PRES LUNAM
Ecole Doctorale STIM
Sciences et Technologies de l'Information et Mathématiques

Spécialité : Informatique
Laboratoire : IRCCyN
Equipe : Systèmes Temps Réel

Device driver synthesis for embedded systems

Tanguy Julien
Mél : julien.tanguy@see4sys.com
Directeur de thèse : Olivier H. Roux

Abstract: Currently, the development of embedded software managing hardware devices that fullfills industrial constraints (safety, real time constraints) is a very complex and error prone task. The automotive industry has tried to solve this issue by defining the AUTOSAR standard for low level drivers & Basic software by defining generic drivers that implements all possible features. However, the level of genericity of such drivers require a lot of configuration code, which is often generated.

The main drawback of this approach is the need of device ressources and configuration complexity. This paper presents a more efficient methodology to solve this issue.

Keywords: *finite automata, control, game, synthesis, embedded systems, device drivers*
Collaborations : See4sys

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 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 when designing device drivers is the device datasheet. Although it is designed to help a driver designer by explaining breifly how the device works, it does not document all possible behaviours. To improve driver correctness and quality, a number of verification techniques [1, 2] have been developed. An alternative to verification is to improve the development process by synthesizing the driver from a formal specification.

This research targets real-time embedded systems with hard timing constraints. 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.

Given these constraints, the embedded systems community have developed configurable architectures — for example, AUTOSAR[3] in automotive industry. These architectures consist of a set of modules having a specific behavior at a given abstraction level. However, these modules usually have a high level of configurability, which greatly increases greatly the complexity of such systems. The configuration of an entire software platform (operating system, memory stack, communication stack, etc.) is an impossible task without the associated configuration tool.

This paper proposes a safer, more application specific approach which reduces the number of abstraction layers between the application and the driver, and generates the sufficient and necessary behaviour, producing a minimal code.

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

In [4], Wang and Malik 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 [5] uses a generic approach to driver synthesis, by specifying a driver in three different specifications. However, Termite-generated drivers work only in the context of a special framework, which simplifies the internal structure of the driver. Although this framework is very convenient for driver generation in the context of desktop computers, the amount of additional computation and memory required by the framework is too much for drivers with high demand.

Our contribution We propose a new approach to device driver synthesis, using an untimed rechability 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. 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.

2 Definitions

Our methodology relies on a model derived from Labeled Transitions Systems, 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 2.1 (Guarded labeled transition system)**.** A guarded labeled transition system (GLTS) is the tuple

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

- 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.

Definition 2.2 (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 behaviour 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;
2. Model the application configurations 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 on 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 behaviour at register level: writing to control and configuration registers, reading from data and status registers, and receiving interrupts from the hardware.

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.

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.
- a set of synchronisation rules, in the form of (P^{sync}, g), where $P^{sync} \subset P$ and $g \in \gamma_P$
- 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 independant 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. It is possible to split valuations of a register field into several properties.

These sub-modes are independant 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 is transformed into an GLTS.

3.2 Driver generation

Once the model of the device and its configuration are defined, well developped control and game theory techniques [6, 7] 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 $\mathcal{D}$ consists of a set of objectives $\mathcal{O}$. An objective represent 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 behavior 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 fullfil this objective. When it is reached, the driver does not take any action until the objective is changed.

Generating the game arena and the game The model of all possible behaviours of the configured device — including changing sub modes — is called the *arena*. It is obtained from the semantics of the asynchronous product of the device model and all modes.

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 [7], 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, i.e. all states from which it is possible to reach a winning state whatever the device does.

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.

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 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 memory footprint, and can be adapted to more complex models.

It relies on a particularity of such systems, which is to be completely defined at compile-time. It is possible to reduce the amount of generated code to a minimum, producing only necessary code.

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 [8]. 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.

References

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

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

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

[4] Shaojie Wang, Sharad Malik, and Reinaldo 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.

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

[6] Peter JG Ramadge and W Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

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

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