Appendix of the paper submitted to ModeVVa 2013

Pascal André Jean-Marie Mottu Gilles Ardourel

July 2013

This document is a web appendix of a **ModeVVa2013** submission. It exhibits details of the experimentation led for the paper.

The experimentation is led on a simplified version of a *platoon of vehicles* case study inspired from [1]. This is an embedded application that should ensure safety properties such as avoiding collisions or not losing a vehicle. The vehicles and the driver are components which interact to know their respective position and speed in order to control their own move. The specification language is Kmelia, a wide-spectrum language dedicated to the development of correct components [2].

After a short presentation of the case study and the modelling language, we specify the case study in Kmelia, and we indicate the verification steps before explainning the test harness construction and execution.

Contents

A A sketch of the case study **2**

B The modeling language **2**

C The Kmelia specification of the platoon system **2**

C.1 System Assembly . . . 2
C.2 Platoon Vehicle . . . 3
C.3 Platoon Leader . . . 8

D Specification Verification **8**

E An Experimentation outlook **9**

E.1 Harness construction . . . 9
E.2 Starting the harness construction . . . 9
E.3 Selecting the services under test . . . 11
E.4 Consistency and Completion process . . . 12
E.5 Test execution . . . 14

A A sketch of the case study

Let consider a platoon of several vehicles, running in a platoon mode only, since the solo mode do not have any kind of interest from a safety point of view. Only the speed and position (X axis only) features of the vehicle are held with the safety constraint that no vehicle have the same position to avoid collision. Another (non-safety) constraint usually applies: once in the platoon, the vehicles always follow the same vehicle except the first one that follows the driver. For sake of simplicity, and maybe reuse, the driver is assumed to be a kind of special vehicle that controls its own values according to a position goal provided at the beginning of the run. Each running vehicle follows it predecessor and control their speed and position by capturing their predecessors (in the train) position and speed. The follower running vehicles control their speed and position by capturing their predecessors (in the train) position and speed. Safe conditions are provided as contracts like *the vehicles have distinct, ordered positions.*

To model the elements at the different specification levels (service contract, interactions, behaviour) we may write specification with UML-like languages or component based languages such as the one used for the CoCOME contest [3]. But to verify their properties we need a formal wide-spectrum language with verification facilities. We chose to specify this support example using Kmelia a multi-service component language.

B The modeling language

Kmelia is an abstract formal component model dedicated to the specification and development of correct components [2]. A component system Kmelia is an assembly of component, which can them selves be composite. The main features of Kmelia are:

- *component*: a component is a container of services; it is described with a state space constrained by an invariant. A component is designed independently from its environment by setting assumptions such as virtual client components or required service specifications;
- *service*: a service describes a functionality; it is more than a simple operation; it has a pre-condition, a post-condition and a behaviour described with a labelled transition system (LTS). Moreover a service may hierarchically give access to other services. The behaviour supports communication interactions, dynamic evolution rules and service composition;
- *assembly of components*: an assembly is a set of components linked via their required and provided services with the aim to build effective functionality. Linking components by their services in assemblies establishes a possible bridge to Service Oriented Architectures. The component assemblies are governed by strict service composability rules;
- *composite component:* a composite component is a component that encapsulates assemblies or other components; it is subject to encapsulation and promotion policies.

The service contracts apply on the component invariant and the links. The component interaction is achieved by the synchronous communications of the service behaviour. The global behaviour is a combination of service behaviours.

Kmelia is supported with an Eclipse-based analysis platform called COSTO[1]. A detailed description of the model and the property verification techniques will be found in [4, 2].

C The Kmelia specification of the platoon system

We present first the system and then the components.

C.1 System Assembly

The assembly of Figure 1 represents three vehicles and a driver. Once initialised in a platoon sequence, a vehicle is enable to run.

The Kmelia description is detailed in Listing 1 The composite component does not provide services, the system is self-controlled. The assembly describes the components and the assembly links between required and provided services (client-server relationship).

[1] http://www.lina.sciences.univ-nantes.fr/aelos/projects/kmelia/

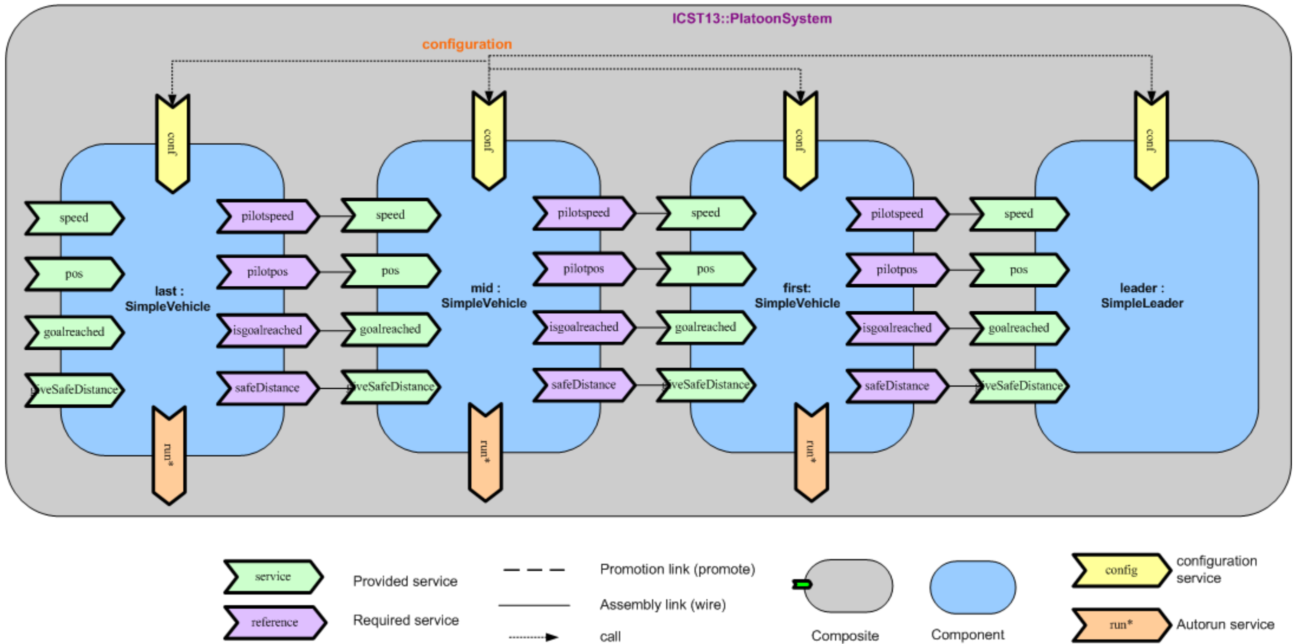

Figure 1: Simplified Kmelia model of the Platoon assembly

Listing 1: Kmelia specification Platoon Assembly

```
APPLICATION  PlatoonSystem
INTERFACE

USES {PLATOONLIB}

SERVICES

END_SERVICES
COMPOSITION
  Assembly
    Components
      leader : SimpleLeader;
      first : SimpleVehicle;
      mid : SimpleVehicle;
      last : SimpleVehicle;
    Configuration
      leader.conf(70);
      first.conf("first",50,0);
      mid.conf("mid",30,0);
      last.conf("last",5,0);
    Links ////////////assembly links//////////
      @dfpos: p-r leader.pos first.pilotpos
      @dfspeed:p-r leader.speed first.pilotspeed
      @dfgoal: p-r leader.goalreached first.isgoalreached
      @dmdist: p-r leader.giveSafeDistance first.safeDistance

      @fmpos: p-r first.pos mid.pilotpos
      @fmspeed:p-r first.speed mid.pilotspeed
      @fmgoal: p-r first.goalreached mid.isgoalreached
      @fmdist: p-r first.giveSafeDistance mid.safeDistance

      @mlpos: p-r mid.pos last.pilotpos
      @mlspeed:p-r mid.speed last.pilotspeed
      @mlgoal: p-r mid.goalreached last.isgoalreached
      @dmdist: p-r mid.giveSafeDistance last.safeDistance
  End // assembly
END_COMPOSITION
```

C.2 Platoon Vehicle

The platoon vehicle is the central component of the system, it is instanciated three times in the system.

A Kmelia component is characterised by a name (the component identifier), a state (variables and an invariant predicate on them), an interface made of *services* and the description of the services. We detail the main elements of the vehicle specification, the full Kmelia specifcation is provide in Listing 5

The state invariant (Listing 2) restricts the speed and position values. Running in a solo mode is not specified here. A running vehicle adapts its position and its speed on its predecessor (its pilot) by requiring the

pilotspeed and pilotpos services as shown in Figure 2. Running vehicles just adapt their position and speed on their predecessor one's by requiring the pilotppeed and pilotpos services as shown by Figure 2. Conversely it provides its speed and pos services. The remaining services are internal, called by the provided services. They compute the new value at each evaluation (a kind of discrete time clock). The vehicle returns in the idle state when called on the stop services. Consequently it has some infinite behaviour.

Listing 2: The state of the SimpleVehicle component

```
COMPONENT SimpleVehicle
INTERFACE
  provides : {run, pos, speed, goalreached, computeSpeed, giveSafeDistance}
  requires : {pilotpos,pilotspeed, isgoalreached, safeDistance}
  config : {conf}
  autorun: {run}
USES {PLATOONLIB}
VARIABLES
  obs goalreached:Boolean;
  vspeed,
  lastpos : Integer;
  vname : String;
INVARIANT
  @borned: 0 <= vspeed and vspeed <= maxSpeed

INITIALIZATION
  goalreached:=false;
  vspeed := 0;
  lastpos := 0;
  vname := "Anonymous";
```

Listing 3 gives the specification of the configuration service that enables to initialise state variables of SimpleVehicle especially in test design.

Listing 3: Configuration service of the SimpleVehicle component

```
provided conf(pvname : String; currentPos : Integer)
Behavior
Init initvi Final initvf
{  initvi -- lastpos := currentPos --> initve1,
   initve1 -- vname := pvname --> initve2,
   initve2 -- display(vname + " is initialised.") --> initvf
}
Post lastpos = currentPos
End
```

Listing 4 gives the specification of the contracts related to the SimpleVehicle component services. Three services are playing, the run service models the running process which controls the vehicle information by calling internally the computeSpeed service and the applySpeed service. These two services compute the new values under safety conditions. Their dynamic behaviour is shown in Figure 2, generated by the COSTO tool. From the service point of view, the functional aspect of the contract is written using assertions (pre/post conditions) and the dynamic aspect covers the communication actions.

The driver is a simplified vehicle that computes its position and speed according to its given (position) goal assuming input devices *e.g.* GPS control. It follows no vehicles but provide useful information to its followers.

Listing 4: The run service of the SimpleVehicle component

```
provided run ()
Interface
  extrequires: {safeDistance, isgoalreached}
  intrequires : {applySpeed, computeSpeed}
Variables
  safeDist:Integer;
  newspeed:Integer;

Behavior
... see figure 2
End

provided applySpeed(thespeed:Integer)
Pre  thespeed >= 0 && thespeed <= maxSpeed
Behavior
... see figure 3
#Post vspeed=thespeed && lastpos = old(lastpos)+thespeed
End

provided computeSpeed(safeDistance:Integer) :Integer
Interface
  extrequires: {pilotpos, pilotspeed}
Pre safeDistance >=0
Variables
distance:Integer;
newspeed:Integer;
```

```
pilotpos, pilotspeed:Integer;
Behavior
... see figure 3
Post
Result>=0 && Result<=maxSpeed &&
((pilotpos-lastpos)<safeDistance) implies Result = 0
End
```

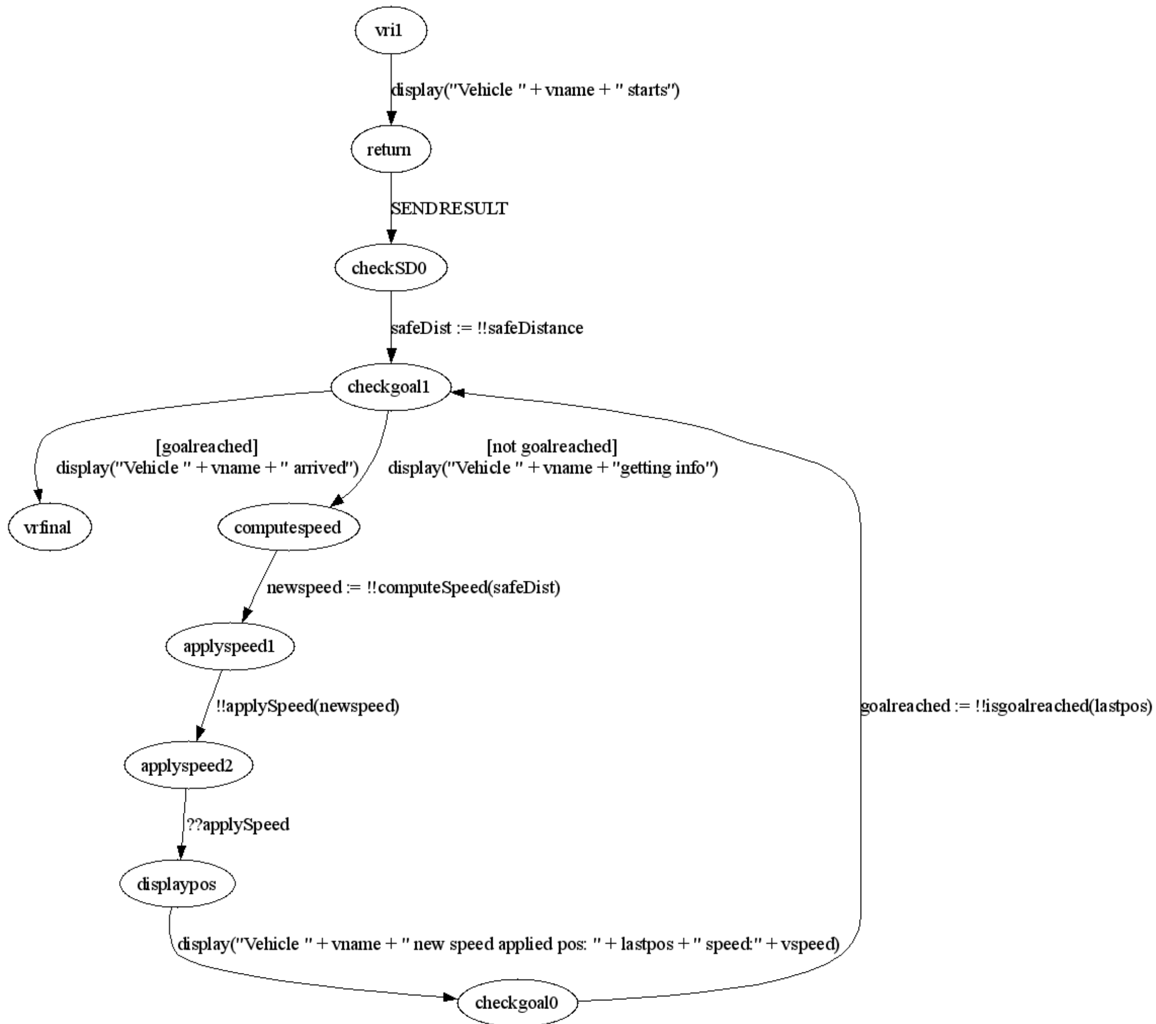

Figure 2: Behaviour of run service in SimpleVehicle

Listing 5: Kmelia specification SimpleVehicle

```
COMPONENT SimpleVehicle
INTERFACE
   provides : {run, pos, speed, goalreached, computeSpeed, giveSafeDistance}
   requires : {pilotpos,pilotspeed, isgoalreached, safeDistance}
   config : {conf}
   autorun: {run}
USES {PLATOONLIB}
VARIABLES
   obs goalreached:Boolean;
   vspeed,
   lastpos : Integer;
   vname : String;
INVARIANT
   @borned: 0 <= vspeed and vspeed <= maxSpeed

INITIALIZATION
   goalreached:=false;
   vspeed := 0;
   lastpos := 0;
   vname := "Anonymous";

SERVICES
########### provided services
```

Figure 3: Behaviour of computeSpeed and applySpeed services in SimpleVehicle

```
provided conf(pvname : String; currentPos : Integer; currentSpeed:Integer)
Behavior
Init initvi Final initvf
{  initvi — lastpos := currentPos —> initve1,
   initve1 — vname := pvname —> initve2,
   initve2 — vspeed := currentSpeed  —> initve3,
   initve3 — display(vname + " is initialised at currentpos: "+currentPos+ "
             speed: "+currentSpeed) —> initvf
}
Post lastpos = currentPos &&vspeed=currentSpeed
End

required safeDistance():Integer
End

required isgoalreached(followerpos:Integer):Boolean
End

provided goalreached(followerpos:Integer):Boolean
//return true if my goal is reached and the caller is within minimal distance of myself
Variables
ok:Boolean;
 Behavior
 Init i
 Final f
 {
 i— ok=goalreached and (lastpos−followerpos<=idealDistance)—>e2,
 e2—SendResult(ok)—>f
}
End

provided run ()
Interface
```

```
  extrequires: {safeDistance, isgoalreached}
  intrequires : {applySpeed, computeSpeed}
Variables
  safeDist:Integer;
  newspeed:Integer;
Behavior
Init vri1 Final vrfinal
{
  vri1 — display("Vehicle "+vname+" starts") —> return,
  return —SendResult() —> checkSD0,
  checkSD0 — safeDist:=!! safeDistance() —>checkgoal1,
  checkgoal0 —goalreached:=!! isgoalreached(lastpos)—>checkgoal1,
  checkgoal1 — [goalreached] display("Vehicle "+vname+" arrived") —>vrfinal,
  checkgoal1 — [not goalreached] display("Vehicle "+vname+"getting info") —>computespeed,
  computespeed — newspeed:= !! computeSpeed(safeDist) —> applyspeed1,
  applyspeed1 — !! applySpeed(newspeed)—> applyspeed2,
  applyspeed2 — ?? applySpeed()—> displaypos,
  displaypos — display("Vehicle "+vname+" new speed applied pos: "+lastpos+" speed:"+vspeed) —> checkgoal0
}
End


provided applySpeed(thespeed:Integer)
Pre  thespeed >= 0 && thespeed <= maxSpeed
Behavior
Init i Final f
{
i— lastpos:=lastpos+thespeed—>e1,
e1— vspeed:= thespeed —>e2,
e2—  SendResult() —>f
}
#Post vspeed=thespeed && lastpos = old(lastpos)+thespeed
End

provided computeSpeed(safeDistance:Integer) :Integer
Interface
  extrequires: {pilotpos, pilotspeed}
Pre safeDistance >=0
Variables
  distance:Integer;
  newspeed:Integer;
  ppos, pspeed:Integer;
Behavior
Init i Final f
{
  i— display("computeSpeed called with safeDistance: "+safeDistance)—>displaystate,
  displaystate— display("Vehicle "+vname+" has lastpos: "+lastpos+" and  vspeed: "+vspeed) —>getppos1,
  getppos1— ppos:=!! pilotpos() —> getppos2,
  getppos2 — display("Vehicle "+vname+" gets pilotpos"+ppos) —> getpspeed1,
  getpspeed1 — pspeed:=!! pilotspeed() —> displayspeed,
  displayspeed—  display("Vehicle "+vname+" gets pilotspeed"+pspeed) —>calcdist,
  calcdist— distance := ppos−lastpos  —> e1,
  e1 — [distance>safeDistance && vspeed<=pspeed] newspeed:=vspeed+distanceStep —> e2,
  e1—  [distance>safeDistance && vspeed>pspeed]  newspeed:= vspeed − distanceStep —>e2,
       // *((distance−safeDistance) div distance)
  e1 —[distance <=safeDistance] newspeed:=0 —>e2,
  e2 —[newspeed<0] SendResult(0) —> f,
  e2 —[newspeed>maxSpeed] SendResult(maxSpeed) —> f,
  e2 — [newspeed<=maxSpeed && newspeed>=0] SendResult(newspeed) —>f
}
Post
  Result>=0 && Result<=maxSpeed &&
  ((ppos−lastpos)<safeDistance) implies Result = 0
End

provided pos():Integer
Behavior
Init i Final f
{ i— _CALLER !! pos(lastpos) —>f }
End

provided speed():Integer
Behavior
Init i Final f
{ i— _CALLER !! speed(vspeed) —>f }
End


provided giveSafeDistance():Integer
Behavior
Init i Final f
{ i— SendResult(10) —>f }
End
//////////// required(/)  (/): In(t/e  (n/d     (d)     red(/)  (/     w     l( )    (w     (/)    g     g       (/)   (/)   (/)   (/)   (/)   (/
```

```
END_SERVICES
```

C.3 Platoon Leader

Listing 6: Kmelia specification SimpleLeader

```
COMPONENT SimpleLeader

INTERFACE
  provides : {pos, speed, goalreached, giveSafeDistance}
  config : {conf}
USES {PLATOONLIB}
VARIABLES
  goal : Integer;


INITIALIZATION
  goal := 0;

SERVICES

provided conf(thegoal:Integer)
Behavior
Init i Final f
{ i-- goal:=thegoal-->f
}
End

########### provided services

provided goalreached(followerpos:Integer):Boolean
//return true if my goal is reached and the caller is within minimal distance of myself
Variables
ok:Boolean;
 Behavior
 Init i
 Final f
 {
 i--  ok:=(goal-followerpos<=idealDistance)-->e2,
 e2--SendResult(ok)-->f
}
End



provided pos():Integer
Behavior
Init i Final f
{ i-- SendResult(goal) -->f }
End


provided giveSafeDistance():Integer
Behavior
Init i Final f
{ i-- SendResult(10) -->f }
End

provided speed():Integer
Behavior
Init i Final f
{ i--SendResult(40) -->f }
End
END_SERVICES
```

D Specification Verification

In Kmelia the components, assemblies and compositions can be analysed according to various facets. The reader will find in [2] a detailed discussion on the multiple kinds of properties to be verified. Among them we focus only on those related to three specification levels (service contract, interactions and behaviour):

1. the *service contracts* are statically checked using the Atelier-B prover, after an extraction/transformation into B specifications, according to the kind of property to check (see [2]);

2. the properties of the *interactions between components* are statically checked using the MEC model checker after an extraction/transformation into MEC specifications, according to the kind of property to check (see [4]);

3. the *service behaviour* dynamic properties are checked using the MEC model checker as above but actions are abstracted because model-checkers cannot handle complex data values in their state exploration without exploding.

The first two levels are checked using the techniques developed in [2] ; the proof details are given in the web appendix. The third level (conformance of the behaviour against the contract) is hardly verifiable because of potentially infinite behaviour of an LTS (in a theorem proving attempt) and the potentially infinite set of values of the data (in a model checking attempt). We unsuccessfully explored both tracks in previous work. We propose to engage a different vision using testing approaches.

E An Experimentation outlook

In this section we illustrate the step by step harness construction. The support example

E.1 Harness construction

The test intention focuses on the goal of the test. Here we want to test the conformance of the computeSpeed (safeDistance : Integer) : Integer service of the mid component with the following *safety property*: *the distance between two neighbour vehicles is greater than a value minDistance*. So we provide a "rich" test intention with input data and oracle predicate (Listing 7).

Listing 7: Provided service computeSpeed

```
TEST_INTENTION PlatoonTestIntention
DESCRIPTION "the vehicle will stop if it is too close to the previous one"
INPUT VARIABLES
```

iPP"PP0BI RBES

p0 0 0s

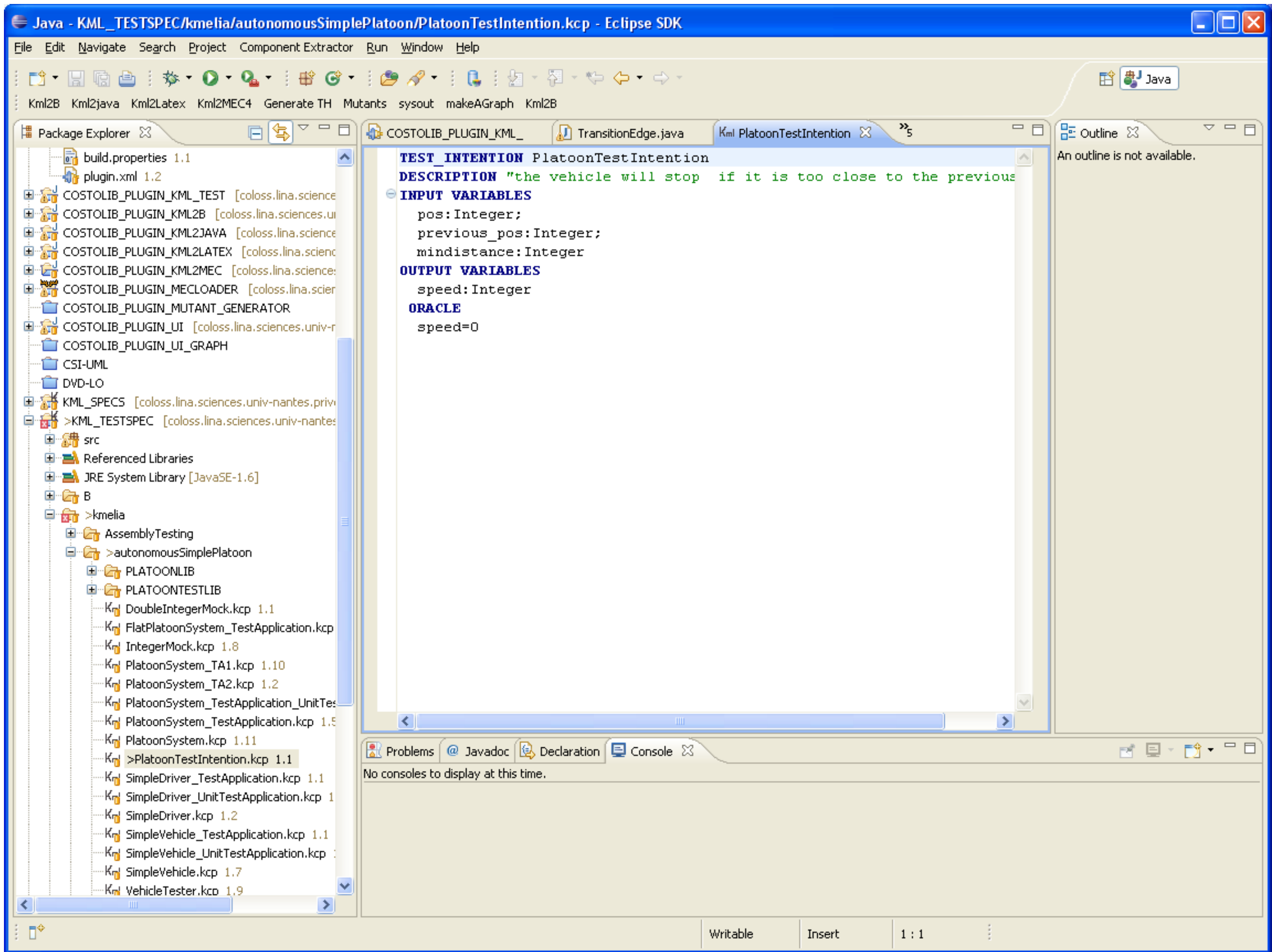

Figure 4: Launching the construction from the test intention

It opens the *Test Harness Creating Process* window (Figure 5).

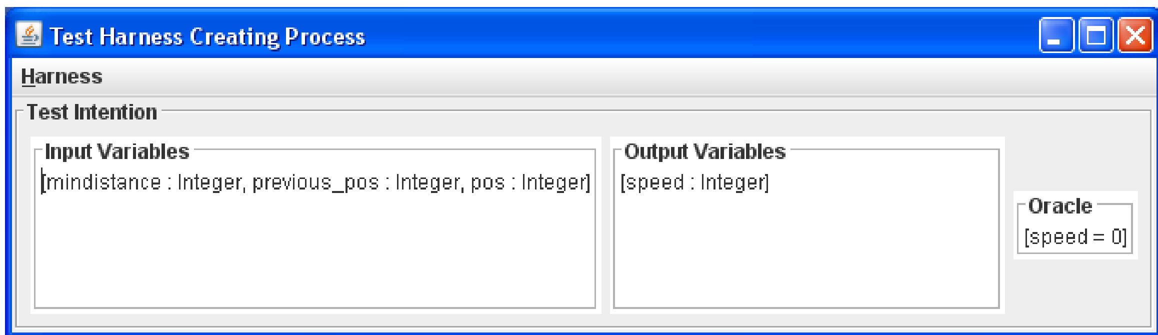

Figure 5: Test Harness Creating Process window

Then we select the system under test (the PIM actually) by calling the `Harness>Select Target` menu command (Figure 6).

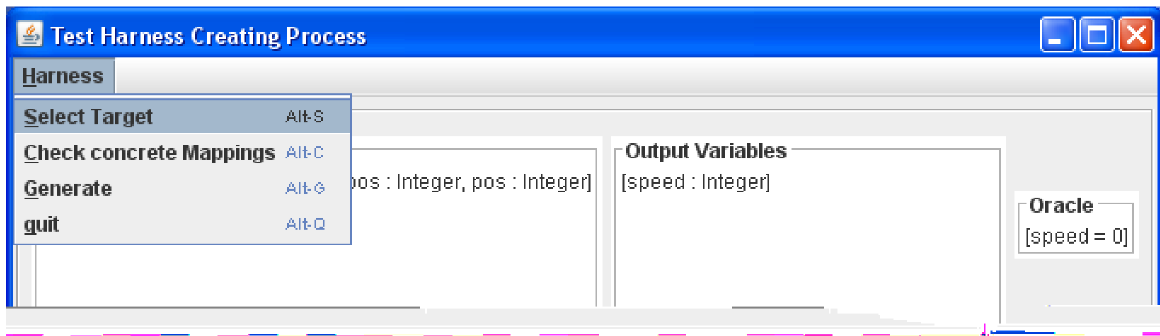

Figure 6: Selecting the PIM target

E.3 Selecting the services under test

In the target system, the test designer choose which (component and) services will belong to the system under test (SUT) (Figure 7).

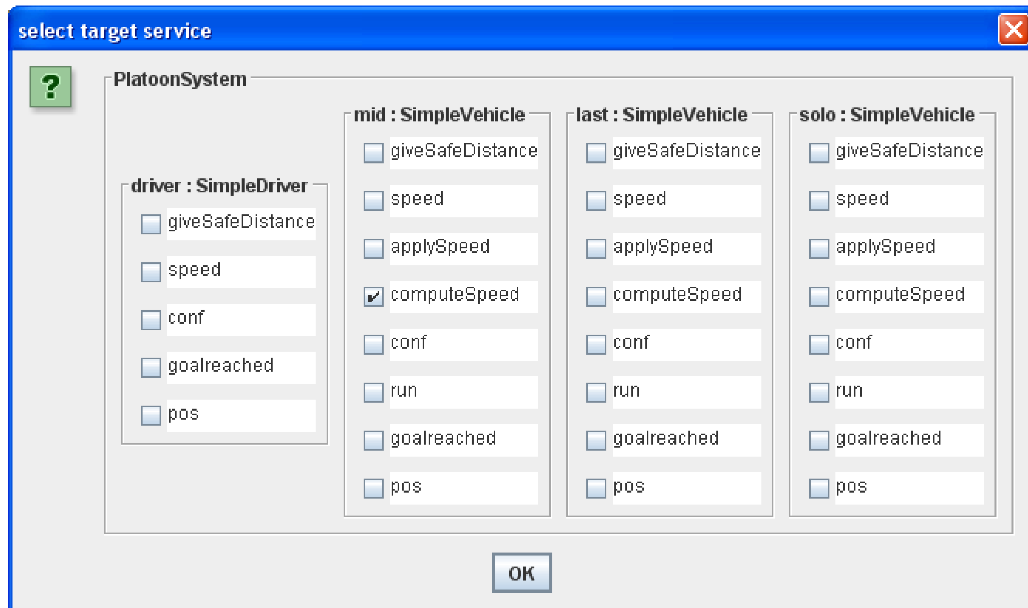

Figure 7: Selecting the PIM target

E.4 Consistency and Completion process

Once the original elements are selected, the building process really starts. The tool displays the board to match the test intention with the SUT (Figure 8).

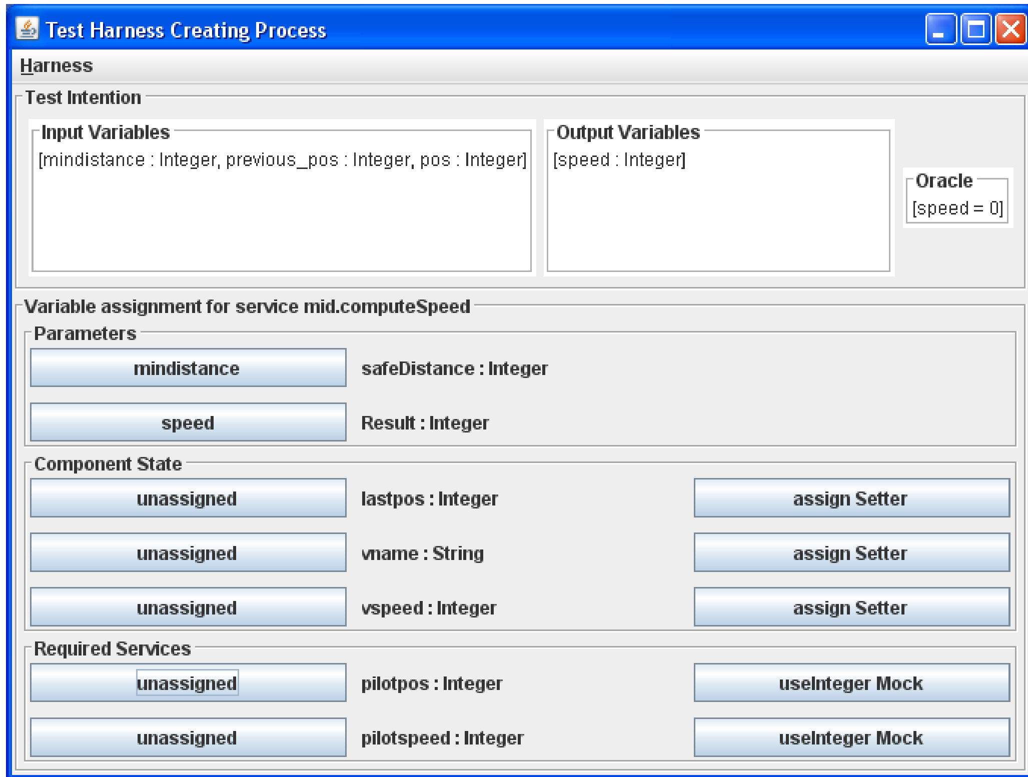

Figure 8: Component and service notation

The test designer binds test data to the test application (TSM) which includes the components of the SUT and test components added by the test designer. The assistant proposes alternatives to the test designer. The system checks the executability properties and proposes candidates (from the libraries) for the missing elements.

Service computeSpeed running depends on the following data: the security distance safeDistance between vehicules, the current speed vspeed and position lastpos, and the predecessor's speed and position. This information can be found in the initial specification of the service or can be extracted by an assistant as illustrated in the Figure 9(a). To complete the test intention, the test designer provides an oracle on these data and on the result of the service running. In our example, the oracle compares the output with the expected output. The assignment of concrete data in such a test case is a vector of values associated to data:
lastpos=10; vspeed=0 pilotpos=10; pilotspeed=80; safeDistance=5; oracledata=45.

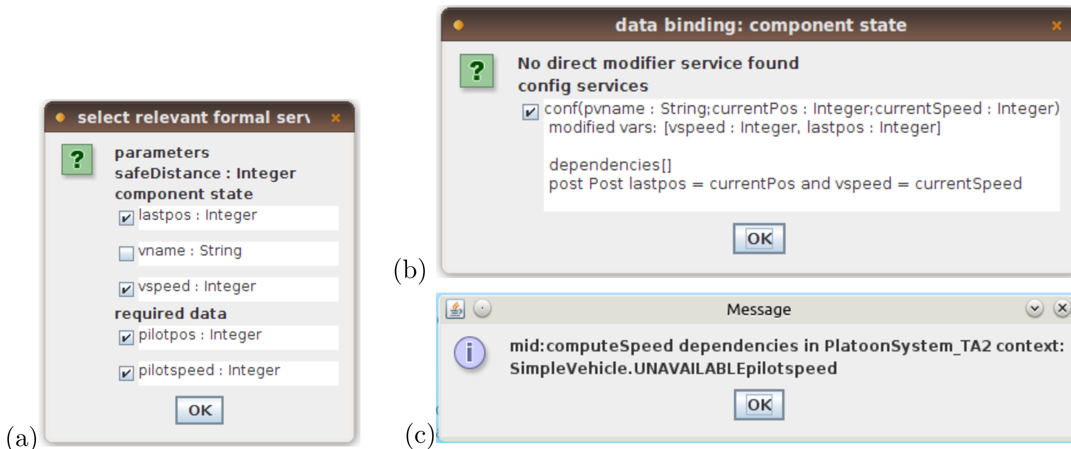

Figure 9: Harness Design Assistants

The harness building process is not entirely automatised. The plugins contain designer assistance functions to add missing dependencies, calculate the service formal data, to propose effective data source for the formal linking of incomplete data (throw the predefined functions libraries), to find the service required to access part of the state domain or to modify a variable. Testing the computeSpeed service, the steps are:

1. The harness is build selecting the SimpleVehicle component to test computeSpeed
2. Service dependencies analyse of computeSpeed identifies two required variables pilotpos and pilotspeed. The process proposes two possibilities:
   (a) Embed the first component into the SUT (Figure 1) and maintain the the current assembly links. In the next iteration, it would then be necessary to satisfy the services required by first which are detected by the verification V1 (illustrated Figure 9(c)).
   (b) Put a mock component. The assistant would suggest two mocks to generate integers. We choose to develop our own mock component im1.
3. Building the test pilot vtd, we iterate on the environment building:
   (a) The assistant generate a service call and we associate the safeDistance data to it.
   (b) Then it is necessary to find a way to initiate the component state and to associate the lastpos and vspeed=0 data to it. For this, the assistant (Figure 9(b)) list the services helping to modify directly or not the state variables. It proposes the initialisation service `conf`, and precise its signature, its dependencies, and additional information (for instance here a post-condition)
4. The oracle calculation is provided by an expression inspired by the post-condition.
5. The result is compared to the value returned by the oracle.

The (system) configuration clause includes the (configuration) service call of component mid and the mock im1. The variables vp et vs of the test intention are bound to the component variables.

The test intention is realised by :

1. a test driver vtd which provides the testcase1 describing the test sequence and the oracle predicate that call the primitive function assert (Listing 8).

   Listing 8: Service describing the test case of the test driver VehicleTestDriver

```
provided testcase1()
Interface
  extrequires: {computeSpeed}
Variables
  computespeedresult:Integer;
Sequence
{ computespeedresult := !!computeSpeed(getData("safeDistance"));
  //init sequence call
  verdict := (computespeedresult = getData("oracledata") );
  //oracle evaluation
  assertT (verdict);      //transmit verdict
  SendResult()    //end of test service
}
End
```

2. a library of concrete functions (included in the PDM) and types
3. a mapping from the abstract (Kmelia) primitives to the concrete functions and types, for example the following lines establish a mapping between the `mylib` Kmelia library and the Java code of the PDM.

```
getData=mylib.PlatoonTestlibMap.getData
assertT=@WithSelf mylib.PlatoonTestlibMap.assertT
```

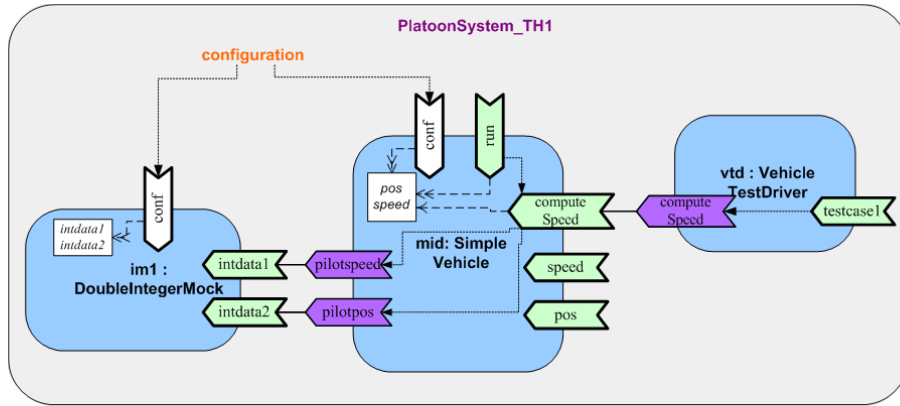

Figure 10: A TSM : test harness for the mid component's service computeSpeed (SUT)

E.5 Test execution

The target platform is a Java runtime framework based on a communication framework where services are threads, component and channels are monitors. The channels have buffers handled by threads in a synchronous interpretation with rendez-vous. The runtime framework metrics are: 7 packages, 50 classes, 540 methods, 3400 LOC.

The data values are provided with text files. To establish the test data of service computeSpeed, we used coverage techniques based on the control flow the (service) behaviour state transition system [5]. We identified 9 paths to cover : 4 cannot be reached by the date, the remaining 5 are reached by 3 test cases each. The 15 test cases have been executed with success.

The executable code is obtained by a Java code transformation which produces generated traceable code over the runtime framework. Each concept of the abstract component model has a concrete representation in the runtime framework which ensure the readability of the execution and the high-level error information. After the code generation, the java generated application is executed and returns the verdict as illustrated by Figure 11.

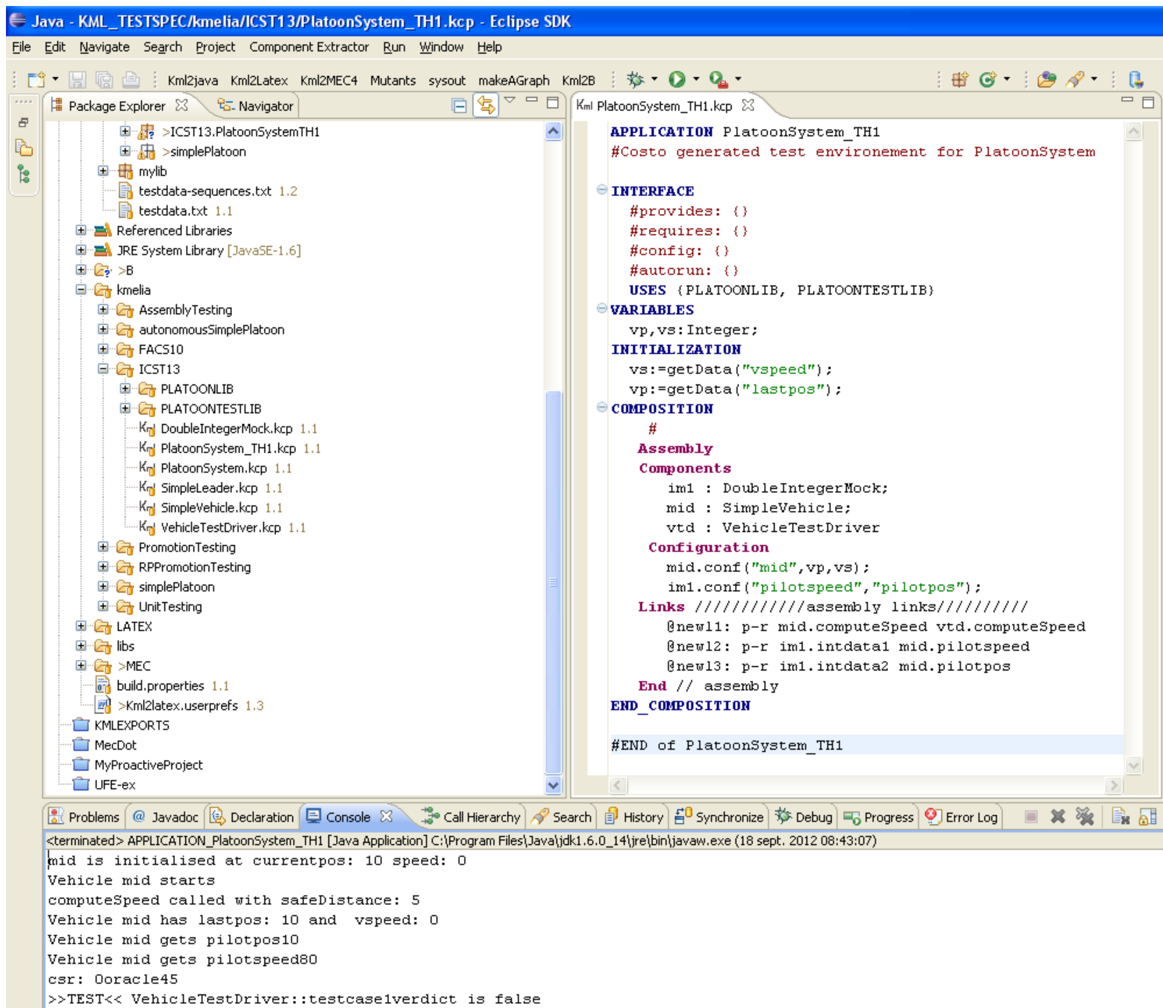

Figure 11: Snapshot of a running test application

References

[1] S. Colin, A. Lanoix, O. Kouchnarenko, and J. Souquières, “Towards validating a platoon of cristal vehicles using csp||b,” in *AMAST*, 2008, pp. 139–144.

[2] P. André, G. Ardourel, C. Attiogbé, and A. Lanoix, “Using assertions to enhance the correctness of kmelia components and their assemblies,” *ENTCS*, vol. 263, pp. 5 – 30, 2010, proceedings of FACS 2009.

[3] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, Eds., *The Common Component Modeling Example: Comparing Software Component Models*, ser. LNCS. Heidelberg: Springer, 2008, vol. 5153.

[4] C. Attiogbé, P. André, and G. Ardourel, “Checking Component Composability,” in *5th International Symposium on Software Composition, SC'06*, ser. LNCS, vol. 4089. Springer, 2006.

[5] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.