# Use of model transformation for the formal analysis of railway interlocking models

T. Xu[1], O. M. Santos[2], X. Ge[2] & J. Woodcock[2]
*[1]State Key Laboratory of Rail Traffic Control and Safety, Beijing Jiaotong University, China*
*[2]Department of Computer Science, University of York, UK*

## Abstract

Model transformation is at the heart of Model-Driven Engineering (MDE). In MDE, the system model is specified using a modelling language, such as UML (Unified Modelling Language) or a DSL (Domain-Specific Language). Once a model is specified, executable code for a computing platform can be automatically generated by means of model transformation (code generation). Besides the support for incremental model development, MDE also enables the formal verification of system properties. In the context of safety-critical systems, such as railway interlockings, the system model (e.g., specified in terms of UML) can be translated to a formal (mathematical) language more amendable to rigorous analysis. This paper presents a model transformation that takes a railway interlocking model (specified in Executable UML (xUML)) as input and outputs a formal model that can be mathematically analysed. This can potentially bridge the gap between well-known modelling languages (such as xUML) and formal languages, which facilitates the systematic development of safety-critical systems in terms of MDE. A small xUML railway interlocking model is used to illustrate the proposed method.
*Keywords: railway interlocking systems, model driven engineering (MDE), executable UML (xUML), formal languages, formal analysis.*

## 1 Introduction

Railway interlocking plays a very important role in establishing high safety for train operations in a railway system, and protecting passengers and equipment from damage. Due to its life-critical application, a rigorous verification phase is required to ensure the correctness of the interlocking system. Among the most common verification methods (including simulation and test) used in the verification phase, formal methods are often recommended for modelling and verifying interlocking systems [1]. Differently from testing and simulation, formal verification allows the analysis of all the possible scenarios that a system can possibly generate. For instance, in terms of railway interlockings, it can check all the possible executable combinations for a given track layout.

However, compared with the use of testing and simulation for the analysis of modelling languages, such as the Unified Modelling Language (UML) or Domain-Specific Languages (DSL), the use of formal verification requires a lot of mathematical expertise from software engineers. In this sense, Model-Driven Engineering (MDE) [2] paves a solid foundation in the use of automatic formal verification for railway interlocking systems by means of model transformation. In this approach, the model of a railway interlocking can be defined using a modelling language, such as Executable UML (xUML), and automatically translate that to the input language of a formal verification tool. The effectiveness of this approach depends on two important aspects, related to the use of: (a) a tool for automating the model transformation and (b) a target formal language with enhanced tool support.

Firstly (a), several model transformation tools can be found in the literature, such as ATL [3] and graph-based model transformation [4], which have gained widely use in MDE. Amongst them, the Epsilon model management framework [5], built on top of the Eclipse platform [6], supports a collection of transformation languages. Including ETL (Epsilon Transformation Language) for model-to-model transformation and EGL (Epsilon Generation Language) for model-to-text transformation.

Secondly (b), tool support for the formal analysis includes several technologies, including: (a) finite-state model checkers; and (b) theorem provers. Examples of such languages and tools include CSP [7], B [8], Alloy [9] and Promela/SPIN [10]. However, due to the required expertise for using them, these languages are not widely used by engineers compared to modelling languages like xUML. In particular, this work focuses on the use of (a) finite-state model checkers. In this formal verification approach, the tool (model-checker) generates and analyses all the possible execution paths, for a given scenario of the model, against a system property. For example, a safety property in a railway interlocking model would be that trains never collide due to a signal error in the interlocking. This property is then formalised following the model's definition in the formal language.

In this paper we describe our approach for analysing interlocking models specified in Executable UML (xUML). xUML has been used for modelling railway interlocking systems in the context of the INESS (INtegrated Europe Signalling System) European project (http://www.iness.eu/). In particular, we show how to translate an xUML scenario representing a track layout for a small interlocking to the formal language Communicating Sequential Processes (CSP). This model can be formally analysed using the FDR2 tool [11].

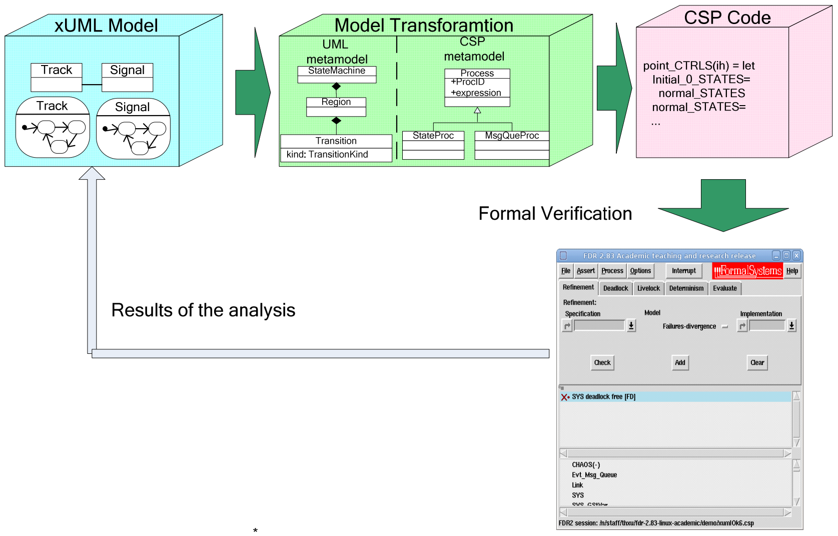
Figure 1 illustrates the framework of the proposed approach. It consists of several steps, starting from the xUML model of a railway interlocking system to the generation of the CSP code. Then, formal verification takes place using the FDR2 tool, which is used to detect and show errors of the xUML model representing the railway interlocking.

## 1.1 Related work

There is a significant body of work in the literature (e.g., [12, 13]) targeting the formal analysis of railway interlocking systems. However, the shortage of user-friendly modelling tools (e.g., UML) and the support for the automatic verification of interlocking models make it difficult to gain wide use in industry. Regarding the use of model transformation for analysing xUML model, the work of Treharne *et al.* [14] is particularly relevant to our work. They present a model transformation that uses as input an xUML model and output a formal model in the CSP || B language. Similarly, Hansen *et al.* [15] describes a formalisation of a subset of xUML in the formal language mCRL2. Both works also target the automatic analysis of the xUML models using tool support.

## 1.2 Organization

The rest of the paper is organized as follows. The next section presents an overview of xUML, used to model railway interlocking systems, and the formal language CSP, utilised for the formal verification of these models. The generation of a CSP models from xUML models is presented in Section 3. Section 4 shows an overview of the formal analysis the proposed approach. Finally, we conclude the paper in Section 5.



```
point_CTRLS(ih) = let
  Initial_0_STATES=
    normal_STATES
  normal_STATES=
  ...
```

```
SYS deadlock free [FD]
```

```
CHAOS(-)
Evt_Msg_Queue
Link
SYS
```

```
FDR2 session: /n/staff/thxu/fdr-2.83-linux-academic/demo/xumlOk6.csp
```

Figure 1: Framework for the verification of railway interlocking models in xUML.

## 2 Background

### 2.1 Executable UML for modelling railway interlockings

Executable UML (xUML) [16] is a coherent subset of UML that has been developed to define the program execution of the model at a higher level of abstraction. This abstraction leaves open the possibility to generate an implementation of the system in different target (computing) languages, including technologies like C, Java and Ada. xUML extends UML with an Action Specification Language (ASL), providing all the conditional logics and primitive needed to manipulate xUML objects, which allows the developer to define the behaviour in sufficient detail so that it can be executed. The other benefit of using an ASL is the ability to describe a system independent of its target platform. Typically, xUML supports the following UML diagrams [17]:

- Use Case Diagrams to capture the requirement of the system;
- Sequence Diagrams to define the interaction among different domains;
- Class Diagrams to describe the classes in each domain;
- State Machines to specify the behaviour of each class.

In this paper, we focus on the use of class diagrams and state machines, since these are the main diagrams used to construct the xUML models of railway interlockings described in this paper. Basically, a class is used to define the structure of an object (e.g., attributes and associations). Every class has an associated state machine, which defines the behaviour of the object.

Figure 2 shows the xUML class diagram of a simple interlocking specification called the Micro interlocking model [15], which has been provided by partners in the INESS project. It contains five classes, named *element*, *track*, *point*, *signal* and *route.* The class *element* is a generalisation of *track*, *point* and *signal*. This means that all the structure and behaviour defined for *element* will be part of the structure and behaviour of these classes. An instance of the Micro model is obtained from the track layout depicted in Figure 3. This small layout consists of three tracks $t_1$, $t_2$, $t_{3,}$ one point $p_1$, one signal $s_1$ and two routes. The first route $r_1$ requires $p_1$ to be positioned left and goes from track $t_1$ to track $t_{3.}$ Route $r_2$ requires $p_1$ to be positioned right and goes from $t_1$ to $t_2$. Both routes have $s_1$ as their entry signal. The model instances thus contains three track objects, one point object, one signal object and two route objects. Every component of the layout corresponds to a particular component of the class diagram.

Figure 4(a) presents the state machine associated to the *route* class. This models the main functionality of the Micro interlocking, i.e., route setting and route cancellation. When a route receives a reserve request, it sends a signal to its left and right points to move into position. When all points are positioned, tracks along the route and entry signal are ready, the route becomes *ready*. When one of the elements associated with the route is no longer in the required *ready* state, or the route is cancelled, the route becomes *idle*. Figure 4 (b) shows the state machine associated to the *point* class. In this state machine, two states *normal* and *error* are used to model the normal and abnormal operations. The presentation of the other state machines is omitted for simplicity.

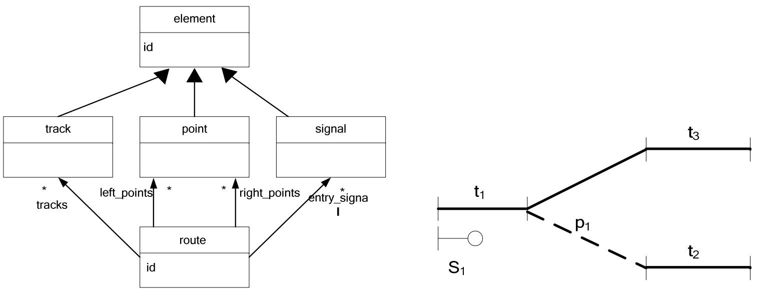
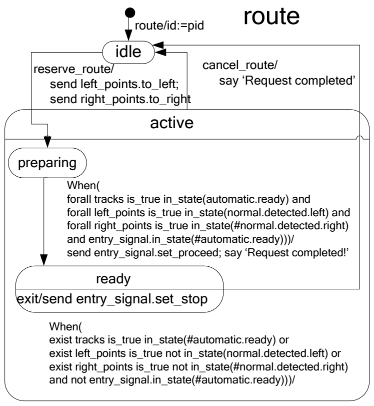
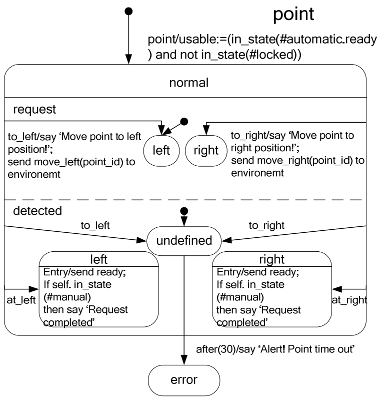


Figure 2: Class diagram for the micro interlocking track layout example.


route
route/id:=pid
idle
reserve_route/
send left_points.to_left;
send right_points.to_right
cancel_route/
say 'Request completed'
active
preparing
When(
forall tracks is_true in_state(automatic.ready) and
forall left_points is_true in_state(normal.detected.left) and
forall right_points is_true in_state(#normal.detected.right)
and entry_signal.in_state(#automatic.ready)))/
send entry_signal.set_proceed; say 'Request completed!'
ready
exit/send entry_signal.set_stop
When(
exist tracks is_true in_state(#automatic.ready) or
exist left_points is_true not in_state(normal.detected.left) or
exist right_points is_true not in_state(#normal.detected.right)
and not entry_signal.in_state(#automatic.ready)))/

(a) state machine for route class.


point
point/usable:=(in_state(#automatic.ready
) and not in_state(#locked))
normal
request
to_left/say 'Move point to left position!';
send move_left(point_id) to environemt
left
right
to_right/say 'Move point to right position!';
send move_right(point_id) to environemt
detected
to_left
to_right
undefined
left
Entry/send ready;
If self. in_state
(#manual)
then say 'Request completed'
right
Entry/send ready;
If self. in_state
(#manual)
then say 'Request completed'
at_left
at_right
after(30)/say 'Alert! Point time out'
error

(b) state machine for point class.

Figure 3: State machines associated to the classes route and point.

## 2.2 CSP

CSP is a notation for describing concurrent systems whose components, called processes, communicate with each other and the environment. A process can be thought of as an independent entity which has interfaces through which the processes interact with the external environment. A process is defined in terms of events – basic elements of CSP.

CSP-M [18] is a machine-readable version of CSP developed as the input language of the FDR2 tool. CSP-M extends CSP with a small but powerful functional language, which offers constructs such as lambda and let expressions. The language provides a number of predefined data types, e.g., Booleans, integers, sequences and sets, and also allows user-defined data types. CSP-M is now the *de facto* standard of machine-readable CSP.

FDR2 is a model checker for CSP. It allows concrete design description to be compared with abstract specification in order to check if the refinement properties are satisfied. If the properties are not satisfied (thus the refinement

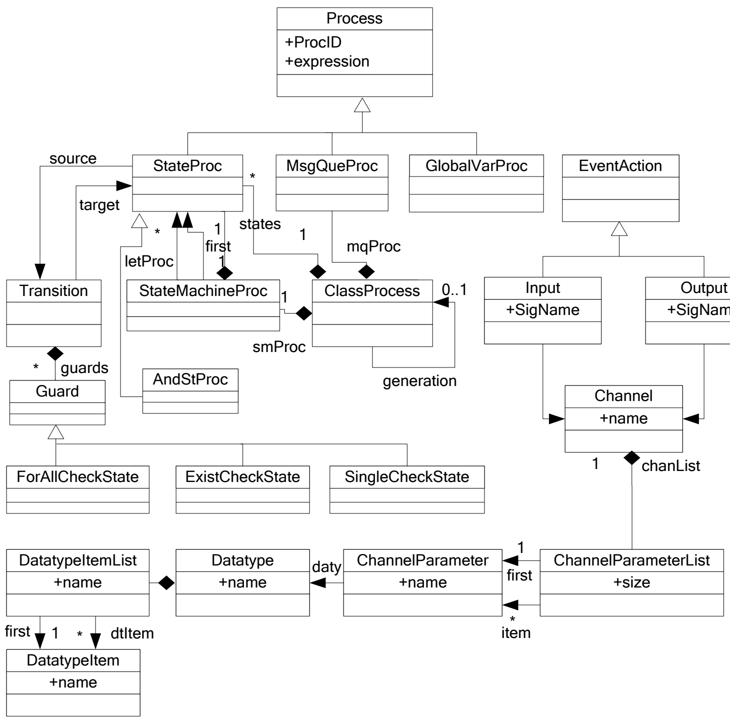


Figure 4: The CSP metamodel.

check fails), FDR2 will generate counter-examples that can be used to pin-point the error that caused the property to fail.

## 3 Model transformation (generating CSP from xUML)

In order to use model transformation technology to generate a CSP model, amenable to formal verification, from the original xUML model, we need to define a meta-model. Basically, a meta-model defines a structure that models must conform to. For instance, a correct model representing the transformed CSP model, must conform to the meta-model (structure) that we have defined for CSP, which will be detail discussed in the following section.

With a target meta-model specified, we define translation rules that are executed automatically. These rules are composed of input and output elements. Input elements correspond to elements found in the xUML model. The output defines what element this particular input should be transformed to. In particular, we have used the Epsilon tool-set for implementing the transformation of the model.

In our work, we have used two different types of transformation. Firstly, we use a model-to-model transformation for generating a target CSP model from the source xUML model. Secondly, we use the newly created CSP model to define a model-to-text transformation that generates code, which can be analysed with the FDR2 model-checker.

### 3.1 CSP metamodel

As described previously, the CSP metamodel is shown in Figure 5. This meta-model bears several similarities to the works of Bisztray *et al.* [19] and Treharne *et al.* [14], in particular the use of the *Datatype* and *DatatypeItemList* classes. Like [14], we model all event and their types, e.g., the event *read.ih?x* corresponds to an instance of *Input* for the channel called read, which has two *EventParameters*, *ih*, and *x*. Modeling this aspect is especially important for our use which requires communication of CSP events with different handlers of instance processes. In addition, the metamodel also contains support for localized definition (*StMachineProc*), which is convenient for representing the behaviour of xUML state machines.

There are however significant differences. A major difference is that the *AndStProc*, *MsgQueProc*, *GlobalVarProc* classes are employed to describe the concurrent state machine, message queue and global variables of the class, respectively. In CSP the process can only refer to its own variables (there are no shared variables), which results in a lot of difficulties in the transformation of xUML models with state information. In favour of simplifying the transformation, the above classes, associated with the global variable and message queue of xUML classes, are explicitly introduced.

### 3.2 Transformation rules

Based on the meta-models for xUML and CSP, we construct transformation rules. The general procedure used for generating CSP models, out of xUML models, is outlined in Algorithm 1. This basically consists of the following transformations:

- Translation of class instances,
- Translation of control flow in the state machines,
- Translation of the signal queue modelling the external environment,
- Translation of global variables for storing state values and attributes of state machines,
- Composition of the all transformed process into the final process SYS (representing the global execution system).

For the purposes of readability we only discuss the transformation of class instances and their associated state machines found in the xUML model.

Algorithm 1. An outline for generating CSP for object instances.

```
for all c in class of xUML model m do
 if c has a state machine sc then
   <c>_SCTRL(ih) = let /*state machine behavior for instance handler ih*/
   for all states, s, in sc do
     <c>_STATES= receive.ih?x -> if (x==trigEvt ) then
                                   else if …
                                   else <c>_STATES
  end for
   within /*initial _STATES process*/
  <c>_QueueS(ih, s) = (#s<M)&generate.ih?x -> … /* the message queue associated with sc*/
              [](#s>0)& receive.ih!head(s) -> …
  <c>_GStVar(ih,v) = …         /*the global variable process to store the state value of sc*/
```

```
13   <c>_Atribute(ih, w) = …   /*the global variable process to store the attribute value of sc*/
14  end if
  /*Composition of all instances in <c>_HANDLER*/
15 SYS=(|||ih ∈ HANDLER <c>_SCTRL(ih)){|generate,...|} (|||ih ∈ HANDLER <c>_QueueS(ih, s))
     {|...|} (|||ih ∈ HANDLER <c>_GStVar(ih,v)) {|...|} (|||ih ∈ HANDLER <c>_Atribute(ih, w))
```

### 3.2.1 Transforming the class instance

A class instance, either active or passive, is translated into a controller process. In our translation from xUML to CSP, each class becomes a process specification *<c>_SCTRL(ih)*, as shown in line 3 of Algorithm 1. Each one of these processes consists of four parallel parts: the first part is the translation of the state machine associated with class (lines 3 to 9), the second part formalises the state machine inherited from the superclass (the detail discussion is omitted here for conciseness), the third part models the message queue associated with the state machine as an event pool (lines 10 to 11), and the fourth part denotes the global variables used to store the state location and update the value of the attribute of the state machine (lines 12 and 13), respectively. Finally, the CSP code associated with the composition of all instances is given in line 15.

### 3.2.2 Transforming the state machine

In this section, based on the semantics of the xUML state machine, we design a transformation of the xUML state machine model to a CSP specification in a compositional manner. Our transformation rules are designed to inductively process the three types of state found in xUML: basic states, OR-states and AND-states. The core transformation rules (defined using the Epsilon Transformation Language (ETL) of the Epsilon tool) are presented is Listing 1.

```
operation StateMachine2CTRLProcess (sr : UML!StateMachine)
: CSP!StMachineProc {
var root : new CSP!StMachineProc;
root.ProcID := sr.name + '_CTRLS';
var states := UML!Vertex.all.select
      (sm1|sm1.containingStateMachine() = sr);
for (st in states){
  var Cont := st.container;
  if (Cont.state.isDefined()) {
    var stProc : new CSP!StateProc;
    if (Cont.state.isOrthogonal) {
      SimConcurent2StProc(st, sr, stProc);
    } else {
      SimComp2StProc(st, sr, stProc);
    }
    root.letProc.add(stProc);
  } …
 }
var startP : new CSP!StateProc;
startP := getStartState(sr);
root.first := startP;
return root;
}
```

Listing 1: Transformation rules for the state machine.

The operation `StateMachine2CTRLProcess` is responsible for transforming the xUML state machine into a localized (*let ... within*) process. In terms of our CSP metamodel, this mainly involves a creation of new instances of `CSP!StMachineProc`. In line 4, we set the name of the name of state machine, e.g., *track_CTRLS*. From lines 7 to 18, each of the states of the state machine are iterated and operations are called to transform the state types.

Note that `st`, `sr,` `stProc` are process parameters and denote the instance of `UML!Vertex`, `UML!StateMachine` and `CSP!StateProc`, respectively. Each application of the above operations returns a process instance `stProc`, which is added to the localized process by the statement `root.letProc.add(stProc).` Finally, we compute the initial state of the class diagram using the `getStartState` operation and we link the new process to the original root.

## 4 Formal analysis of an interlocking model

In this section we exemplify the verification of the xUML Micro interlocking shown in Section 2.1. Listing 2 illustrates a partial translation of the *point* state machine (presented in Figure 4(b)) in terms of CSP. Lines 9 to 18 denote the process `normal_detected_undefined_STATES`, which reacts to the external event *tout* (modelling the time event *after (30)*) and reaches the target process `error_STATES`. At the same time the concurrent process `normal_requested_right_STATES`, representing the behaviour of concurrent region request, is terminated by the process *STOP* (shown in lines 19 to 23). The resulting process *STOP*||| *error_STATES* is equivalent to *error_STATES*, which is used to model the transition from state *undefined* to *stop*. The synchronization communication statement *envGenerate.ih?ok* is used to keep the consistent pace between the concurrent states, i.e., *request* and *detected*.

```
1 point_CTRLS(ih) = let
2   Initial_0_STATES=
3     normal_STATES
4   normal_STATES=
5     normal_requested_Initial_0_STATES
6       |||
7     normal_detected_Initial_0_STATES
8       …
9   normal_detected_undefined_STATES=
10    receive_normal_detect.ih?x ->
11    if(x==tout) then
12      write.ih!error_STATE ->
13      envGenerate_detected.ih?ok ->
14      error_STATES
15    else if(x==at_right) then
16      ……
17    else if(x==at_left) then
18        ……
19  normal_requested_left_STATES=
20    receive_normal_request.ih?x ->
21    if (x==tout) then
22      envGenerate_requested.ih?ok -> STOP
23    else if(x==to_right) then …
24  error_STATES=
       receive.ih?x: EVENT ->
       envGenerate_detected.ih?ok ->
       envGenerate_requested.ih?ok -> error_STATES
   within Initial_0_STATES
```

Listing 2: Partial CSP code for the state machine point.

The CSP code for the composition of all instances in the Micro interlocking model is shown below. Here, different instances of track (i.e., *t1*, *t2*, *t3*), route (i.e., *r1*, *r2*), point (i.e., *p1*) and signal (i.e., *s1*) are executed in parallel.

```
SysCTRL =
   (point_CTRLS(p1) ||| signal_CTRLS(s1)
    |||track_CTRLS(t1) |||track_CTRLS(t2)
    |||track_CTRLS(t3)|||route_CTRLS(r1)|||route_CTRLS(r2))
       [|{|envGenerate, envGenerate_detected,
        envGenerate_requested|}|]
   (element_CTRL_point_CTRL(p1)|||element_CTRL_signal_CTRL(s1)
    |||element_CTRL_track_CTRL(t1)
    |||element_CTRL_track_CTRL(t2)
    |||element_CTRL_track_CTRL(t3))
```

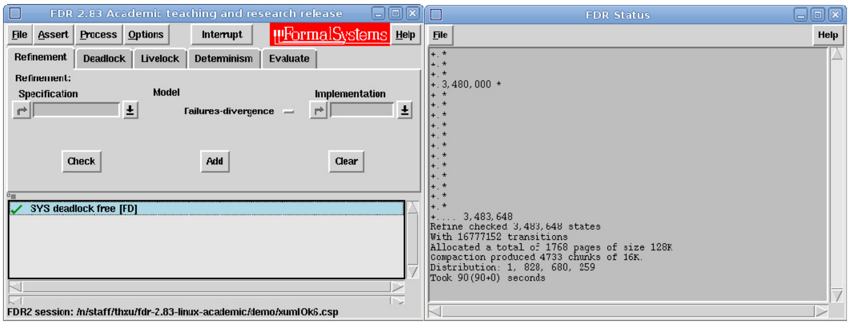
To demonstrate the verification approach, we analyse the model against a very simple property. Basically, we check if the interlocking model never gets to a deadlock situation – where no routes can be further reserved or cancelled. Deadlock checking is implemented with respect to the processing of signals in active objects. For example, we need to check that

$$\mathit{SYSLinkSysState} \underset{\{|\mathit{externalGenerate}|\}}{\parallel} \mathit{ExternalSignals}$$

is deadlock-free. The following partial listing shows the corresponding CSP process *ExternalSignals*, which is used to define the property.

```
External_Point_p1 = externalGenerate.p1!at_right ->
    externalGenerate.p1!to_right -> external_Point_p1
    ……
```

The verification results obtained in the FDR2 tool are shown in Figure 5.



SYS deadlock free [FD]

FDR2 session: /n/staff/thxu/fdr-2.83-linux-academic/demo/xumlOk6.csp

```
+. *
+. *
+. *
+. 3,480,000 *
+ *
+. *
+. *
+. *
+. *
+. *
+. *
+. *
+. *
+. *
+. *
+. *
+.... 3,483,648
Refine checked 3,483,648 states
With 16777152 transitions
Allocated a total of 1768 pages of size 128K
Compaction produced 4733 chunks of 16K.
Distribution: 1, 828, 680, 259
Took 90(90+0) seconds
```

Figure 5: Snapshot of deadlock-free verification in the FDR2 tool.

## 5 Conclusions and future work

Modelling languages, like Executable UML (xUML), can be used for the definition of railway interlocking systems. In particular, modelling languages typically use testing and simulation of the analysis of the system. They tend not to provide analysis based on formal, more rigorous, methods. This is especially needed for the analysis of safety-critical systems, like railway interlockings.

In this paper, we have presented our approach towards the formal analysis of railways signalling specified with xUML. Our approach focuses on the use of model transformation, an integral part of Model-Driven Engineering. Starting from an xUML model, we translate that to the Communicating Sequential Process (CSP) language, used as input to the FDR2 formal verification tool. This enables the formal analysis of the model using FDR2.

Future work will mainly focus on the provision of a transparent verification methodology. For instance, currently, the verification results of the analysis of the system are provided in terms of the CSP model. We want to be able to: (i) specify verification properties in terms of the xUML model; (ii) generate counter-examples, executions of the model that violate the property, provided by FDR2 in terms of the xUML model (using sequence diagrams of UML).

## Acknowledgements

The research is supported from the National Science Foundation of P. R. China under grant No. 60634010, the State Key Laboratory of Rail Traffic Control and Safety of Beijing Jiaotong University within the frame of the project (No. RCS2008ZZ005) and the Technology Funding Project (Beijing Jiaotong University, No. 2007RC101, 2007XM004).

This research is also funded by the European Commission via the INESS project, Seventh Framework Programme (2008-2011).

## References

[1] European Committee for Electrotechnical Standardization (CENELEC). Railways Applications: The speciation and demonstration of dependability, reliability, availability, maintainability and safety (RAMS), 1997.

[2] Jouault, F., Alliaire, F., Bézivin, J., et al., ATL: A model transformation tool. *Sci. Comput. Program*, 72(1-2), pp. 31-39, 2008.

[3] http://www.eclipse.org/m2m/atl/atlTransformations/

[4] Varró D., Automated formal verification of visual modeling languages by model checking. *Softw. Syst Model*, 3(2), pp, 85-113, 2004.

[5] Extensible Platform for Specification of Integrated Languages for Model management (Epsilon). http://www.eclipse.org/gmt/epsilon

[6] http://www.eclipse.org/gmt

[7] Hoare, C. A. R., *Communication Sequential Process*. Prentice-Hall, Englewood Cliffs, 1985.

[8] Abrial, J. R., *The B Book: Assigning programs to meaning*. CUP, 1996.
[9] Software Design Group at MIT: Alloy Analyser 4.1.2, 2008.
[10] Holzmamm, G. J., The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), pp. 1-17, 1997.
[11] http://www.fsel.com
[12] Cimatti, A., Giunchiglia, F., Mongardi, G., et al., Formal verification of a railway interlocking system using model checking. *Formal Aspects Comput*, 10, pp. 361-380, 1998.
[13] Garmhausena, V. H., Campos, S., Cimatti, A., et al., Verification of a safety-critical railway interlocking system with real-time constraints. *Science of Computer Programming*, 36, pp. 53-64, 2000.
[14] Treharne, H., Turner, E., Paige, R. F., et al., Automatic generation of integrated formal model corresponding to UML model. *47th International Conference, TOOLS EUROPE 2009*, pp. 357-367, 2009.
[15] Hansen, H., Ketema, J., Luttik, B., et al., Towards model checking executable UML specification in mCRL2. *Innovation in Systems and Software Engineering,* 6(1-2), pp. 83-90, 2010.
[16] Raistrick, C., Francis, P., Wright, J., et al., *Model Driven Architecture with Executable UML*. Cambridge University Press, Cambridge, 2004.
[17] OMG Unified Modeling Language: Superstructure, version 2.0 – final adopted specification, August 2008. http://www.omg.org.
[18] Scatergood, B., The semantics and implementation of machine-readable CSP. PhD thesis, University of Oxford, 1998.
[19] Bisztray, D., Heckel, R., Ehrig, H., Verification of architectural refactoring rules. *Technical report*, University of Leicester, 2008.