Device driver synthesis for embedded systems

Julien Tanguy, Jean-Luc Béchennec, Mikaël Briday, Sébastien Dubé, Olivier Henri Roux

Device driver synthesis for embedded systems*

Julien Tanguy[†,‡] Jean-Luc Béchennec[‡] Mikaël Briday[‡] Sébastien Dubé[†]
Olivier H. Roux[‡]

[†]See4sys,
Espace Performance La Fleuriaye, 44481 Carquefou CEDEX, France
[‡]LUNAM université, IRCCyN Lab, École Centrale de Nantes
1, rue de la Noë, 44 300 Nantes, France.

E-mail: {julien.tanguy, sebastien.dube}@see4sys.com
{jean-luc.bechennec, mikael.briday, olivier-h.roux}
@irccyn.ec-nantes.fr

Abstract

Currently the development of embedded software managing hardware devices that fulfills industrial constraints (safety, real time constraints) is a very complex task. To allow an increased reusability between projects, generic device drivers have been developed in order to be used in a wide range of applications. Usually the level of genericity of such drivers require a lot of configuration code, which is often generated. However, a generic driver requires a lot of configuration and need more computing power and more memory needs than a specific driver. This paper presents a more efficient methodology to solve this issue based on a formal modeling of the device and the application. Starting from this modeling, we use well-known game theory techniques to solve the driver model synthesis problem. The resulting model is then translated into the actual driver embedded code with respect to an implementation model.

By isolating the model of the device, we allow more reusability and interoperability between devices for a given application, while generating an application-specific driver.

1 Introduction

The development of device drivers in embedded systems is a critical and error-prone task. Because a device driver is the interface between the hardware device and the application or the operating system, the designers must have a knowledge of all those three components in order to develop efficient and safe drivers. The security aspect is emphasized by the execution context of most drivers: Being executed with supervisor privileges, any error in a driver may have a serious impact on the integrity of the entire system.

Another difficulty for designing device drivers is the device datasheet. Although it is designed to help a driver designer by explaining briefly how the device works, it does not document all possible behaviors. For example, a datasheet might specify that a device must be shut down in order to change some configuration registers, but it does not explain the outgoing of a configuration register change while the device is running.

To improve driver correctness and quality, a number of verification techniques [2, 7] has been developed. An alternative to verification is to improve the development process by synthesizing the driver from a formal specification. The verification method ensures that the driver behaves correctly and can check protocol violations between the application and the driver, while the synthesis approach ensures a correct-by-construction driver. However, for configurability and inter-operability reasons, such generated drivers, still conform to the traditional model of a driver consisting of multiple API endpoints, with minimal state.

Our research targets real-time embedded systems with hard timing constraints, mainly but not exclusively for automotive systems. These systems have usually high requirements in terms of functional safety, but on the other hand they have few resources in terms of computing power and memory storage.

An example of such constraints is the reaction time of an airbag controller, which has to be around a few microseconds.

Given these constraints, the automotive industry have developed AUTOSAR, a configurable architecture [4]. It defines a basic software architecture, consisting of several generic modules which implements all possible features.

These modules are usually defined as i) a core of basic functionalities which can do everything, ii) some configuration code which selects or refines the previously defined

*This work was partially funded by the ANR national research program ImpRo (ANR-2010-BLAN-0317).

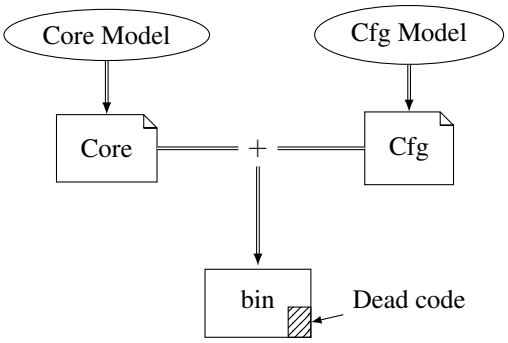

Figure 1. Current development methodology

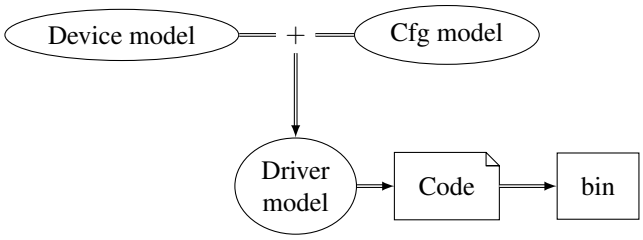

Figure 2. Proposed development methodology

behaviors and wrapper code to encapsulate the module functionalities in APIs — see figure 1. The configuration code is usually generated at compile-time and compiled along the core code, but the specification allows a post-compilation configuration which is passed to the core code by pointers.

This high level of configurability at every level increases greatly the complexity of such systems; they usually require multiple modules and abstraction levels. It can also result in a lot of dead code and if the configuration is not perfectly tuned to the application demands unnecessary behaviors make it into the code and may be executed. This comes at the cost of decreased performance and greater memory footprint, in terms of stack size, ROM and RAM usage. The consistency of the configuration must also be checked in order to be sure that the driver cannot behave in an unspecified way.

Even with consistency checks, there is no certain method to ensure that all behaviors which make their way to the final binary will be used by the application.

However, the automotive standards are quickly evolving and safety constaints are becoming more and more strict.

Driven by the industrial need for more formalism and verification, we have developed a synthesis approach based on a formal modeling of the system. By using a formal model approach, we can use well-developed model-checking techniques to ensure safety constraints functionnaly on the model and on the generated code.

This methodology has the particularity of being a more application specific approach than existing conventional drivers, which reduces the number of abstraction layers between the application and the driver, and generates the sufficient and necessary behavior, producing a small code.

Related works Some work has already been done in driver synthesis.

The Devil language [5] is a Domain Specific Language (DSL) targeted at the description of basic communication protocols with a device. Devil comes with tools to check for consistency of such models. However, being a low-level Domain-Specific Language, it focuses on the interface between the device and the device driver. The latter remains developed in a classical way.

Wang and Malik[10] propose another model which allows to generate full drivers and to check some properties in the model. While the approach is interesting, it targets UNIX-like systems, respecting the traditional driver model for compatibility reasons.

The Termite tool [9] uses a generic approach to driver synthesis, by specifying a driver in three different specifications:

- a device-class specification, which defines the messages used internally for a class of device drivers;
- a device specification, which defines the access protocol with the device;
- an OS specification, defining the communication protocols between the driver and the Operating System.

These three separate specifications allows reusability and exchangeability of devices and operating systems; their specifications depending only on device-class one. However, Termite-generated drivers work only in the context of a special framework, which simplifies the internal structure of the driver. For example, all events going in and out of the driver (API calls, hardware interrupts, etc.) are serialized and dealt with several handlers sequentially. This serialization behaves nicely in the context of UNIX drivers for desktop use, because these systems have enough computing power to handle all events in a reasonable time, but in the context of embedded systems, which have a *very* limited computing power and memory, the additional memory and computing cost cannot be afforded.

Our contribution We propose a new approach to device driver synthesis using an untimed reachability game on a formal model of the device, controlled by the application. Such information is often unavailable until runtime, but in the case of critical embedded systems it is known at compile-time.

By introducing more information from the application, it is possible to reduce the complexity of the exposed API, thus reducing the number of errors that can be made. For instance, instead of having to initialize an analog to digital

converter, setting up the conversions settings and starting the conversion — which is a typical usage of a conventional API — it is better to have a reduced semantic API for sampling a speed value, or sending a temperature message through the network. In this context, the driver would perform such initializations and configuration automatically, depending on the current objective.

This allows to generate more application-specific drivers, and limits the need for abstraction layers, since the driver API is exposed directly to the application.

Outline of the paper This paper is organized as follows: first we present the underlying modeling which supports the methodology presented in section 3. The methodology is explained on a simple example in section 4. At last some concluding remarks and considerations about future work are presented in section 5.

2 Definitions

This methodology relies on a model derived from Labeled Transitions Systems (LTS), in which transitions can have guards. In order to define this model formally, let us define some common terms beforehand.

Let $\mathbb{N}$ be the set of natural numbers. For a finite set E, we denote by 2^E the set of all its subsets. Let γ_P be a propositional logic over the predicates $p \in P$, e.g. of the form

$$\varphi := p | \neg\varphi | \varphi \wedge \varphi, \text{where } p \in P$$

For $A \subseteq P$, we define the semantics of such propositional logic:

- $A \vDash p$ iff $p \in A$;
- $A \vDash \neg\varphi$ iff $A \nvDash \varphi$
- $A \vDash \varphi \wedge \psi$ iff $A \vDash \varphi$ and $A \vDash \psi$

For $g, g' \in \gamma_P$ we say that g and g' overlap if

$$\exists A \subseteq P, \text{ such that } A \vDash g \text{ and } A \vDash g'$$

Definition 1 (Guarded labeled transition system) *A guarded labeled transition system (GLTS) is the tuple*

$$(Q, Q_0, A, P, E, l), \textit{where}$$

- *Q is a set of states;*
- *Q_0 is a set of initial states;*
- *A is a set of actions;*
- *P is a set of atomic properties;*
- *$E \subseteq Q \times \gamma_P \times A \times Q$ is the set of edges between the states;*
- *$l \subseteq Q \times 2^P$ is a labeling function.*

Deriving the definition for standard Labeled Transition Systems, we say that a GLTS (Q, Q_0, A, P, E, l) is deterministic if:

- $|Q_0| = 1$. We denote it as q_0.
- if (q, a, g', q') and $(q, a, g'', q'') \in E$, then g', g'' do not overlap if $q' = q''$.

In the sequel we will only consider deterministic GLTS.

We also define an asynchronous product operation on networks of GLTS. For the following definition, we consider n GLTS $\mathcal{S}_i = \left(Q^i, q_0^i, A_i, P_i, E_i, l_i\right)$, $i \in [\![0, n]\!]$, where $\forall i, j \in [\![0, n]\!], i \neq j \implies A_i \cap A_j = \varnothing$. We denote $A_i^\bullet = A_i \cup \{\bullet\}$, where $\bullet \notin A_i$.

Definition 2 (Asynchronous product of GLTS) *The asynchronous product $\mathcal{S} = \mathcal{S}_0 \times \cdots \times \mathcal{S}_n$ of the n GLTS is the GLTS (Q, q_0, A, P, E, l) where:*

- $Q = Q^0 \times \cdots \times Q^n$,
- $q_0 = (q_0^0, \ldots, q_0^n)$,
- $A = A_0 \cup \cdots \cup A_n$
- $P = P_0 \cup \cdots \cup P_n$
- $((q_0, \ldots, q_n), a, g, (q_0', \ldots, q_n')) \in E$ *such that*

$$\forall i \in [\![0, n]\!], \begin{cases} g = g_i \textit{ and } (q_i, a, g, q_i') \in E \textit{ if } A \in A_i \\ q_i = q_i' \textit{ otherwise.} \end{cases}$$

- *for* $q = (q_0, \ldots, q_n) \in Q$, $l(q) = l_0(q_0) \cup \cdots \cup l_n(q_n)$

Definition 3 (Semantics of a GLTS) *The behavioral semantics of a GTLS (Q, q_0, A, P, E, l) is the LTS $(Q, q_0, A, \rightarrow)$, where $\forall (q, a, g, q') \in E, (q, a, q') \in \rightarrow \iff l(q) \vDash g$.*

3 Methodology

The synthesized driver is derived from two separate models: one of the device, which models the internal behavior of the device, and one of the application settings, which models how the device will be used by the application.

In order to synthesize such drivers, we propose the following workflow:

1. Model the hardware device, with some synchronization primitives. This modeling does not require any knowledge about the application, thus can be done once for a particular device.
2. Model the application configurations, or modes of operations the application needs the device to be in.
3. Define driver objectives;
4. Generate the configured device model and compute strategies;
5. Translate abstract actions into actual code.

The application settings model is the representation of how the device is used by the application. In this model, several functional modes are defined, each mode representing a set of values of the configuration registers.

3.1 Modeling the components

Modeling the device The first step — the device model — models only the device behavior at register level: writing to control and configuration registers, reading from data and status registers, and sending interrupt to the driver.

It is the only reusable model between different applications, and can be part of some sort of model database. It is based only on the device datasheet. As part of the synthesis methodology, a device modeling methodology is proposed.

First, the set of all register fields is partitioned into three sets, depending on the effects a register read/write has on the device:

- the Control Fields. Writing in a control field has an immediate effect on the device's behavior.
- the Configuration Fields. Writing to a configuration field has no immediate effect on the device, but alters future behavior of the device. For example, an input channel selection field, or a device mode fields are considered part of the configuration space.
- the Data Fields, on which reading or writing to has no effect.

For each of the control actions, one or more abstract actions is added to the alphabet A of the model. For example, from a `Power Down` boolean register field, two actions can be defined: *PowerUp* and *PowerDown*. These are called the controllable actions A^C. The uncontrollable actions A^U of the device are also modeled, such as some internal action or hardware interrupts.

For the configurations, a set of atomic properties P is defined such that each atomic property corresponds to a valuation of a register field. The properties are used as guards in the device model, but are attached to states of the application settings model.

Sometimes, the datasheet imposes constraints on the changing of certain register fields in certain states, or it simply does not make sense to allow the modification of some registers while the device is busy. These restrictions are modeled by adding new properties to the states in which changing a register field is allowed, and some application settings model generation rules.

With all these guidelines, it is possible to produce a device model which corresponds to the behavior described in the device datasheet. The use of additional properties is allowed, to mark particular states of the device.

In a nutshell, the device model exposes to the application designer:

- a set of configuration properties P_{cfg}. These properties can be further grouped into sets of semantically related properties. For instance, a 1-bit interrupt mask can be split into two properties *interrupt* and *polling*.
- a set of synchronization rules, in the form of (P^{sync}, g), where $P^{sync} \subset P$ and $g \in \gamma_P$: if one of the properties in P_{sync} is used, then the corresponding GLTS must add g as a guard for every of its transitions. For instance, one might define a rule $(\textit{interruptSync}, \{\textit{interrupt}, \textit{polling}\})$.
- a set of additional informative properties P_{info} about the state of the device, such as *PowerDown*, *Idle*, *Busy*, *Waiting*, etc.

Modeling the application settings Once the model of the device is defined, the application designer has to define how it will be used by the application. The application settings are modeled by a global mode which is split into several independent sub-modes. These sub-modes can represent runtime behavior — e.g. *Low-Power*, *Sleep* — or statically defined properties — e.g. Channel groups in Analog-Digital Conversion, Types of frames in CAN/LIN/SPI communication, etc.

Formally, the global mode $\mathcal{M}$ divided into several sub-modes $\mathcal{M} = (m_1, \ldots, m_n)$. Each of these sub-modes have a set of possible values: $m_i \in m_i^1, \ldots, m_i^{p_i}$. Each value m_i^j of a sub-mode is mapped to a set of atomic properties among those exposed by the device model, representing the required configuration of the device in that sub-mode.

Even though it is possible to split valuations of a register field into several properties (for example, a 1-bit interrupt mask can be split into two properties *interrupt* and *polling*), there is an implicit restriction that only one of these properties can tag a sub-mode. Adding both properties to a sub-mode would render the sub-mode inaccessible, because of the way deterministic GLTS are defined.

These sub-modes are independent in the sense that they have *no* influence on each other, but they are linked by the synchronization constraints of the device.

Once defined, each sub-mode m_i is transformed into a GLTS $(Q^{m_i}, q_0^{m_i}, A_{m_i}, P_{m_i}, E_{m_i}, l_{m_i})$ by the following method:

1. each valuation m_i^j of the sub-mode is mapped to a state Q^{m_i} of the GLTS;
2. all properties tagging any sub-mode tags the corresponding state;
3. a default reset state $q_0^{m_i}$ with no properties attached to it is added;
4. the alphabet of actions A_{m_i} is derived from the state names, e.g. $toLowPower$, $toReset$, etc.
5. the transitions from and to every state are generated with respect to the synchronization rules by adding a conjunction of all the required guards to every transition, i.e. for all synchronization rules (P_k^{sync}, g_k) and all transitions $(q_1, a, g, q_2) \in E_{m_i}$,

$$l_{m_i}(q_1) \cup P_k^{sync} \implies \exists g' \text{ such as } g = g' \wedge g_k.$$

The initial reset state assumes that the state of the device is not known when the driver (re)starts. The fact that it does not hold any property ensures that one of the first

actions taken by the driver is to configure the device into a defined mode before doing any work.

The global mode GLTS $\mathcal{M}$ is obtained by computing the asynchronous product of all sub-modes.

3.2 Driver generation

Once the model of the device and its configuration are defined, well developed control and game theory techniques [3, 8, 6] are used in order to generate the driver model. Although the problem defined by the GLTS model could be reduced to a *shortest path problem* in a graph, this methodology uses a more generic, model-agnostic approach which can be easily extended to timed models by simply changing the underlying modeling and game rules. But first let us define the outline of a driver.

Anatomy of the driver In this model, a driver consists of an arena $\mathcal{G}_a$ a set of objectives $\mathcal{O}$. An objective represents a set of atomic properties which the configured device is to satisfy, for instance the *power down* or *idle* state, a *busy* state while converting a certain analog input, or the end of the sending of a given frame over the network.

For each objective, the driver has a strategy, *i.e.* a sequence of actions to take in order to get from the current state to an objective state. For this model it is sufficient to consider only *memoryless* strategies, *i.e.* strategies in which the actions to take are dictated only by the current state, and not the sequence of states which led to the current one. These strategies are computed with respect to the model of the configured system which represents the possible behaviors of the device and any mode change in the application settings.

At any point in time, the driver has only one active objective, and is taking actions to fulfill this objective. When it is reached, the driver does not take any action until the objective is changed.

More formally, given a model $\mathcal{D}$ of the device, we want to generate a controller $\mathcal{C}$ — the driver — such that the system $\mathcal{D}|\mathcal{C}$ composed of the device controlled by the driver satisfies a given property φ, expressed by the LTL property for all paths:

$$\varphi = \diamond A, \text{with } A \subseteq P.$$

Problem 4 (Control problem) *Given $\mathcal{D}$ and φ, is there any driver (or controller) $\mathcal{C}$ such that $\mathcal{D}|\mathcal{C} \vDash \varphi$?*

Generating the game arena and the game The model of all possible behaviors of the configured device — including changing sub modes — is called the *arena* $\mathcal{G}_a$. It is obtained from the semantics of the asynchronous product Π_{async} of the device model $\mathcal{D}$ and the driver modes $\mathcal{M}$. The product Π_{async} is computed as described in definition 2.

Taking the semantics of this product, we obtain the following LTS:

$$(Q^a, q_0^a, A_a, \rightarrow_a).$$

The game arena is derived from this LTS by partitionning the alphabet A_a of actions in A_a^C and A_a^U. The alphabet of controllable actions groups the controllable actions of the device and all the sub-mode change actions: $A_a^C = A_{\mathcal{D}}^C \cup A_{\mathcal{M}}$. The alpbabet of uncontrollable actions is the uncontrollable actions of the device only: $A_a^U = A_{\mathcal{D}}^U$.

Assuming the initial device model is correctly defined, taking the semantics of the product ensures that any non-specified behavior is not accessible.

The problem reduces to an untimed two-player safety game between the driver, performing controllable actions of the device and all sub modes switches, and the device performing its uncontrollable actions. There are several algorithms to compute a strategy which resolves this game.

One of the most used is the algorithm defined in [6], with the controllable predecessor method.

Intuitively, this method computes iteratively the set of states for which a strategy exists — these are called *winning* states — starting from the set of *goal* states. At each iteration, the algorithm adds to the set of *winning* states all its controllable predecessors.

A controllable predecessor of a set $\mathcal{S}$ of states is a state for which there exists at least a controllable action $a^c \in A^C$ to $\mathcal{S}$ and all uncontrollable actions $a^u \in A^U$ also lead to $\mathcal{S}$.

More formally, the controllable predecessor set $\pi(\mathcal{S})$ of $\mathcal{S} \subseteq Q$ is defined as follows:

$$\forall q \in Q \setminus \mathcal{S}, q \in \pi(\mathcal{S}) \text{ if and only if } \begin{cases} \exists q' \in \mathcal{S}, a \in A^C \text{ s.t. } (q, a, q') \in \rightarrow \\ \forall q'' \in Q, \forall b \in A^U \text{ s.t. } (q, b, q'') \in \rightarrow, q'' \in \mathcal{S}. \end{cases} \quad (1)$$

When computing the controllable predecessors, the algorithm deduces a strategy to execute in order to reach the *goal* states.

This algorithm ends when it has reached a fixpoint, *i.e.* when it cannot add any new state to the *winning* states. The remaining states which could not be added are the *loosing* states. In these state there is no action to take in order to go to a *winning* state, whatever the device does.

More formally, the algorithm is as follows:

$\text{Win}_0 \leftarrow \text{Goal}$ $i \leftarrow 1$
repeat
  $\text{Win}_i = \text{Win}_{i-1} \cup \pi(\text{Win}_{i-1})$
  $i \leftarrow i + 1$
until $Win_i = Win_{i+1}$;

Algorithm 1 (Computing the winning states)

Future work will lift part of the constraints, allowing the driver to wait for an uncontrollable action *because* it will happen eventually, whereas the current hypothesis allows the device to withhold the interrupt and lock the driver indefinitely.

From the computation it is possible to derive a *memoryless* strategy for each objective: each state is either a *goal* state — the driver has nothing to do —, a *losing* state — the driver cannot do anything and may fail into some error recovery mode — or the driver has a controllable action to take in order to reach one of the *goal* states.

4 Example

In this section, we will apply the methodology to a simple example. Let us consider a simple and generic Analog to Digital Converter. This device is part of almost all micro-controllers, and its role is to sample analog signals and convert them into digital values. Usually, a single ADC has several input channels. It can sample and convert its inputs either sequentially or in parallel.

The example ADC has the following features:

- two different clock modes, one half speed and one full-speed;
- a power-down mode, only in which the clock configuration can be changed;
- multiple input channels, converted sequentially in a conversion chain;
- the conversion of each of the channels can be enabled or disabled, while the device is idle or shutdown;
- two conversion modes: *oneshot*, in which only one conversion chain is performed, and *continuous*, in which conversion chains are performed indefinitely until the user stops the conversion (the last chain still ends the normal way)
- the device triggers an *End Of Conversion* (eoc) interrupt at the end of each channel conversion, and an *End of Chain* (ech) interrupt at the end of a chain.

Modeling the device For this high-level model, the granularity is set at chain conversion level, so all the single channel conversions are abstracted.

From the specification, the following alphabet of actions is derived: *abort*, *ech*, *sleep*, *start*, *stop* and *wakeup*.

From the register description, the following properties are defined:

- Clock configuration: *clkFull* and *clkHalf*, for the two values of the speed, and *clkCfg* for the synchronization constraints.
- Conversion configuration: *Os*, and *Cont*, for the oneshot/continuous setting, and *convCfg* for the synchronization constraints.
- Informative properties: *Idle*, *poweroff* and *busy*.

The device model is straightforward, as presented in figure 3.

Modeling the configuration Once the driver model is defined, we can define the application configuration, or the driver modes. For this example, the application usage is as follows:

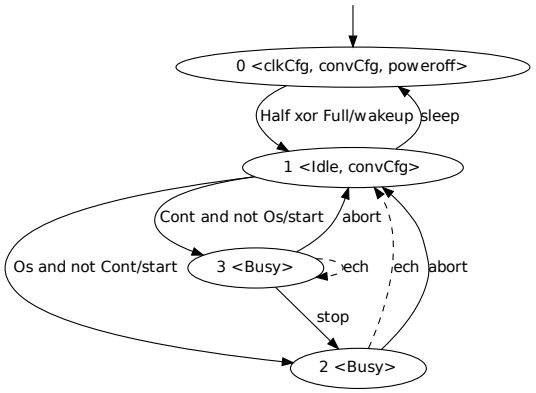

Figure 3. Model of the device. Controllable actions are represented with solid lines, and the uncontrollable actions are represented in dashed lines.

- The driver shall perform conversions fast, so only the *clkFull* setting will be used.
- The driver will convert two groups of signals: one is to be monitored continuously, with the *Cont* setting, while the other corresponds to on demand conversions, using the *Os* setting.

These modes are then translated into GLTS, following the method defined in 3. For the example, the general driver modes is divided into two sub-modes: the conversion sub-mode and the clock sub-mode. Here only the conversion sub-mode is detailed.

First three states are defined: *reset*, *G1* and *G2*. *G1* is tagged with *clkfull* and *Os*, while *G2* is tagged with *clkfull* and *Cont*. Since these states involve the conversion properties, all the transitions must have *convSync* in their guard.

The resulting automaton is presented figure 4. Note that all these transitions will be controllable for the driver, since they represent changes of its internal mode.

Defining driver objectives For this example, the application needs to perform two types of conversions: one for the group G1 and one for the group G2. We will also consider a low-power mode of the driver, where the device is switched off. These two conversion groups and the low-power are then translated into three driver objectives:

1. Go to a state labeled by *poweroff*
2. Go to a state labeled by *G1* and *busy*
3. Go to a state labeled by *G2* and *busy*

Generating the arena and computing strategies Once all the components of the configured system are defined in terms of GLTS, the arena of the game is generated. First,

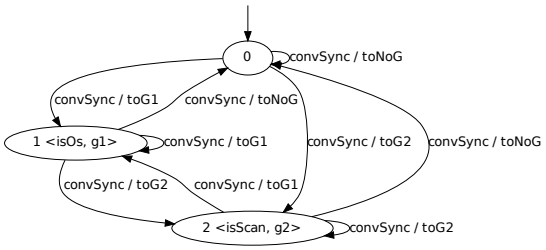

Figure 4. Conversion sub-mode GLTS. This GLTS is generated with 3 modes: One Shot mode (1), Continuous mode (2) and the default reset mode.

all the models are composed into an asynchronous product.

The semantics of the product is shown figure 5. In any of the states the driver can take the controllable actions, represented with solid lines, and the uncontrollable actions are represented in dashed lines.

This product model is then processed with the driver objectives in order to generate adequate strategies. The computed strategies for all the objectives are presented in table 1.

These strategies are similar, except for one *loosing* state for the first two objectives. This is due to the untimed nature and the worst-case hypothesis of the game. In this context, a strategy wins if the driver can force a behavior whatever the device does. But here the untimed strategy does not work because the *zeno* behavior where the device does the *ech* action infinitely often prevents the driver to act. This behavior is obviously unrealistic: in reality, the *ech* interrupt has a minimum period, so the driver has time to cancel an ongoing conversion, or the related interrupt can be masked.

Future improvements of this work will consider timed models of the device, which are more complex to analyze and to compute strategies for.

5 Conclusion

We have developed a generic methodology and supporting models for device driver synthesis. It is designed specifically for embedded real-time systems, with low complexity and small memory footprint, and can be adapted to more complex models.

It relies on a particularity of such systems, which are to be completely defined at compile-time. It is possible to reduce the amount of generated code by performing optimisations at the model level — cutting unreachable states when applying the possible strategies — thus producing only necessary and sufficient code.

State	Objective 1	Objective 2	Objective 3
0	Win	Take *toClkFull*	Take *toClkFull*
1	Win	Take *wakeup*	Take *wakeup*
2	Take *sleep*	Take *toG1*	Take *toG2*
3	Win	Take *toClkFull*	Take *toClkFull*
4	Win	Take *wakeup*	Take *wakeup*
5	Take *sleep*	Take *start*	Take *toG2*
6	Take *abort*	Win	Take *abort*
7	Win	Take *toClkFull*	Take *toClkFull*
8	Win	Take *wakeup*	Take *wakeup*
9	Take *sleep*	Take *toG2*	Take *start*
10	Take *abort*	Take *abort*	Win
11	Loose	Loose	Win

Table 1. Computed strategies on the arena

Future development Future developments of this methodology will include more complex elements into the model. We will include the management of shared data variables and buffers, allowing the definition of safety objectives such as "This buffer shall not overflow".

In order to manipulate finer models, we need to add time to these models, like in [1]. Along with the notion of time, it is possible to express the notion of urgency, adding a whole range of available behaviors to the driver.

At last we will build a complete prototype being able to generate a device driver code from a model. Performance will be evaluated on an actual platform and compared to device drivers developed in a traditional way.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

[3] A. Church. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, pages 23–35, 1962.

[4] F. Kirschke-Biller. Autosar – A worldwide standard current developments, roll-out and outlook. www.autosar.org, 2011.

[5] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An idl for hardware programming. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, volume 4, pages 2–2. USENIX Association, 2000.

[6] A. Pnueli, E. Asarin, O. Maler, and J. Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control. Elsevier*. Citeseer, 1998.

[7] H. Post and W. Küchlin. Integrated static analysis for linux device driver verification. In *Integrated Formal Methods*, pages 518–537. Springer, 2007.

[8] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

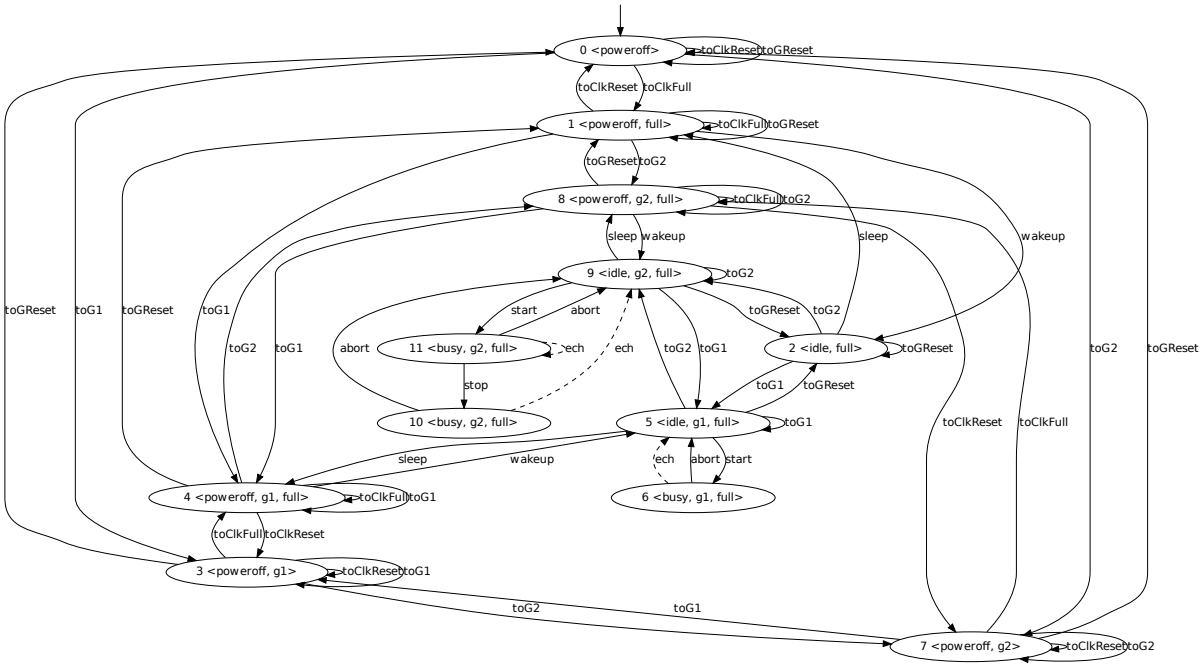

Figure 5. Generated model of the arena

[9] L. Ryzhyk. *On the Construction of Reliable Device Drivers*. PhD thesis, University of New South Wales, 2009.

[10] S. Wang, S. Malik, and R. A. Bergamaschi. Modeling and integration of peripheral devices in embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe*, volume 1, pages 136–141. IEEE Computer Society, 2003.