Developing a Practical Reactive Synthesis Tool: Experience and Lessons Learned

Leonid Ryzhyk*
Samsung Research America
l.ryzhyk@samsung.com

Adam Walker
NICTA and UNSW, Sydney, Australia
adamwalker10@gmail.com

We summarise our experience developing and using Termite, the first reactive synthesis tool intended for use by software development practitioners. We identify the main barriers to making reactive synthesis accessible to software developers and describe the key features of Termite designed to overcome these barriers, including an imperative C-like specification language, an interactive source-level debugger, and a user-guided code generator. Based on our experience applying Termite to synthesising real-world reactive software, we identify several caveats of the practical use of the reactive synthesis technology. We hope that these findings will help define the agenda for future research on practical reactive synthesis.

Introduction

The reactive synthesis method aims to automatically generate an implementation of a reactive system based on a declarative specification of its desired behaviour [54]. While theoretical and algorithmic aspects of reactive synthesis are fairly well understood, there exists virtually no practical experience using this technology to synthesise real-world software and hardware. Existing synthesis tools have been developed as research vehicles for experimentation with new synthesis algorithms rather than as practical tools for use by hardware or software designers [6,10,19,27,35,36]. The gap between theory and practice hinders further advances in the field. For comparison, in the closely related field of automatic verification, the push towards real-world application of the model checking technology has fueled revolutionary advances of the technology itself [31].

For the past several years, our team has been working on using reactive synthesis to automate the development of operating system device drivers. The primary objective of the project has been to improve driver developers' productivity, as opposed to the creation of new synthesis algorithms or tools. However, early on in the project we discovered that none of the existing tools were suitable for our purposes. As a result, we created Termite, the first synthesis tool designed to make reactive synthesis accessible to software developers as a well-defined, predictable methodology.

Our previous publications describe the synthesis algorithm of Termite [69] and its application to the driver synthesis problem [61]. In this paper we focus on our experience designing and using a practical user-friendly synthesis tool. Our contributions are three-fold. First, we present the rationale behind the design of a practical reactive software synthesis toolkit. We consider each step involved in the synthesis process: specification development, strategy synthesis, specification debugging, and code generation, and identify the main barriers to making these tasks accessible to software developers.

Second, we present the key features of Termite designed to overcome these barriers: (1) an imperative C-like specification language, which enables developers to apply conventional programming techniques in writing specifications for reactive synthesis, (2) an interactive source-level debugger, which helps the

*Work completed at the University of Toronto

Submitted to:
SYNT 2016

```
// user has selected a record to play
task void evt_selection() {
  if ((jb.position == jb.selection) &&
      jb.arm_down)
    // selected record already on the
    // turntable-start playing
    jb.cmd_play();
  else if (jb.arm_down)
    // another record is on the
    // turntable, pick it up first
    jb.cmd_lift();
  else
    // rotate drum to selected position
    jb.cmd_rotate(jb.selection);
};
// drum has finished rotating
task void evt_rotated() {
  // place record on the turntable
  jb.cmd_put();
};
// mechanical arm has finished moving the
// record to the turntable or to the drum
task void evt_parked() {
  if(jb.have_selection){
    if (jb.arm_down)
      // record on the turntable-play it
      jb.cmd_play();
    else
      // previous record returned to drum,
      // rotate to the new selection
      jb.cmd_rotate(jb.selection);
  };
};
// record has been played to the end
task void evt_playback_complete() {
  // return it to the drum
  jb.cmd_lift();
};
```

Figure 1: Synthesised implementation of the jukebox controller.

developers to troubleshoot synthesis failures, and (3) a user-guided code generator, which combines the power of automation with the flexibility of manual development.

Third, and most importantly, we identify several caveats of the practical use of the reactive synthesis technology, based on our experience applying Termite to synthesising real-world reactive software. We hope that these findings will help define the agenda for future research on practical reactive synthesis. We formulate these caveats as “lessons learned” throughout the paper.

User-guided synthesis: the user perspective

Termite is intended for synthesis of reactive software modules such as device drivers or robotic controllers. A reactive module is typically embedded in a larger software stack written in C or other imperative language. The module is activated by invoking its event handler functions. It issues control actions via callbacks to the environment.

Example 1 (Running example: a jukebox controller) *Consider the control module of a mechanical jukebox. The jukebox consists of (1) a rotating drum holding up to* 256 *vinyl records, (2) a turntable, (3) a mechanical arm that transfers records from the drum to the turntable, and (4) a number pad for choosing the next record to play. The control module controls the jukebox mechanism by issuing commands to (1) rotate the drum to a specified position, (2) place the record at the current position on the turntable, (3) move the record from the turntable back to the drum, and (4) play the record on the turntable.*

Figure 1 shows our synthesised implementation of the control module with detailed comments. It consists of four event handlers: `evt_selection`*, triggered when the user selects a record to play,* `evt_rotated`*, triggered when the drum finishes rotation to the requested position,* `evt_parked`*, issued when the mechanical arm finishes transferring the record from the drum to the turntable or back, and* `evt_playback_complete`*, which indicates that the record has been played to the end.*

Synthesis workflow

The Termite synthesis workflow is shown in Figure 2. Termite takes as input a specification in the *Termite Specification Language* (TSL). The specification is compiled into a symbolic GR-1 [53] (Section 3), which is then solved by the Termite game solver. The game solver produces either a winning

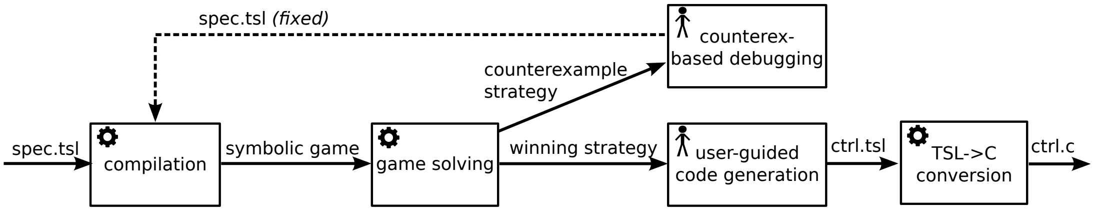


Figure 2: Termite synthesis workflow. The gear and the stick figure distinguish fully automatic and user-guided stages of the workflow.

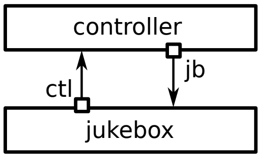
strategy for the controller or, if one does not exist, a counterexample strategy on behalf of the environment. In the former case, the winning strategy is converted into a controller implementation in TSL in a user-guided fashion. The resulting implementation is automatically translated into a C module. In the latter case, the user explores the counterexample strategy using the *Termite visual debugger* in order to track the cause of the synthesis failure down to a defect in the input specification.

The Termite Specification Language

Rationale Our main language design objective is to facilitate the specification and synthesis of a particular class of target systems, namely reactive software modules, while minimising modelling errors. We therefore prioritise modelling convenience over generality. In addition to this main principle, we identify the following requirements for the language:

1. *Modularity.* The ability to model each component of a complex system as a separate module enables specification reuse, which is crucial in practical applications of reactive synthesis, as explained in Section 2.3.
2. *Programmer-friendly syntax.* TSL is intended for use by software developers rather than formal methods experts. As such, it must support modelling using familiar programming concepts and idioms.
3. *Modelling of hybrid systems.* In many applications of reactive synthesis, including device driver development and robotics, we deal with systems that contain both software and hardware components. We therefore designed TSL to facilitate modelling of such hybrid systems.

Key features of TSL Following the first requirement, TSL supports the development of modular specifications through object-oriented modelling. A TSL class, called *template*, consists of *variables*, *methods* and *processes*, which together model a component of the environment or the reactive module that is being synthesised. A TSL specification consists of multiple statically created template *instances* that together model a complete system. Templates communicate by invoking each other's methods via typed interfaces exported through *ports*. The figure on the right shows the two templates involved in our running example. Arrows represent bindings of template ports. Lines 2-3 in Figure 3 instantiate this architecture in TSL.



Following the second requirement, we designed TSL as an imperative C-style language (Section 2.3). To meet the third requirement, we extend TSL with hardware modelling features, such as arbitrary-size bit vectors, bit slices, combinatorial logic, etc. In addition to standard programming language features, TSL supports constructs specific to reactive synthesis, introduced below. A complete description of TSL syntax can be found in the language reference manual [58].

Developing input specifications

A Termite specification consists of at least two templates, which model the environment and the controller. The environment model can be seen as a *test harness* for the controller, which stimulates controller inputs and validates its outputs. TSL allows structuring such a model similar to how one would write a software test harness: it consists of a workload generator process that generates a stream of requests to the controller and callback methods invoked by the controller.

The controller template declares methods of the controller and specifies the control objective. Termite supports GR-1 objectives (see Piterman et al. [53] and Section 3), defined in terms of one or more *goal conditions*, which the controller must satisfy infinitely often. For example, a goal condition may require the controller to eventually complete every request from the environment. In addition, the controller template can provide partial implementation of controller methods. During synthesis, Termite tries to extend this to a complete implementation that satisfies the objective.

Example 2 *In our running example, the environment consists of the jukebox mechanism modelled by the* `jukebox` *template in line 6, Figure 3. The mechanism can be in one of the states listed in lines 9-13. Variables in lines 15-23 model the state of the number pad, the drum and the mechanical arm. The main part of the model is the process in line 24, which generates a stream of events to the controller. The first part of the loop body (lines 27-32) simulates the user selecting the next record to play. In our simple jukebox, record selection is only enabled when there is no current selection yet and the jukebox mechanism is idle (line 27). Even when the event is enabled, it is not guaranteed to occur—it is up to the user when to make a selection. We model such external choice in line 29, where the asterisk ('∗') represents a non-deterministically chosen value. Likewise, the record number selected by the user is modelled as a non-deterministic value passed as an argument to the event handler in line 31.*

The second part of the loop (lines 34-49) generates a completion event for the current operation peformed by the jukebox. For example, if the jukebox is in the `put` *state, transferring the record from the drum to the turntable, it returns to the idle state (line 38), updates arm position, and issues the* `evt_parked` *event, which enables the controller to respond to the state transition.*

Lines 53-72 show environment callbacks that can be invoked by the controller, designated by the `controllable` *qualifier. Each callback contains an assertion that specifies when it is safe to issue the command. For example, the rotate command (line 53) can only be issued when the mechanism is idle and the mechanical arm is in the upward position (i.e., there is no record on the turntable).*

Lines 75-82 show the controller template. The goal condition in line 76 requires that the system is in a state where there is no user selection on the number pad. This can only be achieved by playing each selected record. The bodies of all template methods contain elipsis ("`...`*"), which are placeholders for synthesised code, referred to as* magic blocks.

Magic blocks are regular TSL statements and can be used anywhere in the code. This enables the developer to enforce a preferred implementation structure in advance by provinding a partial implementation of a method, for example:

```
task void evt_selection() {
  if (((jb.position == jb.selection) && jb.arm_down)) {...;}
  else if (jb.arm_down) {...;}
  else {...;};
};
```

Multiple TSL statements are combined in a single atomic transition. The transition terminates upon reaching a special `pause` statement or a magic block. For example, from its initial state in line 25, the `pjukebox` process may execute atomically until calling the `evt_selection` method in line 32,

```
1  template main
2    instance controller ctl(jb);
3    instance jukebox    jb(ctl);
4  endtemplate
5
6  template jukebox(controller ctl)
7    // states of the jukebox mechanism
8    typedef enum {
9      idle,
10     spin,//spinning the drum
11     play,//playing a record
12     put, //moving record to turntable
13     lift //returning record to drum
14   } state_t;
15   state_t state = idle;//current state
16   // record has been selected
17   bool have_selection = false;
18   // selected record number
19   uint8 selection;
20   // mechanical arm position
21   bool arm_down;
22   // current drum position
23   uint8 position;
24   process pjukebox {
25     forever {
26       // Simulate user selection
27       if (!have_selection &&
28           state == idle) {
29         if (*) {
30           have_selection = true;
31           selection = *;
32           ctl.evt_selection();};};
33       // Simulate command completion
34       if (state == spin) {
35         state = idle;
36         ctl.evt_rotated();
37       } else if (state == put) {
38         state = idle;
39         arm_down = true;
40         ctl.evt_parked();
41       } else if (state == lift) {
42         state = idle;
43         arm_down = false;
44         ctl.evt_parked();
45       } else if (state == play) {
46         state = idle;
47         have_selection = false;
48         ctl.evt_playback_complete();
49       };
50       pause;
51     };
52   };
53   task controllable void
54   cmd_rotate(uint8 pos) {
55     assert(state==idle && !arm_down);
56     state = spin;
57     position = pos;
58   };
59   task controllable void cmd_put() {
60     assert(state==idle && !arm_down);
61     state = put;
62   };
63   task controllable void cmd_lift() {
64     assert(state==idle && arm_down);
65     state = lift;
66   };
67   task controllable void cmd_play() {
68     assert(state==idle && arm_down);
69     assert(have_selection &&
70             (position==selection));
71     state = play;
72   };
73 endtemplate
74
75 template controller(jukebox jb)
76   goal play_selection =
77        (jb.have_selection == false);
78   task void evt_selection(){...;};
79   task void evt_rotated(){...;};
80   task void evt_parked(){...;};
81   task void evt_playback_complete(){...;};
82 endtemplate
```

Figure 3: Input TSL specification for the jukebox controller.

where it stops at the magic block in line 78. Alternatively, if none of the conditions in the body of the process holds, the transition stops at the pause location in line 50.

Note that in this example the specification is longer and more complicated than the resulting synthesised controller (Figure 1). We observed a similar effect in synthesising real-world device drivers. This is not surprising, as artificially engineered systems are usually designed to have a simple control strategy. This is in contrast to, for example, competitive games such as checkers, where the winning strategy is much more complicated than the description of game rules.

In order to obtain a productivity improvement with reactive synthesis, one must minimise the effort invested in the development of input specifications. This can be achieved by reusing existing specifications where possible. For example, in device driver synthesis, the driver environment consists of the operating system (OS) and the hardware device [61]. The OS specification is developed once for each class of drivers (e.g., network drivers) and used to synthesise many drivers of this type. The device specification is produced by hardware developers as part of the circuit design process and can be reused, with some modifications, in driver synthesis. The object-oriented nature of TSL comes in handy here, as it allows decomposing specifications into reusable modules.

Lesson 1 *Specification reuse is essential to achieving a productivity gain with reactive synthesis.*

Counterexample-based debugging

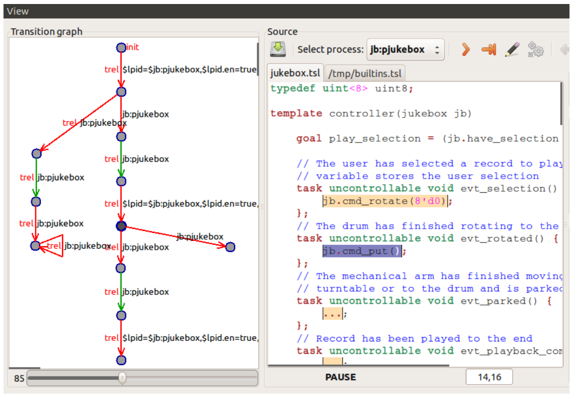
Rationale The input specification may contain defects making it *unrealisable*, i.e., such that there does not exist a specification-compliant controller implementation. In this case, Termite produces a certificate of unrealisability in the form of a counterexample *spoiling strategy* on behalf of the environment. By following this strategy, the environment is able to either force the system into an error state by violating one of its assertions or to permanently keep the system from entering one of its goal regions. By studying the counterexample strategy, the user can trace the synthesis failure to a bug in the specification. However, such counterexample-based debugging is a non-trivial task, which requires special tool support. The counterexample strategy may represent a complex branching behaviour that does not have a compact user-readable representation.

The interactive debugger Termite helps the user to understand counterexample strategies by exploring them interactively. To this end, the Termite debugger simulates a game where the user plays on behalf of the controller, while the debugger responds on behalf of the environment, according to the counterexample strategy. The debugger maintains the look and feel of a conventional software debugger, where the user steps through the code of the system, observing its state and control flow. Unlike a conventional debugger, Termite automatically resolves environment choices in accordance with the counterexample strategy. At every step of the debugging session, it chooses a TSL process to run and assigns concrete values to all non-determimistic expressions (‘$*$’) it evaluates.

When the debugger schedules a process that is currently inside a magic block, it allows the user to choose a controller action to execute. In TSL, a controller action corresponds to an invocation of one of controllable methods. The user specifies the action that, they believe, represents a correct controller behaviour by either typing a line of code or via an interactive dialog.

At some point in the game the user observes an unexpected behaviour, e.g., one of user-provided actions does not change the state of the system as expected or one of environment actions triggers an assertion violation. Based on this information, the user can revise the faulty specification.

The debugger visualises the transition graph explored during the debugging session, as shown in the screenshot on the right. It allows the user to go back to a previously explored state and try a different controller behaviour from there.



```
typedef uint<8> uint8;

template controller(jukebox jb)

    goal play_selection = (jb.have_selection

    // The user has selected a record to play
    // variable stores the user selection
    task uncontrollable void evt_selection()
        jb.cmd_rotate(8'd0);
    };
    // The drum has finished rotating to the
    task uncontrollable void evt_rotated() {
        jb.cmd_put();
    };
    // The mechanical arm has finished moving
    // turntable or to the drum and is parked
    task uncontrollable void evt_parked() {
        ...;
    };
    // Record has been played to the end
    task uncontrollable void evt_playback_com
```

Example 3 *Consider a defective version of our running example where the assignment in line 61 is missing. This is equivalent to the jukebox ignoring the command to put the record on the turntable. Termite detects that the specification is unrealisable and produces a counterexample strategy. During the debugging session, the debugger initialises the* `arm_down` *variable to false and* `position` *to* 0*. It then steps through lines 27-32 of the jukebox model, setting the non-deterministic condition in line 29 to true and the value in line 31 to* 0*, thus simulating a request to play record number* 0 *(i.e., the record in the current position). The debugger stops in the magic block in line 78, passing control to the user. The user attempts to* *complete the request by typing the command `jb.cmd_put()` in the magic block to place the record on the turntable. The user expects the environment to respond by invoking the `evt_parked` callback. However, due to the specification bug, the `(state==put)` condition in line 37 does not hold, and the debugger keeps iterating the `forever` loop without ever invoking any of the controller callbacks. Upon observing this behaviour, the user goes back to the invocation of the `cmd_put()` method and single-steps its execution, finally discovering that it does not set the `state` variable as expected. We have traced the synthesis failure to its root cause and can now easily fix it by adding the missing statement.*

Our counterexample-guided methodology greatly simplifies specification debugging; however it also uncovers an important caveat of reactive synthesis: in order to find specification defects, the developer must have a good understanding of both the input specifications and the expected controller implementation.

Lesson 2 *Synthesis does not replace human expertise, as synthesising a reactive module requires similar knowledge and skills as developing the module manually.*

Furthermore, the above example shows that debugging synthesis failures requires manually providing parts of the controller implementation. In the worst case, the user effectively ends up implementing the entire controller manually before finding a bug in the specification. One way to avoid this is to perform synthesis *incrementally*. To this end, we restrict the input specification to only exercise a subset of the controller functionality (e.g., only initialisation). This can be achieved by commenting out parts of the environment model or by disabling all but one goal condition. Having synthesised the selected fragment of the controller, we gradually enable additional features until a complete controller has been synthesised. If at some point, the specification becomes unrealisable due to a defect, only a modest manual effort is required to locate the defect. Termite supports the incremental synthesis methodology by accepting partially implemented controller templates (as described in Section 2.3).

Lesson 3 *The incremental approach is necessary to streamline specification debugging and reduce manual effort involved in synthesis.*

User-guided code generation

Rationale Early versions of Termite synthesised the reactive module implementation from the input specification fully automatically. Having experimented with this push-button approach for several years, we found that it consistently failed to produce satisfactory implementations. First, it proved hard in practice to capture all important aspects of the desired behaviour in the input specification. For example, the original version of the jukebox controller synthesised by Termite left the last played record on the turntable, i.e., it was missing the `cmd_lift` command in line 37 in Figure 1. This implementation satisfies the specification, but delivers suboptimal user experience due to an increased delay before playing the next selected record. While in principle the input specification can be extended to incorporate such additional requirements, such extensions tend to be awkward and error-prone. In our experience, it is typically easier to achieve the desired behaviour via a manual modification to the generated code than to enforce it indirectly via changes to the specification.

Second, Termite currently does not handle extensions of the reactive synthesis problem, such as quantitative synthesis [17], timed synthesis [14], and synthesis with imperfect information [26, 55]. As a result, it does not automatically enforce non-functional properties related to time, performance, power consumption, state observability, etc. This is a deliberate design choice, as these extensions make synthesis more computationally expensive and hence less practical.

Third, we found it difficult to achieve a clean, human-readable code structure automatically. This is an important concern, as code inspection remains a key quality assurance method even when using automatic synthesis.

Lesson 4 *Given current state of the art in reactive software synthesis, automatic tools are unlikely to replace human expertise in the foreseeable future. A practical synthesis tool must include user input in the synthesis process in a way that combines the power automation with the flexibility of conventional development.*

User-guided code generation Our solution to this problem is based on the following principles:

1. *The two-phase process.* We separate synthesis into a fully automatic *game solving* phase and an interactive *code generation* phase. The former computes a winning strategy for the controller, which maps states of the system to the set of winning moves in each state, and can be seen as a compact representation of all possible specification-compliant controller implementations. The latter extracts one specific controller implementation from the strategy in a user-guided fashion.
2. *The user is in control.* The user interacts with the tool during the code generation phase by arbitrarily changing or amending the generated code. The automatic code generator is invoked on demand. It never changes any of the user-provided code, but rather tries to extend it to a complete specification-compliant implementation.
3. *Enforcing correctness.* The resulting combination of manual and generated code is continuously validated against the input specification to preserve the strong correctness guarantees offered by automatic synthesis.

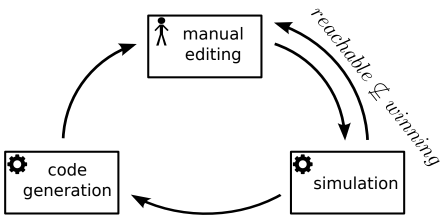


Figure 4: Code generation flow.

Figure 4 illustrates the interactive code generation phase, which iterates over three steps. At the first step, the user edits the partially synthesised implementation inside magic blocks (other parts of the module remain read-only). At the moment, Termite does not allow creating new functions interactively; however such capability can be readily added in the future.

The manual editing step ends when the user chooses a partially filled magic block and invokes the automatic code generator via a GUI button to produce the next code statement. This starts the second step, where Termite symbolically simulates execution of the system, including all of the manual and automatic code produced so far. The simulation returns a symbolic representation of all possible *reachable states* of the system at the control location where code generation has been requested. Termite compares the reachable set against the winning set computed during the game solving phase. If it finds that at least one of the reachable states are not winning, this means that the partial controller implementation generated so far is incorrect due to the manual changes made by the user. Furthermore, Termite is able to produce a counterexample explaining the failure.

During the third step, the Termite code generation algorithm translates the winning strategy in the reachable set into a TSL statement. It first checks if there exists a common winning action in all reachable states. Such an action corresponds to a simple TSL statement, i.e., an invocation of a controllable method. Otherwise, the algorithm computes a partitioning of the reachable set into subsets such that there exists a common winning action in each subset. This corresponds to a conditional branching statement, such as the one in lines 3-14 in Figure 1. Finally, the newly generated statement is inserted at the corresponding control location, and Termite returns to the interactive editing mode.

Example 4 *Starting with the empty template, shown in lines 78-81 in Figure 3, Termite produces code for the* `evt_selection` *and* `evt_rotated` *methods, which does not require any manual changes. Next, it suggests an empty implementation for the* `evt_playback_complete` *method. We are not satisfied with this implementation and manually modify it to issue the* `cmd_lift` *command (line 37, Figure 1) in order to return the current record to the drum. Finally, we proceed to generate the* `evt_parked` *method in line 23. Note that the synthesised implementation takes into account manual changes made at the previous step. In particular, the first branch of the generated code handles completion of the* `cmd_lift` *command issued manually in line 37.*

Termite internals

In this section we describe the internal design of the Termite toolkit, highlighting some of the important implementation issues.

Termite computes a strategy for the controller by representing the synthesis problem as a two-player game between the controller and the environment.

Definition 1 (GR-1 game [53]) *Let $F_t(V)$ be a set of Boolean formulas over variables V in some theory t (at the moment, Termite supports the theory of fixed-size bit vectors). A concrete symbolic GR-1 game $G = \langle X, I, Y_c, Y_u, \delta_c, \delta_u, \Gamma, \Phi \rangle$ consists of a finite set of state variables X, an initial condition $I \in F_t(X)$, finite sets Y_c and Y_u of controllable and uncontrollable action variables, controllable transition relation $\delta_c \in F_t(X, Y_c, X')$, uncontrollable transition relation $\delta_u \in F_t(X, Y_u, X')$ (where X' are the next-state versions of state variables X), $\Gamma = (\gamma_1, \dots, \gamma_k)$, $\gamma_i \in F_t(X)$ is a finite set of goal regions, and Φ is a finite set of fairness conditions $\Phi = (\phi_1, \dots, \phi_k)$, $\phi_i \in F_t(X, Y_u)$.*

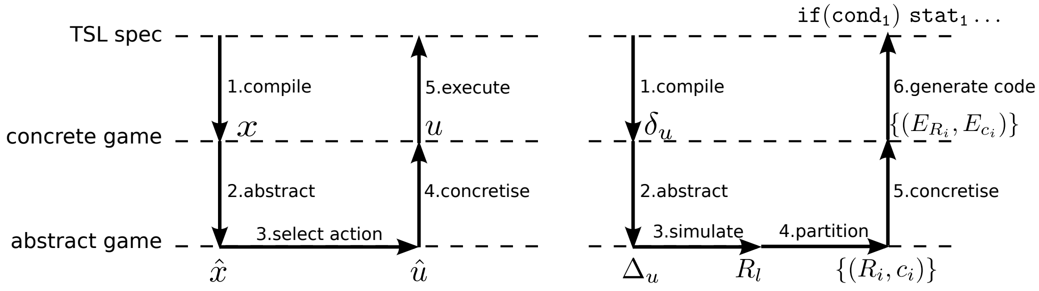
The controller and the environment participate in the game by performing actions allowed by their respective transition relations. To win the game, the controller must infinitely often force the system into each goal region γ_i, assuming fair execution where each of the fairness conditions ϕ_j holds infinitely often.

The TSL compiler converts a TSL specification into a GR-1 game G, where state variables X model the state of the TSL program, controllable actions encode invocations of controllable methods available to the controller, uncontrollable actions model atomic transitions of TSL processes goal sets Γ encode goals declared in the input specification, and fairness conditions Φ enforce fair scheduling by making sure that every runnable process gets scheduled eventually.

We solve the resulting game G using the algorithm by Piterman et al. [53], which involves exhaustive exploration of the state space of the game. The Termite game solver mitigates the state explosion problem by constructing and iteratively refining an abstraction of the game that approximates states and actions using Boolean predicates. Abstraction refinement is performed fully automatically. We refer the reader to our earlier publication [69] for a detailed description of the abstraction refinement algorithm of Termite.

Throughout the synthesis process, Termite maintains three distinct representations of the synthesis problem: (1) a TSL specification, (2) a concrete symbolic game, and (3) an abstract game. All interactions with the user, including specification development, debugging, and interactive code generation, occur at the TSL level. Internally, Termite completes all user requests at the abstract level and lifts results of this computation back to the TSL level, which enables it to achieve interactive performance during debugging and code generation.

As an example, consider one iteration of a debugging session, where the user invokes Termite to pick an uncontrollable action according to the counterexample strategy. It involves the following steps



(a) debugging

(b) code generation

Figure 5: Navigating between different representations of the synthesis problem during debugging and code generation.

(Figure 5a). (1) *Compilation:* map the state of the program in the debugger to a state x of the underlying game. (2) *Abstraction:* compute corresponding state $\hat{x}$ of the abstract game. (3) *Action selection:* pick action $\hat{u}$ from the abstract counterexample strategy in state $\hat{x}$. (4) *Concretisation:* compute a concrete action u that matches the abstract action $\hat{u}$ (there can exist multiple such concrete actions). (5) *Execution:* the resulting uncontrollable action u encodes the selection of a TSL processes to execute, along with choices of all non-deterministic values encountered by the process; the user can now simulate this transition at the TSL level by single-stepping through its statements or by running it to completion.

As another example, Figure 5b shows one iteration of an interactive code generation session, where the user has made some changes to the synthesised code and invokes Termite to generate a code statement for a selected magic block. (1) *Compilation:* invoke the TSL compiler on the modified specification. All code written or generated so far becomes part of the uncontrollable transition relation δ_u (as Termite is not allowed to modify the behaviour of this code). (2) *Abstraction:* compute an abstract approximation Δ_u of δ_u. (3) *Simulation:* simulate all possible executions of the partial implementation generated so far (see Section 2.5): $R = \Delta_u^*(\mathscr{I})$, where Δ_u^* is the transitive closure of Δ_u, $\mathscr{I}$ is the abstract initial state, and R is the set of reachable abstract states of the system. Restrict R to a subset $R_l \subseteq R$ at the control location l where code generation has been requested. (4) *Partitioning:* compute a partitioning $R_l = \dot{\bigcup}_i R_i$ of R_l, such that for each partition R_i, there exists a common winning abstract controllable action c_i. (5) *Concretisation:* convert each pair (R_i, c_i) into an expression $E_{R_i}(X)$ and $E_{c_i}(X, Y_c)$. These expressions encode a subset of concrete states and the concrete action to be performed when in one of these states. (6) *Code generation:* convert each E_{R_i} into a TSL expression $\texttt{cond}_\texttt{i}$ and each E_{c_i} into a TSL statement $\texttt{stat}_\texttt{i}$. Generate a conditional TSL statement of the form: $\texttt{if}(\texttt{cond}_1)\ \texttt{stat}_1\ \texttt{else}\ \texttt{if}(\texttt{cond}_2)\ \texttt{stat}_2 \ldots \texttt{else}\ \texttt{stat}_\texttt{n}$.

Evaluation

We implemented all components of the Termite toolkit, including the TSL compiler, the game solver, the counterexample debugger, the user-guided code generator, and the TSL-to-C compiler, in Haskell. Our implementation uses the CUDD BDD library for efficient symbolic manipulations over Boolean relations, and the Z3 SMT solver for satisfiability queries over the theory of bit vectors, as described in [69]. Termite is publicly available under the BSD license and can be downloaded from the project website `http://termite2.org`.

We evaluate Termite by synthesizing drivers for eight I/O devices: a UVC-compliant USB webcam, the 16550 UART serial controller, the DS12887 real-time clock, the IDE disk controller for Linux, as well as seL4 drivers for I2C, SPI, and UART controllers on the Samsung exynos 5 chipset 2 and SPI

	input spec(loc)	vars(bits)	synt. time(s)	synthesised code (loc)
webcam	487	128 (125565)	215	113
16450 UART	289	81 (407)	210	74
exynos UART	380	80 (1185)	645	37
STM SPI	317	68 (389)	67	24
exynos SPI	327	83 (933)	25	40
exynos I2C	326	65 (303)	45	79
RT clock	370	92 (810)	56	84
IDE	668	114 (1333)	285	94

Table 1: Summary of case studies.

controller on the STM32F10 chipset[1]. These devices are representative of simple peripherals found in many embedded platforms. The main barrier to synthesizing drivers for more advanced devices, e.g., high-performance network controllers, is the current lack of support for synthesis of direct memory access (DMA) code in Termite. Such code cannot be generated using only reactive synthesis techniques and requires support for synthesis of pointer-based data structures such as nested lists, circular packet buffers, etc.

Table 1 summarises our case studies. It shows the size (in lines of code) of the input TSL specification, the complexity of each benchmark in terms of the number of bit-vector variables in the GR-1 game and the total number of bits in these variables, the time the game solver took to compute a winning strategy for the controller, and the size (in lines of TSL code) of the synthesised implementation. As discussed in Section 2.3, input specifications in our case studies are larger than the synthesised implementations. We are able to mitigate the problem by decomposing the specification into reusable device and OS models.

Performance As can be seen from Table 1, Termite is able to solve games with hunderds to thousands of state variables. These results illustrate the effectiveness of our predicate abstraction algorithm combined with the use of BDDs for symbolic game solving. However, the battle for scalable reactive synthesis is far from over. Complex I/O devices can have hundreds of configuration registers, which translates to tens of thousands of state bits, which is beyond the reach of all existing synthesis algorithms, including those used in Termite. We hope that ongoing research into scalable reactive synthesis algorithms will help to close this gap.

Interactive debugging We found counterexample-based debugging to be crucial to streamlining the synthesis process. Before the debugger was available, we relied on code inspection for troubleshooting synthesis failures, which proved to be an unpredictably long process. The Termite debugger streamlines this process, giving us the confidence that any failure can be localised by following well-defined steps. A typical debugging session takes a few minutes and involves entering only a few commands manually before the defect is localised.

User-guided code generation In our case studies, 60% to 90% of the code was generated fully automatically, with the rest produced in a user-guided fashion. The main sources of manual changes were (1) suboptimal performance or structure of the auto-generated code, and (2) handling of partially observable state. The latter problem occurs in device drivers, as the driver strategy often depends on the internal state of the hardware device, which is not directly observable by the driver. As a result, Termite generates invalid code that directly accesses device-internal variables. We modified Termite to report such situations to the user, prompting them to provide a functionally equivalent valid implementation. For example, whenever the auto-generated code accesses a device-internal register, it may be possible to obtain the value of the register by issuing an additional command to the device.

[1] Our case studies are discussed in detail in [61].

Related work

While our work is the first to address the quality of synthesised code in reactive software synthesis, in the adjacent field of hardware synthesis significant progress has been made on the circuit minimisation problem, concerned with converting the synthesised strategy into a circuit with minimal number of gates [7,28].

Previous reactive synthesis tools support specification languages designed to be as simple and expressive as possible. For example, Unbeast [27] and Acacia+ [10] support synthesis from LTL specifications, while RATSY [6] handles specifications written as deterministic Büchi automata. Furthermore, many of these languages are geared towards hardware synthesis. One example is the Extended AIGER format used in the reactive synthesis competition [34]. In contrast, TSL is the first domain-specific language for reactive *software* synthesis.

Counterexample-based debugging is a standard method of troubleshooting synthesis failures. Anzu [40] and RATSY [6] implement an interactive version of this technique. Another line of work explores ways to produce minimal counterexamples that help locate the bug faster [41]. Termite complements these techniques by lifting counterexample-based debugging to the source code level.

Techniques presented in this paper are independent of the choice of the underlying synthesis algorithm and will benefit from any future advances in this area. In particular, recent research has proposed efficient symbolic BDD and SAT-based synthesis algorithms [9,46,48,53]. Another line of work explores the use of abstraction to mitigate the state explosion problem [1,2,26,32].

The idea of automatically completing partially specified programs has been explored in syntax-guided and functional synthesis [39,63], as well as in LTL synthesis [49].

Conclusion

While push-button software synthesis may not be feasible in the short-term perspective, we argue that developers can take advantage of the reactive synthesis technology by combining the power of automation with the flexibility of manual development. We presented the design and implementation of the first synthesis tool that enables such combination via three new techniques: user-guided code generation, code-centric interface, and interactive source-level debugging.

References

[1] Luca de Alfaro, Patrice Godefroid & Radha Jagadeesan (2004): *Three-valued abstractions of games: uncertainty, but with precision*. In: *LICS*, Turku, Finland, pp. 170–179.

[2] Luca de Alfaro & Pritam Roy (2007): *Solving Games Via Three-Valued Abstraction Refinement*. In: *CONCUR*, Lisboa, Portugal, pp. 74–89.

[3] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer & S. K. Rajamani (2001): *Partial-Order Reduction in Symbolic State-Space Exploration*. *Formal Methods in System Design* 18(2), pp. 97–116.

[4] Sidney Amani, Peter Chubb, Alastair Donaldson, Alexander Legg, Keng Chai Ong, Leonid Ryzhyk & Yanjin Zhu (2014): *Automatic Verification of Active Device Drivers*. *ACM Operating Systems Review* 48(1).

[5] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani & Abdullah Ustuner (2006): *Thorough Static Analysis of Device Drivers*. In: *1st EuroSys Conference*, Leuven, Belgium, pp. 73–85.

[6] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan & Richard Seeber (2010): *RATSY–a New Requirements Analysis Tool with Synthesis*. In: CAV, Edinburgh, UK, pp. 425–429.

[7] Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Koenighofer & Florian Lonsing (2014): *SAT-Based Methods for Circuit Synthesis*. In: FMCAD, Lausanne, Switzerland.

[8] Roderick Bloem, Robert Könighofer & Martina Seidl (2013): *SAT-Based Synthesis Methods for Safety Specs*. CoRR abs/1311.3530.

[9] Roderick Bloem, Robert Könighofer & Martina Seidl (2014): *SAT-Based Synthesis Methods for Safety Specs*. In: VMCAI'14, Springer Berlin Heidelberg, San Diego, CA, USA, pp. 1–20.

[10] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2012): *Acacia+, a Tool for LTL Synthesis*. In: CAV, Berkeley, California, USA, pp. 652–657.

[11] Randal E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers 35, pp. 677–691.

[12] Lukai Cai & Daniel Gajski (2003): *Transaction level modeling: an overview*. In: "1st International Conference on Hardware/Software Codesign and System Synthesis", Newport Beach, CA, USA, pp. 19–24.

[13] Michael L. Case, Alan Mishchenko & Robert K. Brayton (2006): *Inductively Finding a Reachable State Space Over-Approximation*. In: Proceedings of the 15th International Workshop on Logic and Synthesis.

[14] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen & Didier Lime (2005): *Efficient On-the-fly Algorithms for the Analysis of Timed Games*. In: CONCUR, San Francisco, CA, USA, pp. 66–80.

[15] Pavol Cerny, Thomas Henzinger, Arjun Radhakrishna, Leonid Ryzhyk & Thorsten Tarrach (2013): *Efficient synthesis for concurrency by semantics-preserving transformations*. In: CAV, Saint Petersburg, Russia.

[16] Pavol Cerny, Thomas Henzinger, Arjun Radhakrishna, Leonid Ryzhyk & Thorsten Tarrach (2014): *Regression-free synthesis for concurrency*. In: CAV, Vienna, Austria.

[17] Krishnendu Chatterjee, Mickael Randour & Jean-François Raskin (2012): *Strategy Synthesis for Multi-dimensional Quantitative Objectives*. In: CONCUR, Newcastle, UK, pp. 115–131.

[18] Mingsong Chen, Prabhat Mishra & Dhrubajyoti Kalita (2007): *Towards RTL test generation from SystemC TLM specifications*. In: HLDVT'07, pp. 91–96.

[19] Chih-Hong Cheng, Alois Knoll, Michael Luttenberger & Christian Buckl (2011): *GAVS+: An Open Platform for the Research of Algorithmic Game Solving*. In: TACAS, Saarbrücken, Germany, pp. 258–261.

[20] Vitaly Chipounov, Volodymyr Kuznetsov & George Candea (2012): *The S2E Platform: Design, Implementation, and Applications*. ACM Transactions on Computer Systems 30(1), pp. 2:1–2:49.

[21] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem & Dawson Engler (2001): *An Empirical Study of Operating Systems Errors*. In: 18th ACM Symposium on Operating Systems Principles, Lake Louise, Alta, Canada, pp. 73–88.

[22] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2003): *Counterexample-guided abstraction refinement for symbolic model checking*. Journal of the ACM 50, pp. 752–794.

[23] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina & Karen Yorav (2004): *Predicate Abstraction of ANSI-C Programs Using SAT*. Formal Methods in System Design 25(2-3), pp. 105–127.

[24] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani & Damien Zufferey (2013): *P: safe asynchronous event-driven programming*. In: 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation, Seattle, Washington, USA, pp. 321–332.

[25] Rayna Dimitrova & Bernd Finkbeiner (2008): *Abstraction Refinement for Games with Incomplete Information*. In: FSTTCS, Bangalore, India.

[26] Rayna Dimitrova & Bernd Finkbeiner (2012): *Counterexample-Guided Synthesis of Observation Predicates*. In: FORMATS, London, UK, pp. 107–122.

[27] Rüdiger Ehlers (2010): *Symbolic Bounded Synthesis*. In: CAV, Edinburgh, UK.

[28] Rüdiger Ehlers, Robert Könighofer & Georg Hofferek (2012): *Symbolically synthesizing small circuits*. In: *FMCAD*, Cambridge, UK, pp. 91–100.

[29] Cormac Flanagan & Shaz Qadeer (2002): *Predicate Abstraction for Software Verification*. In: *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, pp. 191–202.

[30] Archana Ganapathi, Viji Ganapathi & David Patterson (2006): *Windows XP Kernel Crash Analysis*. In: *20th USENIX Large Installation System Administration Conference*, Washington, DC, USA, pp. 101–111.

[31] Orna Grumberg & Helmut Veith, editors (2008): *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg.

[32] Thomas A. Henzinger, Ranjit Jhala & Rupak Majumdar (2003): *Counterexample-guided control*. In: *ICALP*, Eindhoven, The Netherlands, pp. 886–902.

[33] Intel Corporation: *CoFluent Technolofy*. `http://www.intel.com/content/www/us/en/cofluent/cofluent-difference.html`.

[34] Swen Jacobs (2014): *Extended AIGER Format for Synthesis*. *CoRR* abs/1405.5793.

[35] Barbara Jobstmann & Roderick Bloem (2006): *Optimizations for LTL Synthesis*. In: *FMCAD*, San Jose, CA, USA, pp. 117–124.

[36] Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer & Roderick Bloem (2007): *Anzu: A Tool for Property Synthesis*. In: *CAV*, Berlin, Germany, pp. 258–262.

[37] Asim Kadav, Matthew J. Renzelmann & Michael M. Swift (2009): *Tolerating Hardware Device Failures in Software*. In: *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA.

[38] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2009): *seL4: Formal Verification of an OS Kernel*. In: *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, pp. 207–220.

[39] Etienne Kneuss, Viktor Kuncak, Ivan Kuraj & Philippe Suter (2013): *On Integrating Deductive Synthesis and Verification Systems*. *CoRR* abs/1304.5661.

[40] Robert Könighofer, Georg Hofferek & Roderick Bloem (2009): *Debugging formal specifications using simple counterstrategies*. In: *FMCAD*, Austin, Texas, USA, pp. 152–159.

[41] Robert Könighofer, Georg Hofferek & Roderick Bloem (2013): *Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies*. *STTT* 15(5-6), pp. 563–583.

[42] Orna Kupferman & Moshe Vardi (2000): *Synthesis with Incomplete Information*. In: *Advances in Temporal Logic*, 16, Springer Netherlands, pp. 109–127.

[43] Ben Leslie, Peter Chubb, Nicholas FitzRoy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone & Gernot Heiser (2005): *User-level Device Drivers: Achieved Performance*. *Journal of Computer Science and Technology* 20(5), pp. 654–664.

[44] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess & Stefan Götz (2004): *Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines*. In: *6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, pp. 17–30.

[45] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet & Gilles Muller (2000): *Devil: An IDL for hardware programming*. In: *4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, USA, pp. 17–30.

[46] Andreas Morgenstern, Manuel Gesell & Klaus Schneider (2013): *Solving Games Using Incremental Induction*. In: *IFM*, Turku, Finland, pp. 177–191.

[47] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang & Sharad Malik (2001): *Chaff: engineering an efficient SAT solver*. In: *DAC*, Las Vegas, NV, USA, pp. 530–535.

[48] Nina Narodytska, Alexander Legg, Fahiem Bacchus, Leonid Ryzhyk & Adam Walker (2014): *Solving Games without Controllable Predecessor*. In: *CAV*, Vienna, Austria.

[49] Stefan Naujokat, Anna-Lena Lamprecht & Bernhard Steffen (2012): *Loose Programming with PROPHETS*. In: *FASE'12*, Tallinn, Estonia, pp. 94–98.

[50] Mattias O'Nils, Johnny Öberg & Axel Jantsch (1998): *Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces*. Washington, DC, USA.

[51] *16550 UART core*. `http://opencores.org/project,a_vhd_16550_uart`.

[52] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall & Gilles Muller (2011): *Faults in Linux: ten years later*. In: *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, pp. 305–318.

[53] Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2006): *Synthesis of Reactive(1) designs*. In: *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pp. 364–380.

[54] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, Austin, Texas, USA, pp. 179–190.

[55] Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen & Thomas A. Henzinger (2007): *Algorithms for Omega-Regular Games with Imperfect Information*. *Logical Methods in Computer Science* 3(3).

[56] Matthew J. Renzelmann & Michael M. Swift (2009): *Decaf: Moving Device Drivers to a Moderm Language*. In: *USENIX Annual Technical Conference*, San Diego, CA, USA.

[57] Richard Rudell (1993): *Dynamic variable ordering for ordered binary decision diagrams*. In: *ICCAD*, Santa Clara, CA, USA, pp. 42–47.

[58] Leonid Ryzhyk (2014): *TSL2 Reference Manual*.

[59] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur & Gernot Heiser (2009): *Automatic Device Driver Synthesis with Termite*. In: *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA.

[60] Leonid Ryzhyk, John Keys, Balachandra Mirla, Arun Raghunath, Mona Vij & Gernot Heiser (2011): *Improved Device Driver Reliability Through Hardware Verification Reuse*. In: *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA.

[61] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm & Mona Vij (2014): *User-Guided Device Driver Synthesis*. In: *OSDI*, Broomfield, CO, USA.

[62] Michael Sipser (1996): *Introduction to the Theory of Computation*, 1st edition. International Thomson Publishing.

[63] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík & Kemal Ebcioğlu (2005): *Programming by Sketching for Bit-streaming Programs*. In: *PLDI*, Chicago, Illinois, USA.

[64] Michael F. Spear, Tom Roeder, Orion Hodson, Galen C. Hunt & Steven Levi (2006): *Solving the starting problem: device drivers as self-describing artifacts*. In: *1st EuroSys Conference*, Leuven, Belgium, pp. 45–57.

[65] Michael M. Swift, Brian N. Bershad & Henry M. Levy (2003): *Improving the Reliability of Commodity Operating Systems*. In: *19th ACM Symposium on Operating Systems Principles*, Bolton Landing (Lake George), New York, USA.

[66] Synopsys: *Virtual Prototyping Models*. `http://www.synopsys.com/Systems/VirtualPrototyping/VPModels`.

[67] Wolfgang Thomas (1995): *On the Synthesis of Strategies in Infinite Games*. In: *12th Annual Symposium on Theoretical Aspects of Computer Science*, pp. 1–13.

[68] Vayavya Labs: *Device Driver Generator tool*. `http://vayavyalabs.com/products/`.

[69] Adam Walker & Leonid Ryzhyk (2014): *Predicate Abstraction for Reactive Synthesis*. In: *FMCAD*, Lausanne, Switzerland.

[70] Wind River (2010): *Wind River Simics Model Builder reference manual. Version 4.4*.

[71] Wind River (2010): *Wind River Simics Model Builder user guide. Version 4.4*.

[72] *WindRiver Simics DS12887 Model*. `http://www.windriver.com/products/simics`.

[73] Raj Yavatkar (2012): *Era of SoCs, presentation at the Intel Workshop on Device Driver Reliability, Modeling and Synthesis*.

[74] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula & Eric Brewer (2006): *SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques*. In: *7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, USA, pp. 45–60.