/10.4204/EPTCS.157.12

[37] B. Jobstmann and R. Bloem, "Optimizations for LTL synthesis," in *Formal Methods in Computer Aided Design, 2006. FMCAD'06*. IEEE, 2006, pp. 117–124. [Online]. Available: doi.org/10.1109/FMCAD.2006.22

[38] R. Ehlers, "Unbeast: Symbolic bounded synthesis," in *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ser. TACAS'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 272–275. [Online]. Available: doi.org/10.1007/978-3-642-19835-9_25

[39] ——, "Experimental aspects of synthesis," *EPTCS*, vol. 50, 2011. [Online]. Available: doi.org/10.4204/EPTCS.50.1

[40] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J.-F. Raskin, "Acacia+, a tool for LTL synthesis," in *Computer Aided Verification*. Springer, 2012, pp. 652–657. [Online]. Available: dx.org/10.1007/978-3-642-31424-7_45

[41] I. Filippidis and contributors. (2013) List of verification and synthesis tools. [Online]. Available: https://github.com/johnyf/tool_lists/blob/master/verification_synthesis.md

[42] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.

[43] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available: doi.org/10.1145/360933.360975

[44] G. J. Holzmann. Promela language reference (http://spinroot.com/spin/Man/promela.html). [Online]. Available: http://spinroot.com/spin/Man/promela.html

[45] P. Van-Roy and S. Haridi, *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[46] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[47] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev, "NuSMV 2.5 User Manual," Fondazione Bruno Kessler, 18 Via Sommarive, 38055 Povo (Trento), Italy, Tech. Rep., 2010. [Online]. Available: nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf

[48] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, "DiVinE 3.0 – an explicit-state model checker for multithreaded C & C++ programs," in *Computer Aided Verification (CAV 2013)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 863–868. [Online]. Available: dx.org/10.1007/978-3-642-39799-8_60

[49] J. Gennari, S. Hedrick, F. Long, J. Pincar, and R. Seacord, "Ranged integers for the C programming language," Software Engineering Institute, Carnegie Mellon University, Technical Note CMU/SEI-2007-TN-027, Retrieved Oct 07, 2014 2007. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8265

[50] R. M. Keller, "Formal verification of parallel programs," *Commun. ACM*, vol. 19, no. 7, pp. 371–384, Jul. 1976. [Online]. Available: doi.org/10.1145/360248.360251

[51] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, "Alternation," *J. ACM*, vol. 28, no. 1, pp. 114–133, Jan. 1981. [Online]. Available: doi.org/10.1145/322234.322243

[52] D. E. Muller, A. Saoudi, and P. E. Schupp, "Alternating automata, the weak monadic theory of the tree, and its complexity," in *Automata, Languages and Programming.* Springer, 1986, pp. 275–283. [Online]. Available: doi.org/10.1016/0304-3975(92)90076-R

[53] M. Y. Vardi, "Alternating automata and program verification," in *Computer Science Today.* Springer, 1995, pp. 471–485. [Online]. Available: doi.org/10.1007/BFb0015261

[54] ——, "An automata-theoretic approach to linear temporal logic," in *Logics for concurrency: structure versus automata.* Springer, 1996, vol. 1043, pp. 238–266. [Online]. Available: doi.org/10.1007/3-540-60915-6_6

[55] O. Kupferman and M. Vardi, "Safraless decision procedures," in *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, Oct 2005, pp. 531–540. [Online]. Available: doi.org/10.1109/SFCS.2005.66

[56] C. A. R. Hoare, *Communicating sequential processes.* Prentice-hall Englewood Cliffs, 1985–2004, vol. 178. [Online]. Available: http://www.usingcsp.com/cspbook.pdf

[57] H. Søndergaard and P. Sestoft, "Non-determinism in functional languages," *The Computer Journal*, vol. 35, no. 5, pp. 514–523, 1992. [Online]. Available: http://comjnl.oxfordjournals.org/content/35/5/514.abstract

[58] J. McCarthy, "A basis for a mathematical theory of computation," in *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds. North-Holland, 1963, pp. 33–70. [Online]. Available: doi.org/10.1145/1460690.1460715

[59] R. W. Floyd, "Nondeterministic algorithms," *J. ACM*, vol. 14, no. 4, pp. 636–644, Oct. 1967. [Online]. Available: doi.org/10.1145/321420.321422

[60] M. Broy, "A theory for nondeterminism, parallelism, communication, and concurrency," *Theoretical Computer Science*, vol. 45, no. 0, pp. 1 – 61, 1986. [Online]. Available: doi.org/10.1016/0304-3975(86)90040-X

[61] R. McNaughton, "Infinite games played on finite graphs," *Annals of Pure and Applied Logic*, vol. 65, no. 2, pp. 149 – 184, 1993. [Online]. Available: doi.org/10.1016/0168-0072(93)90036-D

[62] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 348–370. [Online]. Available: doi.org/10.1007/978-3-642-17511-4_20

[63] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems.* New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[64] Z. Manna, *Mathematical theory of computation.* McGraw-Hill, 1974.

[65] B. Finkbeiner and S. Schewe, "Bounded synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 519–539, 2013. [Online]. Available: dx.org/10.1007/s10009-012-0228-z

[66] U. Klein, N. Piterman, and A. Pnueli, "Effective synthesis of asynchronous systems from gr (1) specifications," in *Verification, Model Checking, and Abstract Interpretation.* Springer, 2012, pp. 283–298. [Online]. Available: dx.org/10.1007/978-3-642-27940-9_19

[67] A. Pnueli and U. Klein, "Synthesis of programs from temporal property specifications," in *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, July 2009, pp. 1–7. [Online]. Available: doi.org/10.1109/MEMCOD.2009.5185372

[68] N. Halbwachs, *Synchronous Programming of Reactive Systems.* Springer, 1992.

[69] F. Ciesinski, C. Baier, M. Größer, and D. Parker, "Generating compact mtbdd-representations from probmela specifications," in *Model Checking Software*, ser. Lecture Notes in Computer Science, K. Havelund, R. Majumdar, and J. Palsberg, Eds. Springer, 2008, vol. 5156, pp. 60–76. [Online]. Available: doi.org/10.1007/978-3-540-85114-1_7

[70] M. Baldamus and J. Schröder-Babo, "P2B: A translation utility for linking promela and symbolic model checking (tool paper)," in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, ser. SPIN '01. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 183–191. [Online]. Available: doi.org/10.1007/3-540-45139-0_11

[71] Y. Jiang and Z. Qiu, "S2N: Model transformation from spin to nusmv," in *Proceedings of the 19th International Conference on Model Checking Software*, ser. SPIN'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 255–260. [Online]. Available: doi.org/10.1007/978-3-642-31759-0_20

[72] D. Peled and T. Wilke, "Stutter-invariant temporal properties are expressible without the next-time operator," *Inf. Process. Lett.*, vol. 63, no. 5, pp. 243–246, Sep. 1997. [Online]. Available: doi.org/10.1016/S0020-0190(97)00133-6

[73] K. L. McMillan, "Symbolic model checking: An approach to the state explosion problem," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 1992, uMI Order No. GAX92-24209. [Online]. Available: http://www.kenmcmil.com/pubs/thesis.pdf

[74] "open PROMELA implementation." [Online]. Available: http://github.com/johnyf/openpromela

[75] "PROMELA parser in PYTHON." [Online]. Available: http://github.com/johnyf/promela

[76] D. M. Beazley. Ply (python lex-yacc) v3.4. [Online]. Available: http://www.dabeaz.com/ply/ply.html

[77] D. Kroening and O. Strichman, *Decision procedures: an algorithmic point of view.* Springer Science & Business Media, 2008.

[78] ISO/IEC, "Programming languages - c," International Organization for Standardization, International Standard 9989:TC3, 2007. [Online]. Available: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

[79] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Computer Aided Verification.* Springer, 2005, pp. 226–238.

[80] A. S. Lezama, "Program synthesis by sketching," Ph.D. dissertation, Citeseer, 2008.

[81] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD), 2013.* IEEE, 2013, pp. 1–17.

[82] V. Beaudenon, E. Encrenaz, and S. Taktak, “Data decision diagrams for promela systems analysis,” *International journal on software tools for technology transfer*, vol. 12, no. 5, pp. 337–352, 2010. [Online]. Available: doi.org/10.1007/s10009-010-0135-0

[83] S. Maoz and Y. Sa’ar, “AspectLTL: An aspect language for LTL specifications,” in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, ser. AOSD ’11. New York, NY, USA: ACM, 2011, pp. 19–30. [Online]. Available: doi.org/10.1145/1960275.1960280

[84] E. Najm and F. Olsen, “Reactive EFSMs — Reactive Promela/RSPIN,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer, 1996, vol. 1055, pp. 349–368. [Online]. Available: doi.org/10.1007/3-540-61042-1_54

[85] ——, “Protocol verification with Reactive PROMELA/RSPIN,” in *Second SPIN Workshop*, 1996. [Online]. Available: spinroot.com/spin/Workshops/ws96/Ol.pdf

[86] A. Gamatié, *Designing embedded systems with the Signal programming language: synchronous, reactive specification.* Springer, 2009. [Online]. Available: doi.org/10.1007/978-1-4419-0941-1

[87] M. Jourdan, F. Lagnier, R. Maraninchi, and P. Raymond, “A multiparadigm language for reactive systems,” in *Computer Languages, 1994., Proceedings of the 1994 International Conference on.* IEEE, 1994, pp. 211–218. [Online]. Available: doi.org/10.1109/ICCL.1994.288379

[88] M. Baldamus and K. Schneider, “Extending esterel by asynchronous concurrency,” in *Fachtagung zum Entwurf Integrierter Schaltungen*, 1999. [Online]. Available: http://es.cs.uni-kl.de/publications/datarsg/BaSc99.pdf

[89] B. N. Freeman-Benson, “Kaleidoscope: Mixing objects, constraints, and imperative programming,” *SIGPLAN Not.*, vol. 25, no. 10, pp. 77–88, Sep. 1990. [Online]. Available: doi.org/10.1145/97946.97957

[90] B. N. Freeman-Benson and A. Borning, “Integrating constraints with an object-oriented language,” in *Proceedings of the European Conference on Object-Oriented Programming*, ser. ECOOP ’92. London, UK, UK: Springer-Verlag, 1992, pp. 268–286. [Online]. Available: http://dl.acm.org/citation.cfm?id=646150.679346

[91] B. Freeman-Benson and A. Borning, “The design and implementation of kaleidoscope’90-a constraint imperative programming language,” in *Computer Languages, 1992., Proceedings of the 1992 International Conference on*, Apr 1992, pp. 174–180. [Online]. Available: doi.org/10.1109/ICCL.1992.185480

[92] G. Lopez, B. Freeman-Benson, and A. Borning, “Implementing constraint imperative programming languages: The KALEIDOSCOPE’93 virtual machine,” in *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, ser. OOPSLA ’94. New York, NY, USA: ACM, 1994, pp. 259–271. [Online]. Available: doi.org/10.1145/191080.191118

[93] L. Lamport and F. B. Schneider, “Constraints: A uniform approach to aliasing and typing,” in *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 1985, pp. 205–216.

[94] B. N. Freeman-Benson, “Constraint imperative programming,” Ph.D. dissertation, University of Washington, Seattle, WA, USA, 1992, uMI Order No. GAX92-03256.

[95] L. de Alfaro and R. Majumdar, “Quantitative solution of omega-regular games,” *Journal of Computer and System Sciences*, vol. 68, no. 2, pp. 374 – 397, 2004, special Issue on {STOC} 2001. [Online]. Available: doi.org/10.1016/j.jcss.2003.07.009

[96] T. Felgentreff, A. Borning, and R. Hirschfeld, “Babelsberg: Specifying and solving constraints on object behavior,” *Journal of Object Technology*, vol. 13, no. 4, pp. 1–38, September 2013. [Online]. Available: doi.org/10.5381/jot.2014.13.4.a1