Abstract

In this demonstration session, we present a toolkit we have designed around a model-driven language (LE). This relies on formal methods to ease the development of applications in an efficient and reusable way. Formal methods have been advocated as a means of increasing the reliability of systems, especially those which are safety or business critical. It is still difficult to develop automatic specification and verification tools due to limitations like state explosion, undecidability, etc. To face these problems, we provide LE with a constructive semantic that allows the modular compilation of programs into software and hardware targets (C code, VHDL code, FPGA synthesis, Verification tools). Moreover, we also provide software to design, compile and verify LE programs. Our approach is pertinent according to the two main requirements of critical realistic applications: the modular compilation allows us to deal with large systems, the model-driven approach provides us with formal validation.

keywords: model-driven software development, synchronous models, compilation, modularity, verification

1 Introduction

CLEM toolkit[2] has been designed around the Synchronous language [3] LE. It is a model-driven language to allow both efficiency and reusability of system design, and formal verification of system behavior. Its semantic relies on the synchronous hypothesis which assumes a discrete logic time scale, made of instants corresponding to reactions of the system. All the events concerned by a reaction are simultaneous: input events as well as the triggered output events. As a consequence, a reaction is instantaneous (we consider that a reaction takes no time, in compliance with synchronous language class), there are no concurrent partial reactions and so determinism can be ensured. Although synchronous languages have begun to face the state explosion problem, there is still a need for further research on efficient and modular compilation of synchronous languages. Indeed, only few approaches consider a modular compilation because there is a deep incompatibility between causality and modularity. Causality means that for each event generated in a reaction, there is a causal chain of events leading to this generation. No causal loop may occur. Program causality is a well-known problem with synchronous languages, and therefore, it needs to be checked carefully. Thus, relying on semantic to compile a language ensures a modular approach but requires to complete the compilation process with a global causality checking. The originality of our approach is the way we check causality from already checked sub programs and the modular approach we infer.

2 CLEM Modular Approach

The core of CLEM toolkit is the LE language and its modular compilation. LE has been proposed to design applications in different domains: first, a textual language closed to Esterel [6] allows the design of even-driven systems. Second, StateChart like description is a native construct and is well suited to the design of schedulers. Third, data flow applications can be described with the equation syntax the language offers (see [5]).

To reconcile modularity and causality, we first define a well founded “equational” semantic for LE programs. Such a semantic ensures that programs containing no cyclic instantaneous signal dependencies are translated into sorted equation systems. Equations propagate in-formations about signals status. The distinctive specificity of our approach is that the information propagated by equations about signals status does not lead to determine signals as absent or present but also ⊥ (undefined) or ⊤ (error). As a consequence, the
signal dependency propagation leads to build several valid sorting for an equation system. Thus, in our compilation process we record all these eligible partial orders; and composition of already sorted equation systems makes use this information to compute partial orders without sort again the overall system. In practice, we define a new sorting algorithm (based on the well-known PERT [7] technique) to compute all the eligible partial orders of an equation system. In complement, we also define a link algorithm that can compose two already sorted equation systems. As a consequence, we are modular in the sense that we can build a global sorted equation system from two already sorted ones. Thus, when we compile programs including run module instruction, we can offer a modular compilation means.

3 CLEM Toolkit Description

![Figure 1. Compilation Scheme](image)

The CLEM workflow is summed up in the figure 1. LE language help us design programs. Automata can be generated by our automaton editor galaxy. Each LEmodule is compiled in a LEC internal format file and can call instances of sub modules. These latter can have been already compiled in the past by a previous call to the clem compiler. The separated compilation implies to keep the ⊥ status of signals for which no information has been propagated during the compilation phase. Then when compilation is achieved, a finalization process sets all ⊥ signal status to absent before the generation of code for a targeted usage: simulation, hardware description or software code. CLEM toolkit comes with a simulation tool. An attractive consequence of our approach is the ability to perform validation of program behaviors. As a consequence, we can rely on model checking based tools to verify property of LE language. Thus, from lec files, we also generate smv files to feed the NuSMV model checker [1].

4 Related Works and Conclusion

The theoretical aspect of our approach benefit from a great many studies about synchronous language domain since two decades. Polynomial compilation was first achieved by a translation to equation systems that symbolically encode the automata. This approach is the core of commercial tool [6]. Then several approaches translate the program into event graphs or concurrent data flow graphs (see [4] among others) to generate efficient C code. All these methods have been used to optimize the compilation times as well as the size and the execution of the generated code. However, none of these approaches consider a separated compilation process with a dynamic loading of already compiled sub programs at “compilation “ level.

In this work, we introduced the CLEM toolkit that provides us with reusability and verification in the design of control dominated applications. We rely on a constructive equational semantic and on a new sorting algorithm to offer a separated compilation means. The major improvement we aim at is the introduction of data (like sensor handling facilities) first both in the language and in the generated back ends (included also verification).

References