INTEGRATED SPECIFICATION, VALIDATION AND VERIFICATION WITH HYBRIDUML AND OCL APPLIED TO THE BART CASE STUDY

Stefan Bisanz, Paul Ziemann, Arne Lindow
University of Bremen, FB3 (Department of Mathematics and Computer Science)
Address: Postfach 33 04 40, D-28334 Bremen, Germany
Phone: (+49-421) 218-3337, Fax: (+49-421) 218-3054,
e-mail: {*bisanz,ziemann,lindow*}*@informatik.uni-bremen.de*

Abstract: This article proposes the integration of the HybridUML specification formalism and the USE approach for validation of invariant constraints and verification of system states. The benefit is an executable real-time simulation with an integrated verification/validation component, which combines the advantages of the previously separate approaches by providing an accurate, (partially) time-continuous model that can be checked for consistency between static invariants and dynamical behavior in terms of a complete UML model. The integration is illustrated by means of a train system specification – the BART case study.
Keywords: OCL, USE, Time-Continuous Specification, HybridUML, Railway

1 INTRODUCTION

The Unified Modeling Language (UML) [12, 13, 15] is accepted as the standard modeling language in software engineering. It is a graphical language comprising several types of diagrams for modeling different aspects of a system. The Object Constraint Language OCL [5] extends the diagrammatical part of UML with a textual language for formulating additional constraints on the model and thus specifying detailed aspects that cannot be expressed graphically.

A wide variety of CASE tool support exists for UML in general, however, OCL is supported by only a few UML tools. Among the first was the system USE (UML Specification Environment) [14], which is a tool for verifying and validating UML models consisting of class diagrams and OCL invariants and pre- and postconditions. Verification and validation is performed manually by the user by creating and manipulating system states in terms of UML object diagrams and letting the tool check the OCL constraints. The combination of a particular system state with a set of constraints that are not satisfied are either interpreted as (1) a faulty system state which is verified against the given constraint specification, or as (2) a faulty constraint that is exposed by a correct system state. The latter contributes to the validation of the constraint specification. To facilitate this process, USE allows to group state manipulation commands into procedures.

HybridUML [3, 4] is an extension of UML that facilitates the modeling of systems with discrete and time-continuous aspects – so-called *hybrid systems.* The need to model time-continuous behavior arises in the context of (often embedded) real-time systems which interact directly or indirectly with their physical environment. Controllers themselves behave inherently discretely, whereas many aspects of the physical environment appear in an analog fashion. The environmental specification, i.e. the expected behavior of the environment of such systems is an important part of their specification. The first benefit of HybridUML is the ability to model the environment as it is, rather than a discretized interpretation of the environment. Further, controllers observe and affect this environment, which ideally would take place infinitely fast and thus would proceed approximately time-continuously. The deviation of a real controller's discrete implementation from the ideal time-continuous execution is mainly given by the controller's hardware capabilities, therefore it is desirable to create a controller model that abstracts from the particular hardware restrictions, in the same way as the environment is modeled.

HybridUML extends the concept of Hybrid Automata (see e.g. [11]); more precisely, it is based on hierarchical hybrid specifications as proposed in [1]. The latter introduces the notion of hierarchy and therefore facilitates the application to large-scale systems. The widely known hierarchical state machines without continuous time called Statecharts were introduced in [9]; a formal semantics is given in [6]. The Duration Calculus, which is not suitable for direct adaption to the graphical UML, is a noteworthy formalism in that it contributes fundamentally to the understanding of hybrid systems. [18] Further presentations of projects and results in the field of hybrid systems are subsumed by the DFG priority programme Kondisk. [8]

Technically, HybridUML is a *UML 2.0 profile*, i.e. a collection of UML stereotypes that are applied to existing UML 2.0 language elements, because UML itself does not provide adequate means for modeling hybrid systems (see [2] for a survey). However, standard UML benefits like wide acceptance in computer science and engineering as well as tool support are combined with particular important features for (hybrid) systems modeling: (1) Formal semantics are assigned by a transformation to a semantically well-defined low-level language. (2) Transformed models are directly executable. There is no gap between the semantics of the model and its implementation, because the encoded behavior *is* the semantics of the model.

In [19], the abilities of USE were demonstrated by specifying, validating and verifying the train system known from the BART case study. The BART case study description [17] informally describes a portion of the Advanced Automatic Train Control (AATC) system that is being developed for the Bay Area Rapid Transit (BART) system. BART provides a commuter rail service for part of the bay area of San Francisco, California. The overall objective of the case study is to specify a system within the given infrastructure, that controls speed and acceleration of trains in the system. This system has to respect several constraints, two central ones being the following safety requirements: (1) A train should never get so close to another train in front that if the train in front stopped suddenly, the (following) train would hit it. (2) A train should stay below the maximum speed that the current segment of the track can handle. Related case studies are, e.g. the "generalized railroad crossing" [10] or the "radio-based train control" [4]. The latter is one of two case studies within the scope of the DFG priority programme Software Specification. [7]

The specification in [19] comprises three parts: One part describes the structure of the system by means of a class diagram and a set of OCL invariants, which in particular include the safety constraints mentioned above. The second part specifies the behavior of an operations center[1] in terms of an algorithm for calculating safe accelerations and speed commands for the trains. This is given in terms of system state manipulating procedures, which are defined in a Pascal-like notation and make heavy use of OCL expressions. In order to test the specification, the third part specifies how the system state changes after 0.5 seconds have passed, i.e., the physical behavior of trains is specified. The specification parts were tested against each other by repeatedly calculating new commands for the trains on a sample track and moving the trains ahead in time and place. The downside of this specification technique is that the behavioral and physical specifications are quite

[1] Previously called "station computer".

detached from the structural specification.

In this paper, the OCL as it is utilized by the USE tool is merged with HybridUML, therefore the main benefits are:
(1) The usage of continuous time provides a more accurate model, containing a more realistic description of the physical environment.
(2) The behavioral specification is modeled by means of UML, i.e. it is defined by state machines. These are more comprehensible than the a.m. state manipulating procedures. The separation into parallel hierarchic state machines additionally facilitates the handling of the complexity of the behavioral description.
(3) The complete system specification is unified within a single (UML) model. The behavioral specification is attached to the respective classes, OCL constraints are integrated into the corresponding state machines.

The aim is to compile this complete specification into an executable real-time simulation. During execution, the OCL expressions included in the specification are evaluated by the USE component. If the simulation runs into a deadlock because an invariant is violated and no valid action is available, an inconsistency is found and the modeller can be supported in inspecting the relevant part of the model.

In the following sections, (parts of) the BART case study are specified by means of HybridUML and OCL.[2] In section 2, the static structure is modeled. Section 3 adds the behavioral specification, which includes several kinds of OCL constraints. In section 4, the execution semantics are sketched. Finally, the simulation-integrated verification and validation concept is discussed in section 5.

2 ARCHITECTURE

Within the case study, the main building blocks are:

[2] For the purpose of this article, several simplifications are applied to the case study.

TrainModel. There are physical trains that move along physical (track) segments. This constitutes the embedding environment of the computer system.
OperationsCenter. The operations center is the central part of the computer system.
TrainController. A local controller is located on each train. The operations center calculates commanded accelerations and velocities for each train, whereas the local controllers simply perform some safety checks on the commands from the operations center before applying them.

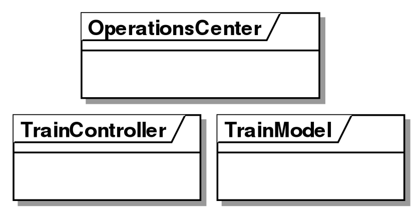

Fig. 1: Packages of the BART model.

The most abstract UML view of the complete model is the set of corresponding packages which is shown in Fig. 1. The static structure of the train model is shown in Fig. 4, and Fig. 2 contains the class diagram of the operations center: An OperationsCenter contains one RemoteTrainController for each physical train to be controlled. Each RemoteTrainController consists of several *basic agent* instances. A basic agent instance is an active object, i.e. an object with a separate thread of control. In contrast, *composite agents* like RemoteTrainController only subsume other agents and have no own behavior. Its components are:
RangeController. This component continuously calculates a velocity-dependant distance. It must be long enough, so that all further calculations for the train's movement only need to consider track segments and obstacles within this range.
SegmentController. This controller keeps track of relevant segments, e.g. the segments

in range are calculated here. Segments may be closed, i.e. the train is not allowed to *leave* the respective segment.[3] The nearest of such segments in front of the train is determined, too.

ObstacleController. The obstacle controller computes a potential acceleration[4] for the train based on obstacles, which are (a) the next closed segment provided by the segment controller and (b) the train ahead. Of course, corresponding hazards should be avoided, therefore the resulting speed must be low enough.

CivilSpeedController. The civil speed controller calculates a proposed acceleration that takes the civil speeds (i.e. the maximum allowed speeds) of the segments in range into account. Again, too high speeds should be avoided.

AccelerationController. It always chooses the lower of both calculated accelerations.

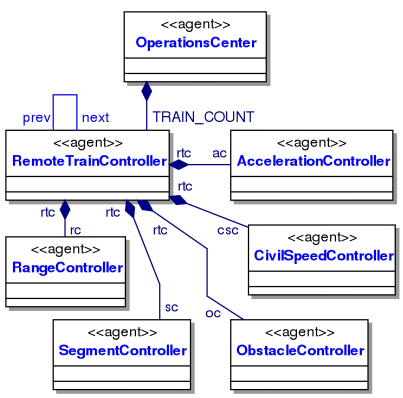

Fig. 2: Architecture of the Operations Center.

Since version 2.0 of UML, the composite structure of classifiers can be defined and illustrated by means of Composite Structure Diagrams. With the HybridUML profile, this diagram type is utilized in order to assign shared variables (and signals, which are not discussed here) to agents, i.e. there are variables that are accessible by several agents. Within class diagrams, these variables appear as attributes of the respective agent. Therefore, shared variables and signals provide mechanisms for interobject communication.

See for example the composite structure view of RemoteTrainController in Fig. 3. Here it is defined that RemoteTrainController provides two variables a_cmd and v_cmd, which is shown by the black rectangular port symbols at the bottom of the diagram. They are defined in the model of the class diagram of Fig. 2 (but not shown there, because the agents' details are hidden in the diagram). Both originate from the contained ac:AccelerationController, i.e. both agents share the respective variable. Since RemoteTrainController is a composite agent and therefore has no own behavior, the value of a_cmd and v_cmd has to be calculated by ac. ac itself shares further variables with csc:CivilSpeedController and oc:ObstacleController which provide each a proposed acceleration (a_civil_cmd, a_obstacle_cmd) and an aimed velocity (v_civil_cmd, v_obstacle_cmd). These variables are only read by ac, as indicated by the *white* rectangular port symbols. In the same fashion, rc:RangeController and sc:SegmentController distribute the (visibility) range s_range and the end of the next closed segment x_end_nextClosedSeg.[3] Further, sc prepares the civil speed v_civil[s] and an acceleration a_civil[s] for each segment s in range. The acceleration a_civil[s] is calculated such that the train's velocity would be v_civil[s] within the current distance to the segment. Each of these variables are read by csc, which chooses the appropriate (i.e. the lowest) acceleration and the corresponding velocity. Finally, note that constants can be modeled by only using white ports, as is done with dot_a_max.

[3]In the original case study description, a closed *gate* corresponds to a closed segment.

[4]The acceleration can be negative, i.e. it can be a deceleration.

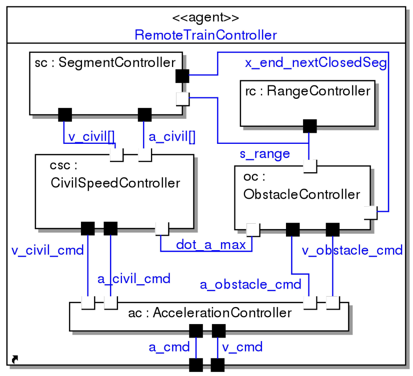

Fig. 3: Composite Structure view of the Remote Train Controller.

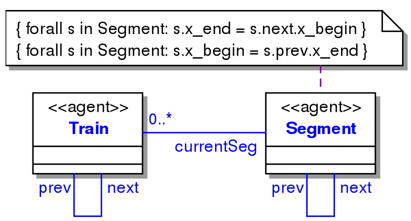

Fig. 4: Architecture of the Physical Train Model.

3 BEHAVIOR

The behavior of each basic agent is defined by a single *top-level mode* which is a special (i.e. a stereotype of) StateMachine. In addition to conventional (discrete) statechart features, a mode contains (1) algebraic and flow constraints as well as (2) invariant constraints. Algebraic and flow constraints define time-continuous evolutions of variables by means of algebraic or differential equations, which are of the form <variable> = <oclexpression> or dot(<variable>) = <oclexpression>, respectively. While the running system resides in the corresponding mode, the values of the respective variables are enforced to accord to these constraints. An invariant constraint controls if the system is allowed to reside in a certain mode. Technically, iff the invariant constraints of all active modes evaluate to true, time can pass. Otherwise, only discrete steps (i.e. firing of transitions) – which are assumed to take no time – are allowed, until a mode configuration is reached so that all invariant constraints are satisfied.

Figure 6 shows the behavior of ObstacleController. Since its top-level mode Running is always active, the variables x_obstacle and s_obstacle are continuously updated: x_obstacle is the absolute position of the nearest obstacle on the track. Besides the nearest closed segment in range, the preceding train (if there is one) and the destination segment (i.e. the target of the journey) is considered. This algebraic constraint is specified using OCL: the subexpression Set(rtc.sc.dest.x_end) uses navigation in the current system state (via the RemoteTrainController instance to the SegmentController instance) to construct a set including only the position of the destination of the controlled train. Then the position of the next closed gate (kept in the attribute x_end_nextClosedSeg) and the position of the next train's rear (rtc.next.x_act-rtc.next.l) are added to this set. The min() operation is not predefined in OCL, it is rather used here as a shortcut for a larger expression that evaluates to the minimal element of a set of real numbers, ignoring undefined elements. Note that the evolution of x_obstacle defines mathematically a continuous function with respect to time, except for discontinuous points that (mandatorily) coincide with discrete steps defined by SegmentController's behavior (which is not shown here).[5] From x_obstacle, the relative position s_obstacle is directly being derived.

Inside Running, the obstacle controller distinguishes between two modes: Decelerate is active, iff there is an obstacle in range, otherwise Accelerate is the active submode. This is enforced by the combina-

[5] It is the end position x_end_nextClosedSeg of the nearest closed segment in range that introduces discrete points to the function.

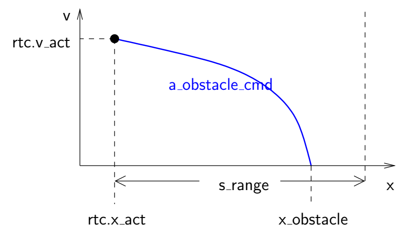

Fig. 5: Snapshot of calculation of acceleration for obstacle in range

tion of the respective invariants and transition guards, which model an urgent mode change. Without the mode invariants, the transitions would have been non-urgent, i.e. they would not have been mandatory to fire when enabled.[6] While decelerating, the required (negative) acceleration is calculated such that the train stops exactly in front of the obstacle. Particularly, an additional acceleration that originates from the segment's (physical) grade is taken into account. The target velocity is constantly set to zero. Figure 5 illustrates the velocity evolution resulting from the calculated acceleration a_obstacle_cmd; rtc.x_act is the actual position of the train, rtc.v_act is the actual speed.

In the mode of acceleration, the target velocity is set to v_max which is the maximum speed of the train. It is assumed that the Agent CivilSpeedController always overrides the target speed by a lower value, therefore v_max will not become effective. The applied acceleration is progressive, thus the acceleration itself increases while accelerating.[7] This is directly specified within a *flow* condition by its derivative dot(a_obstacle_cmd) with respect to time. This dot notation is introduced here as an extension of OCL. Expressions using this extension can be evaluated numerically in an environment comprising a sequence of system states representing the system's configuration at succesive points in time, i.e. during a discretized execution of the model. Since HybridUML's low-level language is based on C/C++, tools like Matlab can be used to create the appropriate code.

The behavior of the basic agent CivilSpeedController is shown in Fig. 9. Its specification is more elaborate than that of ObstacleController, but the applied language features are the same.

It determines an appropriate commanded acceleration a_civil_cmd that does not cause a violation of civil speeds. Therefore, two aspects are taken into account: (1) the civil speed of the current segment, and (2) the civil speeds of the following segments in range. Assuming that the train runs slower than current civil speed (minus a relative velocity dv_safe which defines a lower bound for a range of critical velocities), the submode AdjustingForCivilSpeedAhead is active. The incoming transition activates the mode, and the mode's invariant condition allows the controller to stay there. Within the mode, the resulting acceleration a_civil_cmd is set to the minimum acceleration from the set of accelerations a_civil[]. Each acceleration of this set corresponds to the civil speed of a segment in range. The calculation is based on the segment's distance from the train. AdjustingForCivilSpeedAhead is illustrated by a scenario, shown in Fig. 7 as a graph of the velocity with respect to the position on the track: Provided that the train's position is rtc.x_act and its velocity is rtc.v_act, there are three segments in range.[8] From the distances s_civil[] (between the train and the respective start positions of the segments in range) and the velocities v_civil[], the accelerations a_civil[] are calculated. They result in the shown velocity evolutions. These values are provided by agent SegmentController and are based on the formula `(segInRange.at(s).v_civil^2 - rtc.v_act^2) / (2*s_civil.at(s))`.

The agent CivilSpeedController keeps

[6] A sketch of HybridUML's simulation semantics is given in section 4. For a detailed description, the reader is referred to [4].

[7] The acceleration itself changes constantly, given by dot_a_max.

[8] The current segment technically is *not* in range.

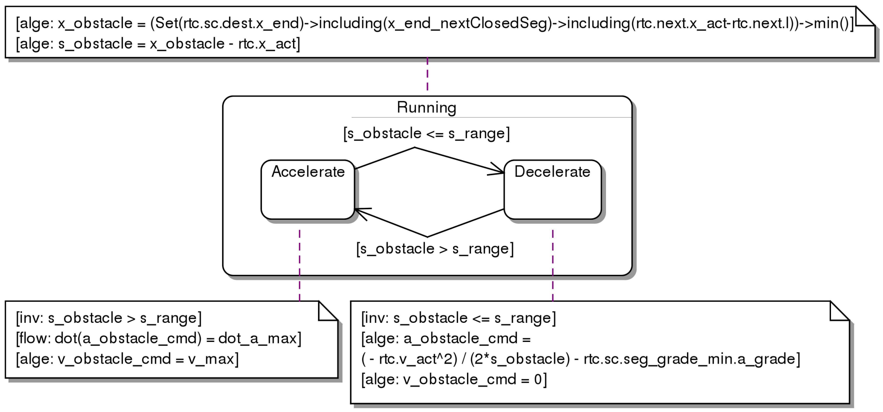

Fig. 6: Behavioral specification of the Obstacle Controller.

track of the minimum acceleration a_civil_min by updating it every time a lower acceleration is provided through a_civil[]. This is modeled by the transition connected to mode Running and its invariant. The calculation is done by recalcCivilMin, which is specified by its post condition:

```
context
CivilSpeedController::recalcCivilMin
post:
index_civil_min =
Sequence{1..MAX_SEG_IN_RANGE}->iterate
(i; i_min = 1
 | if (a_civil.at(i) <
       a_civil.at(i_min)
   then i
   else i_min)
and
a_civil_min =
  a_civil.at(index_civil_min)
and
v_civil_min =
  v_civil.at(index_civil_min)
```

In submode AdjustingForCivilSpeedAhead, the commanded acceleration a_civil_cmd is adjusted by an additional acceleration which is expected to be effective on the segments in range, due to their physical grade. From the segments in range, the minimum grade acceleration is chosen. Further, the civil speed v_civil_min that corresponds to a_civil_min is chosen as the commanded velocity.

If the actual velocity of the train is getting critical, i.e. if it is at least the civil speed of the current segment minus dv_safe (`rtc.sc.currentSeg.v_civil - dv_safe`), then the CivilSpeedController changes to mode RestrictingToCurrentCivilSpeed and stays there unless the actual velocity falls below this speed again. Within RestrictingToCurrentCivilSpeed, the controller commands either (a) Maintaining of the current velocity or (b) Decelerating. The cooperation of these two submodes is modeled by their invariant constraints and the two transitions between them: Maintaining may be active, as long as the current speed is below the civil speed. *Exactly when* the civil speed is reached, Decelerating becomes active. Deceleration *may* proceed as long as the velocity is critical, but *can* be pre-empted by Maintaining. This way, the transition from Maintaining to Decelerating is modeled as *urgent* transition, whereas the transition originating at Decelerating is *non-urgent*.

While Decelerating, the maximum de-

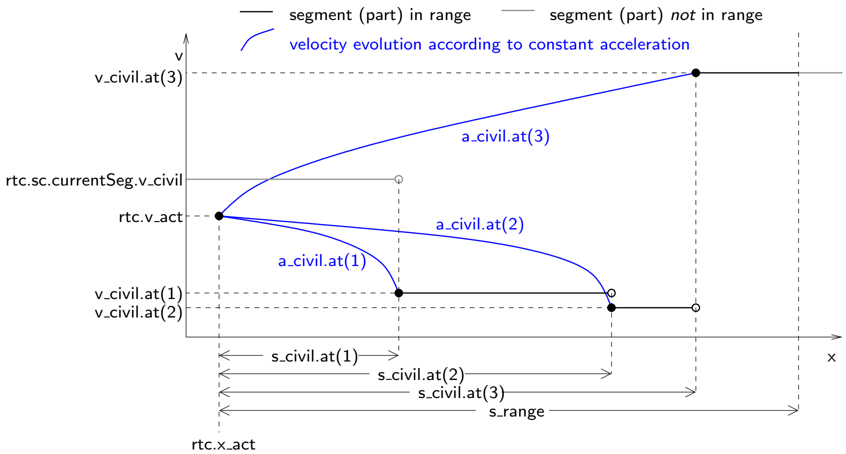

Fig. 7: Snapshot of calculation of acceleration for civil speed

celeration is utilized in order to reach the lowest critical speed (again). In mode Maintaining, a zero acceleration is applied. Since here no additional grade acceleration is considered, the train may particularly become faster, and therefore a mode change to Decelerating can occur again. Figure 8 shows an example of how the restriction to the current civil speed is realized: The assumed setting consists of two segments with the same civil speed, but with different grades segment.a_grade $= 0$ and segment.next.a_grade > 0. First, the controller adjusts for the civil speed of a segment in front (1). Then, the current speed is maintained and remains constant, because no additional acceleration is effective (2). On the next segment, the grade acceleration increases the velocity up to the civil speed (3), which results in decelerating (4). A second iteration of maintaining (5) and decelerating (6) occurs, but the controller now decides to switch to Maintaining before the lower velocity bound is reached (7).

As mentioned earlier, the *physical environment* is an integral part of the complete model. It is specified in the same way as the controller part(s), by means of agents and modes. For example, the expected behavior

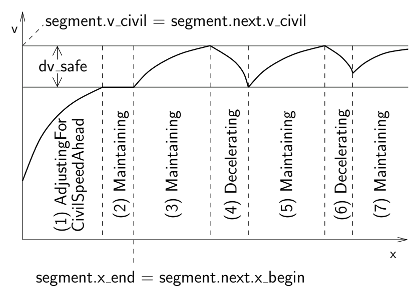

Fig. 8: Illustrated functionality of submode RestrictingToCurrentCivilSpeed of the Civil Speed Controller.

of the physical train is shown in Fig. 10, which is associated with the basic agent Train of Fig. 4.

The train is assumed to be either accelerating or braking. It gets the required (positive or negative) acceleration a_ctrl from its controller and decides, depending on the acceleration's algebraic sign, whether to apply propulsive power or to activate the brake. In any case, the train satisfies physical laws: (1) Its overall acceleration is the sum of all effective accelerations – the braking deceleration, the propulsive

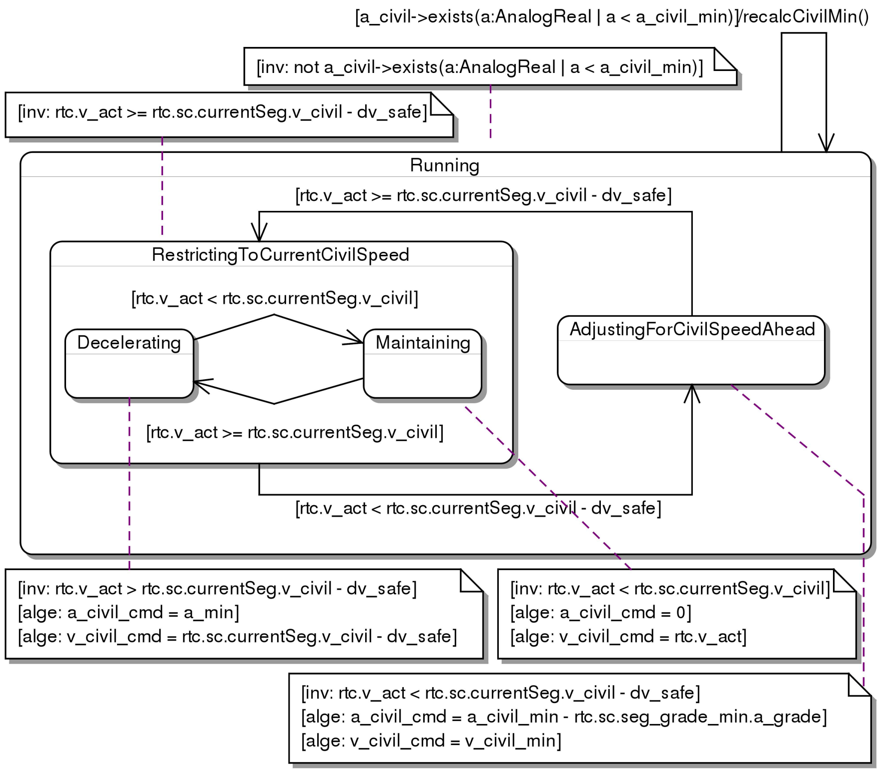

Fig. 9: Behavioral specification of the Civil Speed Controller.

acceleration and the environmental acceleration introduced by the current segment's grade. (2) The train's velocity always depends on the overall acceleration, i.e. its rate of change *is* the acceleration. (3) The train's position always is the consequence of its velocity, i.e. its alteration rate *is* the velocity. Additionally, the position of the train implies the current segment on which the train is located. This is modelled by both the transition and the invariant attached to mode Running.

There are two top-level invariants which define fundamental safety constraints for the complete model:
(1) The train's velocity is never allowed to exceed the current segment's civil speed.
(2) The train must not touch (i.e. crash into) the train in front.
The benefit of these invariants is discussed in section 5. Invariant (2) uses a side-effect free operation wcsd() of the Train instance that calculates the worst case stopping distance of that train. It is specified by an OCL post condition:

```
context Train::wcsd():Real post:
return = -(v^2)/
  2*(a_wcsp+currentSeg.a_grade)
```

It is assumed that even under the worst condition a train is able to brake with a deceleration of a_wcsp. The acceleration

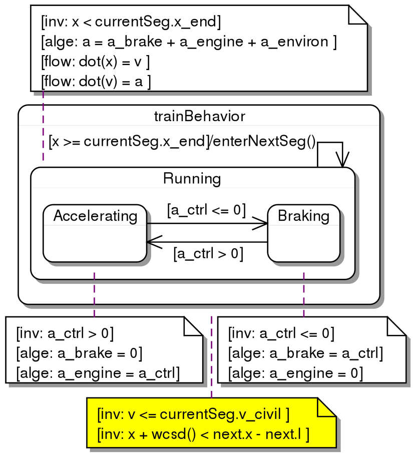

Fig. 10: Expected Behavior of the Physical Train.

caused by the grade of the current segment is added to this value.

4 SEMANTICS

An execution of a HybridUML model is an interleaving of discrete and continuous steps. A discrete step is taken by a single basic agent by means of firing one transition. When all invariants of all agents' actual mode configurations are satisfied, a continuous step can be taken – all agents concertedly let time pass and apply the currently active algebraic and flow conditions. Thus, the execution semantics of a HybridUML model is defined as follows; for a detailed description, the reader is referred to [4].
(1) It is determined if a continuous step would be admissible, i.e. the active invariants are checked. If any invariant is violated (i.e. evaluates to false), then no continuous step is allowed and therefore time cannot pass.
(2) For each agent, the set of enabled transitions is calculated.
(3) If at least one transition is enabled, one enabled transition is chosen non-deterministically. It constitutes the possible discrete step.
(4) The choice between discrete and continuous step is non-deterministic, if both are possible. If only one is allowed, that one is applied. If neither is possible, the simulation is deadlocked.
(5) The chosen step is taken, the respective calculations are performed and the execution continues with enumeration step (1).
Note that not only strict alternations of discrete and continuous steps are allowed, but also are sequences of discrete steps and sequences of continuous steps.

Amongst discrete steps, there is *no* prioritization. This design decision is made for two reasons: The semantics are easier, and the dispute for the right prioritization[9] is avoided.

5 VERIFICATION AND VALIDATION

A shortcoming of HybridUML's execution semantics – which is no speciality of HybridUML but a usual problem with behavioral specifications – is the possibility of deadlock as mentioned in enumeration step (4) of section 4. This situation particularly occurs, when the execution is in an unstable state (i.e. a state that does not allow time to pass) but offers no execution continuation that establishes a stable state again. Being in an unstable state means that at least one of the invariants within the current mode configurations evaluates to false. In order to analyze the situation (and to enhance the model), the relevant portions of the affected invariants have to be identified. This can be achieved by means of the USE tool: USE supports the evaluation of OCL invariants with respect to a given system state. An implementation for simulating HybridUML models including OCL can internally employ USE methods to evaluate

[9] UML proposes higher priority for inner transitions, whereas outer transitions have priority according to the Statemate semantics [6].

OCL expressions.

When a deadlock occurs because of violated invariants, the interpretation is either (1) that the system state in not correct with respect to the invariants. Assuming that the invariants are yet validated and therefore properly describe static aspects of the model, the checking for deadlock is a verification step. However, if the system state is considered correct, then (2) the failed invariants have to be faulty themselves. This is the start of a validation step which has to be continued by an informal analysis of the respective invariants in order to correct them.

Proper GUI components of USE can then assist the user in inspecting the current system state in a diagrammatic way as well as in inspecting the violated OCL constraints, e.g. by means of its evaluation browser that shows an OCL expression as a tree of subexpressions together with their values.

Note that the top-level invariants from section 3 are used to restrict the admissible state space of the model directly, since there cannot be any transition exiting a top-level mode. Therefore, these invariants always must evaluate to `true`.

6 CONCLUSION

This paper has demonstrated the benefits of combining the specification formalism HybridUML and the USE approach for validation and verification. OCL has been applied to specify constraints on the modes, i.e. invariants, algebraic and flow constraints, as well as guard conditions attached to transitions and post conditions defining the result of operations. Since OCL is the standard UML supplement for the specification of constraints on (parts of) a model, it was a straight-forward approach to apply OCL to HybridUML.

As an application example, a train control system according to the BART case study was chosen. Parts of the HybridUML/OCL specification have been presented in detail, in order to (1) explain the features of HybridUML and (2) to exemplify how OCL constraints can be embedded into the specification. Particularly, top-level invariant constraints are attached to top-level modes of basic agent's behavioral specification. They directly act as constraints to the complete model, because a top-level mode is always active and cannot be left.

In contrast to the previous work in [19] the HybridUML/OCL model comprises structural and behavioral specifications in one single unified model, completely specified by means of UML. It consists of class and composite structure diagrams and state machines.

HybridUML facilitated the modeling of time-continuous properties of the system in terms of flow and algebraic conditions. The model has got its semantics by transformation into a semantically well-defined low-level language. This results in an executable program that simulates the specified system in real-time. The simulation semantics have been introduced briefly.

The adaption of HybridUML's semantics wrt. automated test data generation is currently investigated: The aim is to trigger selected executions of the system rather than random simulation runs. Additionally, the system can be separated into a simulation part and an external part, e.g. into an externally implemented (hardware) controller and its simulated environment. This approach facilitates not only a complete system simulation but also the specification-based automated test of external controllers. Such kinds of tests are applied yet for discretized real-time systems, e.g. to Airbus aircraft controllers. [16]

Is has been pointed out that the USE approach is suitable to (1) verify system states with respect to the top-level invariant constraints and to (2) validate the top-level invariant constraints themselves. It is a matter of interpretation whether the comparison between system state and invariant constraints is a verification or validation step: If the invariant constraints are assumed to reflect proper restrictions on the model, then the system state is *verified.* If the system state is supposed to reflect an admissible snapshot of the object model, and invariants are violated, then the comparison contributes to the *validation* of the invariants. Of course, the (failed) invariants have to be analyzed informally.

Since the executable HybridUML simulation fundamentally relies on the evaluation of OCL expressions, and particularly of the invariant constraints of the model, an (implemented) integration of the USE tool into the executable simulation as a “USE component” would be highly desirable. The USE component could support the analyzing of invariant violations in terms of graphical presentations of the system state. Modelers could inspect the failed expressions in order to backtrack the cause of a failed execution, which is indicated by a deadlock of the simulation.

Further topics on the verification and validation of HybridUML/OCL models to be investigated in future work are amongst others: (1) Deadlocks are not the only way a HybridUML simulation may fail. For example, livelocks can occur, when no continuous simulation step *is and never will* be possible, but always a discrete step is allowed. Detection of these situations is desirable. (2) The HybridUML specification is accompanied by an *architectural specification* and by a *physical constraints specification* (which are not discussed in this article) that define environmental restrictions for the execution of HybridUML models, e.g. available CPUs and cluster nodes as well as required frequencies for the discrete representation of time-continuous calculations. The integration of these parts into the UML model with utilization of OCL has to be investigated.

REFERENCES

[1] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 33–48, 2001.

[2] Kirsten Berkenkötter. Using UML 2.0 in real-time development - a critical review. SVERTS Workshop at the ≪UML≫ 2003 Conference, October 2003. http://www-verimag.imag.fr/EVENTS/2003/SVERTS/.

[3] Kirsten Berkenkötter, Stefan Bisanz, Ulrich Hannemann, and Jan Peleska. HybridUML Profile for UML 2.0. SVERTS Workshop at the ≪UML≫ 2003 Conference, October 2003. http://www-verimag.imag.fr/EVENTS/2003/SVERTS/.

[4] Kirsten Berkenkötter, Stefan Bisanz, Ulrich Hannemann, and Jan Peleska. Executable HybridUML and its Application to Train Control Systems. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*. Springer Verlag, September 2004. ISBN: 3-540-23135-8.

[5] Boldsoft, Rational Software Corporation, and IONA. Response to the UML 2.0 OCL RfP (ad/2000-09-03), January 2003. http://www.klasse.nl/ocl/ocl-subm.html.

[6] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. *Lecture Notes in Computer Science*, 1536:186–238, 1998.

[7] Priority Programme Software Specification – Integration of Software Specification Techniques for Applications in Engineering. http://tfs.cs.tu-berlin.de/projekte/indspec/SPP.

[8] S. Engell, G. Frehse, and E. Schnieder, editors. *Modelling, Analysis and Design of Hybrid Systems*, volume 279 of *Lecture Notes in Control and Information Sciences*. Springer Verlag, 2002. ISBN 3-540-43812-2.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 120–131. IEEE Computer Society, 1994.

[11] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[12] OMG. UML 2.0 Infrastructure Specification, OMG Adopted Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-15.pdf, September 2003.

[13] OMG. UML 2.0 Superstructure Specification, OMG Adopted Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf, August 2003.

[14] Mark Richters. A UML-based Specification Environment, last revision 2001. http://www.db.informatik.uni-bremen.de/projects/USE.

[15] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference Manual, Second Edition*. Addison-Wesley, 2004.

[16] Verified Systems. RT-Tester 6.x – User Manual. Technical Report Verified-INT-014-2003, Verified Systems International GmbH, Bremen, 2004.

[17] Victor L. Winter and Sourav Bhattacharya. *High Integrity Software*. Kluwer Academic Publishers Press., 2001.

[18] Chaochen Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In *Hybrid Systems*, pages 36–59. The Computer Society of the IEEE, 1993. Extended abstract.

[19] Paul Ziemann and Martin Gogolla. Validating OCL specifications with the USE tool - an example based on the BART case study. In Thomas Arts and Wan Fokkink, editors, *Proc. 8th Int. Workshop Formal Methods for Industrial Critical Systems (FMICS'2003)*, volume 80 of *ENTCS*. Elsevier, 2003.