Tool-based Development of Light-weight Fault-tolerant Embedded Systems

Chih-Hong Cheng*, Christian Buckl†, Alois Knoll*
*Unit 6: Robotics and Embedded Systems, Department of Informatics
Technischen Universität München, D-85748 Garching, Germany
†fortiss GmbH, Guerickestr. 25, D-80805 München, Germany
Email: chengch,knoll@in.tum.de, buckl@fortiss.org

***Abstract*—In this report, we present Gecko, a framework for the model-based development of embedded systems, focusing on light-weight fault-tolerance and dependability aspects. Given a high-level sketch of the system, including models of the hardware, software, expected faults, and a set of predefined mechanisms, Gecko can perform a series of refinement processes, including model annotation and concretization over specified mechanisms, to generate executable code over dedicated platforms.**

To supplement the design flow, several analysis techniques are embedded or experimented in Gecko as modules, including (a) generation of dependability models acceptable by verification engines, (b) worst-case transmission time calculation for networks demonstrated on the example of CAN bus, and (c) generation of integrity constraints using program analysis to estimate the effect of sensor imprecision. Gecko also interfaces standardized ESL design methodology by generating SystemC fragments (currently loosely timed TLM 2.0 models) to support detailed analysis.

I. Introduction

In this report, we present Gecko, a framework for model-driven development of embedded (distributed) systems, focusing on fault-tolerance and dependability aspects. The goal of Gecko is to introduce fault-tolerance aspects into existing design flows of embedded systems with minimal overhead.

The predecessor of Gecko is called FTOS, which was developed as a text-based modeling tool for the support of designing fault-tolerant systems. Technical details about FTOS can be found in [8]. Since then, we have extended and reimplemented the tool with the following goals:

- Closing the visibility gap between models and implementations. As FTOS works over dedicated models of computation, a single-step transformation from model sketches to executable code is non-intuitive. By introducing intermediate models (functional and architectural) and several analysis techniques, we acquire stronger guarantees over the resulting artifact[1]. Furthermore, it enables a generalized usage for any application specific models to be introduced in the Gecko development flow.
- Giving more control to the user with ease of use. The tool should allow users to inspect intermediate results and to perform modifications on-the-fly. The latter case can be useful, in case an automatic global optimization of system aspects would require a search in a huge state space, while developers can find a nearly optimal solution based on their experience.

The resulting tool has been developed as a plug-in under the Eclipse Modeling Framework (EMF) [1], [3], which is now called Gecko. It automates the implementation of extra-functional aspects with a focus on fault-tolerance, integrates formal verification, but also enables developers to manually optimize the system.

II. Architecture of Gecko

We introduce the design flow and components currently included in Gecko using figure 1.

A. Functional Concretization

In the first step, a given *application-specific model* in high-level descriptions is concretized into an *abstract actor model* that is based on concepts similar to actor-oriented languages, and contains explicit timing information based on specifications included in the original models or information added by the user within this step. In fig. 1, the application-specific model consists of four partial models, namely HW, SW, Fault, FT (for fault tolerance). During the concretization, only SW and FT are involved, where we mainly generate required actors, ports, and token transfers (similar to connectors in Ptolemy [10], differences mentioned later) concretizing user specified fault-tolerance mechanisms. Fig. 2 illustrates an example from model sketches (upper part) to functional models (lower part). For the timing specification, we use the information contained in the SW model, which is based on the concept of logical execution time [13], [14].

The *actor model concretizer* in Gecko translates fault-tolerance mechanisms into according actors represented by a tree structure, whose elements can reference other elements of the model. To generate executable code for actors in later stages, for each platform only a generalized tree interpreter is required. This greatly reduces the possibility of programming errors when introducing new platforms, as no parameterized skeletons for each actor and each platform are required[2].

[1]We are aware of theoretical frameworks of composition and refinement process of systems, whilst it is not our current focus.

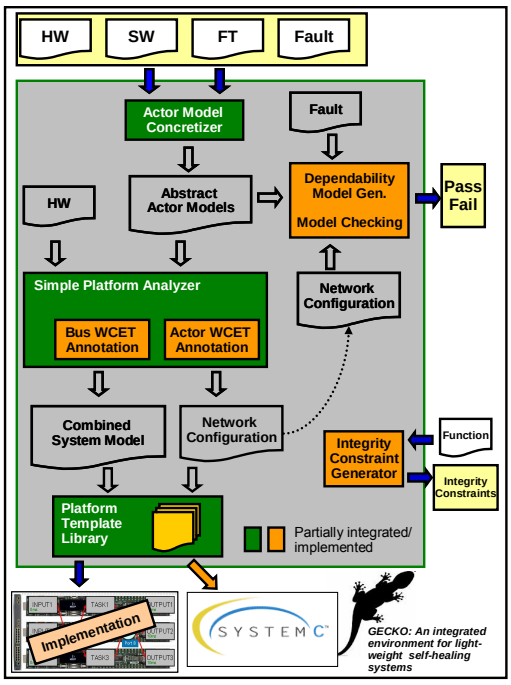

Fig. 1. An overview for the flow of model transformations, code generation, and analysis scheme in Gecko.

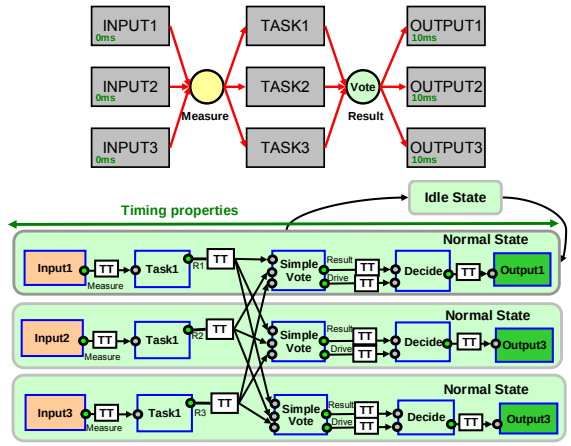

Fig. 2. An example for concretizing FTOS models to functional models with timing annotations (some details omitted). TT stands for token transfer, which will later be refined during function-architecture mappings.

B. Function-Architectural Mapping under Local Scale

After the first transformation, different application-specific models are mapped to unified abstract actor models, allowing us to follow the same transformation scheme to generate *computational (software) models*. These models concretize the implementation of the communication between software components and the timing without concrete parameters. These parameters are derived in the next phases, when the concrete hardware is considered. These steps are performed by the *simple platform analyzer* as depicted in fig. 1.

The above process can be challenging concerning design space exploration with multi-objective function optimization. Therefore, we constrain ourselves, and assume that a major part of the functionalities are mapped to architectures by user-guided information, as it is currently specified in the FTOS software and hardware model. Our goal is to perform a ***local-scale*** design space exploration concerning extra-functional properties, i.e., we consider how to integrate newly generated actors (for dependability, fault-tolerance) and newly introduced token transfers, such that the specified design constraints are still satisfied. Currently Gecko generates the mapping based on a small set of user-predefined parameters, and the analysis is done separately after the concretized model is generated. The resulting artifact is a combined model including network configurations (e.g., message groupings and priorities in the CAN bus).

The timing behavior in Gecko is based on (1) the required time for executing actor blocks (WCET) plus (2) the required time for token transfers (WCTT), which is grouped into messages in the architecture model. We follow two different approaches: a hard real-time approach based on temporarily disabling the system interrupts (not applicable for all applications) to assure the claimed WCET of actor blocks or a less stricter approach based on simulation to acquire confidence about the system behavior.

1) With respect to timing behavior of the network, currently only CAN buses are supported with analysis techniques. We implemented the *CAN bus WCTT analyzer* using the classical real-time scheduling theory [9], [15] based on resource contention. Nevertheless, the analysis technique does not scale to industrial examples, as in this idealized case, the CAN controller must have enough TxObjects to accommodate all the outgoing message streams. Our implementation template deploys the system based on this assumption, and raises alarms if it is violated. The mentioned problem is mainly an issue for models-of-computation with aperiodic behavior.
2) With respect to execution time of our automatically generated actors, to have stronger confidence for predictable timing, files containing the WCET of these actors (with parameterized form) can be read into Gecko as supplementary files.

C. Generation of Dependability Models

Regarding dependability, currently we are interested in whether a system with equipped fault-tolerant mechanisms is sufficient to resist faults defined in the fault-hypothesis. To achieve this goal, Gecko generates the dependability model from a combination of functional models, fault models, and network/hardware configuration files; the result is created in

[2]For instance, when concretizing actors in UPPAAL verification models, all &, <, > tokens are replaced by &, <, >. This can be easily achieved by a partial modification of the tree interpreter.

formats acceptable by the verification engine UPPAAL [7]. Fig. 1 indicates required components for the generation of the dependability model:

1) The *network and hardware configurations* generated on the architectural level provide a sufficient abstraction on how faults are actuated. For instance, when a fault with type `MessageLoss` is considered, token transfers grouped in one single message can be lost simultaneously, which should be faithfully modeled.
2) The *abstract actor model* can be naturally translated into automata as input for formal verification engines. Gecko supports this translation and uses logical time / action causalities to avoid excessive use of variables in the verification model. Both actor actions and token transfers are represented by edges[3].
3) The *fault model* is used to annotate non-deterministic edges on the original model. As we try to eliminate the extensive use of clock variables, the sporadic behavior for the occurrence of faults requires appropriate abstractions to be tailored into the functional model.

Limitation: Currently, Gecko only generates partial stubs of the whole model. Specifications, as well as user defined functions must be annotated manually to the UPPAAL model, such that meaningful analysis results can be derived.

D. Deriving Integrity Constraints

With the above framework, we plan to investigate new techniques and algorithms and construct them on top of Gecko. In this section, we give an example how the tool can benefit from integration of these techniques.

In the fault-tolerance community, *integrity constraints* are conditions that hold in normal operation, but may fail to hold in the event of a fault [12]. As sensors may have imprecisions, deriving integrity constraints to distinguish between effects caused by imprecisions or errors is crucial to prevent false alarms. Here a simplified scenario in fig. 3 is given to motivate the discussion and indicate our current progress.

In fig. 3, three identical sensor units `s1`, `s2`, and `s3` with imprecision ranging between $[-0.5, 0.5]$ perform readings from the environment, and pass their readings to the corresponding functional unit $z = (x+y)/2$. Results from each functional unit are transformed to the voter to detect erroneous results, and a copy of the previous result directly from port z is stored in the memory for next stage processing (similar to the integral part of the PID control). It can be observed that due to accumulative sensor imprecision, values on ports A (similarly B and C) can deviate from the actual value with a maximum ranging over $[(\frac{(-0.5+0)}{2} + \frac{(-0.5+\frac{(-0.5+0)}{2})}{2} + \ldots), (\frac{(0.5+0)}{2} + \frac{(0.5+\frac{(0.5+0)}{2})}{2} + \ldots)] = [-\Sigma_{i=2}^{\infty}\frac{1}{2^i}, \Sigma_{i=2}^{\infty}\frac{1}{2^i}] = [-0.5, 0.5]$. Therefore, when $|\alpha - \beta| > 1$ occurs, where $\alpha, \beta \in \{A, B, C\}$, we are certain that either α or β contains a erroneous sensor reading which leads to the result (or faulty in our definition)[4];

[3]An edge update takes zero time; time consumption is explicitly added, which is similar to discrete-event simulation in Ptolemy or SystemC [2].

[4]Here a single fault-occurrence assumption is applied, meaning that at most one sensor can generate erroneous readings at any instance.

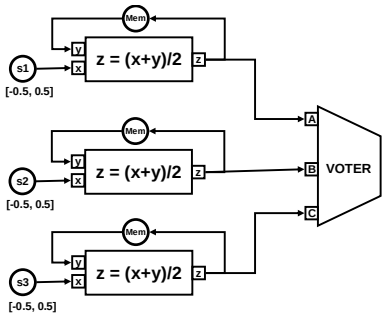

Fig. 3. An example where integrity constraints are considered over the voter component.

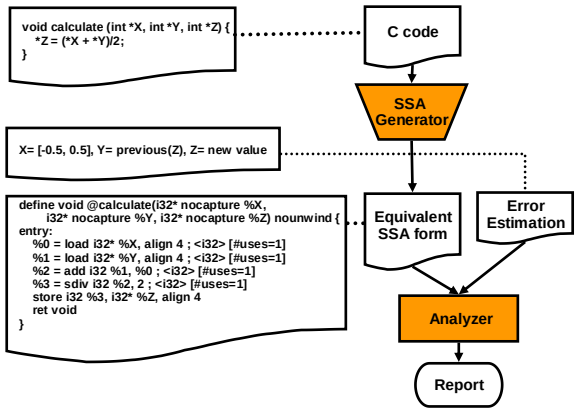

Fig. 4. Process flow for the generation of integrity constraints.

the integrity constraint can be designed based on the above information. The absence of such a criteria (no bounded interval available), indicates design errors. Following this example, in Gecko the process flow of generating integrity constraints (algorithm omitted) is sketched in fig. 4:

1) *(Preprocessing)* By adapting techniques in compiler technologies, the equivalent static single assignment (SSA) form using LLVM [5] can be generated, which enables efficient manipulations in the next stage.
2) *(Analysis)* The analyzer module gives each register variable (in fig. 4, `%0`, `%1`, `%2`, `%3`, `%X`, `%Y`, `%Z`) two values $(value, imprecision)$, and update these values based on the following two types:
   - *Atomic machine instructions* (in fig. 4, `load`, `store`, `add`, `sdiv`). Consider in fig. 4 with the instruction `%2 = add i32 %1, %0 ;`. Given `%0` with imprecision $[-0.5, 0.5]$, `%1` with imprecision $[-0.25, 0.25]$, `%2` would have imprecision ranging over $[-0.75, 0.75]$.
   - *Global iterations* (e.g., a sensor reads one new value from the environment, load memory, which is recorded separately in the information "Error Estimation" in fig. 4). Note that as fixed points may never be reached, the number of iterations can be set explicitly.

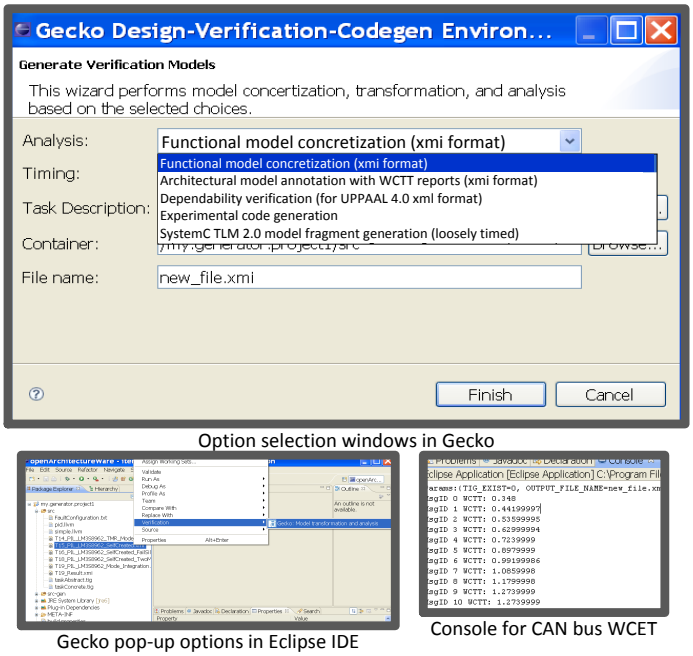

Fig. 5. Screenshots when designing systems using Gecko.

Limitation: The integrity constraint generator module currently operates over functions under *affine transformations*; for conditional operations, a decision by taking both edges is approximated. The analysis works with sensor imprecision modeled using both intervals and Gaussian distribution, which is also closed under linear transformations.

III. Evaluation and Extension

The above functionalities of Gecko have been implemented (preliminary version and demo available at [4]) under the Eclipse platform, and fig. 5 shows some screenshots when designing with Gecko. Stepwise refinement operations can be performed to generate models or executable code. A complete demonstration using the tool over Luminary Micro LM3S8962 evaluation boards is also available, where a model containing distributed voting using the CAN bus is designed and deployed. To support detailed analysis, Gecko also enables designers to generate SystemC modules from Gecko models; currently only loosely-timed modeling methodology (using `b_transport()` primitives) is available.

To provoke free extension and usage, the package will be released as an Eclipse add-on under GPL 2.0. For extensions, introducing new mechanisms requires users to design a parameterized skeleton using EMF, which can be easily adapted by referencing existing mechanisms; introducing new platforms requires users to implement links from abstract instructions (e.g., `Send()`, `Wait()`) to concretized mechanisms.

IV. Brief Overview on Related Work

In general, different tools focus on different aspects in the overall design flow. For instance, Metropolis [6] focuses on the joint modeling of applications and architectures, and Ptolemy focuses on the unified semantics for the execution of heterogeneous models of computation. Our focus, compared to others, is different: we expect to offer simplified (light-weight) means to achieve dependability and fault-tolerance under a unified design platform or to experiment novel techniques.

V. Concluding Remarks

Our contributions are summarized as follows:

- We presented a framework for embedded systems to equip with fault-tolerance with reduced design efforts. We bind actor-oriented methodologies and fault-tolerant patterns [11] using tree-structures, and perform local-scale design space exploration (for communication), and tree translation (for actors) in the refinement process.
- Analysis techniques are introduced (dependability model generation, simple timing analysis), or experimented (integrity constraint analysis) in Gecko.
- The tool enables interfacing and integration with other tooling by sharing a common platform (Eclipse IDE) and by translating Gecko models into SystemC models.

The introduction of model-driven development (MDD) is to facilitate the design process by providing an abstraction of the system behavior, where complexities of the system construction can be reduced. The Gecko tool, which is based on MDD and focuses on fault-tolerance and dependability aspects, will continue to be improved and extended.

References

[1] Eclipse Modeling Framework. *http://www.eclipse.org/modeling/emf/*.
[2] Open SystemC Initiative (OSCI). *http://www.systemc.org/*.
[3] openArchitectureWare Project. *http://www.openarchitectureware.org/*.
[4] Temporarily folder for the Gecko Project. *http://www6.in.tum.de/~chengch/Gecko/*.
[5] The LLVM Compiler Infrastructure Project. *http://llvm.org/*.
[6] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, pages 45–52, 2003.
[7] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
[8] C. Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. PhD thesis, Technische Universität München, Oct 2008.
[9] R. Davis, A. Burns, R. Bril, and J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.
[10] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
[11] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Software Patterns Series, 2007.
[12] I. Hayes. Dynamically Detecting Faults via Integrity Constraints. In M. Butler, C. B. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2009.
[13] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.
[14] A. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press New York, NY, USA, 2008.
[15] K. Tindell, A. Burns, and A. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.