Rapport

June 3, 2016

Contents

Abstract

1 Introduction

The wide application of Internet of Things [?] raises the complexity of embedded systems today rapidly. Model-Driven Engineering (MDE) is recognized as an efficient means to dealing with the complexity. MDE [9] promotes abstraction and automation. The latter often relies on chaining transformations from source models at high level abstraction to target models and finally to code. Those two techniques are identified as model to model (M2M) and model to code (M2C) transformations. Among modeling languages, Unified Modeling Language (UML) is the most widely used. UML and its diagrams are much more useful to design such systems than any other text-based languages. Especially, in embedded system domains such as vehicle controlling, UML State Machine, extended from computational state model, is used as a powerful means to describing the dynamic behavior of such complex systems.

Ideally, a full model-centric approach is preferred by the MDE community due to its advantages [?]. However, in industrial practice, there is significant reticence [?] to adopt it. The reticence is due in part to the perceived gap [1] between diagram-based languages and textual languages. On one hand, programmers prefer to use the more familiar textual programming language. On the other hand, software architects, working at higher levels of abstraction, tend to favor the use of models, and therefore prefer graphical languages for describing the architecture of the system (see Fig. 1).

The survey described in [5] polled stakeholders in companies who use MDE approaches. The survey reveals that UML is the prominent and dominating language. 85 % of the respondents used UML for various purposes, especially problem understanding and automation such as code generation (88,2%). It notes that 70% of the respondents primarily work with models, but still require manually-written code to be integrated. Furthermore, 35% of the respondents answered that they spend a lot of time and effort synchronizing model and code. The latter is critically required by systems today. Besides the synchronization aspect, efficiency of code generated from models is a concern for the respondents.

The code modified by programmers and the model are then inconsistent. Round-trip engineering (RTE) [4] is proposed to synchronize different software artifacts, model and code in this case [10]. RTE enables actors (software architect and programmers) to freely move between different representations [10] and stay efficient with their favorite working environment.

The back-and-forth switching between diagram-based and textual languages, as well as between model and code, has hindered the adoption of MDE in industrial practice. To solve this issue, we believe the sharp distinction between model and code must be blurred. We feel that a collaborative solution must be offered to different categories of developers (e.g. architects,

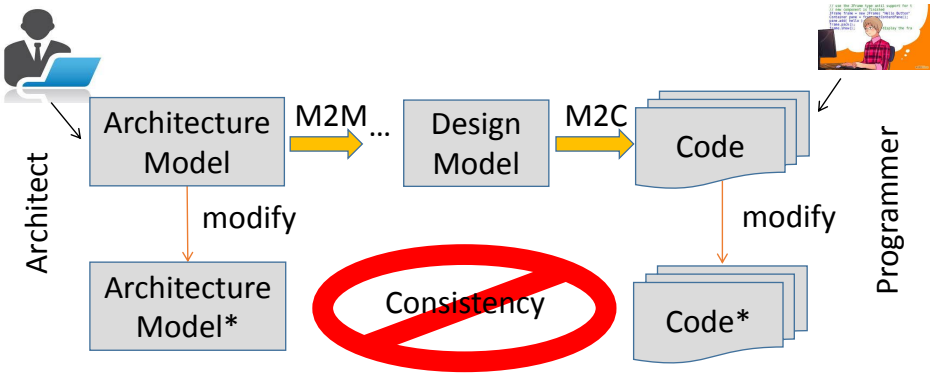

Figure 1: Concurrent modifications made to different artifacts

programmers), who use different development practices.

Collaboration between developers producing different types of artifacts, in different languages, using different tools, raises the issue of artifact synchronization. This is a well-known concern with round-trip engineering [11]. It is related to traditional software engineering disciplines such as forward and reverse engineering [2].

Round-trip engineering achieves synchronization between related artifacts that may evolve concurrently by incrementally updating each artifact to reflect editions made in the other artifacts.

Usually, a direct synchronizing between the architecture model and the code is hard because of the large abstraction gap. The synchronization is therefore decoupled into several synchronizations of artifacts participating to one of the transformation phases of the chain.

To sum up, the thesis can be divided into two related major parts as followings:

1.1 Major topics

- **Artifact synchronization**: As previously discussed, this is critical for step-by-step integrating MDE into current software development practices and collaborating these with MDE methodologies. Particularly, synchronization of artifacts is realized by means of incremental model-to-model and model-to-text transformations. Architecture models are often at higher level abstraction than low level implementation code. Mappings from the architecture are usually not one-to-one. This raises the difficulty in mapping code-side elements back to architecture model-side elements. The existing synchronizations mainly focus on static parts while the dynamics are missing.

- **Improving code generation solutions and towards synchronization**: UML is widely used in industry, especially the use of State Machine for reactive, real-time embedded systems. However, on the one hand, OMG seeks to raise the usefulness of UML State Machine by providing more modeling concepts supporting software architects for modeling and designing complex systems. On the other hand, the support of existing generation approaches, over the years of research and development, is still limited to simple cases, especially when considering concurrency of *doActivity* and orthogonal regions, pseudo states such as history, and different events. This again enlarges the gap between the UML State Machine semantics and the actual generated code. Furthermore, the synchronization between UML State Machine and generated code is not supported by any other approaches, even though it is critical.

1.2 Objective

The transformation from architecture models to code directly involves chaining model-to-model and model-to-text transformations. Therefore, to synchronize the architecture models and code, the following objectives need to be done:

- A generic methodology, which synchronizes two different system artifacts (model and code). This method will be used in different specific synchronization cases developed in the thesis deployment.
- In order for wider integrating MDE into software development today, one task, among other important tasks, is needed to seamlessly bring the theory specified by the UML specification and the Precise Semantics of UML Composite Structures (PSCS) [8] into practice of complex software development with respect to the semantics as much as possible. To do it, the objective of this part is to provide a complete generation solution, especially considering the concurrency supporting for real-time systems. From the solution, a specific methodology combined with the generic one as above is also desired to provide one more step for integrating MDE into industry.

1.3 Contributions

Contributions of this report are followings:

- An extensible generic methodology, which synchronize two different system artifacts (model and code).

- A complete generation solution from UML State Machine to code with respect to the Precise Semantics State Machine (PSSM).
- A specific methodology derived from the generic one supporting the synchronization of UML State Machine and code.

1.4 Structure of the report

2 A generic model-code synchronization pattern

2.1 Preliminary definition

In this section we define the actors who will use our model-code synchronization approach to collaborate during development. Some basic concepts related to the actors and use-cases are also defined in this section.

For the sake of generality, we postulate that the architect and programmer are actors with starkly opposite development practices. This allows the approach to be used even in cases where model and code can both be used for the full implementation of a system, rather than just architectural design for the former, and code implementation for the latter. First, we introduce the concepts of *development artifact* and *baseline artifact*.

Definition 2.1 (Development artifact)**.** A development artifact is a artifact, as defined in [7], that can be used for the full implementation of the system.

In our work, we assume that model and code are both development artifacts. A development artifact may be the baseline artifact, defined in this paper as follows:

Definition 2.2 (Baseline artifact)**.** A baseline artifact is one which may be edited manually. All other artifacts are produced from the baseline artifact through some process, and only through a process. Manual edition of artifacts other than the baseline artifact is forbidden.

Two primary actors, called *model-driven developer* and *code-driven developer*, are introduced.

Definition 2.3 (Model-driven developer)**.** A model-driven developer is an actor in a software development process for whom the baseline artifact is the model.

The code must always be produced from the model automatically by some process that guarantees that the code is consistent with the model. A software architect is a kind of the model-driven developer who edits the model to specify the architecture of the system.

Definition 2.4 (Code-driven developer)**.** Code-driven developer is an actor in a software development process for whom the code is the baseline artifact.

A programmer is a specialization of the code-driven developer. Indeed, programmers may modify the code, such as editing method bodies.

There are some use-cases for manual edition of artifacts. The `Edit Artifact` use-case implies that the IDE must have some tool to let the developer manually edit an artifact. The `Edit Model` and `Edit Code` use-cases are specializations of the `Edit Artifact` use-case where the artifact is the model or code.

There are also some use-cases related to the synchronization of artifacts. The `Synchronize Artifact` use-case is the synchronization of two artifacts by: (1) comparing them, (2) updating each artifact with editions made in the related artifact, and (3) reconciling conflicts when appropriate. The `Synchronize Model` and `Synchronize Code` use-cases are specializations where, respectively, the model or the code are the artifacts being synchronized.

The `Generate Code` use-case is related to forward engineering. It is the production of code in a programming language from a model. The developer can either use `Generate Code (Batch)` or `Generate Code (Incremental)`.

Definition 2.5 (Batch code generation [3])**.** Batch code generation is a process of generating code from a model, from scratch. Any existing code is overwritten by the newly generated code.

Incremental code generation is a specialization of incremental model transformation, which is defined in [3] as model transformation that does not generate the whole target model from scratch but only updates the target model by propagating editions made in the source model.

Incremental code generation is defined in this paper as follows:

Definition 2.6 (Incremental code generation)**.** In-cremental code generation is the process of taking as input an edited model, and existing code, and then updating the code by propagating editions in the model to the code.

Finally, the `Reverse Code` use-case is related to reverse engineering. `Reverse Code` is the production of a model, in a modeling language, from code, written in a programming language. The developer can either use `Reverse Code (Batch)` or `Reverse Code (Incremental)`, which are defined in this paper as follows:

Definition 2.7 (Batch reverse engineering)**.** Batch reverse engineering is a process of producing a model from code, from scratch. The existing model is overwritten by the newly produced model.

Definition 2.8 (Incremental reverse engineering)**.** Incremental reverse engineering is the process of taking as input a edited code, and an existing model, and then updating the model by propagating editions in the code to the model.

In the next section, the use-cases of the IDE are integrated into a process that covers model-code synchronization.

2.2 Processes to synchronizing model and code

We propose two synchronization strategies for this scenario. The general approach behind our strategies is to represent one artifact in the language of its corresponding other artifact. These two can then be compared. For this, we define a concept of a *synchronization artifact*:

Definition 2.9 (Synchronization artifact)**.** An artifact used to synchronize a model and its corresponding code is called a synchronization artifact. It is an image of one of the artifacts, either the model or the code. In this context, an image I of an artifact A is a copy of A obtained by transforming A to I. A and I are semantically equivalent but are specified in different languages.

For example, a synchronization artifact can be code that was generated from the edited model in batch mode. In that case, it is code that represents an image of the edited model.

Using the concept of synchronization artifact, two strategies are proposed in this paper: one in which the synchronization artifact is code, and the other in which the synchronization artifact is a model. We propose two strategies so the developer can choose to either use the `Synchronize Code` or `Synchronize Model` use-cases of the IDE. The choice may be determined by preferred development practices or the availability of suitable tools (e.g. the programmer may prefer to synchronize two artifacts, both represented in the same programming language, because he prefers to work exclusively with code).

Figure 2 shows the first synchronization strategy based on using code as the synchronization artifact. The general steps of the process shown in Figure 2 are described as follows:

Step 1 Both the model and code may be edited concurrently. (To simplify Figure 2, we don't show the Read and Write interactions for this step.) After both artifacts have been edited concurrently, we need to synchronize them.

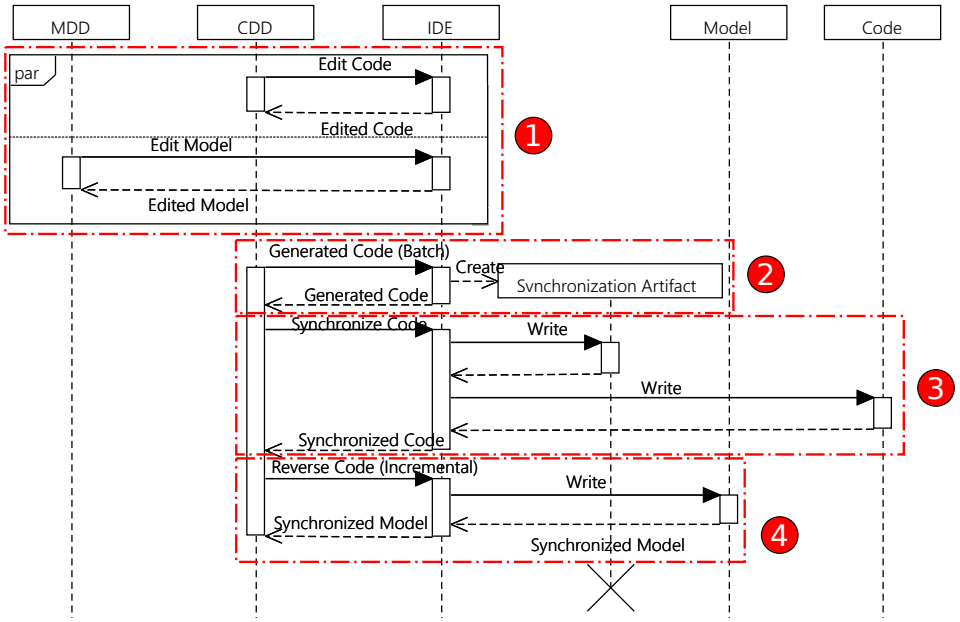

Figure 2: Synchronization process, in which the model and the code are concurrently edited with code as the synchronization artifact (CDD = Code-Driven Developer, MDD = Model-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

Step 2 First we create a synchronization artifact from the edited model by generating code in batch mode. This synchronization artifact is code and it is an image of the edited model.

Step 3 The synchronization artifact is synchronized with the edited code. Since the synchronization artifact is code itself, this step is done with the `Synchronize Code` use-case of the IDE.

Step 4 Once synchronization artifact and edited code are synchronized, the former is reversed incrementally to update the edited model.

The second strategy, based on using model as the synchronization artifact, is the opposite of the first strategy. In the second strategy, the synchronization artifact is obtained by reversing the edited code in batch mode. Afterwards the synchronization artifact is synchronized with the edited model. Finally, we generate code incrementally from the synchronization artifact to update the edited code.

We propose two strategies based on the preferences of the developers. They may even use both strategies, successively, as a kind of hybrid strategy. This may be useful when developers want to synchronize parts of the system using one strategy, and other parts using the other strategy.

3 A complete code generation solution for UML State Machine

3.1 Background definition

Definition 3.1. A directed graph $G = \{V, Ed\}$ consists of a finite set V of vertexes, and a set Ed of edges. An edge connects a source vertex to a target vertex. The source and target vertexes of an edge ed are obtained by $src(ed)$ and $tgt(ed)$.

Definition 3.2. A UML vertex $v \in V$ has a kind $v.kind \in$ *{initial, final, state, comp, conc, join, fork, choice, junction, enpoint, expoint, history}*. Each vertex v has a name and we write $v.name$.

Definition 3.3. A region $r \in \mathcal{R}$ is composed of one or several vertexes, and contained by a state s. We write $ctner(r) = s$ and $subvts(r)$ is its sub-vertices set.

Definition 3.4. A UML state s is a vertex v where v.kind $\in$ state, comp, conc. s has an *entry*, an *exit* and a *doActivity* action. A composite state cs contains one or more vertexes. We write $subvts(cs)$ is a set of vertexes contained by cs and $ctner(v)$ refers to the containing state of the vertex v. A concurrent state contains more than one region.

Definition 3.5. An action $act \in ActLang$ is a set of statements written in an object-oriented programming language $ActLang$. A guard is a boolean expression written in $ActLang$.

Definition 3.6. A transition $t \in T$ is an edge connecting two vertexes. A transition has a guard $guard(t)$, an effect $effect(t)$, and is associated with a set of events $\subset$ E. We write $events(t)$ as the associated set of events. A transition has a type $t.type \in \{trigger, triggerless, guardless\}$ and a kind $t.kind \in external, local, internal$.

Definition 3.7. An event is one of the followings:

- A *TimeEvent te* specifies the time of occurrence d relative to a starting time. The latter is specified when a state, which accepts the time event, is entered.
- A *SignalEvent se* is associated with a signal sig, whose data are described by its attributes and is occurred if sig is received by a component, which is an active UML class.
- A *ChangeEvent che* is associated with a boolean expression $ex(che)$ written in $ActLang$. che is emitted if $ex(che)$ changes from true (false) to false (true).
- A *CallEvent ce* is associated with an operation $op(ce)$. ce is emitted if there is a call to $op(ce)$.

Suppose that for each vertex $v \in V$, its incoming and outgoing transition lists are extracted by the functions $incomings$ and $outgoings$, respectively. For a list l, the function $head$ is used to get the first element of the list. If $v.kind = conc$, suppose $regions(v)$ is the region set contained by v. Given a transition t:

- $t.type = trig$ if $\#events(t) > 0$.
- $t.type = tless$ if $\#events(t) = 0$.
- $t.type = gdless$ if $(guard(t) = true \vee guard(t)$.
- $t.type = triggdless$ if $\#events(t) = 0 \wedge (guard(t) = true \vee guard(t))$.

The behavior of an active class C is described by using a state machine whose definition is as following:

Definition 3.8. A state machine sm is a graph specified by $\{V, T\}$ associated with a set of events E. A state machine is a special composite state which has no incoming and no outgoing transitions. A root vertex v is a direct subvertex of the state machine, $ctner(v) = sm$. The set of regions contained by sm is written $\mathcal{R}$.

Table 1: State machine constraints

- If $v.kind = initial$ then $\#T_{outs}(v) = 1 \wedge \#T_{ins}(v) = 0 \wedge t_{first}.type = triggdless$.
- If $v.kind = final$ then $\#T_{outs}(v) = 0$.
- If $v.kind \notin \{state, comp, conc\}$ then $\forall t \in T_{outs}(v) : src(t) \neg = tgt(t)$.
- If $T_{auto} = \{t \in T_{outs} | \#events(t) = 0\}$, $T_{ng} = \{t \in T_{auto} | guard(t) = true \vee guard(t)\}$ then $\#T_{ng} <$
- $\#T^{+}_{ins}(v) > 0 \vee \#T_{outs}(v)^{+} > 0$.
- If $v.kind = comp$ then $\#subvertexes(v) > 0$.
- If $v.kind = conc$ then $\#regions(v) > 0 \wedge (\forall r \in regions(v) : \#subvertexes(r) > 0)$.
- $\#regions(sm) = 1$.
- If $v.kind = fork$ then $\#T_{ins}(v) > 0 \wedge \#T_{outs}(v) > 1 \wedge (\forall t \in T_{outs}(v) : t.type = triggdless \wedge ctne$
- If $v.kind = join$ then $\#T_{ins} > 1 \wedge \#T_{outs}(v) = 1 \wedge (\forall t \in T_{ins}(v) : t.type = triggdless \wedge (\exists s \in ctne$
- If $v.kind \in choice, junction$, then $\#T_{ins}(v) > 0 \wedge \#T_{outs}(v) > 1 \wedge (\exists! out \in T_{outs}(v) : out.type =$
- If $v.kind \in enpoint, expoint$, then $ctner(v).kind \in \{comp, conc\} \wedge \#T_{ins}(v) > 0 \wedge \#T_{outs}(v) = 1$
- If $v.kind = history$ then $ctner(v).kind \in \{comp, conc\} \wedge (if v.kind = comp$ then $\exists! v \in ctner(v).$

For each vertex $v \in V$, we write the following sets $T_{ins}(v) = incomings(v), T_{outs}(v) = outgoings(v)$, $t_{first} = head(t_{outs})$; transitive transition sets $T^{+}_{ins}(v)$ and $T^{+}_{outs}(v)$ are sets of transitions incoming to and outgoing from, respectively, v or direct or indirect sub-vertexes of v.

Definition 3.9. Transitive container $ctner^{+}(v)$ of a vertex v of a state machine sm is defined as following:

$$ctner^{+}(v) = \begin{cases} sm & ctner(v) = sm \\ ctner(v) \cup ctner^{+}(ctner(v)) & otherwise \end{cases} \quad (1)$$

Similarly, $subvts^{+}(v)$ is a set of transitive sub-vertexes.

A state machine $sm = \{V, T\}$ associated with E is validated if, for each $v \in V$, the constraints listed in Table 1 are hold. These are evaluated before the generation phase is taken into account.

Definition 3.10. Current active configuration Cfg of a UML state machine sm is a set of candidate UML states which are able to process an incoming event.

3.2 Thread-based Concurrency for UML State Machine

3.2.1 Thread-based concurrency analysis

While concurrency is an important aspect defined by the UML State machine specification, especially hierarchical and concurrent state machines with *doActivity*s for states, most of existing approaches do not take into account. This is non-trivial since concurrency is dynamic in UML state machine since the number of threads used for concurrency is non-deterministic.

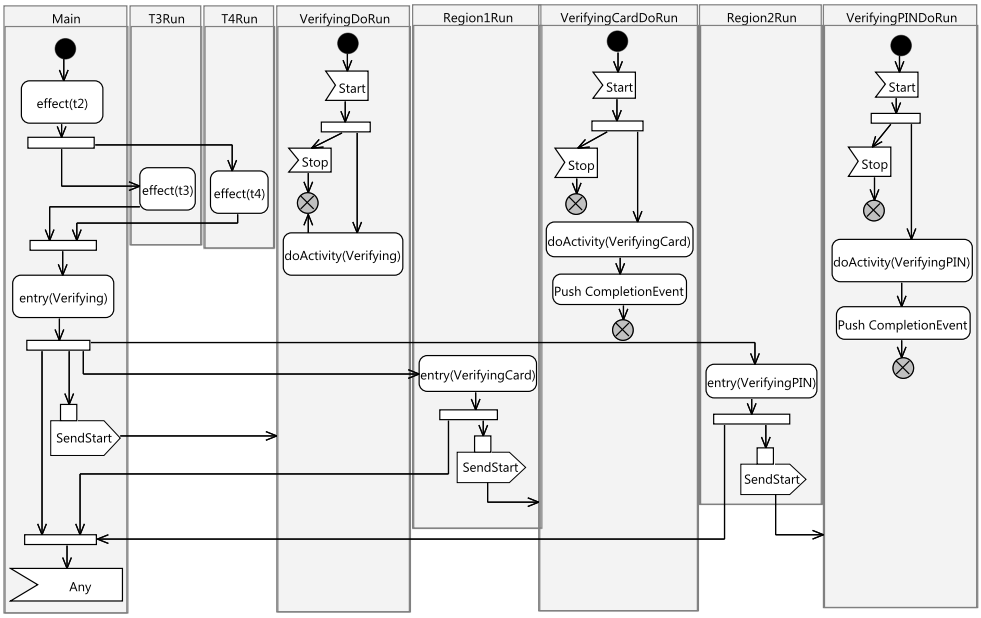

Figure 3: Concurrency of the ATM when receiving the *verifyingPIN* event

Let us give an analysis on the state machine example in Fig. **??**. Assuming that *Idle* is the current active state and a *verifyPIN* event is coming. The *doActivity* behavior of *Idle doActivity(Idle)* (if has) is terminated, *exit(Idle)* and the *effect(t2)* (*T2Effect*) are executed sequentially. These actions are run in a state machine main thread which reads incoming events from a "first in, first out" (FIFO) priority queue. Fig. 3 shows the activity diagram representing the concurrency processing *verifyPIN*, in which each partition represents a thread. *effect(t3)* and *effect(t3)* are run concurrently in two threads *T3Run* and *T4Run*, respectively, since the transitions owning these effects outgo from a fork pseudo state. The entry action *entry(Verifying)* is executed following the termination of the two threads.

UML says that *doActivity(Verifying)*, *entry(VerifyingCard)* and *entry(VerifyingPIN)* should be concurrently executed upon the completion of *entry(Verifying)*, which is represented by a fork node, in which a *Start* signal is sent to *VerifyingDoRun* in order for commencing *doActivity(Verifying)*. The executions of the *doActivity*s of the states *VerifyingCard* and *VerifyingPIN* are also concurrent. Also, upon the completion of *entry(VerifyingCard)* and *entry(VerifyingPIN)*, the main thread completes the processing of the *verifyPIN* event, reads next events from the queue or waits for next event occurrences.

If no event is coming, and *doActivity(VerifyingCard)* and *doActivity(VerifyingPIN)* are long actions (e.g. forever loops inside), the state ma-

chine remains its active configuration and three concurrent actions including *CheckForEvents*, *doActivity(VerifyingCard)*, and *doActivity(VerifyingPIN)* are permanently run.

The termination time of *doActivity(VerifyingCard)* and *doActivity(VerifyingPIN)* is non-deterministic. However, a completion event is generated and pushed to the event queue whenever one of the two completes. For illustration, assuming that *doActivity(VerifyingCard)* terminates before *doActivity(VerifyingPIN)*. As in Fig. 4, the Main thread checks the *CompletionEvent*. *exit(VerifyingCard)* and *effect(t5)* are then executed sequentially. If *cardValid* is computed as true as the result of *doActivity(VerifyingCard)* and *exit(VerifyingCard)*, the Main thread simply executes *effect(t6)* and *entry(CardValid)* before waiting for other events.

In contrast, Main sends a signal to stop *doActivity(VerifyingPIN)* and *doActivity(Verifying)*, executes exit, transition and entry actions in an appropriate order (see Fig. 4) and waits for other events.

We see that the number of concurrent actions is not constant but changes timely. Each action can either deterministically or non-deterministically terminate. In this sense, deterministic actions (DAs) prevent the Main thread from going to the waiting-for-event point. In other words, pending events in the queue are only read and processed once all deterministic actions complete. Therefore, we re-define the run-to-completion paradigm of UML state machine as following:

Definition 3.11. Run-to-completion means that, in the absence of exceptions or asynchronous destruction of the context class object or the state machine execution, a pending Event occurrence is dispatched only after the completion of all deterministic actions commenced by the processing of the current event. At this point, a stable state configuration has been reached

In the example, some of DAs are as followings: *effect(t2)*, *effect(t4)*, *entry(Verifying)*, *entry(VerifyingCard)*, and non-deterministic actions (NDAs) as followings: *doActivity(Verifying)*, *doActivity(VerifyingCard)* and *doActivity(VerifyingPIN)*.

3.2.2 Thread-based design of generated code

Each NDA is run in parallel with the main thread which reads and dispatch events from the event queue. Each is associated with a thread which is initialized at the state machine initialization moment. The number of threads associated with NDAs is therefore equal to that of the NDAs. The design of threads is based on the thread pool pattern, which initializes all threads at once, and the paradigm "wait-execute-wait". In the latter, a thread **waits** for a signal to **execute** its associated method and goes back to the **wait**

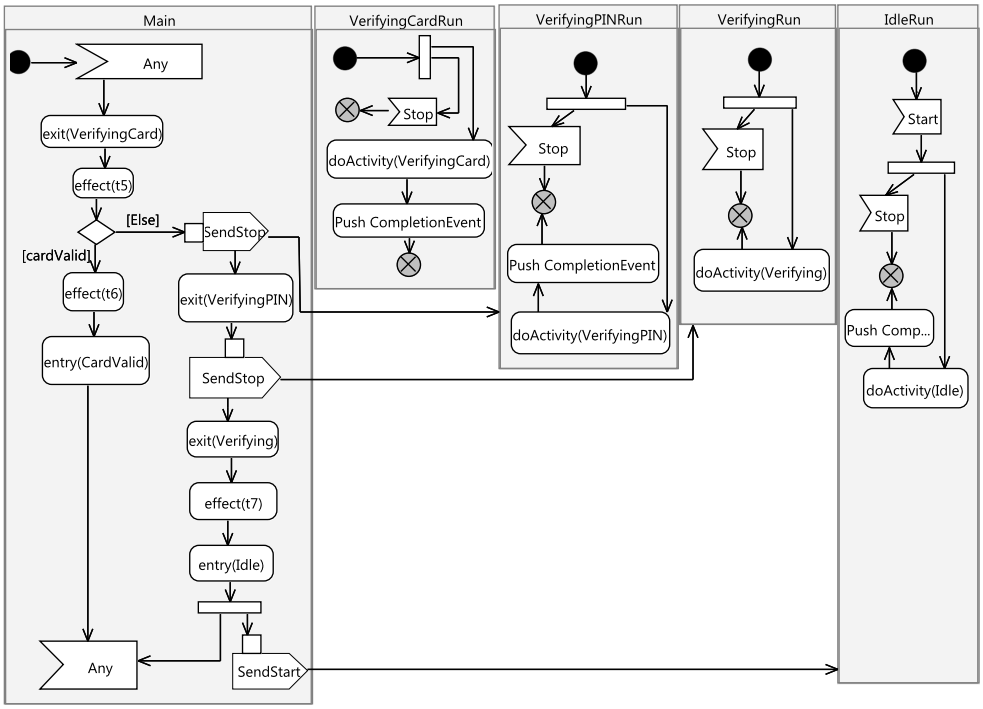

Figure 4: Concurrency of the ATM when *doActivity* of *VerifyingCard* completes before that of *VerifyingPIN*

point if it receives a stop signal or its associated method completes. An NDA is one of the followings:

- *doActivity* of each state if has. The number of *doActivity* $n_{do} = \#\{s \in V | \exists doActivity(s)\}$
- Sleep function associated with a *TimeEvent* which counts ticks and emits a *TimeEvent* once completes: $n_{te} = \#\{e \in E | e \text{ is a time event}\}$.
- Change detect function associated with a *ChangeEvent* which observes a variable or a boolean expression and pushes an event to the queue if changes happen: $n_{che} = \#\{e \in E | e \text{ is a change event}\}$.

Therefore, the concurrency has the number of initial threads $n_{threads} = n_{do} + n_{te} + n_{che}$ plus a main thread which reads events from the event queue, and sends start and stop signals to these initial threads.

Now we consider spontaneous threads which are created to run DAs, joined until and destroyed once DAs complete.

The spontaneous threads follow a paradigm in which if a thread *parent* creates a set of threads *children*, *parent* must wait until *children* complete their associate methods. These threads are created in one of the following cases:

- A thread is created for each effect of transitions' outgoing from a *fork* or incoming to a *join*.
- Entering a concurrent state s, after the execution of $entry(s)$, a thread is also created for each orthogonal region.
- Exiting a concurrent state s, before the execution of $exit(s)$, a thread is also created for each region to exit the corresponding active sub-state.
- An event is processed by active states, in which a thread per orthogonal region.

3.2.3 Deadlock avoidance

Each NDA thread is associated with a mutex for synchronization communication in the multi-thread-based generated code. The mutex must be locked before the method associated with the thread is executed. The mutex associated with the main thread protects the run-to-completion semantics since some event such as *CallEvent* can be processed synchronously and some asynchronously. Each event processing must lock the main mutex before executing the actual processing.

3.3 Assumption

Assuming that we want to generate from the state machine to an object oriented programming language *ActLang*, which is a C++-like and supports multi-threading as following functions and resource control as mutexes.

- A mutex has three methods *lock*, *unlock*, and *wait*, which automatically unlocks the mutex and waits until it receives a signal.
- *FORK(func)* creates a thread (lightweight process) associated with the function/method *func* and *JOIN(theThread)* waits until the method associated with the thread *theThread* completes.

3.4 State transformation

A common state interface *IState* is created. The interface contains three methods, namely, *entry*, *exit*, and *doActivity* respectively corresponding to three state actions. To preserve the hierarchy of composite states, the interface also has two attributes called *actives* and *pres* referring to current and previous active sub-states in case of the presence of history states, and a list of deferred event identifiers.

Each UML state is transformed into an instance of the interface associated with a state ID (which is a child element of an enumeration) inside the active class C. When initialization, each instance refers its methods to the actual methods implemented in C. In C++, this referring is done by using the powerful mechanism function pointer. In other object-oriented languages such as Java, this is done with anonymous sub-classes of the interface. Listing 1 shows the interface and its instances, in which S0 is one of *lstates*. *NUM_STATES* is the number of states in the state machine. The actions of the states are named depending on the name of the states. In the following sections, we only consider *ActLang* as a C++-like. The discussion of other object-oriented languages are much similar since these share the same concepts.

Each of state's *doActivity* is associated with a thread and a mutex. The *doActivity* thread is initialized, waits for a start signal, executes the *doActivity* code, generates a completion if the state is atomic and still active, and goes back to the waiting point as the paradigm above. Listing 2 shows a code segment for *doActivity* threads. The method in the Listing takes as input a state id to use and call the appropriate mutex and *doActivity*, respectively.

//discuss the limitation of doActivity implementation vs specification.

```
typedef struct IState {                                                    1
  int pres[2];   int actives[2];                                           2
  EventId defEvents[2];                                                    3
  void (C::*entry)();  void (C::*exit)();  void (C::*doActivity)();        4
} IState;                                                                  5
```

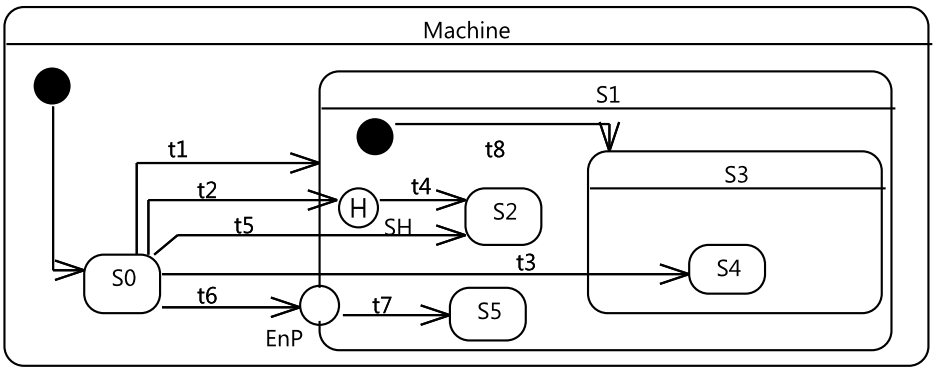

Figure 5: Example illustrating different ways entering a composite state

```
class C {                                                                    6
private:                                                                     7
  IState states[NUM_STATES];                                                 8
public:                                                                      9
  C() {                                                                      10
    states[S0_ID].entry = &C::S0_entry;                                      11
    defEvents[0] = EVENT_MAX; defEvents[1] = EVENT_MAX;                      12
    ...                                                                      13
  }                                                                          14
  void S0_entry {...}                                                        15
}                                                                            16
```

Listing 1: IState interface and function pointers in C++

```
void doActivity(int stateId) {                                               1
  isStarts[stateId] = false;                                                 2
  while(true) {                                                              3
    mutex[stateId].lock();                                                   4
    while(!isStarts[stateId]) {                                              5
      mutex[stateId].wait();                                                 6
    }                                                                        7
    states[stateId].doActivity();                                            8
    isStarts[stateId] = false;                                               9
    mutex[stateId].unlock();                                                 10
    if (!isStops[stateId]) {                                                 11
      if (stateId == IDLE_ID || stateId == DISPENSEMONEY_ID ...) {           12
        pushCompletionEvent(stateId);                                        13
      }                                                                      14
    }                                                                        15
  }                                                                          16
}                                                                            17
```

Listing 2: Example code generated for doActivity

3.5 Region transformation

Each region is transformed into an entering and exiting method. While the entering method controls how a region r is entered from an outside transition t ($src(t) \notin subvts(r)$), the exiting method exits completely a region by executing exit actions of sub-states from innermost to outermost.

A region r is entered by either a transition t ending at the border of its containing state or on a sub-vertex (direct or indirect), depending on how

the state machine is designed. The following lists different ways r may be entered:

- Way 1: entering by default: $tgt(t) = ctner(r) \wedge src(t) \notin subvts(r)$.
- Way 2: entering on a direct sub-vertex: $tgt(t) \in subvts(r) \wedge src(t) \notin subvts(r)$.
- Way 3: entering on an indirect sub-vertex: $ctner(tgt(t)) \in subvts^{+}(r) \wedge src(t) \notin subvts(r)$.

All of the entering ways execute the entry action of the containing composite state $entry(ctner(r))$ after $effect(t)$. *doActivity(ctner(r))* is then signaled to be run in its associated thread. The actions afterward are different from each way. To illustrate, we use an example as in Fig. 5 with *S1* as a target composite state. *t1*, *t3* and *t6* are in the ways of 1 and 3, respectively, while *t2, t5* in the way 2.

The entering method associated with the region r of *S1* has a parameter *enter_mode* indicating how actions should be executed. *enter_mode* takes values depending the number of transitions coming to the composite state $S1$: $\#values(s) =$
$\#\{v \in subvts(s) | v.kind = initial\} +$
$\#\{v \in subvts(s) | \exists t \in T_{ins}(v), src(t) \notin subvts(s)\} +$
$\#\{v \in subvts^{+}(s) \setminus subvts(s) | \exists t \in T_{ins}(v), src(t) \notin subvts^{+}(s)\}$. In this case these values are $\{DEFAULT = 0, SH\_MODE = 1, S2\_MODE = 2, S4\_MODE = 3, ENP_MODE = 4\}$. Listing 3 shows the C++-like example code generated for r.

```
void S1Region1Enter(int enter_mode){                                    1
if (enter_mode == DEFAULT) {                                            2
  states[S1_ID].actives[0] = S3_ID;                                     3
  states[S3_ID].entry();  sendStartSignal(S3_ID);                       4
  S3Region1Enter(DEFAULT);                                              5
} else if (enter_mode == S2_MODE) { //entry                             6
  states[S1_ID].actives[0] = S2_ID;                                     7
  states[S2_ID].entry();  sendStartSignal(S2_ID);                       8
} if (enter_mode == SH_MODE) {                                          9
  StateIDEnum his;                                                      10
  if (states[S1_ID].pres[0] != STATE_MAX){                              11
    his = states[S1_ID].pres[0];                                        12
  } else {                                                              13
    his = S2_ID;                                                        14
  }                                                                     15
  states[S1_ID].actives[0] = his;                                       16
  states[his].entry();  sendStartSignal(his);                           17
  if (S3_ID == his) {                                                   18
    S3Region1Enter (S3_REGION1_DEFAULT);                                19
  }                                                                     20
} else if (enter_mode == S4_MODE) {                                     21
  states[S1_ID].actives[0] = S3_ID;                                     22
  states[S3_ID].entry();  sendStartSignal(S3_ID);                       23
  S3Region1Enter(S4_MODE);                                              24
} else if (enter_mode == ENP_MODE) {...}                                25
```

Listing 3: Example code generated for the region of S1

By default, the active sub-state of the region is set after the execution of any effect associated with the initial transition, $S3$ is set as active sub-state

of $S1$. Entering on a direct sub-state (*S2*) sets the active sub-state of *S1* directly to *S2*. In case of an indirect sub-state ($S4$), the entry action of $S3$ is executed before $S4$ is set as the active-sub state of $S3$ and the execution of $entry(S4)$. It is worth noting that after the execution of each entry, a start signal is sent to activate the waiting thread associated with *doActivity* of the corresponding state.

Transitioning from a vertex to a sub-vertex of the composite state (transition from $S0$ to SH is a particular case) is not as simple as that of two states. It needs a systematic approach which generates code for a transition outgoing from a vertex to any other one. This is detailed in the next section.

3.6 Event and transition transformation

3.6.1 Events

An event enumeration *EventId* is created whose children are event identifiers associated with events. Each event e) is also transformed into a method mtd_e in the context class C. Suppose *levents* is the list of events which can be processed by the state machine *sm*. Besides the explicitly defined events of the state machines, *levents* contains a special event called *CompletionEvent*, which is implicitly implemented as a *CallEvent*. For each event type, the transformation is realized as followings:

- *CallEvent ce*: The associated operation $op(ce)$ can be either synchronous or asynchronous. When $op(ce)$ is called, it waits and locks the main mutex protecting the run-to-completion semantics, and executes mtd_{ce}. Contrarily, the parameters of the asynchronous operation are used to create a signal which is transformed similarly to the case of *SignalEvent*.

- *SignalEvent se*: *SignalEvent* is asynchronous. The signal associated with *se* is written into the event queue of the active class C by an operation which takes as input the signal.

- *TimeEvent te*: A thread *teThread* associated with *te* is created and initialized at the initialization of the state machine. Within the execution of *teThread*, the method associated *te* waits for a signal, which is sent after the execution of the entry of a state $s \in \{v \in V | \exists t \in T_{outs}(v), te \in events(t)\}$, to start sleeping for a duration d of *te*. When the duration expires, *te* is emitted and written to the event queue if s is still active.

- *ChangeEvent che*: Similar to *TimeEvent*, a thread *cheThread* is initialized at initialization but the associated method *mtd* does not wait

for a signal to start. *mtd* periodically checks whether the value of the associated boolean expression $ex(che)$ changes by comparing the current value with the previous value. If a change happen, *che* is committed to the event queue.

- *Any*: any of the above events can trigger the associated transitions.

As above presented, all asynchronous incoming events are stored in a FIFO priority queue, in which each event type has a configurable priority. *CompletionEvent* always has the highest priority. Others are equal by default.

3.6.2 Transitions

Each event triggers a list of transitions. We suppose $T_{trig}(e)$ is the transition list triggered by the event e, and $S_{trig}(e) = \{src(t)|t \in T_{trig}(e)\}$. In other words, $S_{trig(e)}$ is a set of states which are the source states of the transitions in $T_{trig}(e)$. To present how the body of event methods is generated, we define functions as followings:

- Vertex depth $dp(v)$ is defined as:

$$dp(v) = \begin{cases} 1 & v \text{ is a root vertex} \\ dp(ctner(v)) + 1 & otherwise \end{cases} \quad (2)$$

- $Map_e(s) \subset S_{trig(e)}|\forall sub \in Map_e(s) : ctner(sub) = s$, $Prt(e) = \{s \in V|Map_e(v) \neq \emptyset\}$. $Prt(e)$ is an ordered list whose length is $len(Prt\{e\})$ and elements are accessed by indexes. The order of $Prt(e)$ is defined as: $\forall i, j \leq len(Prt\{e\})$,
  if $i < j, dp(Prt(e).get(i)) \geq dp(Prt(e).get(j))$.

```
∀ item ∈ Lm(e)                                                    1
  ∀s ∈ Map_e(item)                                                2
    T_s = {t ∈ T_trig(e)|src(t) = s}                              3
    ∀t ∈ T_s                                                      4
      GENERATE_STATE_EVENT_CHECK(s, t, e)                         5
      GENERATE_GUARD(t)                                           6
      GENTRANS(s, t, tgt(t))                                      7
```

Listing 4: Generation process for an event

The procedure in Listing 4 describes how the generation process works with an event. It first finds the innermost active states which are able to react e by orderly looping over Lm_e. For each transition outgoing from an innermost state, code for active states and deferral events, guard checking and transition code segments are generated by $GENERATE\_STATE\_EVENT\_CHECK$,

$GENERATE\_GUARD(t)$ and $GENTRANS$, respectively. If the identifier of e is equal to one of the events listed in $defEvents$ of the corresponding state (not shown in this paper), it is deferred by putting it to a deferral event queue managed by the main thread, which also pushes the deferred events back to the main queue once one of the pending events is processed.

Depending on the target of t, $GENERATE\_STATE\_EVENT\_CHECK$ can generate single or multiple active state checking code. The latter occurs if $tgt(t)$ is a *join*. The detailed discussion on these is not presented due to space limitation. Listing 5, line 1-2 and Listing 6, line 1 show examples for single and multiple checking, respectively.

Generally, $GENTRANS$ generates code for transitions between any vertexes satisfying the constraints described in Section 4. The detail of $GENTRANS$ is not presented here. Each pseudo state is transformed as the followings:

- *join*: Use $GENTRANS$ for v's outgoing transition.
- $fork$: Use $FORK$ and $JOIN$ for each of outgoing transitions of v.
- *choice*: For each outgoing, an $IF-ELSE$ is generated for the guard of the outgoing together with code generated by $GENTRANS$ (see Listing 6).
- *junction*: As a static version *choice*, a *junction* is transformed into an attribute $junc_attr$ and evaluated before any action executed in compound transitions (see Listing 6). The value of $junc_attr$ is then used to choose the appropriate transition at the place of *junction*.
- *shallow history*: The identifiers of states to be exited are kept in *pres* of $IState$. Restoring the active states using the history is exampled as in Listing 3. The entering method is executed as default mode at the first time the corresponding composite state is entered (see Listing 3). *pres* is updated with the active state identifier before exiting the region containing the history.
- *deep history*: Saving and restoring active states are done at all state hierarchy levels from the composite state containing the deep history down to atomic states. Updating *pres* is committed before exiting the region, which is directly or indirectly contained by a parent state, in which a deep history is present.
- *enpoint*: If *enpoint* has no outgoing transition, the corresponding composite state is entered by default. Otherwise said, $GENTRANS$ is called to generate code for the outgoing transition.

- *expoint*: The code for the unique transition outgoing from *expoint* is generated by using $GENTRANS$.
- *terminate*: The code executes the exit action of the innermost active state, the effect of the transition and destroys the state machine object.

3.6.3 Example Code

Listing 5 shows a code segment generated for the processing of $verifyingPIN$. Single checking (line 1) checks whether $Idle$ is the current active state, in which $activeStateID$ is the identifier of the current root active state. The *doActivity* behavior of $Idle$ is then stopped upon receiving a stop signal. The effect of $t2$, $effect(t3)$ and $effect(t4)$ are then executed after $exit(Idle)$. The execution of $emtry(Verifying)$ then follows the changing of root active state to $Verifying$. $doActivity(Verifying)$ is triggered and followed by concurrently entering the two orthogonal regions of $Verifying$ with appropriate modes.

```
if (activeStateID == IDLE_ID) {                                          1
  sendStopSignal(IDLE_ID);  exit_Idle();                                 2
  effect_t2();                                                           3
  thread_t3 = FORK(effect_t3);  thread_t4 = FORK(effect_t4);             4
  JOIN(thread_t3);  JOIN(thread_t4);                                     5
  activeStateID = VERIFYING_ID;  entry_Verifying();                      6
  sendStartSignal(VERIFYING_ID);                                         7
  th_r1 = FORK(R1Enter(VERIFYINGCARD_MODE));                             8
  th_r2 = FORK(R2Enter(VERIFYINGPIN_MODE));                              9
  JOIN(th_r1);  JOIN(th_r2);                                             10
}                                                                        11
```

Listing 5: Example code generated for event $verifyingPIN$

The discussion of Listing 6 is similar to Listing 5 except that a multiple checking is executed (line 1-2) instead of a single one. The evaluation for $Junction1$ is executed (line 4) before any other actions (semantic conformance) to decide the decision should be taken (line 10-17).

```
if((states[VERIFYING_ID].actives[0]==CARDVALID_ID)                       1
   &&(states[VERIFYING_ID].actives[1]==PININCORRECT_ID)) {                2
  Junction1 = 0; //else outgoing transition of Junction1                 3
  if (tries < maxTries) {Junction1 = 1;}                                 4
  FORK(R1Exit); FORK(R2Exit);                                            5
  //JOIN ...                                                             6
  sendStopSignal(VERIFYING_ID); exit(VERIFYING_ID);                      7
  FORK(effect_t11); FORK(effect_t13);                                    8
    //JOIN ...                                                           9
  if (Junction1=1) {                                                     10
    tries++;                                                             11
    activeStateID = IDLE_ID;     entry(IDLE_ID);                         12
    sendStartSignal(IDLE_ID);                                            13
  } else {                                                               14
    cardValid = false;                                                   15
    activeStateID = IDLE_ID;     sendStartSignal(IDLE_ID);               16
  }                                                                      17
}                                                                        18
```

Listing 6: Example code generated for $Join1$ and $Junction1$

4 Ongoing work

References

[1] N. C. C. Brown, M. Kolling, and A. Altadmri. Position paper: Lack of keyboard support cripples block-based programming. In *Proceedings of the 2015 Blocks and Beyond Workshop*, Atlanta, GA, USA, 2015.

[2] E. J. Chikofsky, J. H. Cross, and others. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[3] H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, Genova, Italy, 2006.

[4] T. Hettel, M. Lawley, and K. Raymond. Model synchronisation: Definitions for round-trip engineering. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5063 LNCS, pages 31–45, 2008.

[5] J. Hutchinson, J. Whittle, and M. Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, Sept. 2014.

[6] G. Mussbacher, D. Amyot, R. Breu, J.-m. Bruel, B. H. C. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, J. Kienzle, and M. Schöttle. The Relevance of Model-Driven Engineering Thirty Years from Now. *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 183–200, 2014.

[7] OMG. Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0. http://www.omg.org/spec/SPEM/2.0/PDF, 2008.

[8] OMG. Precise Semantics Of UML Composite Structures. (October), 2015.

[9] B. Selic. What will it take? A view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, Oct. 2012.

[10] S. Sendall and J. Kuster. Taming Model Round-Trip Engineering.

[11] S. Sendall and J. Küster. Taming model round-trip engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, 2004.