Journal of Universal Computer Science, vol. 10, no. 10 (2004), 1383-1407
submitted: 30/6/04, accepted: 15/10/04, appeared: 28/10/04 © J.UCS

Platform Modeling and Model Transformations for Analysis

Tivadar Szemethy
(Institute for Software-Integrated Systems, Vanderbilt University,
tivadar.szemethy@vanderbilt.edu)

Gabor Karsai
(Institute for Software-Integrated Systems, Vanderbilt University,
gabor.karsai@vanderbilt.edu)

Abstract: The model-based approach to the development of embedded systems relies on the use of explicit models in the design process. If these models faithfully represent the components of the system with respect to their properties as well as their interactions, then they can be used to predict the dynamic behavior of the system under construction. In this paper we argue for modeling the execution platform that facilitates the component interactions, and show how models of the application and the knowledge of the platform can be used to translate system configurations into another abstract formalism (timed automata, in our case) that allows system verification through model checking.

Keywords: Model-Based Development, Model Transformations, Models of Computation, Graph Transformations, Software Verification
Categories: D.2.2

1 Introduction

Model-based development of embedded systems [Karsai, 03a] facilitates design-time analysis. Because the models are abstractions of the system that the designer is building, analysis algorithms can be applied to these models and properties of the system can be computed and thus the behavior of the system predicted. Typical properties include schedulability, lack of deadlock, maximum usable data rates, worst-case reaction times to events, end-to-end latency and others. In spite of the apparent validity of the principle of model-based analysis, there are very few [Larsen, 97][Amnell, 02][Gu, 03] actual and practical design tools that fully implement this principle.

In this paper, we are focusing on the model-integrated construction of component-based embedded systems. In this context "model-integrated" means that models play an integral role in the design, analysis and synthesis of the embedded application, while "component-based" means that the system constructing from components that interact with each other using a well-defined model of computation [Lee, 97]. This model of computation is implemented and facilitated by an underlying component infrastructure (e.g. real-time CORBA [RTCORBA]) that strictly enforces the rules of component interaction. We argue that the analysis of the design should be explicitly based on the knowledge and precise understanding of this platform. By "understanding" we mean having a formal, operational semantics of the platform, which may also include performance metrics, like worst-case context switching delays. If the operational semantics of the platform is known, one can create a formal model of it, and this explicit *platform model* could serve as the conceptual foundation for realizing the analysis approach for the component-based system.

In this paper we will show how a platform model could be defined, and how it can be used to perform the analysis. Platform models can be explicit or implicit. As we use a general-purpose analysis engine (which is based on the concept of timed automata), the platform models are implicit and captured in the form of the model transformations rules that map system models into analysis models.

2 Model-integrated development of component-based embedded systems

In model-integrated development one uses domain-specific modeling languages to capture salient properties of the system to be constructed. These models are then used both in analysis of the design and in synthesis of the final application. The key concept here is that the models of the system are explicit (i.e. a designer effectively creates them), and they are also an abstraction of the system.

In component-based design the system is constructed from components that have a well-defined behavior and interfaces through which they interact with each other. This applies to component-based embedded systems as well. Components interact with each other through precisely defined interaction patterns, called the model of computation. In single-threaded software designs the main (sometimes the only) method of component interaction is the procedure call, with the usual call-return semantics. In embedded system designs, where concurrency is prevalent, more complex interactions patterns are needed. The hardware typically gives direct support for the call-return interactions via the subroutine call/return instructions, but very rarely gives direct support for more complex concurrency patterns. These concurrency patterns are implemented using some underlying computational framework or infrastructure that we will call the *platform*.

The platform provides an abstraction layer above the raw hardware (which may include multiple processors with communication services, etc.), and defines the interaction patterns among components. We make the assumption here *that all component interactions happen via the platform*, and no other component interactions are allowed. Thus, component based systems are constructed as shown in Figure 1. In the figure, broad arrows indicate the physical interactions (between the component and the platform), while thin dashed arrows indicate the logical interactions (between components). Obviously, the latter are implemented by the former. Note that this platform-oriented view is compliance with the principles of platform-based design [Sangiovanni-Vincentelli, 01].

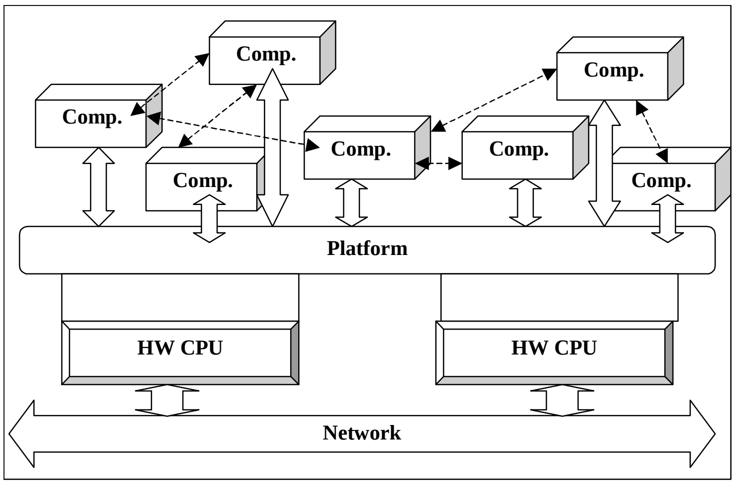


Figure 1: Component-based system

In the embedded design process we often need to perform an analysis on the design. This analysis can be used to answer important questions about the system: Is it schedulable? Does it meet all the required deadlines? Will it deadlock? Is there a danger for priority inversion? What are the maximum usable data rates for the system? What is the maximum data latency between signal flow endpoints of the design? What is the maximum latency between the arrival of an asynchronous event and generating a response to that?

Obviously, the precise (and ultimately satisfying) answers to these questions could be given only after analyzing the code of the entire system, but this is often not feasible. The next best thing one can do is to analyze the model of the system and draw conclusions about the final system based on the models. This approach assumes that models are valid abstractions, and recognizes that a proof obtained is valid only if the models are valid.

Note that there are (at least) two kinds of models needed: models for the components and a model for the platform. Note that the component models are not necessarily the same as the designer has created: the designer can work on a higher level of abstraction, while lower-level models could be necessary for analysis. In terms of the language of the Model-driven Architecture [MDA], while the designer could work with Platform-Independent Models (PIM), for analysis one needs Platform-Specific Models (PSM) for the components.

For the platform, one needs a model that unambiguously captures the platform's behavior and describes how the components interact with each other via the platform. Note that the overall system is a composition: $C_1 \parallel C_2 \parallel \ldots \parallel C_n \parallel P$ (i.e. the components are composed with the platform). The objective of the analysis is to compute and verify the properties of the composed system consisting of the components and the platform (models).

Note that the platform model is abstract (in the general sense), but it must be made concrete for the configuration of specific applications. The abstract platform model captures how components interact in general, without fixing any specific component configuration. However, we are always interested in concrete, specific systems, where components are "wired" in a specific way. Therefore the platform model must always be concretized for the application one wants to analyze.

To summarize, the analysis of component-based embedded system designs necessitates that platforms are modeled, in addition to components. Platforms define the rules how the components interact with each other. For the analysis of specific designs one needs to have the concrete model of the platform, which includes the models for specific component interactions.

The requirement for concretizing the platform models led us to solve the problem using a transformational approach. In the subsequent sections, we describe our approach in general, and show some concrete examples how it works on a simple platform.

3 SMOLES: A Simple Modeling Language for Embedded Systems

In the remaining part of the paper we are going to use a simple modeling language that was used to model component-based embedded system designs. We introduce the language here informally.

SMOLES was designed as a simple modeling language that allows constructing small, embedded systems from components. The components are assumed to be concurrently executing objects that communicate and synchronize with each other. Furthermore, objects can perform I/O operations in which they wait for the result, while other objects can execute. Communication between components means passing data from a source component to a destination component, which is then enabled to run, in order to process the data. In addition to data triggering, periodic timers can also trigger the components.

SMOLES follows an event-triggered execution model: computations are activated when token(s) arrive at components. Tokens carry data that the component can retrieve.

SMOLES is a modeling language that focuses on component composition and coordination. As such, it has to work together with an underlying procedural language, which is used to implement specific details of algorithms, data structures, etc, used in the final application.

Specifically, the language consists of *Components* and *Assemblies*. Components are the elementary building blocks, and contain:

- *Input and output ports*, which are used to receive/send data tokens from/to other components
- *Attributes*, which are data members of the component objects
- *Methods*, which are operations that the component implements, written in an underlying procedural language
- *Triggers*, which specify how the arrival of data on input port(s) will trigger the invocation of methods or behaviors.

The components work as follows. *Input ports* feed into *triggers* that activate a single *method*. The triggers specify whether all or at least one (any) input port(s) must have a data token in order to fire: If multiple ports are fed into a single trigger, then *all* of the ports have to have data available to activate the trigger. If multiple triggers are connected to a method, then the method executes if *any* of these triggers becomes active.

When the condition is satisfied, the activated method executes. *Methods* can also be connected to output ports, on which they can send data tokens to downstream components. The number of tokens produced by a single method execution could be specified as a range of integers. One of the methods can be marked as *initial*, which will be executed when the component is instantiated for the first time. Methods can access the attributes as instance variables of their owner the component object, while the input and output ports are accessed via API calls.

Assemblies contain *components*, and describe how they are interconnected. Like components, assemblies can have their own input and output ports, and assemblies can contain other assemblies. The wiring of ports of assemblies and components shows the data flows among components in the system. A port can be connected to precisely one other port. If data produced by one port should be sent to multiple other ports, or if multiple data streams should be merged into a single port a *Queue* has to be inserted. A queue is a multi-writer, multi-reader queue data structure that non-deterministically merges input data streams into a single one, which then can be read by multiple readers. An item is removed from a queue if all the readers have received it. Component input ports in an assembly can also be connected to *Timers*. Timers produce data tokens with a fixed period and trigger downstream components. Assemblies can be organized into a containment hierarchy, and the various components and assemblies in the hierarchy communicate with each other through the data flows via ports. If there is a need for accessing a non-local data flows in a specific assembly, one can place a queue reference object into the assembly that acts as a pointer to the remotely declared queue object.

4 Formal Definition of the SMOLES Modeling Language

Following the informal introduction to the language, its formal definition follows below. In the following definitions, we assume all sets to be finite, and *N* is the set of non-negative integers.

4.1 Syntactical definitions

A SMOLES model consists of the following elements:

1) A set of Component definitions:
A Component $\boldsymbol{C}=(\boldsymbol{I}, \boldsymbol{O}, \boldsymbol{Tr}, \boldsymbol{M}, \boldsymbol{InTr}, \boldsymbol{TriM}, \boldsymbol{MOut}, \boldsymbol{m_0})$ is a tuple:

$\boldsymbol{I}$ is a set of input ports, which are data token entry points.

$\boldsymbol{O}$ is a set of output ports, which are data token exit points.

$\boldsymbol{Tr}$ is a set of triggers.

$\boldsymbol{M}$ is a set of method instances M_i. A method instance is $M_i = (m_i, bcet_i, wcet_i)$ where m_i is the executable code of the method, and $bcet_i$, $wcet_i \in N$ correspond to the best- and worst-case execution times, respectively. Note that $bcet_i \leq wcet_i$ and $bcet_i > 0$

InTr $\subseteq (I \times Tr)$ is a set of *Input→Trigger* connections

TriM $\subseteq (Tr \times M)$ is a set of *Trigger→Method* connections

MOut is a set of annotated *Method→Output* connections: ***MOut****=(m,o,MinT,MaxT),* where $m \in M$, $o \in$ O, and *MinT*, $MaxT \in N$. Integers *MinT,MaxT* correspond to the minimum and maximum number of tokens emitted on *o* during execution of *M*. Note that $MinT \leq MaxT$.

$\boldsymbol{m_0}$: a trigger-less initial method. $\boldsymbol{m_0} \in M$ and $\neg\exists\, t \in Tr \mid (t,m_0) \in TriM$.

For Components we also use the following definitions: *Inputs(C):=I, Outputs(C):=O, Ports(C):=*$I \cup O$.

2) A set of Assembly definitions

An Assembly ***A*** *= (I, O, As, Co, Cl, Q, Df, Tt)* is a tuple:

I is a set of input ports, which are data token entry points.

O is a set of output ports, which are data token exit points.

As is a set of Assemblies contained in ***A***.

Co is a set of Components contained in ***A***.

Cl is a set of Cl_i instances, where $Cl_i = (timer_i, p_i)$, where $timer_i$ is a timer type, and $p_i \in N$, $p_i > 0$ is its period.

Q is a set of *Queues*

$$\boldsymbol{Df} \subseteq ((I \cup Q \cup \bigcup_{i=1..n} Outputs(As_i) \cup \bigcup_{j=1..m} Outputs(Co_j)) \times$$

$$(O \cup Q \cup \bigcup_{i=1..n} Inputs(As_i) \cup \bigcup_{j=1..m} Inputs(Co_j))\ (\text{with } m=|Co|, n=|As|)$$

is a set of *Dataflow* connections within the Assembly between ports and Queues.

$\forall (o,i) \in Df : o \notin Q \Rightarrow (\neg\exists j : (o,j) \in Df \wedge i \neq j)$. That is, a port can be connected to precisely one port and only Queues can be connected to multiple destinations.

$$\boldsymbol{Tt} \subseteq ((Cl \times (O \cup Q \cup \bigcup_{i=1..n} Inputs(As_i) \cup \bigcup_{j=1..m} Inputs(Co_j)),\quad (\text{with } m=|Co|,$$

$n=|As|)$ is a set of *TimeTrigger* connections connecting Timers to Ports and Queues.

$\forall (t,i) \in Tt$: $\neg\exists j$: $(t,j) \in Tt \wedge i \neq j$: a Timer can be connected to precisely one destination.

For Assemblies, we also define $(n=|Co|, m=|As|)$:

$$Inputs(A) := I \cup \bigcup_{i=1..n} Inputs(As_i) \cup \bigcup_{j=1..m} Inputs(Co_j)$$

$$Outputs(A) := I \cup \bigcup_{i=1..n} Outputs(As_i) \cup \bigcup_{j=1..m} Outputs(Co_j)$$

$$Ports(A) := Inputs(A) \cup Outputs(A)$$

Functions *Assemblies(A), Components(A), Queues(A), Dataflows(A)* and *TimeTriggers(A)* are defined in an analogous, recursive manner.

4.2 Semantics

Below, we define the execution semantics of the SMOLES language on a non-preemptively scheduled, uniprocessor platform. Data tokens are (for the scope of this discussion) atomic units of data used in component interactions in SMOLES. The set of all possible data tokens is denoted as *Dt*. On this platform, time is measured in integer units.
First we define the *state* of some objects:

State of a port (Input or Output) is an ordered set: $St \subseteq (N \times Dt)$ where *Dt* is the set of *Datatokens* and the integer is a consecutive sequence number: $n=|St|$, $i \in N$ such as:

if $(n=0) \Rightarrow St = \emptyset$,
else if $(n=1) \Rightarrow St = \{(1,d)\}, d \in Dt$
else $\forall\ (i,e),(j,f) \in St \Rightarrow 1 \leq i,j \leq n,\ i \neq j,\ e,f \in Dt$

The state of an actual port *p* is denoted as *St(p).*

State of a Component is the collection of the states of its ports, such that there is one state associated with each port:
$St(C) \subseteq (Ports(C) \times St)$ such as: $r \in Ports(C) \Leftrightarrow (r,St(r)) \in St(C)$

State of a Timer instance is its value: $St(Cl_i) \in N$

State of an Assembly is defined similarly to a component's state, extended with the state of the timers:
$St(A) \subseteq ((Ports(A) \cup Timers(A)) \times (St \cup N)$ such that
$r \in (Ports(A) \cup Timers(A)) \Leftrightarrow (r,St(r)) \in St(A)$

The firing of a Component is one execution of precisely one of the methods of the component, which has its trigger condition satisfied. During execution, the first data token is removed from all input ports connected to the activating trigger, and to each output port a non-deterministic *k* number of tokens is sent, where $MinT \leq k \leq MaxT$. As a side-effect the execution takes $bcet \leq c \leq wcet$ time units, as specified by the method instance.
For an assembly $\boldsymbol{A} = (I_A, O_A, As, Co, Cl, Q, Df, Tt)$ and component
$\boldsymbol{C_f}=(I,\ O,\ Tr,\ M,\ InTr,\ TriM,\ MOout,\ m_0)$, where $C_f \in Components(A)$, and $M_i=(m_i,bcet_i,wcet_i) \in M$ is a method instance, the behavior is as follows.

A ***firing cycle*** is an *action* $St(A) \xrightarrow{fire} St(A)'$ that executes as follows.

1. If there is a trigger connected to Method M_i is with data tokens available on all its input ports:
   $\exists\, t,\ s.t.\ (t,M_i) \in TriM$ and $\forall\, p_n$ where $(p_n,t) \in InTr,\ (1, d_n) \in St(p),$
   then

a) the data token is removed from the connected ports:

(p_n is the same as above: for $\forall p_n$ where $(p_n,t) \in InTr$, $(1, d_n) \in St(p))$

$St(p_n)' := \emptyset \cup ((\forall j: 2 \le j \le |St(p_n)|, (j,d_j) \in St(p_n)) \Rightarrow (j-1, d_j) \in St(p_n))'$,

$|St(p_n)'| = |St(p_n)| - 1)$, and

b) the unconnected input ports don't change:

$(\forall p_m$ where $\neg \exists t \,|(p_m,t) \in InTr)$: $St(p_m)' := St(p_m)$.

2. Next

a) DataTokens are written to the connected output ports:

$(\forall o \in O, (M_i,o,MinT,MaxT) \in MOut$ and $d_k \in Dt)$:

$St(o)' := St(o) \cup \bigcup_{i=0..k} \{(n+k),d_k)$, where $n = |St(o)|$, $MinT \le k \le MaxT$, and

b) the unconnected ouput ports remain unchanged:

$(\forall o \in O, (M_i,o,MinT,MaxT) \notin MOut)$:

$St(o)' := St(o)$

3. The updated Component state is the union of the updated Port states:
$St(C_f)' \subseteq (Ports(C_f) \times St)$ such as: $r \in Ports(C_f) \Leftrightarrow (r,St(r)') \in St(C_f)'$

4. All timers of the system advanced:
$\forall (cl,u) \in Timers(A)$: $St(cl)' := St(cl) + c$ where $c \subseteq N$ and $bcet \le c \le wcet$

5. Non-member Assembly ports remain unchanged:
$(\forall p \in Ports(A)$ where $p \notin C_f)$: $St(p)' := St(p)$

As the Components fire, they read tokens from their input ports and produce tokens on their output ports. For the next firing round, these data tokens need to be delivered to their destinations (which are input ports of components). Thus, in order to define the state update for Assemblies, we need to introduce the notion of *connected output ports:* the set of output ports (or timers) which feed forward into a given input port through a *direct data path*. A direct data path is a contiguous sequence of dataflow connections (with the exception that the first segment can be a TimeTrigger connection if the source is a timer), connecting port and Queue objects, as the data path is routed through the component-assembly hierarchy.

For component input port i within Assembly $\boldsymbol{A}$: $o \in Connected(i)$ **iff**

1) i is a component input: $\exists C_x \in Components(A)$ such that $i \in Inputs(C_x)$,

2) o is a component output port or a Timer: $\exists C_y \in Components(A)$ such that $o \in Outputs(C_y)$ **or** $o \in Timers(A)$, and

3) there exists a direct data path: $\exists P = <(p_1,q_1),(p_2,q_2)\ldots(p_n,q_n)>$ such that:

a) P originates in o and ends in i:

$(p_1 = o, q_n = i)$,

b) the first element is either a Dataflow or a TimeTrigger, and the rest are Dataflows:

$(p_1,q_1) \in (Dataflows(A) \cup TimeTriggers(A))$ and

$\forall$ j: $2 \leq j \leq n$ $(p_j, q_j) \in Dataflows(A)$, and

c) the path is connected:

$p_{j+1} = q_j$

The kernel step of an Assembly consists of two phases

1. ***Delay*** until there are data tokens to be propagated between ports (can be zero)
2. The actual data token ***propagation***, when data tokens from component output ports are propagated to the connected component input ports, and data tokens are generated for expired Timers.

For an Assembly $\boldsymbol{A} = (I, O, As, Co, Cl, Q, Df, Tt)$:
A Kernel step is an *action* $St(A) \rightarrow St(A)'$ consisting of:

1) Delay phase:

a) if there are data tokens to be propagated between output and input ports, or there is an expired timer:

```
if  $\exists (i,o)$ where $i \in Inputs(A)$, $o \in Connected(i)$ and
        $St(o) \neq \emptyset$ or $\exists (c,p) \in Timers(A)$ where $St(c) > p$:
        $d := 0$ (no delay)
else
        delay 1 unit after the first timer $c$ to expire:
        $(c,p) \in Timers(A)$ and $\forall (t,r) \in Timers(A)$: $(t - St(t)) \geq p - St(c)$
        $d := p - St(c) + 1$
endif
```

b) expired timers are reset:

($\forall (u,q) \in Timers(A)$ where $St(u) + d > q$): $St(u)' := 0$

c) all other timers advance by $\boldsymbol{d}$

($\forall (u,q) \in Timers(A)$ where $St(u) + d \leq q$): $St(u)' := St(u) + d$

2) Propagation phase:

a) data tokens from connected output ports are propagated to input ports:

($\forall (i,o)$ where $i \in Inputs(A)$ and $o \in Connected(i)$, $o \in Outputs(A)$ and $St(o) \neq \emptyset$):

$$St(i)' := St(i) \cup \bigcup_{j=1..k} \{(n+j), d_j)\}, \quad \text{where} \quad n = |St(i)|, \quad d_j \in Dt,$$

$$k = |St(o)|, \quad (j, d_j) \in St(o)$$

b) empty and unconnected output ports do not propagate:

($\forall i \in Inputs(A)$ where $\neg\exists o$ such that $o \in Connected(i)$, $o \in Outputs(A)$, $St(o) \neq \emptyset$):

$St(i)' := St(i)$

c) expired timers (that are already reset) insert a data token into the connected input ports:

($\forall i \in Inputs(A)$ where $\exists (c,p) \in Timers(A)$ such that $(c,p) \in Connected(i)$ and $St(c)' = 0$):

$St(i)' := St(i) \cup \{(n+1, d)\}$ where $d \in Dt$ and $n = |St(i)|$

d) all output tokens were propagated, so all output ports are empty
$(\forall\, o\in Outputs(A): St(o)' := \emptyset)$

If data tokens are propagated between input and output ports, the action takes no time.

Execution for an Assembly: an execution is a finite or infinite sequence of alternating Kernel and Firing steps:

$$<St(A)\rightarrow St(A)'|St(A)' \xrightarrow{fire} St(A)''|St(A)''\rightarrow St(A)'''|St(A)''' \xrightarrow{fire} St(A)''''|..>$$

where the initial conditions are:
$(\forall\, p\in Ports(A): St(p)= \emptyset)$, $(\forall\, (c,r)\in Timers(A): St(c)= 0)$

The sequence is initiated with a Kernel action that waits for the first timer to expire. If there are multiple components available for firing, one of them is selected non-deterministically.

The sequence ends if there is no such Kernel action that can enable a Component to run, i.e. when the expiration of any of the Timers (and the corresponding generation of data tokens) is insufficient to enable a component to fire. In most practical systems the firing sequence repeats infinitely.

To reduce complexity, three important, simplifying assumptions were made in the above definitions:

1) The token holding capacity of input/output ports is unbounded. This assumption can be eliminated by modifying the definition of firing/kernel steps to generate/propagate a limited number of data tokens. This extension is fairly straightforward and fits well into the above definition scheme.
2) The token propagation takes zero time. This assumption can be eliminated by trivially modifying the “Delay” part of Kernel step. For faithful modeling, the delay introduced should be a function of *(A,St(A))*, as it depends on both the state and configuration.
3) The semantic definition of queues is implicit through the *Connected()* function: the result also contains the ports connected through queues, and the necessary propagations are performed. To define different queue semantics, the function and the Kernel action needs to be modified.

5 Transforming component models into TA

In order to perform automated model verification, we need to transform the design models into a format accepted by verification tools, such as timed automata (TA)[Alur, 94]. The TA model is another abstraction of the system, where the *assumed* (real-time) properties of the components and the *desired* (real-time) properties of the system are captured. Note that we assume that the timing properties of components are known (and captured in the models), the system composition is known, and the analysis will be used to determine if the desired properties hold for the system. Other system properties (such as the execution of a component method) are abstracted into attributes like best- and worst-case execution times on the given platform. The most significant abstractions were as follows:

- The data content of data-flow tokens was not considered, and only the number of tokens was represented. Therefore, e.g. data-dependent execution times were not considered.
- Each processing step (method invocation) was represented with its worst and best case execution time (*bcet*, *wcet*), and a generalized data token production/consumption scheme was used: methods consume data tokens when they are started, and produce data tokens when they finish.

The transformation applied to the model of the application results in a network of concurrent, strictly synchronized timed automata. There is one automaton for each component, and one automaton for the platform model that coordinates component interactions (referred to as the 'Kernel'). The component TA models are reusable (on the same platform), and the Kernel is unique for each system as it contains direct references to the components of the actual system. The actual runtime semantics including the behaviors for process scheduling, resource handling, concurrency and communication etc. are encoded in the Kernel TA. The Kernel TA model is constructed similarly to the component models: the same quantities (e.g. time) are considered, and expressed in a similar manner. Modeling certain properties (delays, the runtime system's own resource requirements) becomes straightforward, since we use the same apparatus to express those as we used in the component modeling. This way, the Kernel becomes a "super-component", lending itself to the same verification techniques as those applied to the component model.

The details of platform semantics are encoded in the translation algorithm, i.e. they are implicit. For each platform to be modeled, a different translation algorithm has to be devised. In general, the translation algorithm starts from a TA "skeleton" containing default states (e.g. *Start*, *Idle*, etc). Then component and platform-specific states are added to the skeleton, for example, to represent each method invocation. Finally, state transitions are generated, implementing the fine details of the platform: the formulation of transition guards, synchronizers and reset functions takes care of establishing the platform-specific behavior for the resulting network of timed automata.

5.1 Timed Automata in the UPPAAL verification tool

In our examples, the TA are generated for the popular UPPAAL model-verification tool. The concepts mentioned earlier are mapped onto UPPAAL [Larsen, 97] automata structures as follows. For each component and the Kernel a TA template is created. The Kernel controls execution through synchronization *channels*, and global *integer variables* represent the number of tokens on each buffer. *Clock variables* are used to represent time, and transition guards using clock values implement delays like data transmission and method execution delay.

5.2 Implementation of the translation

As mentioned earlier, the platform semantics are encoded into the transformation rules, as each platform requires a different set of rules. We were looking for a language to express these rules of transformation from the system's design language to timed automata.

Both the source and destination domains of the translation use annotated graphs to describe and visualize the models and the automata. Therefore, using a graph transformation (GT) language [Rozenberg, 97] seemed to be a natural choice. Pattern-matching GT languages are formal, high level languages, and they are a good fit for formal model verification: the algorithms described in them lend themselves to formal verification.

In the examples we have worked on, we used the Graph Rewriting and Transformations (GreAT) graph transformation framework [Karsai, 03b][Karsai, 03c], which is a part of the GME modeling toolset [Ledeczi, 01]. It offers a visual language integrated with GME's modeling facilities, and allows the description of graph-rewriting rule-sets based on graph pattern matching.

Using GME, one can build a domain-specific modeling environment by providing a UML meta-model describing the modeling language. GReAT provides a set of tools for graphically defining transformation programs that operate on models (that follow the composition constraints specified by their meta-models). The GReAT interpreter is used to apply graph-rewriting rules (constituting a model transformation program) on the GME models. The result is a newly created model graph that can be converted (trivially) into an XML file, or any other data file. The output model graph must also have a UML meta-model that describes its composition.

6 Verifications performed via TA model checker

Having the system model converted into a TA network enables using a model checker to verify timing and other properties of the system. Note that the specific verification queries one can pose depend on both the platform's model of computation as well as the model checker's capabilities. This means that queries have to be formulated by taking these into consideration.

In general, the following topics are subject of further research: how to express the desired properties and requirements in the high-level modeling apparatus, and how to propagate these expressions through the different abstractions of the system. One practical difficulty is that model checkers typically verify logical expressions, while the designer often wants to deduce quantitative properties (e.g. fastest data rate, smallest memory heap etc). Below, we highlight some typical properties the designer might be interested in and illustrate the way we expressed them in our experiments.

Checking for latency. A typical verification question is as follows: "Is the delay between invocations of a certain method larger than *n* time units?" To answer this question, we have to extend the generated TA by adding a dedicated clock variable and reset it at each invocation. Then, a model checker query can be formulated such as (in CTL [Clarke, 01]) "E◊ (myClock > 5)" meaning "can the clock value ever exceed 5?"

Checking for resource usage and conflicts. In our examples, we annotated each system activity (method) with its *wcet* and *bcet*. In addition to CPU time usage, one can annotate them with other resource requirements, and the Kernel can implement the accounting for that resource. Dynamic memory allocation is a good example: each component TA increments/decrements the size of the dynamic memory pool upon entering/leaving states corresponding to method invocations. Using a model checker, the designer can verify the system's memory requirements.

Bounding the number of tokens on dataflow links. In our examples we use a dataflow oriented runtime platform. Many system properties (such as deadlock) can be verified by formulating queries on the number of "tokens" on the dataflow links (or queues). If the number of tokens grows without bounds (e.g. it exceeds a large enough constant), the system produces more tokens than it consumes. Another example might be as follows: if the number of tokens on a link driven by a periodic timer ever exceeds one, then the destination component is not able to process the timer token on time and the system could have missed a timer tick.

Checking schedulability. Although the definition of schedulability is general, non-schedulability can manifest itself on different platforms and systems in different ways. Fortunately, the question can usually be formulated within the model checker as a straightforward Boolean condition that verifies that all tasks meet their deadlines.

7 Transforming component models into TA: Illustrative examples

We illustrate the discussed method on two simple models: both models are created in the SMOLES (**S**imple **Mo**deling **L**anguage for **E**mbedded **S**ystems) language environment, and translated into UPPAAL timed automata. The runtime platform modeled by the transformation is a non-preemptive token-passing dataflow-oriented execution environment, referred to as DFK (**D**ata**F**low **K**ernel) in the following discussion. The first, very simple system will help understand the key ideas behind the transformation, and the second one demonstrates a simple real-life system implemented and built using the SMOLES and DFK framework.

7.1 Implementation of SMOLES: Modeling, generators, and execution platform

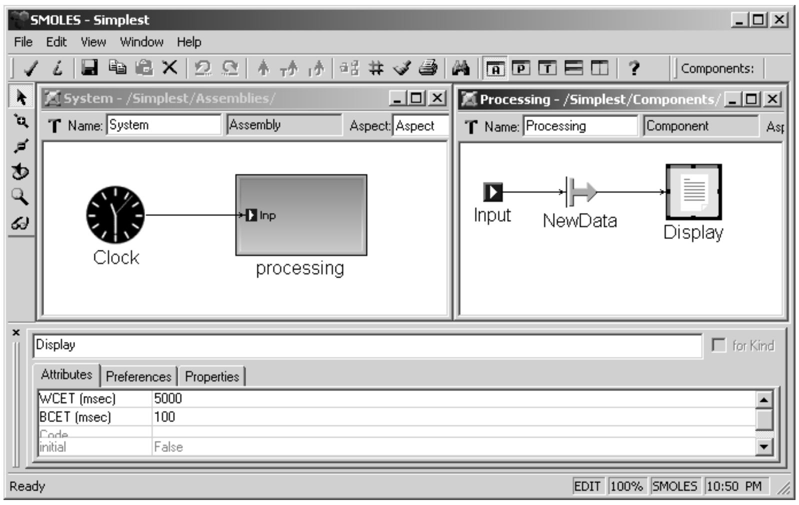
SMOLES was implemented in the GME modeling environment. It is based on token-passing dataflow semantics, and multiple generators were developed to generate code implementing SMOLES models for various platforms (C++ and Java-based DFK). The generated code provides the data (and control) flow mechanisms and structures, and implements the components specified in the SMOLES model. The examples provided here were synthesized for the DFK platform, which is implemented in C++, using OO concepts.

C++ code generation from SMOLES models is done by deriving the component (actor) objects from the predefined DFK abstract objects; then extending them with the user-specified code for the methods in the model, and generating the list of required dataflow connection objects.

On the visual SMOLES models input ports are on the left, output ports are on the right, oversized arrows represent triggers and notebook icons stand for user-supplied methods that execute procedural code. User functions without connections are initializers (run on the first execution only). Simple lines with arrows represent dataflow links.

On UPPAAL timed automata diagram, circles represent states (locations), possibly annotated by their time invariant condition. Arrows stand for transitions, with the guard statements written above the arrows and the assignment/update statements below the arrows. In expressions *?* and *!* stand for the synchronization channel operations in the usual CSP manner: read and write, respectively.

7.2 Example 1: A simple SMOLES system



Name: System | Assembly | Aspect: Aspect

Name: Processing | Component

Display

WCET (msec)	5000
BCET (msec)	100
Code	
initial	False

Ready EDIT 100% SMOLES 10:50 PM

Figure 2: A simple SMOLES system in GME

This is one of the simplest working systems: it consists of a periodic *Timer* and a *Component* containing a single method *Display*. The *Clock* generates data tokens periodically and those are delivered to the Component's *Input* port. This *triggers* the execution of the *Display* method (with user-specified code). The figure shows the component interaction diagram (*Clock*, and *Processing* components) and the internals of the Processing component. Here, the *Display* method is connected to the *Input* port through the *NewData* trigger.

For this model, the resulting UPPAAL automata are simple enough to present them in detail. The global variable declarations are shown below.

```
const nProcs 1;   // number of processes in the system
const IdleTick 10;      // Kernel idle sleep duration
const DefaultChanSize 3;      // dataflow buffer size

const ClockPeriod 10;   // Clock period
const processingPID 0;  // PID 'processing' process

// no. of tokens on the inter-component dataflow link
int[0, DefaultChanSize] ClockOut_processingInput:=0;

int[-1, nProcs-1] running:=-1;      // active process
int[0,nProcs] nInitializedProcs:=0; //

clock WallClock;        // global time
clock ClockClk;         // timer clock period
chan processingRun;     // sync for context switch
```

The number of processes is one, because in this environment the *Clock* is not a separate entity: it is implemented internally in the Kernel. The following figure shows the generated TA for the *Processing* component:

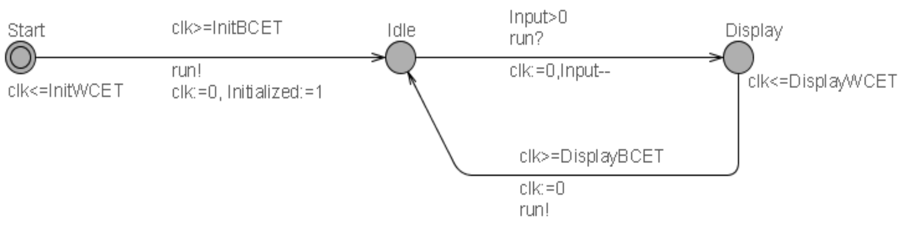


Figure 3: Generated UPPAAL TA for the 'Processing' component

The lifecycle of the component is as follows: It stays in state *Start* for at least *InitBCET* a most *InitWCET* time, then, it proceeds to *Idle*, synchronized with the Kernel's scheduler. In Idle, it waits until there's data on port Input (Input > 0), and the scheduler has released the component (run!). Then, it executes Display method, reading a data token from the input port (Input--). The parameter variable Input is a reference to the global *ClockOut_processingInput* integer, representing the number of data tokens on the inter-component dataflow links.

The next figure shows the TA generated for the Kernel:

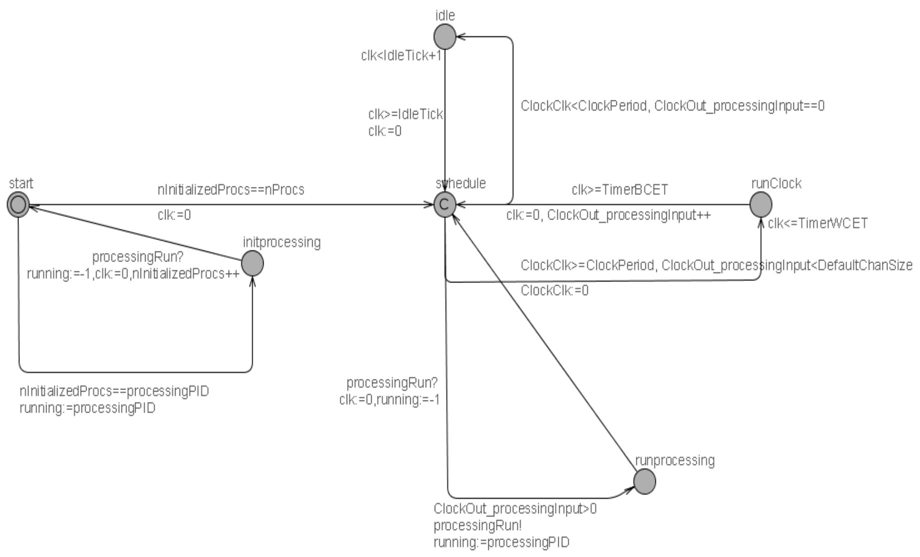

idle
clk<IdleTick+1
clk>=IdleTick
clk:=0
ClockClk<ClockPeriod, ClockOut_processingInput==0
start
nInitializedProcs==nProcs
clk:=0
schedule
C
clk>=TimerBCET
clk:=0, ClockOut_processingInput++
runClock
clk<=TimerWCET
initprocessing
processingRun?
running:=-1,clk:=0,nInitializedProcs++
ClockClk>=ClockPeriod, ClockOut_processingInput<DefaultChanSize
ClockClk:=0
nInitializedProcs==processingPID
running:=processingPID
processingRun?
clk:=0,running:=-1
runprocessing
ClockOut_processingInput>0
processingRun!
running:=processingPID

Figure 4: Generated UPPAAL TA for the 'Simplest' kernel

To understand the automaton, we have to recall that *ClockOut_processingInput* is the same as *Input* in the previous component, and *processingRun* is the same as *run* for the component.

After finishing initialization, the Kernel TA proceeds to state *schedule*, where it chooses from three possible transitions:

- If there is data on the dataflow link (*ClockOut_processingInput > 0*), then it will schedule the component to run
- If the clock's period has expired, and there's available buffer capacity for the token (*ClockClk >= ClockPeriod and ClockOut_processingInput < DefChanSize*), then one token will be placed on the buffer, and the *ClockClk* reset
- Otherwise it goes to idle where it spends exactly *IdleTick* amount of time

If the conditions for both of the first two choices are satisfied, then the scheduler chooses one non-deterministically. This is a crucial point for the accuracy of the platform model, as it faithfully models an important property of the actual runtime system. The designer (driven by a false intuition) might assume that the Clock has higher priority (since it is internal to the Kernel) than the Component. On this actual platform this is not true, and using verification technique such errors could be identified.

7.3 Example 2: Visual feedback camera positioning system

Our next example is a simple 3-component system: a pan and tilt capable camera tracks a moving object.

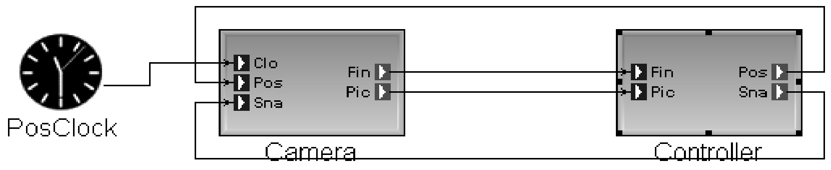


Figure 5: Components of the camera positioning system

The Camera component is a wrapper for the camera API: it is capable of moving the camera head (*Positioning*), and signal the completion of the positioning (*FinishedPos*). It can also take *SnapShots*, even while *Positioning* is in progress. The Controller inputs a picture, and outputs positioning data (coordinates). The Camera translates the coordinates into step commands (for the pan and tilt step motors), whose timing is important and driven by the periodic PosClock.

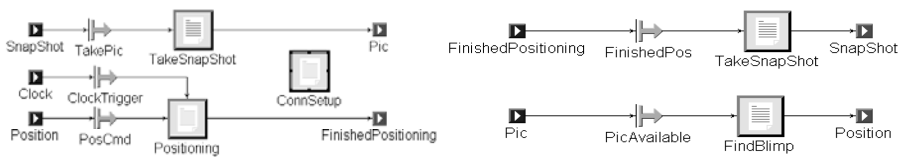


Figure 6: Component models of the camera positioning system

The following figures show the generated UPPAAL TA for the components.

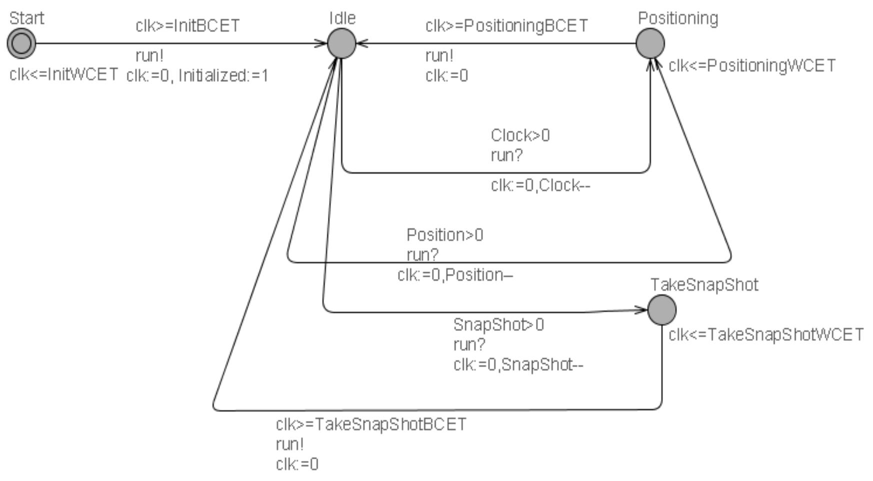


Figure 8: Generated UPPAAL TA for the Camera component

On the Camera TA it's worth observing how the two alternative triggers for Positioning method are translated into two alternative translations from Idle to Positioning: the guard is formulated using the respective port name and the input token is read from there.

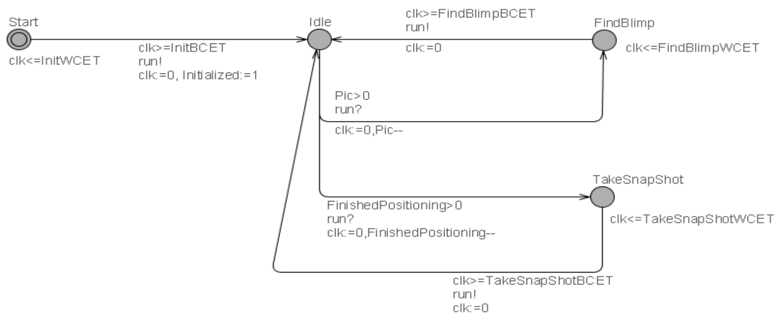


Figure 9: Generated UPPAAL TA for the Controller component

The relation of his TA to the corresponding component above is straightforward. The figure below shows why it is important to perform the translation by computer even for such small and simple systems:

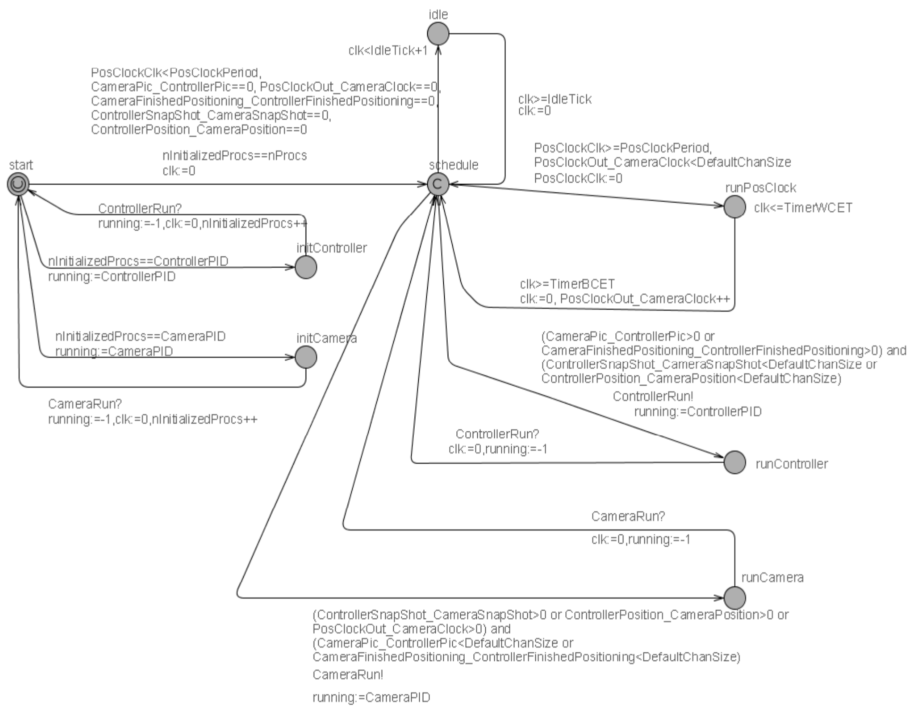

idle
clk<IdleTick+1
PosClockClk<PosClockPeriod,
CameraPic_ControllerPic==0, PosClockOut_CameraClock==0,
CameraFinishedPositioning_ControllerFinishedPositioning==0,
ControllerSnapShot_CameraSnapShot==0,
ControllerPosition_CameraPosition==0
clk>=IdleTick
clk:=0
start
nInitializedProcs==nProcs
clk:=0
schedule
PosClockClk>=PosClockPeriod,
PosClockOut_CameraClock<DefaultChanSize
PosClockClk:=0
runPosClock
clk<=TimerWCET
ControllerRun?
running:=-1,clk:=0,nInitializedProcs++
initController
nInitializedProcs==ControllerPID
running:=ControllerPID
clk>=TimerBCET
clk:=0, PosClockOut_CameraClock++
nInitializedProcs==CameraPID
running:=CameraPID
initCamera
(CameraPic_ControllerPic>0 or
CameraFinishedPositioning_ControllerFinishedPositioning>0) and
(ControllerSnapShot_CameraSnapShot<DefaultChanSize or
ControllerPosition_CameraPosition<DefaultChanSize)
ControllerRun!
running:=ControllerPID
CameraRun?
running:=-1,clk:=0,nInitializedProcs++
ControllerRun?
clk:=0,running:=-1
runController
CameraRun?
clk:=0,running:=-1
runCamera
(ControllerSnapShot_CameraSnapShot>0 or ControllerPosition_CameraPosition>0 or
PosClockOut_CameraClock>0) and
(CameraPic_ControllerPic<DefaultChanSize or
CameraFinishedPositioning_ControllerFinishedPositioning<DefaultChanSize)
CameraRun!
running:=CameraPID

Figure 10: Generated TA for the Kernel in the camera positioning system

The Kernel's automaton is quite complex, especially the transition guard conditions for the scheduler. One observation: the Kernel makes scheduling decisions on the component level: it will schedule a component to run in any "input available, output possible" situation, regardless of the components' internal trigger configuration (i.e. even if none of the methods inside could run). This is in compliance with the dataflow platform implementation being modeled. The runtime environment cannot see inside the components, and allows very simple component trigger conditions that are faithfully modeled in the above TA.

8 Performing the transformation from SMOLES to UPPAAL using GReAT

As mentioned earlier, we perform the transition using a visual graph transformation (GT) language: GReAT. It is integrated with the GME toolset, and allows creation of sequenced translation rules that operate based on pattern matching. The GT rules are expressed using elements of the two meta-models: the meta-model of SMOLES, and a TA meta-model corresponding to the abstract syntax of the UPPAAL language. An auxiliary tool was developed to convert the resulting models (in UPPAAL meta-model format) into UPPAAL's native XML format. This tool also generates an appealing visual layout for the resulting UPPAAL models.

Below, we give a short overview of the translation algorithm, and explain a few selected rule-blocks in detail. The translation rule-set consists of the following steps:

1) Start with locating the top-level Assembly and create the corresponding NTA
2) *Handle Templates*: Match all the Components within the Assembly, and create UPPAAL templates
    a) Create the *Template* skeleton with the default states (Start, Idle)
    b) Add *Method* states, triggers and output transitions based on the component
3) *Handle Processes*: Enumerate al SMOLES components and create UPPAAL global declarations: PIDs, Timer clocks, synchronization channels
4) *Handle Channels*: Find all timer-triggering and dataflow connections, create UPPAAL counterparts
5) *Handle Kernel*: create and populate the kernel TA
    a) Create skeleton (*start, schedule, idle*), compile parameter list
    b) Create and connect states corresponding to Timers
    c) Create component invocation states and transition, generate scheduling guards
6) *Finalize*: generate process instantiation code for the components and Kernel

For an illustrative example of a translation rule, let us consider the formulation of guard conditions in TA templates according to trigger conditions as shown on Figure 11. This is called *enumTriggers*, and it is part of the *Handle Templates* step from the algorithm above.

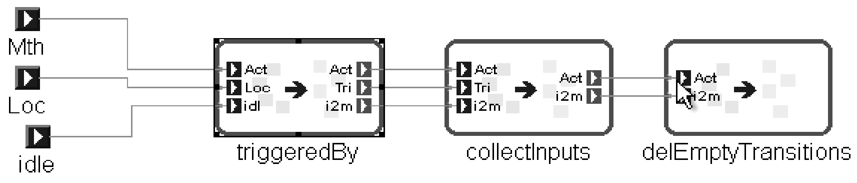
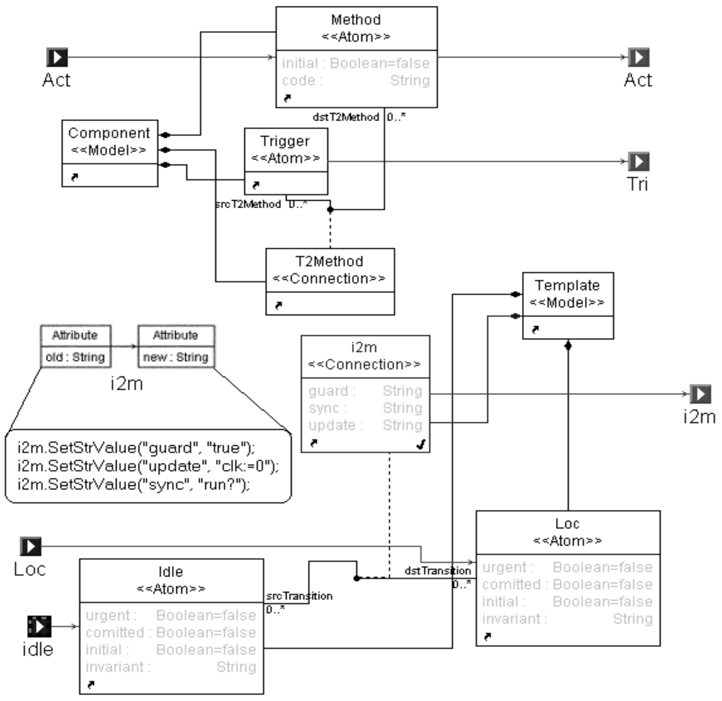


Figure 11: GReAT rule block: enumTriggers

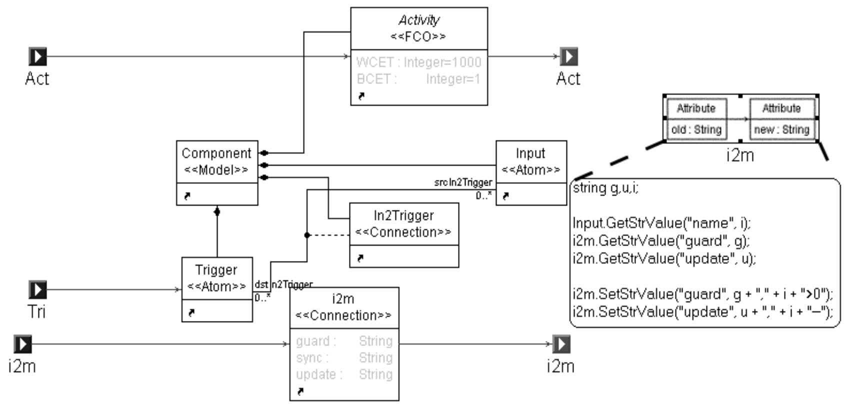
The rule block gets a SMOLES Method and two TA locations as input (shown on the left): the location corresponding to the method’s invocation and the default *idle* location. The TA and the locations were created by the preceding rules.



```
i2m.SetStrValue("guard", "true");
i2m.SetStrValue("update", "clk:=0");
i2m.SetStrValue("sync", "run?");
```

Figure 12: GReAT rule 'triggeredBy'

The first rule of the chain matches the method, and finds a matching Trigger→Method connection within the component (at the top of Figure 12). The two Locations are also matched, and a new Transition (*i2m*) is created within the TA template. The newly created object is indicated with a small “tick” mark in the lower right corner of the icon. After the rule has successfully matched and the target objects have been created, procedural *attribute mapping* code is executed that can modify the created objects. Here, the attributes of the newly created Transition are set by the code captured in the “attribute mapping” box (*i2m*, with code shown in the exploded view) that initializes the “guard” attribute to true. This rule will match on each Trigger→Method condition (where the method is given) and will create a transition for each. The rule propagates all matching Trigger and corresponding *i2m* transitions connections to the following rule(s).



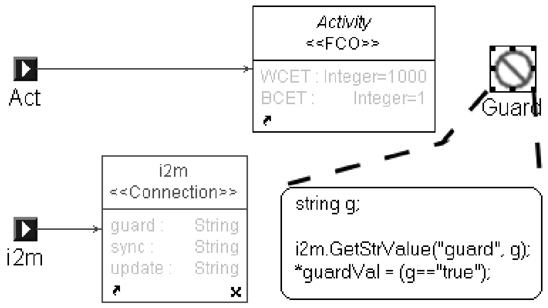
```
string g,u,i;

Input.GetStrValue("name", i);
i2m.GetStrValue("guard", g);
i2m.GetStrValue("update", u);

i2m.SetStrValue("guard", g + "," + i + ">0");
i2m.SetStrValue("update", u + "," + i + "--");
```

Figure 13: GReAT rule 'collectInputs'

Figure 13 will match the Method (*Activity*), the Trigger matched by the previous rule, and the *i2m* connection just created, and it will find all Input ports in the SMOLES component, which are connected to the Trigger. The *guard* and *update* statements of the transition are updated: a condition is added to the guard specifying that more than zero tokens have to be available on the port matched by *Input*, and the update will express the consumption of a token. This rule propagates the Method and the connection.



```
string g;

i2m.GetStrValue("guard", g);
*guardVal = (g=="true");
```

Figure 14: GReAT rule 'delEmptyTransitions'

The following, terminal rule (Figure 14) features a GreAT *guard* code block: this rule is executed only if the graph elements match *and* the guard condition evaluates true. Note that this "guard" is evaluated when the transformation is executed and it is coincidental that the transformed models also have a concept called "guard". In this case, the GreAT guard condition looks at the value of the "guard" attribute of the *i2m* object (which is the transition's UPPAAL guard). If that value is true, then the *collectInputs* rule failed to match any Input ports to the trigger (so the transition's guard remained unchanged from rule *triggeredBy*) therefore this transition doesn't represent a valid method invocation. In this case it is deleted from the target graph. The delete operation is indicated by the small "x" in the icon's lower left corner.

Figure 15 shows how rule *triggeredBy* is applied when the Camera component is being translated:

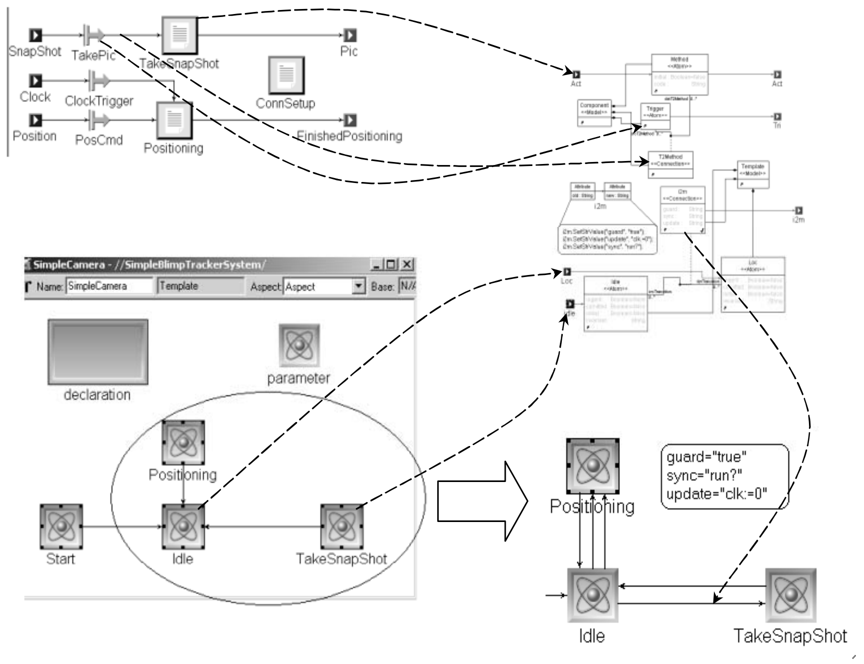


Figure 15: GReAT rule application example

At top left the SMOLES component can be seen. Below is the corresponding UPPAAL TA being built: the previous rules created the states and other elements, but not all the transitions are specified. The arrows show one possible match and indicate the transition generated. In this case there are 3 possible matches since method *Positioning* has two triggers connected. As the resulting (RHS) graph shows, this rule generates three new transitions. At this point they all have the same attribute set (shown bottom left). The next rule (*collectInputs*) will customize them according to the input ports connected.

9 Results and Conclusions

In this paper we argued for the modeling of the execution platform of embedded systems, showed how (the knowledge of) platform models could be used to transform embedded system application models into models that could be subjected to formal verification through model checking, and provided details about implementing the translation algorithm itself. We have used the approach to verify some properties of small systems, but we believe further testing and work is necessary.

There are at least two important research directions we need to consider in the future. One is about making the platform models explicit. In the current system platform models are implicit in the translation algorithm, and for every new platform (semantics) a new algorithm has to be developed. Obviously there is a need for making the platform models explicit such that the translation could be retargeted to different platforms easily.

The other research direction of interest relates to scaling. Even simple problems produced sizeable TA-s, which indicates that the straightforward translation might not be the best approach for systems with hundreds or thousands of components. We believe this problem should be attacked from two sides: (1) by building more scaleable verification tools, and (2) by using clever techniques in the translation process to reduce the complexity of the resulting models. However, both of these directions necessitate further research.

Acknowledgement

The NSF ITR on "Foundations of Hybrid and Embedded Software Systems" has supported, in part, the activities described in this paper. The effort was also sponsored by DARPA, Air Force Research Laboratory, USAF, under agreement number F30602-00-1-0580.The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon. The views and conclusions contained therein are those of authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of the DARPA, the AFRL or the US Government.

References

[Karsai, 03a] Model-integrated development of embedded software, Karsai, G.; Sztipanovits, J.; Ledeczi, A.; Bapty, T.; Proceedings of the IEEE, Volume: 91, Issue: 1 , Jan. 2003 Pages:145 – 164

[Larsen, 97] Uppaal in a Nutshell. Kim G. Larsen, Paul Pettersson and Wang Yi. In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.

[Amnell, 02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - A Tool for Modelling and Implementation of Embedded Systems. In proceedings of 8th International Conference, TACAS 2002, part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 (Grenoble, France, April 8-12, 2002), pages 460-464, Springer-Verlag, 2002. Lecture Notes in Computer Science, Vol.2280.

[Gu, 03] Zonghua Gu, Shige Wang, Sharath Kodase, and Kang G. Shin, An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS'03)

[Sangiovanni-Vincentelli, 01] Sangiovanni-Vincentelli, A.; Martin, G.: Platform-based design and software design methodology for embedded systems, Design & Test of Computers, IEEE , Volume: 18 , Issue: 6 , Nov.-Dec. 2001 Pages:23 - 33

[Lee, 97] E. A. Lee and A. Sangiovanni-Vincentelli, ""A Denotational Framework for Comparing Models of Computation," ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997.

[MDA] Model-Driven Architecture, available online at www.omg.org/mda .

[RTCORBA] Real-time CORBA specification documents, available online from http://realtime.omg.org/rfp/real-time_realtime_corba_1_0.html

[Alur, 94] R. Alur and D. Dill: “Automata for modeling Real-Time Systems.” Theoretical Computer Science, 126(2):183-236, April 1994

[Rozenberg, 97] G. Rozenberg, “Handbook of Graph Grammars and Computing by Graph Transformation”, World Scientific Publishing Co. Pte. Ltd., 1997.

[Karsai, 03b] Karsai, G., Agarwal, A., Shi, F., Sprinkle, J: On the Use of Graph Transformation in the Formal Specification of Model Interpreters,. Journal of Universal Computer Science, Volume 9, Issue 11, 2003.

[Karsai, 03c] Agrawal, G. Karsai, F. Shi: “Graph Transformations on Domain-Specific Models”, Technical Report, available online at http://www.isis.vanderbilt.edu

[Ledeczi, 01] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G.: “Composing Domain-Specific Design Environments”, Computer, pp. 44-51, November, 2001.

[Clarke, 01] E.Clarke et. al: Model Checking, MIT Press 2001.