Program Sketching via CTL* Model Checking

Andreas Morgenstern and Klaus Schneider

University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern, Germany
email: {morgenstern,schneider}@cs.uni-kl.de

Abstract. Sketching is an approach to automated software synthesis where the programmer develops a partial implementation called a sketch and a separate specification of the desired functionality. A synthesizer tool then automatically completes the sketch to a complete program that satisfies the specification. Previously, sketching has been applied to finite programs with a desired functional input/output behavior and given invariants. In this paper, we consider (non-terminating) reactive programs and use the full branching time logic CTL* to formalize specifications. We show that the sketching problem can be reduced to a CTL* model checking problem provided there is a translation of the program to labelled transition systems.

1 Introduction

Formal verification is mandatory for software in safety critical applications. In particular, this is the case for embedded systems that implement difficult and concurrent control problems. These systems are often reactive real-time systems that must react sufficiently fast to the stimuli of their environment. For this reason, their temporal behavior is of essential importance for their correctness.

In the past two decades, many verification procedures based on model checking the temporal behavior of reactive systems have been developed [13], and this research already lead to tools that are are now routinely used in industry. One of the most popular specification logics is CTL* [6,5] that subsumes the well-known temporal logics LTL and CTL and is strictly more expressive than both of them.

In this paper, we want to go one step beyond CTL* model checking. Instead of developing a full implementation and afterwards checking that the implementation satisfies a specification, a programmer may develop a partial implementation, which is called a program sketch. It is the aim of this paper to develop a procedure that completes this partial implementation such that a given CTL* specification is satisfied. The motivation behind the sketching approach is that in many cases, programmers have an idea of an algorithm, but often fail to determine the full details that are required to implement that algorithm without errors. For example, loop bounds are often missed by plus or minus one, so that one often speaks of plus/minus one bugs, or expressions are used that are 'almost' the correct ones, but fail for some corner cases. Typically, programmers

feel whether an expression may not be correct, but they usually have no means in programming languages to express their doubts. Hence, instead of forcing a human programmer to determine the error-prone details, we employ computers to solve this difficult task in that we want to develop a synthesis procedure that selects among a set of given program expressions one that satisfies the specifications.

The term "sketching" has been coined by Solar-Lezama et. al [16,18,17]. The difference to our approach is that we consider reactive systems, i.e., systems with typically non-terminating behaviors that are specified by temporal logic specifications. Clearly, there is also a relationship to infinite games [7,12,11,2,10]. However, we want to emphasize that we solve a completely different problem: In infinite games, the players may select in each position of the game among a set of inputs and there is no restriction on this set of inputs with respect to the previously chosen inputs. Our problem is totally different: we seek *one* program expression that is used in the desired occurrences of the program and this program expression may not change whenever we reenter this program position again[1].

There is also a relationship to programming with angelic nondeterminism: in [3] the authors propose to use Floyd's nondeterministic choose operator to allow the formulation of nondeterministic (finite) programs. The question they consider is whether this choose operator can be replaced by a deterministic program expression such that the specification is fulfilled. However, their procedure is more or less an example-generator: given a partial (nondeterministic) program with choose operator their procedure generates a finite execution trace such that the specification is satisfied and the programmer is still left with the problem of determining the correct program expressions based on this execution trace. In contrast, we use the ability of model checkers to generate counterexamples to actually generate correct program expressions (respectively to select one among a list of given proposals).

To be independent of the underlying programming language in this paper, we develop our theory purely on the basis of Kripke structures which are labelled transition systems. Hence, one is able to use our approach with any programming language provided there is a convenient representation as a transition system. In order to leave the freedom to choose between different program expressions, we endow the Kripke structure with a set of oracles and the synthesizer has to choose among these oracles to fulfill the specification. We have implemented our

[1] Moreover, the known algorithms for the solution of infinite games solve the synthesis problem in an unsatisfactory way: as they construct a strategy that solves the synthesis problem, their result is a (nondeterministic) Moore (or a Mealy) automaton reading the uncontrollable inputs and generating the controllable inputs of the system as its outputs. Deriving a program expression from this automaton to replace the controllable inputs is a challenging task, in particular, if the resulting automaton is given symbolically, e.g., as a BDD. While a hardware circuit can be directly derived from this description, a program expression at the level of the source code, where more abstract data types than booleans should be used, cannot be directly obtained.

approach for the Quartz programming language that is a descendent of Esterel, which is a very good choice for implementing reactive systems. Examples given in the paper have been worked out with our Averest[2] system.

The outline of the paper is as follows: In the next section, we motivate the problem considered in this paper and indicate how we solve it. Section 3 recalls then formal definitions of the temporal logic CTL*. We also formally define Kripke structures with oracles in that section. Section 4 contains the core of the paper where we prove that our problem can be polynomially reduced to CTL* model checking, so that the PSPACE-completeness of CTL* defines also the complexity class of our problem. In Section 5, we then show how we implemented the approach on top of our Averest system. We finally briefly discuss related and future work in our conclusions.

```
macro M = 10;
macro N = 2;

module BinarySearch([N]int{M} ?a, int{M} ?v, nat{N+1} i) {
    nat{N} left,right,mid;
    left = 0;
    right = N - 1;
    i = N;
    while(left<right & i==N) {
        mid = (left + right) / 2;
        if(a[mid]<v)
            next(left) = mid+1;
        else if(a[mid]>v)
            next(right) = mid-1;
        else // v has been found
            next(i) = mid;
        pause;
    }
}
```

Fig. 1. Faulty Implementation of Binary Search

2 Motivating Example

As already stated, we consider the situation where a programmer has an idea of how the algorithm should work, but fails to provide the full details in terms of correct program expressions. For an example, consider the faulty implementation of the well-known binary search algorithm. In Figure 1, we have chosen an

[2] Available at http://www.averest.org.

implementation in the synchronous programming language Quartz [14], but the choice of the programming language is not relevant for the errors on the program. The algorithm searches a desired value `v` in a sorted array `a` and returns the index `i` of the array element `a[i]` if `v` is contained in `a`, and terminates with `i==N` which is no legal array index, since the array has elements `a[0]`, ..., `a[N-1]`.

The local variables `left` and `right` store the borders of the sub-array `a` that has to be searched through in future iterations. Hence, initially, we have `left==0` and `right==N-1` so that the entire array is searched through. Since `a` is sorted, the idea is to compare the middle element of the current sub-array with the desired value `v`. To this end, the index of the middle position `mid` is computed as `mid = (left + right) / 2`. If `v` is less than `a[mid]`, one assigns `right=mid` and if `v` is greater than `a[mid]`, one assigns `right=mid`, and if `v` is equal to `a[mid]`, one assigns `i = mid<N`. These computations are iterated until `left>=right` or no longer `i == N` holds.

While this algorithm seems to be plausible, and while it is contained in many textbooks on algorithms, and is even taught (and verified!) each year in many courses in computer science, it is nevertheless wrong: For example, consider an array consisting of two elements 0 and 1 and the value search for is `v=1`. The algorithm starts with `left=0` and `right=1` and computes then `mid=0`. We then have `a[mid] == 0 < 1 == v`, so that `left` is assigned the value `mid+1`, i.e., 1. Since we then have no longer `left<right`, the loop terminates, but still `i==N` holds and thus, `i` points to the wrong index. The value is therefore not found even though it is contained in the array. This example suggests that the loop condition should be rather `left<=right & i==N` instead of `left<right & i==N`, so that another iteration is performed.

However, even with the modified loop condition `left<=right & i==N`, the algorithm is still not correct: Assume `a` consists of the two values 2 and 3, and we search for the value `v=1` that is therefore not contained in the array. Again, the algorithm starts with `left=0` and `right=1` and computes then `mid=0`. We then have `a[mid] == a[0] == 2 > 1 == v`, so that `right` is assigned the value `mid-1`, i.e., -1. This is very dangerous since this leads to an out-of-bound error in the next iteration.

In a recent 'Google research blog', it has been reported [15] that almost all binary search algorithms are broken. The problems are usually that arithmetic over- and underflows may occur in the computation of the index of the middle position. While it seems to be unavoidable at a first glance, [15] proves that the expressions `x+(y-x)/2` and `(x+y)>>1` correctly implement that position without overflows in case of signed integers, and the expression `(x and y) + (x xor y)/2` can be safely used in case of unsigned integers.

These problems with the binary search algorithm are well-known: In [9], it is mentioned that the main idea of binary search has been published as early as 1946. However, the first correct implementation has been given probably in 1960 [9]. A similar problem with respect to integer arithmetic remained undiscovered in the binary search implementation of the Java library for nearly 9 years. While

such problems are typically found by model checking, removing the errors is still a hard and error-prone task.

Our approach to the problem is as follows: For example, it is pretty clear that the loop condition should contain either `left<right` or `left<=right`. Hence, the programmer might use `(sel ? left<right : left<=right)` by introducing a new boolean oracle variable `sel`. This conditions equals to either `left<right` or `left<=right` depending on whether `sel` is true or false, respectively. In a similar manner, we might introduce new oracle variables `add` and `sub` that represent natural numbers (either 0 or 1) that are then used to increment or decrement the `mid` value.

The question we are going to solve in this paper is then to select constant values for the oracle variables `sel, add, mul` to replace the variables `sel, add, mul`.

3 Preliminaries

In this section, we recall the formal definitions required for the formulation of our problem. In particular, we start with the definition of the syntax and semantics of the temporal logic CTL* which we use for specifying the reactive programs that we want to develop. We will also consider a 'translation' from automata to Kripke structures (labelled transition systems). Finally, we provide a formal definition of Kripke structures with oracles which we use as a representative for incompletely specified programs.

3.1 Syntax and Semantics of CTL*

As already mentioned, we start the preliminaries with the definition of the syntax and semantics of the temporal logic CTL* [5]. The following mutually recursive definitions introduce the set of path formulas $\mathtt{PF}_\Sigma$ and the set of state formulas $\mathtt{SF}_\Sigma$ over a given finite set of variables V_Σ.

- The set of path formulas $\mathtt{PF}_\Sigma$ over the variables V_Σ is the least set which satisfies the following properties:
  - state formulas: each state formula is a path formula, i. e. $\mathtt{SF}_\Sigma \subseteq \mathtt{PF}_\Sigma$
  - boolean operators: $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi \in \mathtt{PF}_\Sigma$, if $\varphi, \psi \in \mathtt{PF}_\Sigma$
  - temporal operators: $\mathsf{X}\varphi$, $[\varphi\ \mathsf{U}\ \psi]$, $[\varphi\ \mathsf{B}\ \psi] \in \mathtt{PF}_\Sigma$, if $\varphi, \psi \in \mathtt{PF}_\Sigma$
- The set of state formulas $\mathtt{SF}_\Sigma$ over the variables V_Σ is the least set which satisfies the following properties:
  - variables: each variable is a state formula, i. e. $V_\Sigma \subseteq \mathtt{SF}_\Sigma$
  - boolean operators: $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi \in \mathtt{SF}_\Sigma$, if $\varphi, \psi \in \mathtt{SF}_\Sigma$
  - path quantifiers: $\mathsf{E}\varphi, \mathsf{A}\varphi \subseteq \mathtt{SF}_\Sigma$ if $\varphi \in \mathtt{PF}_\Sigma$

The set of CTL* formulas over the set of variables V_Σ is the set of state formulas $\mathtt{SF}_\Sigma$. CTL* formulas are interpreted over Kripke structures, which are labelled transition systems as defined below:

Definition 1 (Kripke Structures). *A Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ for a finite set of variables $\mathcal{V}$ is given by a finite set of states $\mathcal{S}$, a set of initial states $\mathcal{I} \subseteq \mathcal{S}$, a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and a label function $\mathcal{L} : \mathcal{S} \to 2^V$ that maps each state to the set of variables that hold in this state.*

To define the semantics of CTL*, we have to define infinite paths through a Kripke structure:

Definition 2 (Paths). *Given a set of atomic propositions $\mathcal{V}$, an infinite path is a function $\pi : \mathbb{N} \to 2^{\mathcal{V}}$. We denote the i-th state of the path π as $\pi^{(i-1)}$ for $i \in \mathbb{N}$. Using this notation, paths are often given in the form $\pi^{(0)}\pi^{(1)} \ldots$. The path starting at t is moreover written as $(\pi, t) := \pi^{(t)}\pi^{(t+1)} \ldots$*

Note that a path as defined above is nothing but a sequence of assignments over the variables V_Σ. The semantics of path formulas is informally given as follows (see e. g. [13] for a full definition): $\mathsf{X}\varphi$ holds at a path π at position t_0 if φ holds at position t_0+1 on the path. $[\varphi \,\underline{\mathsf{U}}\, \psi]$ holds at t_0 iff ψ holds for some position $\delta \geq t_0$ and φ holds invariantly for every position t with $t_0 \leq t < \delta$ i. e. φ holds *until* ψ holds. The *weak before* operator $[\varphi \,\mathsf{B}\, \psi]$ holds at t_0 iff either φ holds before ψ becomes true for the first time after t_0 or ψ never holds after t_0. Finally, $\mathsf{E}\varphi$ holds in a state s of a Kripke structure if there is a path π starting in s such that φ holds on π, and analogously, $\mathsf{A}\varphi$ holds in a state s of a Kripke structure if all paths starting in s satisfy φ.

Finally, a Kripke structure satisfies a CTL* formula Φ if all initial states satisfy Φ [3].

3.2 Relating Automata to Kripke Structures

While Kripke structures are convenient for model checking, they are not useful in system design where state-based systems must still be executable for means of simulation and synthesis. To this end, we must distinguish between input and state variables, to keep track of the causality and the actors that determine the values of these variables. Thus, one typically prefers automata over Kripke structures, where in case of reactive systems, these automata read infinite words over letters chosen from an input alphabet:

Definition 3. *An* automaton *$\mathfrak{A} = (Q, \Sigma, Q_0, \Delta, \mathcal{L})$ over the alphabet Σ and a set of variables $\mathcal{V}$ is given by a finite set of states Q, a set Q_0 of initial states, a transition relation $\Delta \subseteq Q \times \Sigma \times Q$ and a labeling function $\mathcal{L} : Q \to 2^{\mathcal{V}}$.*

[3] Other temporal operators can be defined as syntactic sugar in terms of the above ones: $\mathsf{G}\varphi := [\mathsf{0}\ \mathsf{B}\ \neg\varphi]$, $\mathsf{F}\varphi := [\mathsf{1}\ \underline{\mathsf{U}}\ \varphi]$, $[\varphi\ \mathsf{B}\ \psi] := \neg[\neg\varphi\ \underline{\mathsf{U}}\ \psi]$ $[\varphi\ \mathsf{U}\ \psi] := [\psi\ \mathsf{B}\ (\neg\varphi \wedge \neg\psi)]$, $[\varphi\ \underline{\mathsf{B}}\ \psi] := [\neg\psi\ \underline{\mathsf{U}}\ (\varphi \wedge \neg\psi)]$ $[\varphi\ \mathsf{W}\ \psi] := [(\varphi \wedge \psi)\ \mathsf{B}\ (\neg\varphi \wedge \psi)]$ $[\varphi\ \underline{\mathsf{W}}\ \psi] := [\neg\psi\ \underline{\mathsf{U}}\ (\varphi \wedge \psi)]$. For example, $[\varphi\ \mathsf{U}\ \psi]$ is the *weak until* operator that can be alternatively defined as $[\varphi\ \mathsf{U}\ \psi] := [\varphi\ \underline{\mathsf{U}}\ \psi] \vee \mathsf{G}\varphi$, i. e. the event ψ that is awaited for may not hold in the future. To distinguish weak and strong operators, the strong variants of a temporal operator are underlined in this paper (as done above).

Using standard terminology, we say that $\mathfrak{A}$ is *deterministic*, if exactly one initial state exists and for each $q \in Q$ and each input $\sigma \in \Sigma$, there exists exactly one $q' \in Q$ with $(q, \sigma, q') \in \mathcal{R}$. In that case, we write $\mathfrak{A} = (Q, \Sigma, q_0, \delta, \mathcal{L})$ with an initial state q_0 and a deterministic transition function $\delta : Q \times \Sigma \to Q$.

Obviously, there is a close relationship between automata and Kripke structures that is captured in the following definition [13]:

Definition 4 (Associated Kripke Structure of an Automaton). *Given an automaton $\mathfrak{A} = (Q, \Sigma, Q_0, \Delta, \mathcal{L})$ over the alphabet Σ, and a set of variables $\mathcal{V}$, we define the associated Kripke structure $\texttt{Struc}(\mathfrak{A}) = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L}')$ as follows*

- $\mathcal{S} = Q \times \Sigma$
- $\mathcal{I} = Q_0 \times \Sigma$
- $\mathcal{R}((q, \sigma), (q', \sigma')) \leftrightarrow \Delta(q, \sigma, q')$
- $\mathcal{L}'(q, \sigma) = q$.

Intuitively, the states of the associated Kripke structure consist of the configurations of an automaton, i.e., pairs (q, σ) consisting of a state q and an input letter σ that is read in that state. While automata are the natural result of translations of finite state programs, they are not well-suited for the definition of the semantics of temporal logics. For this reason, we make use of both automata and Kripke structures and use the above definition to relate both when necessary.

3.3 Kripke Structure with Oracles and Implementations

The previous subsection gives us a blueprint to define our problem: we assume that we are given an automaton where some of the inputs are oracle variables. Those oracle variables may represent "angelic" nondeterminism that may be resolved in favor to satisfy the specification. Hence we seek a deterministic implementation of those nondeterministic variables such that the specification is fulfilled. Translated to Kripke structure, we may assume that the state set of a Kripke structure $\mathfrak{A}$ is the product of a state set $\mathcal{Q}$ and a oracle set $\mathcal{O}$ and an implementation of $\mathfrak{A}$ is a Kripke structure that selects exactly one element of the oracle set and fixes this element throughout the execution.

Definition 5 (Kripke Structure with Oracles). *A Kripke structure with oracles is a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ as before where $\mathcal{S} = \mathcal{Q} \times \mathcal{O}$ is the product of a state set $\mathcal{Q}$ with a oracle set $\mathcal{O}$.*
An implementation *$\mathcal{K}_o = (\mathcal{S}_o, \mathcal{I}_o, \mathcal{R}_o, \mathcal{L})$ for some $o \in \mathcal{O}$ is a restriction of $\mathcal{K}$ given by*

- $\mathcal{S}_o = \mathcal{S} \cap \{o\}$
- $\mathcal{I}_o = \mathcal{I} \cap \{o\}$
- $\mathcal{R}_o((q, \hat{o}), (q', \hat{o}')) \leftrightarrow \mathcal{R}((q, \hat{o}), (q', \hat{o}')) \wedge \hat{o} = \hat{o}' = o$

The question we are going to solve in this paper is to find an answer to the following question: Given a Kripke structure $\mathcal{K}$ with oracles and a CTL* formula Φ, is there an implementation $\mathcal{K}_o$ of $\mathcal{K}$ such that $\mathcal{K}_o \models \Phi$?

4 Implementations for Kripke Structures with Oracles

In this section, we show that the problem to determine whether a system has an implementation can be reduced to a CTL* model checking problem.

Theorem 1. *Let $\mathcal{K} = (\mathcal{Q} \times \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ be a Kripke structure with oracle set $\mathcal{O}$ and a CTL* formula Φ. There exists a Kripke structure $\mathcal{K}' = (\mathcal{S}', \mathcal{I}', \mathcal{R}', \mathcal{L}')$ whose size is polynomial in the size of $\mathfrak{A}$ such that the following holds: there exists an implementation $\mathcal{K}_o$ of $\mathcal{K}$ satisfying Φ if and only if $\mathfrak{A}' \models \mathsf{EXAX}\Phi$.*

Proof. We define $\mathcal{K}' = (\mathcal{I}', \mathcal{S}', \mathcal{R}', \mathcal{L})$ by adding new states q_0 and q^o_{sel} for each value of the oracle set $o \in \mathcal{O}$ as follows:

- $\mathcal{S}' = \mathcal{Q} \times \mathcal{O} \cup \{q_0\} \cup \{q^o_{sel} \mid o \in \mathcal{O}\}$
- $\mathcal{I}' = \{q_0\}$
- $\mathcal{R}' = \begin{pmatrix} \{(q_0, q^o_{sel}) \mid o \in \mathcal{O}\} \cup \\ \{((q^o_{sel}, (q, o)) \mid q \in \mathcal{I} \wedge o \in \mathcal{O})\} \cup \\ \{((q, o), (q', o)) \mid ((q, o), (q', o)) \in \mathcal{R} \wedge o \in \mathcal{O}\} \end{pmatrix}$

The intuitive idea behind the construction of $\mathcal{K}'$ is as follows: The only initial state is q_0, and q_0 has transitions to every state q^o_{sel} for each value o of the oracle set $\mathcal{O}$. In each state q^o_{sel}, a choice is made for the oracle set that is kept invariant on all transitions of $\mathcal{K}'$: The transition relation $\mathcal{R}'$ of $\mathcal{K}'$ is defined such that from q^o_{sel} there are only transitions to states (q, o) where q is an initial state of $\mathfrak{A}$ and o is the value chosen by q^o_{sel}. Moreover, the transitions of $\mathcal{K}$ are pruned so that a transition leads from a state (q, o) to a state (q', o') only if $o = o'$ holds (and there was already this transition in $\mathcal{K}$). Hence the sub-structure starting at q^o_{sel} is bisimulation equivalent to the implementation $\mathcal{K}_o$.

Now remember that a Kripke structure $\mathcal{K}$ satisfies a CTL* specification Φ if and only if all initial states of $\mathcal{K}$ satisfy Φ. Hence there does exist a implementation $\mathcal{K}_o$ if and only if the sub-structure rooted at $q^o_{sel} \models \mathsf{AX}\Phi$ and accordingly if and only if $\mathcal{K}' \models \mathsf{EXAX}\Phi$. □

The following corollary determines the complexity of the problem:

Corollary 1. *Deciding whether a Kripke structure with oracles has an implementation satisfying a CTL* property is PSPACE-complete.*

Proof. The problem of CTL* model checking (which is PSPACE-hard) can be reduced to the problem of deciding the existence of an implementation over a singleton Oracle set that has no effect. Hence PSPACE-hardness follows. The problem can be solved in polynomial space, since the constructed Kripke structure of the previous theorem is polynomial in the size of the original Kripke structure and CTL* model checking can be solved in polynomial time. □

5 Sketching Programs in Practice

In this section, we show how a program in a synchronous reactive programming language has to be modified to obtain a Kripke structure with oracles. To this

end, we make use of our synchronous language Quartz [14], a descendent of Esterel. As will be seen the changes are only needed in the program and can be moreover automatically done by a source code transformation. Moreover, note that the principle can be applied to any programming language, and it could also be performed on an automaton that has been obtained by translation of the program. We start with a short introduction to the synchronous language Quartz, and will then show how the construction given in Theorem 1 can be performed at the source code level of Quartz programs.

5.1 Essentials of the Synchronous Programming Language Quartz

Synchronous languages are becoming more and more attractive for the design and the verification of reactive real time systems. There are imperative languages like *Esterel*, data flow languages like *Lustre*, and graphical languages like certain Statechart variants like *SyncCharts*.

Synchronous languages are well-suited for the design of reactive systems since they provide language constructs for parallel execution of processes and a synchronous concurrency that matches with the product computation of automata and Kripke structures. Moreover, they have a formal semantics that allows one to translate synchronous programs to extended finite state machines, and if the program has only finite data types, then even to finite state automata. These translations allow one to directly apply model checking techniques to the automata obtained from synchronous programs or to generate hardware circuits that implement the desired automata.

The common paradigm of synchronous languages is the perfect synchrony [8,1] which means that the execution of programs is divided into macro steps that are usually interpreted as logical time. As this logical time is the same in all concurrent threads, the threads run in lockstep, which leads to a deterministic form of concurrency. Macro steps are divided into finitely many micro steps that are atomic actions of the programs. Moreover, variables change synchronously in macro steps, i.e., variables have unique values in each macro step.

In the following, we give a brief overview of the synchronous programming language Quartz. We do, however, not describe the entire language, and refer instead to [14]. Provided that S, S_1, and S_2 are statements, ℓ is a location variable, x is a variable, σ is a boolean expression, and α is a type, then the following are statements (keywords given in square brackets are optional):

- *nothing* (empty statement)
- $x = \tau$ and $next(x) = \tau$ (assignments)
- $assume(\varphi)$ (assumptions)
- $assert(\varphi)$ (assertions)
- $[\ell :]pause$ (start/end of macro step)
- *if* (σ) S_1 *else* S_2 (conditional)
- $S_1; S_2$ (sequential composition)
- *do* S *while*(σ) (iteration)
- $S_1 \parallel S_2$ (synchronous concurrency)

– *[weak] [immediate] abort* S *when* (σ) (abortion)

– *[weak] [immediate] suspend* S *when* (σ) (suspension)

– $\{\alpha\ x;\ S\}$ (local variable y with type α)

The ℓ : *pause* statement defines a control flow location ℓ which is a boolean variable that is true iff the control flow is currently at the statement ℓ : *pause*. Since all other statements are executed in zero time, the control flow can only rest at these positions in the program, and therefore the possible control flow states are the subsets of the set of locations. Hence, the variables ℓ used to name these control flow locations are state variables that encode the control flow of the program.

There are two variants of assignments; and both evaluate the right-hand side τ in the current macro step. While immediate assignments $x = \tau$ immediately transfer the value of τ to the left-hand side x, delayed assignments $\mathit{next}(x) = \tau$ transfer this value in the following step.

In case the value of a variable is not determined by an assignment, a default value is determined by the declaration of the variable. To this end, declarations provide a storage class in addition to the type of a variable. There are two storage classes, namely *mem* and *event* that choose the previous value or a default value (determined by the type), respectively, in case no assignment determines the value of a variable. Available types are booleans, signed and unsigned integers (both with limited and unlimited bounds), bitvectors, as well as arrays and tuples of types.

In addition to the control flow constructs that are well-known from other imperative languages like conditionals, sequences and loops, Quartz offers synchronous concurrency $S_1 \parallel S_2$ and sophisticated preemption and suspension statements, as well as further statements to allow comfortable descriptions of reactive systems (see [14] for the complete syntax and semantics). In $S_1 \parallel S_2$, both statements S_1 and S_2 must run in lockstep as long as both are active, and the entire statement terminates when the last one of the contained statements terminates. Preemption statements attach a guard to a statement and either suspend or abort it if that guard is true. The immediate forms do already check for preemption at starting time, while the default is to check the preemption only after starting time. The weak variants allow all actions of the data flow to take place even at the time of preemption, while the strong variant forbids them at the time of preemption.

In Quartz programs, we may also add temporal logic specifications in section `satisfies` that follow a Quartz module (see examples below). The specifications listed in these sections can be endowed by an observer statement that makes it often easier to express properties that are otherwise hard to formulate in temporal logic alone. We consider however only specifications that stem from the temporal logic CTL* in this paper.

5.2 Examples

In Theorem 1, we have described a reduction of the oracle selection problem of CTL* properties to CTL* model checking. In this section, we show how that construction can be performed at the source code level of Quartz programs.

The Hello World Program of Sketching To start with, consider the "Hello World" program of sketching that has been introduced in [16] which is reformulated in Quartz as follows:

```
macro N = 5;
module HelloWorld(nat{N} ?x, nat{N*2} !y) {
    y = x * ??;
    q_1:pause;
}
satisfies{
    s1: assert (y == x+x);
}
```

We define an input `x` and an output `y`, both as natural numbers with a fixed range: `x` may take values $0, 1, \ldots 4$, and `y` may take values $0, 1, \ldots 9$. The task of our synthesis procedure is to determine the value of the expression `??` such that the output `y` equals 2 times `x` in the initial step. To solve that problem, we divide our program into two modules. The first one is essentially the sketch given above where an additional input variable `mul` has been introduced to replace the `??`. The second module implements the construction given in Theorem 1 and is responsible for chosing a proper value for `mul`:

```
macro N = 5;
module HelloWorldSketch(nat{N} ?x, nat{N*2} !y,nat{3} ?mul,) {
    y = x * mul;
    q_1: pause;
}

module HelloWorldSelector(nat{N} ?x, nat{N*2} !y,nat{3} ?mul,){
    nat{2} m;
    q_sel: pause;
    m = mul;
    q_1:pause;
    HelloWorldSketch(x,y,m);
}
satisfies{
    spec1: assert E X A X (y == x+x);
}
```

According to the construction described in the proof of Theorem 1, we introduce a new state `q_0` and for each oracle value `m` a new state `q_sel`. Note that in the above program, `q_sel` is a unique control flow state, that is however multiplied

with the possible data values that can be stored in the new local variable m. Note further that we have added a new input variable mul. The idea is that the environment is able to determine the input mul in every possible way that is consistent with its type, i.e., mul may have any of the values 0,1,2. The program takes only care of this value when the control flow is at the pause statement q_0, where the value of the input mul is stored in the local variable m. This local variable m is a memorized variable, and since there are no further assignments to m, this variable stores the value of the input variable mul at time t=1. Figure 2 shows the Kripke structure of the modified program.

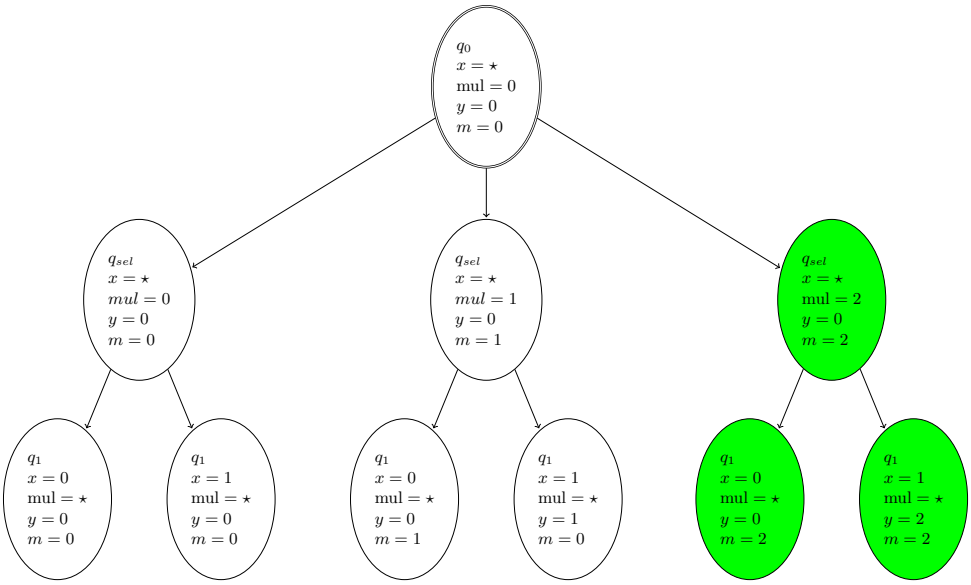

Fig. 2. Kripke structure for the Hello World Example

In order to simplify the picture, we reduced N to 2 and additionally replaced each "don't care" variable with a $\star$. Hence, each state labeled with some $\star$ represents more than one state. For example, we have two initial states, one labeled with $x = 0$ and one with $x = 1$. As can be seen, this Kripke structure satisfies the modified specification: for all successor states of $x = \star, mul = 2, y = 0, s = 2$, we have $y == x+x$. The idea behind this modified program is the same as in the previous section: In the selection state q_sel, the internal variable m stores the values of mul. Afterwards, m will never change and therefore stores the implementation if one exists (note that since mul is a input variable, every possibility is considered by the model checker). From each of those selection states, all initial states of the original program are reachable. Since we fixed the value of m, we try, in principle, each possible strategy in parallel.

It is not difficult to prove that the modifications we made lead to a program that is translated to an automaton whose corresponding Kripke structure is the result of the construction described in the proof of Theorem 1. Thus, we can easily perform the construction as a source code transformation which is very convenient since the Quartz compiler performs the remaining translation, and thus, the essential construction of Theorem 1, implicitly for us.

Checking this toy example with the model checker NuSMV is done in a couple of milliseconds and gives the correct result that the sketch can be completed to satisfy the specification. In order to actually synthesize a solution, all we have to do is to negate the specification. The values of each possible counterexample in the second state then determine a valid completion of the sketch.

```
macro M = 10;
macro N = 2;
module BinarySearchSketch([N]int{M} ?a, int{M} ?v, nat{N+1} i, ?sel, nat{1} →
     ? add,? sub) {
    nat{N} left,right ,mid;
    left  = 0;
    right  = N − 1;
    i  = N;
    while((sel ?  left <right :  left <=right) & i==N) {
        mid = (left  + right)  / 2;
        if(a[mid]<v)
            next(left)  = mid+add;
        else if(a[mid]>v)
            next(right) = mid−sub;
        else // v has been found
            next(i) = mid;
        w: pause;
    }
}
module BinarySearchSelector([N]int{M} ?a, int{M} ?v, nat{N+1} i, →
     ?sel,nat{1} ? add,? sub){
    bool s;
    nat{1} ad;
    nat{1} su;
    q_sel :  pause;
    s  = sel;
    ad=add;
    su=sub;
    q_1:pause;
    BinarySearchSketch(a,v,i,s ,ad,su);
}
```

Fig. 3. A Sketch for Binary Search

A Sketch for Binary Search In a same way, we obtain a sketch for the binary search algorithm we considered in Figure 1. The result is shown in Figure 3 where we replaced the problematic loop condition and the addition / subtraction of proper values by oracle variables. Again, those oracle variables are kept constant in a separate selector module so that we see how this principle "rewriting" on the source code level is performed. Solving the problem using NuSMV is done in under a minute and gives the correct values `sel=false, add=1, sub=0`.

```
macro N = 21;

module SingleRowNIMSketch(
    nat{N} numA, numB,
    nat{N+1} matches,
    nat{5} modulus,
    bool turnA
) {
    turnA = true;
    matches = N;
    while(matches>0) {
        if(turnA) {
            numA = (matches % modulus);
            next(matches) = matches−numA;
            next(turnA) = not turnA;
        } else {
            if(0<numB & numB<=matches) {
                next(matches) = matches−numB;
                next(turnA) = not turnA;
            }
        }
        pause;
    }
}

satisfies {
   A_always_wins: assert E X A X A G (modulus>0 & (matches==0 −> →
        !turnA));
}
```

Fig. 4. A Sketch for the Single Row NIM Game

Single Row NIM Game In the single row NIM game, N matches are put on a desk. The game is played by two players A and B that take either 1,2, or 3 matches in alternating turns from this pile. The player who will take the last

match has won. It is well-known that there is a optimal solution: the strategy is to choose a number of matches such that the remaining matches is a multiple of four. This is always possible for player A if initially the number of matches is not a multiple of four. Otherwise, player B will have the winning strategy.

Now suppose a programmer remembers that the solution has to do with some modulo operation. So, he might propose the sketch given in Figure 4 (where we omitted the selector module to safe space).

Notice that we have to additionally ensure that `modulus>0` to ensure that player A selects a value in the range 1,2,3. Again, checking that there does exist a implementation is done in a couple of seconds using NuSMV and generated a valid implementation with `modulus=4`. To see that this is the correct solution, remember from our hint that the number of remaining matches must be a multiple of four. Hence removing (matches $\mod 4$) piles ensures this invariant.

The Dutch Flag Problem Dijkstra presents the Dutch Flag Problem in [4], we follow here the description given in [3]): given an array of `n` pebbles, each of which is either red, white or blue, the algorithm must sort them in-place, in order of the colors in the Dutch flag: first red, then white, then blue. The algorithm may inspect at most one pebble at once and can only move the pebbles by swapping. A crucial constraint is that only a constant amount of storage can be used to remember the color of pebbles that have been examined.

We propose to use the sketch given in figure 5. The intended meaning of the variables is that `i` is the index-variable that is used to swap the array indices, all pebbles to the left of `R` are red and all pebbles to the right of `B` are blue.

There are two problematic program expressions: the first one is the loop condition while the second is when to increment `i`. We solve the problem using sketching by introducing two new oracle variables `op_loop` and `op_inc` that encode the right expression for the problematic cases. A natural guess would be that `i` should be incremented always. However, this is not true: whenever we found that position `i` contains a blue pebble, it is swapped with some other pebble from the right for which we have not yet determined the color. So it may be the case that we have to swap this pebble again. So can you imagine a right condition? With sketching, a programmer need not. He can simply let this hard work done by a computer as done in figure 5. The loop condition is not as problematic as the `i` increment condition, however looping until `N-1` is not true: instead we have to loop until `i>B`, since otherwise we would overwrite the blue zone.

Solving this sketching problem is harder than the previously considered: it took approximately 12 minutes on a 3.0 GHz Quad Core Pentium Duo which is however still acceptable and indeed, NuSMV generated us the correct program expressions **while** `(i<=B)` {`...`} and **if** `(a[i]!=blue) next(i)=i+1` .

```
macro red=0;
macro white=1;
macro blue=2;

macro N=10;

module DutchSketch ([N] nat{3} a,nat{3} op_loop,nat{7} op_inc){
    nat{N} i;
    nat{N} R;
    nat{N} B;
    i=0;
    R=0;
    B=N-1;
    while((op_loop==0) & (i<=(N-1)) |
          (op_loop==1) & (i<=R) |
          (op_loop==2) & (i<=B)){
        if (a[i]==red) {
            next(a[i])=a[R];
            next(a[R])=a[i];
            next(R)=R+1;
        }
        else if (a[i]==blue){
            next(a[i])=a[B];
            next(a[B])=a[i];
            next(B)=B-1;
        }
        if (
            (op_inc==0) |
            (op_inc==1) & (a[i]==red) |
            (op_inc==2) & (a[i]==white) |
            (op_inc==3) & (a[i]==blue) |
            (op_inc==4) & (a[i]!=red) |
            (op_inc==5) & (a[i]!=white) |
            (op_inc==6) & (a[i]!=blue)
           )
            next(i)=i+1;
        pause;
    }
}
satisfies{
    correct : assert E X A XA F ((forall (i=0 .. (N - 2)) (( (a[i]==white) -> →
        ( a[i+1] == white) | (a[i+1] == blue)) &
                                    (a[i]==blue) -> a[i+1]==blue)));
}
```

Fig. 5. Dutch Flag Problem

6 Conclusions

We considered the problem how one can determine desired program expressions of a synchronous reactive program so that given temporal logic specifications in CTL* are satisfied. We proved that the problem can be reduced to CTL* model checking, so that we can employ model checkers for the solution of the obtained CTL* model checking problems. We then have shown how the reduction to CTL* model checking can be even performed at the source code level of programs so that existing compilers actually perform this construction for us. All that has to be done is to introduce a new control flow location `q_sel` at the point of time where the choices for the oracle variables are made and these choices are stored in local variables at that position. These local variables are kept invariant in the following so that the right solution is stored in those local variables. Using state-of-the-art model checkers, we are then able to check whether a suitable choice exists, and if so, we can determine one by means of counterexample generation offered by model checkers.

Our work has been inspired by the sketching approach to program synthesis [16,18,17] that describes how a partial finite program can be completed such that a specification is satisfied after termination of the program. However, in contrast to [16,18,17], we consider reactive programs that have non-terminating behavior specified by temporal logic formulas which makes our problem harder than the one imposed in [16,18,17].

As already mentioned, algorithms to compute solutions of infinite games like [7,12,11,2,10] are able to deal with reactive systems and temporal logic specifications as well. However, the known approaches construct strategies represented as transition systems which do not allow one to easily derive the desired program expressions. Moreover, they allow the controller to make a different choice each time the expression is evaluated. In the binary search example, one could therefore use a different loop condition in each iteration, which is not useful for completing the program.

We therefore proposed an efficient solution to the automatic synthesis of program expressions that can be used to significantly simplify programming of difficult concurrent reactive programs. We see future work in two directions: the first direction is to define new language constructs that allow to automatically derive sketches. So instead of manually adding a oracle variable to encode different program expressions, we want the compiler to construct the expressions with the oracle variables based on more convenient expressions that only list the potential choices.

The more challenging future work is to develop new procedures so solve the sketching problem. It is well-known that SAT-based model checking procedures do often scale better in comparison to BDD-based model checking procedures. This is especially the case when there are many variables (which we introduce whenever many oracle variables are needed). A slightly modified bounded model checker may be a good starting point to simplify the problem so that on this simplified problem, BDD-based model checkers might succeed.

References

1. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
2. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 190:3–16, 2007.
3. R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In M. Hermenegildo and J. Palsberg, editors, *Principles of Programming Languages (POPL)*, pages 339–352, Madrid, Spain, 2010. ACM.
4. E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
5. E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier, 1990.
6. E. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *Principles of Programming Languages (POPL)*, pages 84–96, New Orleans, Louisiana, USA, 1985. ACM.
7. E. Filiot, N. Jin, and J.-F. Raskin. Compositional algorithms for LTL synthesis. In A. Bouajjani and W.-N. Chin, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 6252 of *LNCS*, pages 112–127, Singapore, 2010. Springer.
8. N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.
9. D. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1998.
10. A. Morgenstern and K. Schneider. A LTL fragment for GR(1)-synthesis. In *Intern. Workshop on Interactions, Games and Protocols (IWIGP)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), 2011.
11. A. Pnueli and R. Rosner. Distributed systems are hard to synthesize. In *Foundations of Computer Science (FOCS)*, pages 746–757, St. Louis, Missouri, USA, 1990. IEEE Computer Society.
12. S. Schewe and B. Finkbeiner. Bounded synthesis. In K. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *LNCS*, pages 474–488, Tokyo, Japan, 2007. Springer.
13. K. Schneider. *Verification of Reactive Systems - Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.
14. K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.
15. E. Shade. Size matters: lessons from a broken binary search. *Journal of Computing Sciences in Colleges (JCSC)*, 24(5):175–182, May 2009.
16. A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, California, USA, 2008.
17. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit streaming programs. In M. Hall, editor, *Programming Language Design and Implementation (PLDI)*, pages 281 – 294, Chicago, Illinois, USA, 2005. ACM.
18. A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, San Jose, California, USA, 2006. ACM.