Formal Aspects of Component-Based Design of Embedded Real-Time Systems

Andrey Kruglyak
Dept. of Computer Science, Electrical and Space Engineering
Luleå University of Technology, 971 87 Luleå, Sweden
E-mail: kruglyak@mac.com

Motivation and Previous Work

An embedded system is traditionally defined as a computational system that is a part of a larger device and serves a specific function. Such systems are often safety-critical, and a vast majority of them include at least some hard real-time tasks, where correctness depends on timeliness of computations.

In recent years, there has been a surge in complexity of embedded systems, often further complicated by resource sharing between independent tasks; there is also a growing demand for concurrency in system operation. The challenge is to achieve that without compromising correctness and reliability.

Component-based design (CBD), successfully used in development of general-purpose software, seems to be a natural choice. However, despite a considerable body of research (see, for example, work on Rubus Component Model [2] and Ptolemy II [1]) and a number of commercial tools available in the market, CBD is rarely used for development of embedded systems outside some specific domains (e.g., Koala [3] has been used by Philips in consumer electronics). This calls for a new, formal approach to CBD addressing the important characteristics of embedded systems. Such an approach should:

- allow for an accurate modeling of interaction between software and hardware, which is key to defining functionality of most embedded systems and is the usual source of hard real-time requirements,
- incorporate timing specifications, central to development of most embedded systems, directly into the model at all levels,
- provide an intuitive and “safe” mechanism for introducing concurrency into system operation,
- ensure a good correspondence between a component model and its implementation in a programming language, in order to facilitate component and system verification, and
- define a graphical representation of a component model of a system that can be used throughout the design process (in most tools in use today, a graphical model can no longer be used once program code generated from it is modified manually).

Our work on CBD of embedded real-time systems is a part of the ongoing development of the programming and modeling language Timber [4] at Luleå
University of Technology and Chalmers University of Technology, Sweden. Timber specifically targets hard real-time systems. It is an object-oriented language that combines purely functional evaluation of expressions with an imperative-style command layer, incorporating system I/O and message passing between objects. Importantly, it has been developed to meet some of the requirements on modeling embedded systems discussed above. Firstly, it adopts a reactive execution model, i.e. a reaction (execution of an object’s method) is triggered either by an external event or by a message from another object; this makes Timber particularly suitable for specifying interaction between software and hardware. Secondly, it combines object-level concurrency with a complete encapsulation of an object’s state, which results in a fairly simple implicit concurrency model. Thirdly, timing specification can be incorporated into Timber code by specifying a permissible execution window – a baseline and a deadline – for each method invocation; this specification is preserved in the compiled code and can be used to guide scheduling at run-time as well as to perform static schedulability analysis.

Research Goals

Our goal is to formulate a general framework for component-based design of embedded real-time systems and to include support for CBD directly in the Timber language. This involves a number of tasks:

- a formal component model should be defined using existing syntactic constructs or by introducing new ones to the Timber language, with component composability verified by static type-checking (Timber has an advanced type system with automatic type inference); the unambiguous semantics of such a model is guaranteed by the operational semantics of the language;
- a language for specifying timing behavior at system and component levels should be defined so that it is possible to verify (a) component composability with respect to timing requirements, and (b) that a particular implementation of a component (in terms of Timber code with permissible execution windows for method invocations) conforms to timing specification at the component level;
- a graphical representation of a component model should be defined, as well as its translation to Timber code and vice versa; the existence and quality of this representation is seen as key to usability of the proposed approach.

Some of our recent results obtained while working on the third task are presented below. These results are of a general nature and are not specific to Timber-based component models.

A Formal Approach to Defining a Graphical Representation

A formal component model is typically defined in a text-based language (in our case – in the programming language Timber), but its usefulness depends on the existence of a graphical representation. True usability, however, comes with the
ability to perform graphical editing operations at any stage of the development process, i.e. even after program code has been added and/or edited manually (which is not supported by the majority of existing design tools). The challenge here is two-fold: firstly, information that should be visualized in a graphical model is often implicit in program code (e.g., a relation between two components can hinge on the value of some variable), and secondly, a graphical model should be simple and hence the translation from program code is inevitably lossy and in many cases irreversible.

To meet this challenge, we propose to introduce an intermediate representation $G$ and, given a set of abstract syntax trees of syntactically correct programs $P$ and a set of models $\hat{G}$, expressed as labelled graphs (such as formal component models or any other models with unambiguous graphical representations), we define:

- a translation $\tau : P \rightarrow G$ that augments an abstract syntax tree of a program with an explicit representation of information needed for the graphical model (e.g., links between a component definition and its instantiations); we should require that for every syntactically correct program there exists exactly one intermediate representation;
- an inverse translation $\tau^{-1} : G \rightarrow P$ that removes the vertices and edges added by $\tau$ (these may be labelled differently from labels on vertices and edges of abstract syntax trees to make $\tau^{-1}$ trivial), such that
  \[(\forall p \in P)[\tau^{-1}(\tau(p)) = p]\]
- an “erasure” $\delta : G \rightarrow \hat{G}$ that hides some of the information available in $G$ (e.g., pure computations that do not affect the component structure) by “collapsing” certain vertices into edges and by completely removing some other vertices and edges;
- graphical editing operations $\gamma_1, \ldots, \gamma_n$, defined not on $\hat{G}$ but on the more complete intermediate representation $G$; for each operation it is necessary to prove that it preserves syntactical correctness of a program:
  \[(\forall p \in P)(\exists p' \in P)[\tau^{-1}(\gamma_i(\tau(p))) = p']\]

It can be shown that this approach ensures that a graphical representation can indeed be used at any stage of the development process, even after program code has been edited manually.

References