

Model-Based Design of Hybrid Systems

Bernhard Schätz

Fakultät für Informatik, TU München
email: schaetz@in.tum.de

Abstract. Model-based development adds the structuring and preciseness of formal approaches to the development process while being driven by the models of the application domain instead of mathematical formalisms. In this article we investigate how a model-based approach can help to define system boundaries, specify an abstract behavior of the system and the environment, ensure the suitability of the model, and repeatedly enhance the system with design decisions.

1 Introduction

Model-based development, i.e., the use of an abstract, generally domain-specific model to describe the system under development, is the foundation of many current approaches, ranging from the UML [FS97] and Model Driven Architecture [MB02], to most CASE-tools (e.g., Ration Rose RT, Rhapsody) [SHH⁺03] for embedded systems. However, model-based development is more than using graphic design languages to describe systems or to generate code from those descriptions. The main objective of this article is to demonstrate the methodical support gained from the application of formal techniques to engineering methods in the development of reactive systems. We focus on how the complexity of the design process can be reduced by breaking it up into different steps, each concentrating on a special aspect of development, and how CASE support can simplify those steps.

As illustrating example we use the BART (Bay Area Rapid Transfer) case study, introduced in [WB01], informally describing the requirements of a system controlling the speed and acceleration selection for fully-automated trains. A train receives commands from a station computer responsible for the area the train is in, and executes this command until a new command is received or the current command expires; if a command expires, the train brakes to halt. The station computer has to select speed and acceleration commands based on status information sent from the trains to avoid, e.g., train collisions, even in case of a breakdown of the communication network. To ensure these safety conditions, the station computer issuing those commands computes a worst case stopping

distance, based on the behavior of the train in case no new command arrives (i.e., executing the command until timeout and braking to halt). The aspects of the BART case study dealt with here include:

- defining the system interface including the communicated data
- constructing a model of the environment (trains, train controller) and the system (station computer)
- analyzing the constructed models concerning conceptual properties ensuring soundness of the models and qualitative safety conditions imposed on the environment
- defining an architectural partitioning (VSC/NVSC components)
- refining the reactive behavior of the system and increasing robustness

While the demonstrated techniques are not tool-specific, here we use the AutoFOCUS approach described in [BLSS00] to illustrate them. We therefore give a short introduction to the AutoFOCUS description techniques and the principles underlying its approach in the remainder of this section. In Section 2 we then describe the steps of a model-based development of a station computer, from modeling the environment to establishing a robust design. Section 3 gives a short discussion of the results.

1.1 Views and Notations: Description Techniques

AutoFOCUS offers hierarchically structured description techniques to form a comprehensive and structured picture of a system, described from different points of view and on different levels of abstraction, including *system structure diagrams* (SSDs), *data type definitions* (DTDs), *state transition diagrams* (STDs), and *extended event traces* (EETs), each one covering different—yet not necessarily disjoint—aspects the system. The integration of the views on a common semantic basis leads to one formal specification of the system. AutoFOCUS supports hierarchical development of systems. Depending on the granularity, views (e.g., SSDs) can be atomic, or consist of sub-views themselves (e.g., SSDs assigned to components in an SSD). Therefore, AutoFOCUS allows to switch between different levels of granularity by using the hierarchical description techniques described in the following.

A (distributed) system consists of its components and the communication channels between them. To describe the architectural aspects of distributed systems, viewing it as a network of interconnected components exchanging messages over channels, we use system structure diagrams as shown, e.g., in Figure 2. In AutoFOCUS, the types of the data processed by

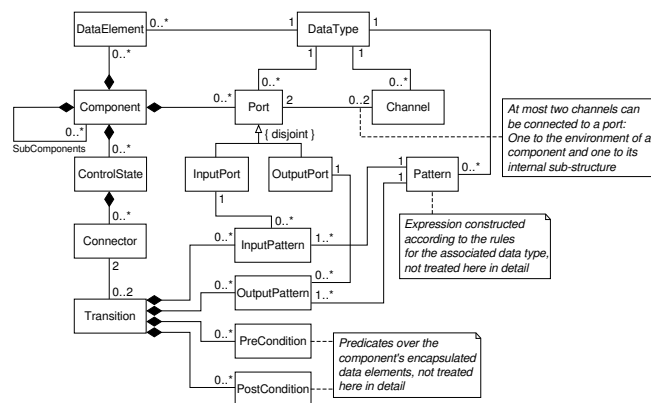


Fig. 1. Simplified Conceptual Model

a distributed system are defined using textual notations called data type definitions. A functional variant is used to declare new types and functions operating on them. An STD, as shown in Figure 3, characterizes the behavior of a system or component, reacting to input received from its environment by producing output sent to the environment, depending on the its current state and setting a new state. Because an STD describes an extended finite state machine using variables local to the characterized component, the state of a component is defined both by the control state (the state of the finite state machine) and the data state (values of the local variables) of the component. EETs, as shown in Figure 5, define a component- and communication-oriented view, describing system behavior by sample communication histories between components. Boxes can be used to reference a set of sub-EETs.

1.2 Views, Models, and Consistency Conditions

Developers create and manipulate models by means of description techniques. These techniques (e.g., STDs or EETs) provide views upon these models, possibly even several different views on the same parts of the model (e.g., an EET including a scenario description of a component, and an STD, which describes the complete behavior of the component). To describe how the elements of those views are related ‘syntactically’, we use the *conceptual model*. Similar to an UML meta model, it defines the ‘abstract syntax’ of the information given by the notations used and ensures that all views that are constructed as an instance of it are ‘well-formed’ and fit together. Figure 1 shows a simplified version of the AutoFOCUS

conceptual model. Besides making sure that views are conceptually well-formed, we also want to make sure that they fit together semantically. Here, the *semantical model* defines the interpretation of a set of views and is therefore used, e.g., for the generation of code or specifications to be verified by proof tools. Similar to the conceptual model, it is also used to ensure that the views of a system fit together semantically.

While views are introduced to reduce the complexity of a system by breaking it up into manageable parts, *consistency* is used to make sure that those parts fit together to form a suitable model. Since we have two kind of ‘meta models’ guiding the development process, we have two corresponding classes of consistency conditions: *Conceptual consistency conditions*, expressed completely within the conceptual model; *Semantic consistency conditions*, expressed using the semantical model.

AutoFOCUS ensures invariant conceptual consistency conditions (e.g., ‘Transitions only use ports of the corresponding component’ by construction of the model; variant conceptual consistency conditions (e.g., ‘All transitions leaving a state have disjoint patterns thus ensuring deterministic behavior’) are defined in ODL (Operation Definition Language) and mechanically checked. Using the description techniques introduced here, two forms of semantic inconsistencies can occur between the introduced views: *Inconsistencies of scenarios*, e.g., EETs cannot be executed as described by corresponding STDs; *Inconsistency of refinement*, e.g., the network of sub-components described by an SSD can exhibit some behavior that is not possible with the abstract component. To check for inconsistencies of scenarios or refinement, as well as general temporal logic properties, AutoFOCUS offers a connection to different proof tools (e.g. μ cke, SMV, Isabelle, and, currently developed, STeP) [BLSS00].

2 Development Steps

The development of the control system is split up into steps, each illustrating how a model-based approach helps

- to get a clear understanding of the system boundaries and to make implicit assumptions about the environment explicit.
- to define an abstract system behavior without limiting the design space for possible implementations.
- to check the soundness of the constructed models.
- to establish the qualitative properties required to hold for the system.
- to support design decisions concerning the structure and the behavior of the system.

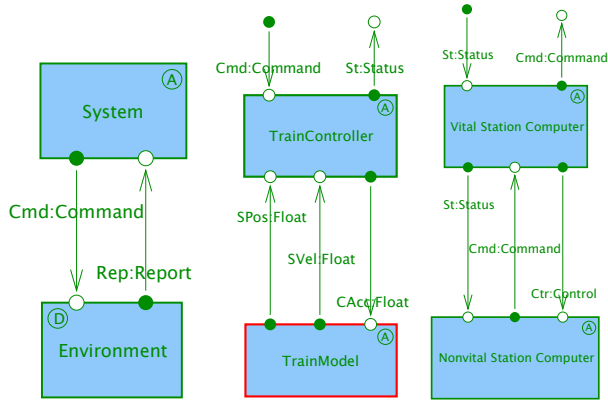


Fig. 2. Overall Structure, Train Structure, System Structure

- to increasing robustness by considering undefined behavior.

2.1 Modeling the Environment

When developing an embedded system, the model of the environment of the system under development is as important as the system itself. On the one hand, the model of the environment is used to *explicitly represent the state* of the environment to express properties about the environment (e.g., positions of the trains) rather than the system (e.g., estimated positions). On the other hand, *assumptions are made explicit*, which can be used to validate whether we have a suitable model of the environment (e.g. by running simulations) and which are needed to establish the required behavior of the system (e.g., reliability of the messaging system). As shown in Figure 2, in the AutoFOCUS approach, system and environment are modeled in form of a control loop similar to the four-variable model of [PM95], with input read from the environment and output fed to the environment; the environment, like the system, can be structured into state-based subcomponent with local variables. The environment contains additional parts (e.g., a communication network) besides the trains. The train consists of a `TrainController` and a `TrainModel` (describing the train physics).

The local data of the environment, forming the *environmental variables*, are used to model the continuous variables and a continuous behavior of the hardware parts as well as the discrete variables and behavior of the software. Here, the environmental variables of the physical model

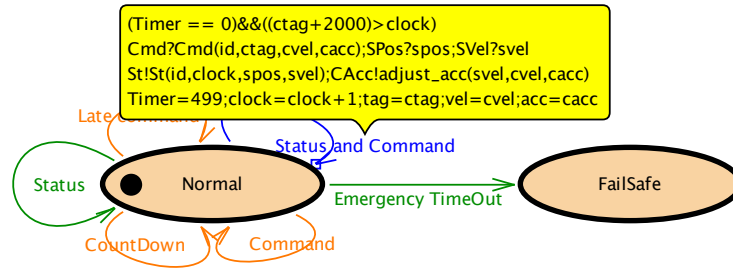


Fig. 3. Behavior of TrainController

of the train (*TrainModel*) include, e.g., the continuous variable for the speed (*vel*) for each train; the discrete variables of the controller of the train (*TrainController*) include, e.g., a timer for the generation of status reports (*Timer*). Similarly, the *interface variables* can be identified by the channels between the system and the environment, e.g, the channel for the command messages from the system to the environment (*Cmd*). As a part of the description of the variables, their type definition is added, e.g.

```
Command = Cmd(CID:ID,CMOTT:Time,CVel:Float,CAcc:Float);
```

The next step in constructing the model of the environment consists in assigning behavior to all components of the environment. The STDs of AutoFOCUS can be used to model continuous parts like the train model as well as discrete parts like the train controller. To define the train controller, we assign a standard AutoFOCUS STD to the *TrainController* component. As shown in Figure 3, it consists of two states (*Normal*, *FailSafe*) for normal and fail safe operation. As defined in the expanded transition with label *Status and Command* a command received in time before fail safe mode is executed with the beginning of a new cycle as long as the command is valid:

Precondition $(Timer==0)\ \&\&((tag+2000)>clock)$ stating that a new cycle begins with the failsafe timeout not yet occurred

Input $Cmd?Cmd(id,tag,cvel,cacc);SPos?spos;SVel?svel$ stating that a new parameterized command as well as the current status (*spos*, *svel*) of the train

Output $St!St(id,clock,spos,svel);CAcc!adjust_acc(svel,cvel,cacc)$ stating that a report is generated and the adjusted acceleration command is set to be executed by the train

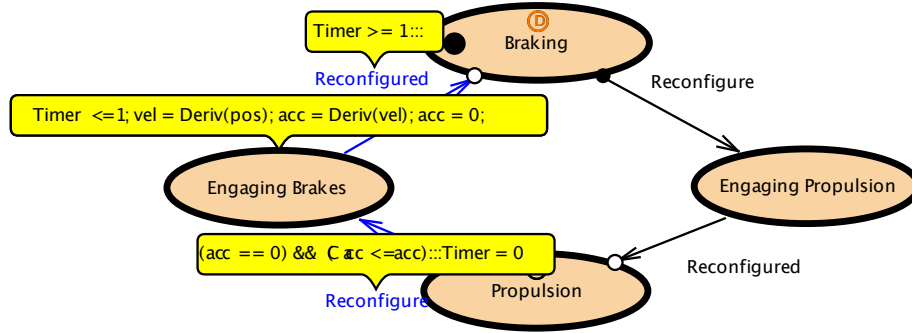


Fig. 4. Behavior of TrainModel

Postcondition $\text{Timer}=499; \text{clock}=\text{clock}+1; \text{tag}=\text{ctag}; \text{vel}=\text{cvel}; \text{acc}=\text{cacc}$ stating that a new 500ms round is started while the local clock is advanced; additionally, the command is stored.

STDs can also be used to model the continuous part of the environment. However, here the interpretation differs from STDs for discrete components. While a discrete component performs its computation (and the corresponding execution of transitions) only once each cycle, a continuous component has no discretized behavior. Therefore, additional to the transitions, behavior for continuous components can be defined using *activities*. Activities are assigned to states and are performed continuously (while the guard is satisfied) until a transition is executed for this state. Generally, differential equations are used to describe the invariant condition in this state. Figure 4 shows the STD describing the behavior of the train model component `TrainModel` with four states characterizing different continuous behavior with the corresponding activities describing the invariance conditions, e.g., `Engaging Propulsion` with activity $\text{Timer} \leq 1; \text{vel} = \text{Deriv}(\text{pos}); \text{acc} = \text{Deriv}(\text{vel}); \text{acc} = 0$. Transitions like `Reconfigure` are defined using enabling conditions ($\text{acc} == 0$) and assignments ($\text{Timer} = 1$); the execution of a transition is assumed to take no time. To formalize the semantics of the hybrid environment, hybrid transition systems (e.g., [MP95]) are used. Hybrid transition system differ from discrete transition systems (as used in `Auto-FOCUS` to model discrete behaviors) by supporting continuous variables, i.e. variables that change continuously over time (e.g., train position `pos`). Additionally, a continuous variable $T \in \mathbb{T}$ is used to describe the physical time. To describe the system state, a variable *state* is used. Accordingly,

transition **Reconfigure** is expressed as

$$\begin{aligned} state &= \text{Braking} \wedge \text{acc} = 0 \wedge \\ state' &= \text{Engaging Propulsion} \wedge \text{Timer}' = 0 \wedge T = T' \end{aligned}$$

using primed variables (e.g., $state'$) to describe the post state. Accordingly, e.g., the invariant of state **Engaging Propulsion** is expressed as a transition with

$$\begin{aligned} (\forall \tau. 0 \leq \tau \leq tick \Rightarrow \text{Timer} + \tau \leq 1) \wedge \\ T' = T + tick \wedge \text{Timer}' = \text{Timer} + tick \wedge \\ \text{pos}' = \text{pos} + \int_0^{tick} \text{vel} dt \wedge \text{vel}' = \text{vel} + \int_0^{tick} \text{acc} dt \wedge \text{acc} = 0 \end{aligned}$$

Since this formalization corresponds to the semantics used with the temporal theorem prover STeP [BBC⁺97], in Subsection 2.3 we will use STeP to verify safety requirements.

Finally, we can now formulate qualitative properties about the environment, e.g., that train speeds and accelerations are selected ensuring that the train does not enter the worst case stopping distance to a leading train. This requirement can be defined as temporal property with variables depending on a time point $\tau \in \mathbb{T}$ stating

$$\forall \tau \in \mathbb{T}. \forall t_1, t_2 \in \text{Trains}. \quad t_1.\text{pos}_\tau + \text{Nose} + \text{wcsd}(t_1, \tau) \neq t_2.\text{pos}_\tau - \text{Tail} \quad (1)$$

(with *Nose* and *Tail* describing the length from the position of a train to its nose and its end, resp., and *wcsd* defining the worst case stopping distance defined in terms of the current state of a train).

2.2 Abstract System Behavior

To define the behavior of the **System** component we identify the *overall behavior* of the system, add the *computations performed during each step* and the *data state* of the controller, and finally describe its *control state and transitions*. To describe the overall behavior we use the interaction view of Figure 5 describing that the behavior basically consists of two phases (described by the boxes *Messages* with some of its instances, and *Commands*) which are cyclically repeated every 500ms.

The computations to be issued by the system must ensure that the imposed safety requirements as defined in Subsection 2.1 in terms of the model of the environment are not violated. Since, however, the station computer cannot directly access the environment variables but only the commanded speed and acceleration, it must use an approximation of those

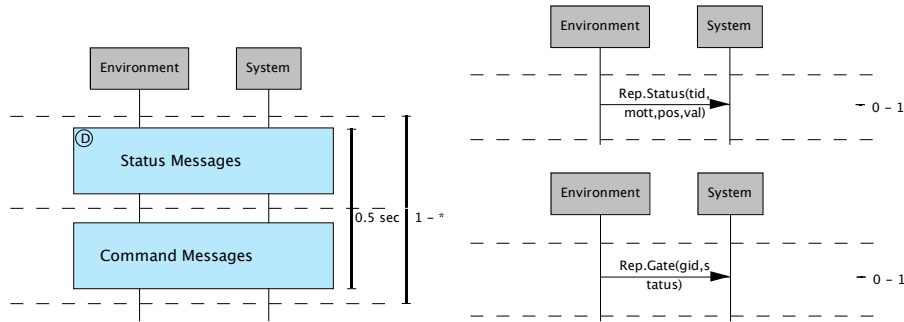


Fig. 5. Overall Behavior and Instance of *Messages*

variables, deduced from the available input and output of the system. Typically for embedded systems, these deduced approximations form an *internalized model* of the environment. When constructing the internalized model of a train used by the station computer, we basically introduce a corresponding variable for every variable used in the model of the environment; accordingly, corresponding computation steps are introduced. To simplify the computation, e.g., to cut execution time, or to deal with non-determinism of the environment, we can use a more abstract model by fixing a variable of the environment to the value leading to a maximum worst case stopping distance with respect to that variable.

Based on the internalized model the computations performed by **System** are defined as a counterpart of the safety property identified in Subsection 2.1. Correspondingly, a function *wcsd* operating on the internalized model is defined and applied in relation **Bound**, to ensure the worst case stopping distance to the leading trains for a commanded speed and acceleration. To define the data state, the arguments of the computations are used.

Using the overall behavior, the data state, and the computations, the control state and the transitions between them can now be defined, as shown in Figure 6: Receiving reports while in **Collect**, computing speed and acceleration commands for registered trains while in **Compute**, and sending the computed commands to the registered trains and updating the internal model while in **Issue**. Note that according to the STD description of transition **Compute**, the controller does not actually compute commands but simple idles; however, since in this step we only model the externally visible behavior, this is an adequate description of the behav-

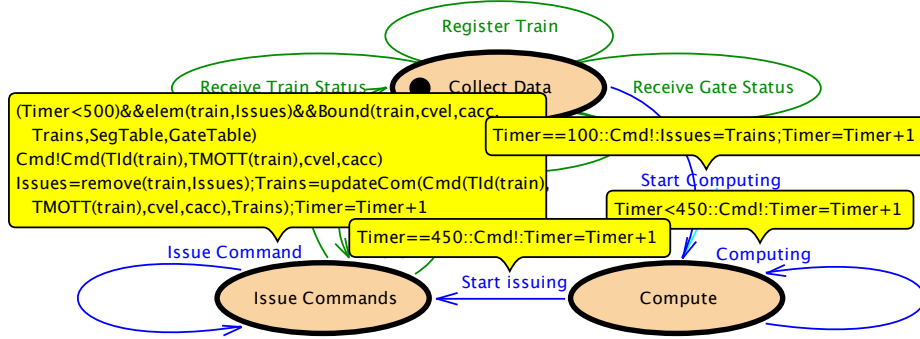


Fig. 6. Behavior of the System

ior. In the following steps we will consider more detailed descriptions of the computing process.

2.3 Well-Formed Models

Before checking whether the introduced model meets the requirements, we make sure that the model is a ‘reasonable’ model. To validate that the model of the environment is a reasonable approximation of reality, simulation or formal proof of properties about the environment can be used. Furthermore, a reasonable model should have a consistent description; here, model-based development supports the detection of different forms of inconsistencies in the models. Besides general conceptual consistency, we are especially interested in two properties: *Input Enabledness* ensuring that the environment (system) cannot restrict the actions of the system (environment) (e.g., by not accepting certain commands issued by the system as input); *Responsiveness of Behavior*, ensuring that neither system nor environment can be taken to an undefined state.¹ For methodical reasons, we treat the model of the environment and the model of the system differently when analyzing completeness of their behaviors. While we require an explicitly completely defined behavior of the environment, for the system we distinguish between

Input Definedness: If for some state acceptable input is only partially defined (i.e., some precondition/input pattern is not accounted for), we allow the system to react arbitrarily.

¹ Note that the environment may still show non-deterministic behavior, e.g., due to sporadic wheel slippage.

Output Responsiveness: If an explicit reaction is defined for some input, the corresponding output and post condition must be defined (i.e., the functions (or relations) over the local data state/ input must be total).

To check output responsiveness, sufficient criteria for the totality of constructive (sub-) functions can be used, like the totality of look-up tables (e.g., for the table of track segments, there is an entry for each possible track position). Those checks can be automatized using, e.g., an ODL condition.

While incomplete input definedness is acceptable in requirements analysis and early design, undefined behavior generally is a possible source of faulty behavior and therefore has to be removed in later steps.

2.4 Verification of System Properties

Having introduced a suitable model of the environment and the system, we can now use these models to verify that the behavior of the system leads to the desired behavior of the environment.

The proof of property of Equation 1 follows a generally outline for this class of properties and systems. To verify invariant properties about the state space of a system, induction is applied. In the *Base Step* step, we show that the system is initially in a safe state. In the *Induction Step*, we show that whenever the system is in a safe state, any possible transition will take it to another safe state. For a system using an internalized model of the environment, the induction step can be broken up into three sub-steps. Since the internalized model used by the system sets the basis for the calculation of the commands, as a first proof step we establish that the *physical model* of the train is always bound by the *internalized model* of the system when a command is executed. This includes establishing properties like the estimated position of the train executing an issued command is an upper bound of its real position at that time. Because these properties relate the status reported by a train to its current state when executing a new command, the proof is established by creating a chain of states, starting at the time of the status report and ending at the time an issued command is executed.

After linking the models, in the second step we relate the *estimated effect of a command* (i.e., distance $wcsd$ used by **System**) to the *desired effect on the environment* (i.e., distance $wcsd$ used as invariant). Here, we ensure that, $wcsd$ is an upper bound of $wcsd$, when an issued command is executed.

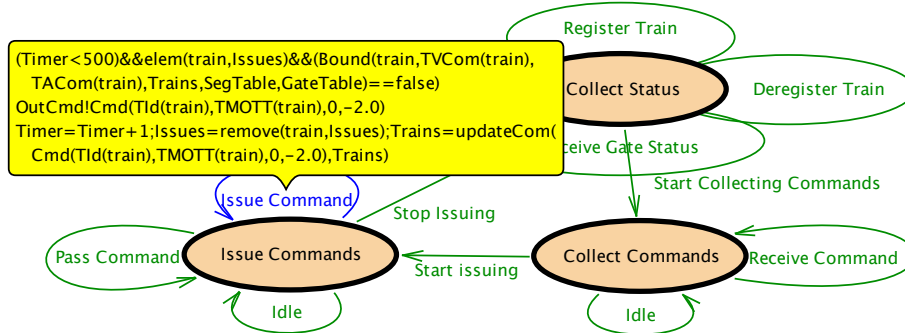


Fig. 7. Behavior of Vital Station Computer

Using these results, in the final step we check whether all *transitions of the environment are safe* concerning the invariant property. Since we have two different kinds of transitions, we use two different techniques:

Command Processing: This part ensures the safety invariance when executing the *Command* transitions of Figure 3. Since these transitions are the only ones controlled by the *System*, here the station computer calculations and therefore the previous results are applied.

Controlling Train: Here, we show that each transition maintains the invariance condition by relating the variables changed (especially the distance covered) when taking a transition, to the corresponding change concerning the *wcsd* defined over those variables.

Since our models are not accessible to fully-automatic model checking, a more general proof support is needed. Therefore, as mention in Subsection 2.2, here the STeP prover system is applied using both automatic and interactive techniques to establish the proof. To ease the understanding of the proof in spite of its complexity, the Verification Diagrams of STeP help to structure the first and third proof step.

2.5 Refining the System

According to platform requirements, the system is refined as shown in Figure 2 by two components: a *Vital Station Computer* (VSC) delegating command calculation to the NVSC, and checking whether suggested commands respect the worst-case bounds; a *Non-Vital Station Computer* (NVSC) computing improved commands and forwarding them to the VSC for validation. As shown in Figure 7, the behavior of the VSC exhibits

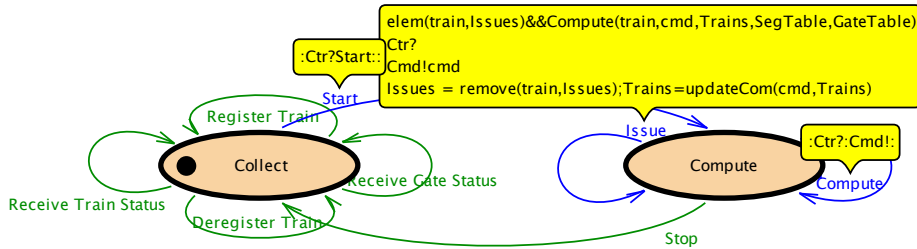


Fig. 8. Abstract Behavior of the NVSC

the same states and similar transitions as the behavior of **System**. Besides the delegation of commands to the NVSC, the major change is the substitution of the **Issue Command** transition of Figure 6 by two transitions:

Pass command: passing on a command suggested by the NVSC if it meets the safety bounds

Issue command: issuing a brake command (with speed 0 and acceleration -2.0), if the safety bounds are exceeded

To show that the refined system of VSC and NVSC implements the original **System**, it therefore basically suffices to show that those two transitions implement the original one, which corresponds to showing the brake command meets the safety bounds. Assuming arbitrary behavior of the NVSC, we can even establish an equivalence between both systems. To formally verify this implementation relation, again STeP can be used.

After refining the structure of the **System**, we now define a suitable behavior of **Non-vital Station Computer** by refining the behavior of the system, eliminating some underspecification. Figure 8 shows a simple behavior to start from, just sticking to a basic interaction protocol with **Vital Station Computer**. With reactive systems, we have two different forms of behavior and corresponding notions of refinement:

Computation: Like procedural programs, reactive systems perform computations, calculating the values of variables depending on the values of others, performed in one (uninterrupted) step. In **AutoFOCUS**, computations, e.g., occur in form of the functions used in the pre and post conditions of the transitions.² If a computation is underspecified, like relation **Compute** in Figure 8, a more refined implementation can restrict this non-deterministic computation.

² For complex computations, **AutoFOCUS** uses a description technique not introduced here to describe the computational data flow.

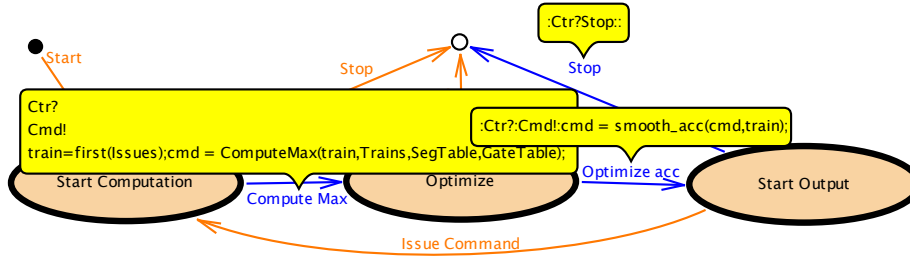


Fig. 9. Refined Behavior of Compute

Interaction: In contrast to procedural programs, reactive systems perform computations repeatedly. Here, the timing and duration of each computation in relation to the interactions with the environment are important. In AutoFOCUS, this corresponds to the repeated execution of transitions. If an interaction is underspecified – like by means of the non-deterministic choice between the **Compute** and **Issue** transitions in Figure 8 – allowing different possible sequences of interaction, a mode refined behavior can restrict this non-deterministic choice.

To refine the computation, we break it up in two steps: first we compute the maximum commanded speed and acceleration respecting the check performed by the VSC; then the speed selection is optimized to ensure, e.g., a smoother ride. To ensure the refinement relation, as usual, we check that the refined computation only results in a (non-empty) sub-set of the original version.

By splitting the computation into different parts, these parts can be assigned to consecutive steps in the behavior of the NVSC. Using STDs, e.g., we assign three sub-states to the **Compute** state, and distribute the computations and interactions on their connecting transitions, as shown in 9. Similar to the component/sub-component refinement step, we can now verify whether the modified behavior of NCSV refines its original behavior.

2.6 Increasing Robustness

Since the operations of the **Vital Station Computer** are safety critical, we want to make sure that we have a completely defined input and output behavior. To ensure a completely defined input, we have to make sure that in each (control and data) state of **Vital Station Computer** for each message received at its channels there is a corresponding transition.

AutoFOCUS currently offers input definedness analysis based on symbolic techniques.

The **Vital Station Computer** exhibits some undefined behavior caused by the status reports to be handled by the station computer. Checking for completeness reveals, e.g., the missing input pattern for the **St** channel in the **Collect** and **Issue** states. According to our assumptions incorporated in the model of the environment, status reports are sent out only during the protocol slot reserved for status messages. Therefore, operational completion as default treatment may be seen as sufficient: non-definedness is resolved by ignoring the received message. Since faulty behavior (clock drift of a train, jitter in the network, etc.) however may compromise these assumptions, we are interested in complete input definedness capable of detecting such situations. By adding corresponding transitions, we can add appropriate reactions (e.g., issuing a warning to the operator desk).

Once having established a complete behavior, we eliminate another source of possible defects by making sure that the transitions are deterministic. By comparing the enabling conditions of all pairs of transitions, AutoFOCUS checks whether two transitions are enabled by the same (control and data) state and messages received. While this does not necessarily result in non-deterministic or even faulty behavior, in embedded systems this situation generally is unwanted. However, since both components already are defined deterministically, no further steps are required here to end up with a robust implementation.

3 Conclusion and Outlook

As sketched, model-based development adds the structuring and preciseness of formal approaches to the development process while being driven by the models of the application domain instead of mathematical formalisms. We introduced the AutoFOCUS approach to model-based development of reactive systems ([SPHP02], and [BLSS00] give further details on the process and tool support). Model-based approach as presented here offers:

Improved product: By moving away from an implementation biased view of a system, the developer can focus on the important issues of the product under development. This results in thinking in terms of the domain-specific conceptual model (state, component, interaction, etc.) instead of the coding level (objects, method calls, etc.).

Furthermore, invariant consistency conditions (e.g. separation of environment, interface, and system) and variant consistency conditions (e.g., absence of undefined behavior) are limiting the possibility of making mistakes while building models and refining them.

Improved process: Using a better structured product model helps to identify more defects earlier. Additionally, higher efficiency can be obtained mechanically ensured conceptual consistency conditions (by construction, e.g., interface correctness or on demand, e.g., completeness of defined behavior) as well as semantical consistency conditions (e.g., checking whether an EET can be performed by system). Finally, transformations of specification (e.g. inserting standard behavior for undefined situations) are interactively carried out by the CASE tool.

The limit of model-based support is defined by the sophistication of the underlying conceptual and semantical model: by adding more domain-specific aspects to it, even more advanced techniques can be offered like supporting reliability analysis for the BART system or calculating safe task schedules for the Vital Station Computer.

References

- [BBC⁺97] Nikolaž Björner, Anca Browne, Eddie Chang, Michael Col'on, Bernd Finkbeiner, Arjun Kapur, Zohar Manna, Henry Sipma, and Thom'as Uribe. *STeP The Stanford Temporal Prover Educational Release Version 1.3 User's Manual*, 1997.
- [BLSS00] Peter Braun, Heiko Lötzbeyer, Bernhard Schätz, and Oscar Slotosch. Consistent integration of formal methods. In *Proc. 6th Intl. Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, 2000.
- [FS97] Martin Fowler and Ken Scott. *UML Distilled*. Addison-Wesley, 1997.
- [MB02] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.
- [MP95] Zohar Manna and Amir Pnueli. *Temporal verification of Reactive Systems: Safety*. Springer, 1995.
- [PM95] David Parnas and Jan Madey. Functional Documents for Computer Systems. *Science of Computer Programming*, 1(25):41–61, October 1995.
- [SHH⁺03] Bernhard Schätz, Tobias Hain, Frank Houdek, Wolfgang Prenninger, Martin Rappl, Jan Romberg, Oscar Slotosch, Martin Strecker, and Alexander Wisspeintner. CASE Tools for Embedded Systems. Technical Report TUMI-0309, TU München, Fakultät für Informatik, 2003.
- [SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-Based Development of Embedded Systems. In J.-M. Bruel and Z. Belahsene, editors, *Advances in Object-Oriented Information Systems - OOIS 2002 Workshops*. Springer LNCS, 2002.
- [WB01] Victor Winter and S. Bhattacharya. *High Integrity Software*. Kluwer Academic Publisher, 2001.