Code Generation From ECDAR

Rasmus Berntsen, Florian Biermann, Lauge Groes, Nicolas Wainach Lundqvist Jensen, Thomas Kokholm, Piort Stawiski, and Wiktor Michal Zdziechowski

IT University of Copenhagen, Denmark
`{raber, fbier, lgro, nicl, tkok, psta, wmic}@itu.dk`

Abstract This paper present a framework for code generation, based on timed automata modeled in ECDAR (Environment for Compositional Design and Analysis of Real Time Systems). ECDAR is a graphical tool based on UPPAAL that allows to visually create models of real-time systems.

The motivation for doing this project is, that the current version of the ECDAR framework is lacking an important feature: The possibility to utilize models for code generation, as a way to develop software solutions based on visually represented models.

The general approach of this project is to solve this aforementioned challenge: Show how one can generate true code, in this case Java, from an ECDAR model (3). This is done in two closely related parts. The first part is a framework build to match the single parts of the ECDAR specification (3.2). The second part presents how to generate compilable code from an ECDAR model. The code basis of the generated code is from the framework(3.4).

We furthermore created the notation of tasks as an extension to the ECDAR model, within the framework (3.1).

The functionality of the framework is evaluated through a series of tests (4.1).

From our findings in the evaluation (4) **[Finish it when we have the evaluation in place]**

1 Introduction

In the preceding years a new level of abstraction in development have been evolving. Utilizing a higher level of abstraction, than high-level programming languages, we have Model Driven Development. This new paradigm is combining a focus on automation and code generation, to enable a new way of black boxing solutions within a multitude of specialists outside traditional programming while securing platform independency.

1.1 Real Time Computing

Real-time computing is the study of hardware and software systems that must satisfy explicit response-time constraints or risk severe consequences, including failure. Timed systems are used in a wide range of domains including communications, embedded systems, real-time and automated control. They can be easily found in our environment; one of simple examples is the airbag system in a car. The real-time constraint in this system is the reaction time between crash sensors receiving input and the deployment of airbags. Among some of the important characteristics of real-time systems we can distinguish extreme reliability and safety as they are very often safety-critical.

1.2 ECDAR

The “Environment for Compositional Design and Analysis of Real Time Systems” (ECDAR) - is a graphical tool based on UPPAAL TIGA [**?**] that allows to visually create models of real-time systems. Unlike UPPAAL [**?**], it is implementing a complete specification theory for real time systems [**?**,**?**]. In ECDAR, components of the system are described as automatons extended with clocks (timed automata), that can be combined to form larger comprehensive system descriptions. Correct specification of composition is supported by well defined compositional reasoning theory, consisting of operators like: parallel composition, conjunction, satisfaction checking and refinement. On the top of that, the tool allows for scalable verification of models by querying the implementation with verification questions [**?**].

1.3 Project

This paper will follow an implementation of code generation from ECDAR to Java. The proposed implementation of the ECDAR code generator is split up in two parts. The first part is a framework of abstract classes, implementing in as much detail as possible the single parts of ECDAR specifications (i.e. edges, locations, TIOA). The second is the actual code generation. Our code generator generates sources which inherit from the abstract framework to minimize the amount of code that needs actually to be generated. The paper will also detail the different testing issues and benefits of implementing code generation for TIOA, and reflect upon how this could be implemented concretely for Real Time Systems. However, to write programs that meet the demands of real-time systems in Java the Real-Time Specification for Java (RTSJ) is needed. RTSJ is a set of interfaces and behavioral specifications that allow for real-time programming in the Java programming language. Real-time in the RTSJ context means "the ability to reliably and predictably respond to a real-world event". So "real-time" is about predictable timing. In our implementation we don’t utilize RTSJ, so we can’t for instance guarantee thread prioritization and timing. RTSJ is only commercially available but can be integrated in to our framework in the future.

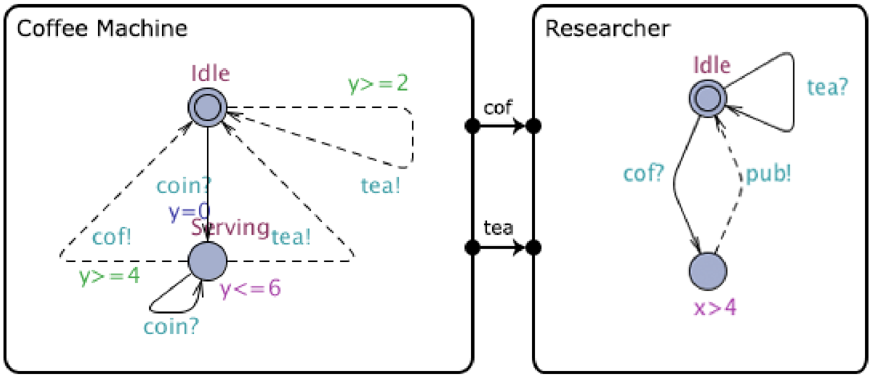

Figure 1. Model of beverage-serving machine and researcher.

2 Background

2.1 Timed Input/Output Automata

The "Timed Input/Output Automata" is a basic, mathematical specification framework for description and analysis of real time systems. In this framework, system is represented by non-deterministic, possibly infinite-state, state machine referred as “timed I/O automaton” (TIOA) [?]. TIOA has been implemented as the modeling language in ECDAR [?].

The preceding figure (see Fig. 2.1) illustrates the system consisting of two automatons: *Coffee Machine* and *Researcher*. The *Coffee Machine*, given a coin (*coin*), it serves either coffee (*cof*) or tea (*tea*) to the *Researcher* within a given time interval. Moreover, free tea is served once in a while. The *Researcher* is producing publications (*pub*), once provided a timely stimuli in form of preferred beverage (*cof*). In the example, the *Coffee Machine* - TIOA consists of and two locations represented by circles: *Idle* and *Serving*. *Idle* represents the starting location, and the state of machine waiting for coin input (*coin?*). Analogously, the *Researcher* is in *Idle* state expecting either coffee (*cof*) or tea (*tea*) provided by *Coffee Machine*. The flow of each TIOA is controlled by three types of labels: *invariants*, *guards* and *clock-reset operations*. Invariants are defined on locations ($y \leq 6$ and $x > 4$) and represent constraints for the clocks in order for the control to remain in particular location until time requirement is fulfilled. Guards are located on the edges ($y \geq 2$ and $y \geq 4$) and express conditions on the values of clock that must be satisfied in order for the edge to be taken. When the condition is satisfied, the transition occurs and action (*cof!*, *tea!* or *pub!*) is triggered. Clock-reset operations ($y = 0$) are simple clock value manipulations in form of assignment that enforce progress in the system.

In ECDAR, the specification interface is leveraging the UPPAAL TIGA language [?] to describe TIOA. However, the following constraints are retained[1]:

[1] See `http://people.cs.aau.dk/adavid/ecdar/examples.html#lang`

- Invariants may not be strict.
- Inputs must use controllable edges.
- Outputs must use uncontrollable edges.
- All channels must be declared broadcast.
- The system is implicitly input enabled due to broadcast communication but for refinement checking purposes the relevant inputs must be explicit in the model.
- In the case of parallel composition of several components, a given output must be exclusive to one component.
- For implementations, outputs must be urgent.
- For implementations, every state must have independent time progress, i.e., progress must be ensured by either an output or infinite delay.
- τ-transitions (no output or input) are forbidden.
- Global variables are forbidden.

2.2 Code Generation

In order to clarify what code generation is one need to understand what a model transformation is, as this is a fundamental part of code generation. Briefly explained a model transformation can be seen as a process of converting an input model, that complies to a certain metamodel, to a new model. The generation is an automated way to produce code from models. The actual generation is defined by the software developer, thus it is defined what the output should be, but the input and the data is not. The reasons for doing code generation are many: One could mention code quality, cost-effective, less error-prone and generally easier to understand for non-domain experts.

There are generally two ways perform a model transformation, that is "model to model" and "model to text", the former known as M2M and the latter M2T. M2M is a transformation of a number of models to a given number of new models – from X number of models to Z number of new models. M2T is the transformation of a number of models to text, the text could for instance be code – which is why the process sometimes is known as "model to code".

There are also a lot of other tools and techniques for transformation, which should not be confused with model transformations. One could mention an XSLT-transformation as an example, where the base input is an XML-document and the final output is another XML-document, often XHTML, with a predefined XML-Schema.

3 Implementation

The proposed implementation of the ECDAR code generator is split up in two parts. The first part is a framework of abstract classes, implementing in as much detail as possible the single parts of ECDAR specifications (i.e. edges, locations, TIOA). The second is the actual code generation. Our code generator generates source code which inherit from the abstract framework to minimize the amount

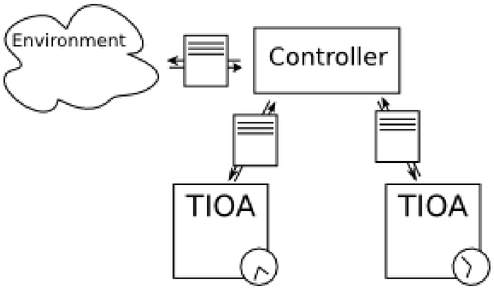

Figure 2. Schematic of the architecture of our ECDAR implementation.

of code that needs to be generated. This means that nearly all design decisions have been made prior to generating code, reducing space for possible errors. This section describes our implemented subset of ECDAR and the code generator in detail.

3.1 Tasks

ECDAR defines the behavior of a system as a state machine. This behavior is, however, still too abstract to justify code generation. We can generate code which implements the behavior of state machines, but in essence, the system would then only produce messages.

To make this tool more useful, we introduce the notion of tasks as an extension to the language. Each location is assigned exactly one task. A task is a procedure which will be executed as soon as an automaton traverses over an edge, arriving at a new location. Such a task could for example be a procedure that heats up the water in the coffee machine from Fig. 1 or some code that extends the undercarriage of a plane when approaching the landing strip.

Tasks can either be preemptive or non-preemptive. This property becomes important for defining behavior of automata when they are notified about input by the controller.

ECDAR is input-enabled (see Sect. 1.2) and therefore, the system is required to react to input immediately. As a consequence, there must also be a well defined reaction to input during the execution of a task.

When an automaton is executing a task and it receives an input message which it accepts, it may stop the currently executed task and proceed as originally defined in ECDAR (i.e. traverse the corresponding edge), if and only if the task is preemptive. Otherwise, the given input will be ignored and the execution of the task continues.

To determine if a task is preemptive is up to the designer of the system to decide. By default all tasks are non-preemptive.

```java
public class UniversityController extends IController {

  public UniversityController(ITIOA[] automata) {
    super(automata);
    IController.controllerInstance = this;
  }
}
```

Figure 3. Example of controller code.

3.2 The ECDAR Framework

The architecture we chose is based upon the work of Amnell et al.[**?**] with some modifications. Communication between automata is implemented as message passing between a controller and automata, where automata send messages to the controller by traversing over output edges. This is different in actual ECDAR, where automata communicate through message passing directly between each other. However, by choosing a slightly different architecture, we can unify the message system in the implementation, handling all messages central and also not distinguishing between a message coming from the environment or a message send by an automaton. The resulting behavior is equal.

Automata need to execute in quasi-parallel. In the implementation they are run in a classical threading architecture that does not require multiple processor units. Since there is no communication between automata directly, we can minimize synchronizing between threads (see more on synchronizing in Sec. 3.3).

The following overview will give further implementation details on each component of ECDAR as we implemented it. Each component is illustrated with a short code example, implementing ECDAR's "University" example[2]. For clarity, we omit the framework implementation and focus on the generated code.

Controller. The controller holds all automata given in the specification and notifies them about received messages. It is a singleton, accessible in a static fashion. This property is useful for sending messages to the controller.

TIOA. The implementation of timed I/O automata holds a set of locations and a reference to the location it is currently at. The TIOA is executed by a thread that keeps checking for available edges and traverses along these as soon as they become enabled. To check if an edge is available, we check only the edge's associated guard. Though it is tempting to also check the invariants of the source and target locations of the edge, there are valid reasons against both.

For the source edge, the check is redundant, since we know that if the invariant was violated, the system would already be in a deadlocked state. The reason for not checking the target edge's invariant is a bit more subtle. An edge can

[2] `http://people.cs.aau.dk/adavid/ecdar/examples.html#university`

```java
public class Machine extends ITIOA {
  ILocation idle, serving;

  public Machine() {
    super();

    idle = new Idle(this);
    serving = new Serving(this);

    idle.setupEdges();
    serving.setupEdges();

    current = idle;
  }
}
```

Figure 4. Example of TIOA code.

update the clock of the automaton in which it resides on traversal. If a target edge's invariant is checked before the edge pointing to it as been traversed, this change is not taken into account. The specification, however, always takes this update on the clock into account. Therefore, only checking the edge's guard is valid.

Additionally, the automaton has the ability to return the current local clock state (see 4.2) and to reset the clock. We use the same notion of clocks as [**?**], where time on the local clock is the difference between the current time on the system clock and the time the local clock was started. Resetting the local clock means to use the current system clock time as the new start time (see Fig. 4).

Locations. Each location is associated with a task (see Sec. 3.1). Task execution is implemented in a separate thread so that the execution of the automaton is never blocked. Locations are implemented as objects holding an array of edges that point away from it. (See Fig. 5)

Edges. An edge holds a reference to the location which the parent automaton will be at after traversing this very edge. Edges can be asked if they will be available at a given time. This is implemented to enable lazy waiting in the automaton's traversal checker. Each edge is associated with some input. If an edge is controllable, it will be triggered if the automaton is notified at this input. If it is uncontrollable, it will send its input to the controller. Furthermore, edges have access to the clock of the parent automaton to reset it appropriately.

The implementation makes a class-wise distinction between input edges (i.e. edges that are traversed when a corresponding message was received) and output edges (edges that send messages on traversal) and hard-codes the behavior in the framework, e.g. messaging the controller (see Fig. 6).

```java
class Idle extends ILocation {

  public Idle(Machine parent) {
    super("idle", parent);
  }

  public void setupEdges() {
    outputEdges = new IOutputEdge[]{new Idle_TEA_Idle()};

    inputEdges = new IInputEdge[]{new Idle_COIN_Serving()};
  }

  public boolean checkInvariant(long time) {
    return true;
  }

  public boolean isPreemptive() {
    return false;
  }

  public void task() { }
}
```

Figure 5. Example of location code.

3.3 Synchronization

In the framework implementation, the Java keyword *synchronized* is used for making certain operations quasi-atomic. That means, that a set of instructions may not be interrupted by the execution of another thread – i.e. traversals over edges.

This is mainly used for the logging of signals, so that logging time is preserved and the output appears in the right order. *Synchronized* is also used, to set some internal states on TIOA, where the internal state consists of multiple values that need to be set at the same time.

Synchronized is furthermore used to prioritize handling of input. The method on the controller object, that is handling the signal, as well as those on the TIOA that react to a signal if it is accepted, are modified with *synchronized*. This ensures that, before everything else, the input is processed.

3.4 Code Generation

In the implementation the generation of source code is done through a model to text transformation. The generation outputs compilable Java based on input from an ECDAR file.

The Eclipse Modeling Framework (EMF) is utilized for the process. EMF is a modeling framework and code generation facility for building applications

```
class Idle_COIN_Serving extends IInputEdge {

  public Idle_COIN_Serving() {
    super(serving, "coin");
  }

  public boolean checkGuard(long time) {
    return to.checkInvariant(time) && true;
  }

  public void onTraverse() {
    resetTime();
  }
}
```

Figure 6. Example of edge code.

based on a structured data model. From a model specification described in the XMI-format, EMF provides tools and run time support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model and it provides a basic editor. The core EMF framework includes a meta model -in Ecore- for describing models and run time support for the models including: change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically. In the implementation presented in this paper an Xtext environment is generated from the ECDAR Ecore model. Xtext is a framework for development of programming languages and domain specific languages.

In order to generate code from the model it's imperative to follow a process of multiple steps: Get the input from ECDAR, translate this to Xtext ECDAR DSL, setup a workflow that manages the process and finally a Xpand-template is needed to define how the transformation output should look like. Each step is described in more detail in the following section.

Transformation Process The initial output from the ECDAR tool is in XML-format. The XML-output contains a complete definition of the model with locations, edges, variables, transformations etc. In order to work with these files and do the actual code generation, a conversion to Xtext ECDAR DSL is needed. For this conversion we are using a converter (courtesy of Bastian Müller) that simply takes the ECDAR XML-file and converts it to ECDAR DSL. The ECDAR DSL syntax is defined in our Xtext ECDAR environment. With the combination of the Ecore meta model and the Xtext syntax a workflow can be defined. This workflow is describing how to handle the generation process. This is done with the help of the "Modeling Workflow Engine 2" (MWE2). Also referenced in the workflow is the template that describes how the actual output is going to

```
«IMPORT ecdarText»
«DEFINE main FOR ecdarText::ETSpecificationDefinition»
```

```
«FILE this.name + ".java"»
```

```
public «"Edge" + iter.counter1»() {
  //Target: «edge.target.name»
  «FOREACH
      edge.io.eContents.typeSelect(ecdarText::ETReference)
    .target.eContainer.eContents.typeSelect(ETVariableID)
    .toList() AS var ITERATOR iterVars-»
  «IF iterVars.firstIteration-»
      super(«edge.target.name», "«var.name»");
  «ELSE-»
```

Figure 7. Snippets from Xpand-template

look like. The templates are written using Xpand. Xpand is a statically typed template language. Conveniently Xpand supports code-completion directly connected to the Ecore model defined in the MWE2 workflow, but also comes with syntax coloring, refactoring and error highlighting. The output generation results, for the system presented in this paper, are based on several workflows and templates to do the rather complex transformations: One set of workflows and templates for respectively the Specification, Controller and Environment.

More specifically in the workflow-file one defines what model to use, a slotname to refer to later and an entry point. The entry point defines which class element is the top or root element. The entry-element that is specified for the three aforementioned workflows is "ETSpecificationDefinition". Also defined in the workflow is how to use the entry-element. For instance in the specification workflow it is defined that for each "ETSpecificationDefinition" a transformation is done using the Xpand template for this particular generation. The end result is generated output for each specification that was initially described and modeled from within the ECDAR XML-file.

With the workflow fully configured, the next step is to write the transformation. This is done in a Xpand template. The snippets in Fig. 7 shows some important steps.

The arrows, known as guillemots (“«” and “»”), indicates where in figure 7 the XPAND language is in place. First of all an import of the model is done in the first line, referenced as ecdarText. We then proceed to one of the central concepts of Xpand by using the define-block; This is where we define our template. We only use one template in this specific file, but it could have contained multiple, which would have resulted in multiple define-blocks. In the next and last snippet we jump to a part where we are iterating through each edge and create a constructor for the current class. In the first line we create the constructor by inserting the text "Edge" and add the number the iterator has reached. We then iterate through a list of the current Edges variables, which should be one signal, and returns the results as a list. We furthermore use a new iterator to keep track of this iteration. Afterward we use a check too make sure it's the first iteration, and if it is, we print out the current Edge target name and the variable signal, such as "super(C, "signal");".

The notion of tasks as previous described in section 3.2. is accounted for in our generated output. In the controller we generate functions that will be invoked at each location. A task is a procedure which will be executed as soon as an automaton traverses over an edge, arriving at a new location. There can only be one task for each location. The idea behind having all methods in the controller is for a better overview and a centralized customizable file.

4 Evaluation and Discussion

4.1 Testing

In order to test our software solution three different testing methods have been arranged. Our test case is the "University" example as introduced in Sec. 2 and 3.

Compilation To test code generation, our first step in validating our approach is checking if the generated code compiles. If the code does not compile, we know that the code generator does not generate valid code and that our templates are wrong.

Manual Comparison A first approach to validating correctness is manual validation of generated log files. A log file contains each action that occurs in a specified system with local time, i.e. the clock time on the automaton, and global time, i.e. time since the system started.

We employed this approach on the "University" example, together with Andrzej Wasowski, who authored ECDAR and the given example specification.

Log File Analysis As means to analyze the generated log files in a more automated and valid fashion we conceived a JUnit test program. This program is tuned especially for the "University" example and therefore does currently not apply to any other specifications.

The analyzer converts a log file into a list of *Events*, more exactly *Signals*, i.e. messages that are send from the environment or from automata to the controller, which can be easily compared to in terms of type, source and global time. These events are stored in a list sorted by time of occurrence. For each occurring signal, there is a follow-up signal defined in the specification (e.g. "grant" is followed by "coin") within a certain time frame.

The list of events is then searched until an occurrence of the follow-up signal is found. For each signal encountered, one assertion for a follow-up signal is issued. If a signal is missing its follow-up signal, the assertion evaluates to false and the test case will result in an error.

We assume, that after one cycle through the specification, beginning with "grant" and ending with "patent", the system is behaving correctly. Therefore, we let the test terminate as soon as "patent" is encountered.

All runs of automated analysis for logs generated with our implementation succeeded. However, it needs to be noted that currently, we do not verify the timing constraints but only that the general behavior of the system is correct.

4.2 Presumptions and Resulting Motivations

Our implementation represents only a subset of actual ECDAR. Currently, the implementation assumes only one clock per automaton. Also, we assume the specification to be valid, since there are other tools that verify correctness[3].

ECDAR features a variety of operators on specifications [**?**]. The only operator fo these that our code generator performs upon is the parallel composition operator. Let M be the type ECDAR specification. Then all operators in ECDAR are of type $M_i \otimes M_j \rightarrow M_{ij}$. It is impractical to implement parallel composition as a model-to-model transformation, since it produces the cross-product of two models [**?**]. These models are size $|M_i| \cdot |M_j|$ and generating code for them would consume a large amount of memory and raise complexity of the actual implementation. This would be inappropriate for an embedded system. Opposite, if we used model-to-model transformations for the parallel composition operator, we would not have to handle timing on threads or synchronization between them.

ECDAR specifications are written on the assumption of the synchrony hypothesis (see Sect. 1.1) [**?**]. This is an important property for code generation, as reasoning about time differences in execution becomes unnecessary for the developer. However, we still kept overhead low in the implementation to achieve reasonable fast performance.

5 Related Work

The aim of this project is to develop a tool for automatic synthesis of Java-code, from the modelling tool: ECDAR. However there already exist tools similar too ECDAR.

TIMES[4]: is a graphical tool set for modelling, schedulability analysis for implementation of embedded systems. It allows users to model a system and the abstract behaviour of its environment. Like ECDAR, TIMES is based on timed automata (See section 1.2 on page 2 for more details on automaton). Similarly TIMES derive from UPPAAL and is based on the standard for modelling real-time systems [**?**]. Times is extended by real-time tasks, checking the reachability

[3] `http://people.cs.aau.dk/adavid/ecdar/`

[4] `http://www.timestool.com`

and schedulability of a modelled automata. It too can simulate models and validate dynamic behaviour of a system: Users can see how tasks executes according to time. The simulator shows a graphical representation of the generated trace, showing the time points when the tasks are released, invoked, suspended, resumed, and completed.

TIMES can be used for code-generate, as illustrated Tobias Amnell et al [**?**]. They generate C-code for a Lego Mindstorms system using TIMES, checking for reachability and schedulability. Simulor to what we trying to do with ECDAR.

There are certain differences between TIMES and ECDAR, despite the missing feature of code generation in ECDAR. In contrast to TIMES's programmable logic controllers, a scheduling approach is not suf?cient to deal with unpredictable behaviour in ECDAR. Looking at what are taking as input by the controller - TIMES simple takes a model, checking reachability and schedulability, whereas ECDAR synthesise each component in a model.

Composable Code Generation for Model-Based Development by Kirk Schloegel et al. present a framework for generating code[**?**]. They emphasize how utilizing their framework, code generators aren't programs separated from a corresponding graphical model as it often have been in the past. Our code generator isn't based on this framework, however their approach on developing code generators with focus on graphical models is related to our approach with ECDAR.

Code Synthesis for Timed Automata by Tobias Amnell et al. present a framework for the development of real-time embedded systems[**?**]. Their work is similar to our project. In the article the illustrate how their framework is based on timed automata and real-time tasks - relative to our concept with ECDAR.

Controller Program Synthesis for Industrial Machines a PhD thesis report utilizing timed automata to control different state of logic in [**?**].

6 Conclusion

In this project we have developed a code generator for a TIOA modeller: ECDAR. We have successfully generated code from simplified model into Java. See section 4.1 on page 11.

7 Acknowledgements

The XML to Xtext ECDAR DSL, as mentioned in section 3.4 on page 9, is developed by Bastian Müller, student at IT Unviversity of Copenhagen.

The project has been carried out with supervision from Andrzej Wasowski, Associate Professor at IT University of Copenhagen.

A First Appendix