Critical Systems Design with UML$^{\text{light}}$

Jan Jürjens[⋆]

Software & Systems Engineering, Dep. of Informatics
Munich University of Technology, Germany

Abstract. Despite years of successful research into using formal methods for the development of critical concurrent systems, there are still too many failures of critical systems in practice. Part of the reason is that use formal methods is often seen to be to costly.
The Unified Modeling Language offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context, since many developers are trained in UML and are using it already.
Our aim is to aid the difficult task of developing dependable systems in an approach based on a formal fragment of the Unified Modeling Language called UML$^{\text{light}}$. We extend the notation to capture dependability requirements and related physical properties. This way we encapsulate knowledge on prudent dependability engineering and make it available to developers which may not be specialized in dependability. One can also go further by checking whether the constraints associated with the stereotypes are fulfilled in a given specification, by performing a formal analysis.

1 Introduction

There is an increasing desire to exploit the flexibility of software-based systems in the context of critical concurrent systems where predictability is essential. Examples include the use of embedded systems in various application domains, such as fly-by-wire in Avionics, drive-by-wire in Automotive etc. .

Given the high dependability requirements in such systems, a thorough design method is necessary, since failures may have quite severe consequences. Since there are faults in any operational system, fault-tolerance is used at execution time "to provide, by redundancy, service complying with the specification in spite of faults occurred or occurring" [Lap92]. Forms of redundancy commonly employed include space redundancy (physical copies of a resource), time redundancy (rerunning functions) and information redundancy (error-correcting codes). To resolve

[⋆] `http://www.jurjens.de/jan – juerjens@in.tum.de` the used redundancy one may require complex protocols whose correctness can be non-obvious [Rus94]. Mistakes in the use of redundancy and the design of these protocols can thus again lead to problems. Therefore reliability mechanisms cannot be "blindly" inserted into a critical system, but the overall system development must take reliability aspects into account. Furthermore, sometimes dependability mechanisms cannot be used off-the-shelf, but have to be designed specifically to satisfy given requirements (for example on the hardware). Such mechanisms are notoriously hard to design correctly, even for experts, as many examples of protocols designed by experts that were later found to contains flaws show.

Spectacular examples for software failures in practice include problems with Ariane 5 rockets: An independent inquiry board set up to investigate the explosive failure in 1997 said the flight control system failed because of errors in computer software design.[1] Whether the Dec. 12 2002 fatal failure relates to software problems as well is currently being investigated.

Any support to aid dependable systems development is thus dearly needed. In particular, it would be desirable to consider dependability aspects already in the design phase, before a system is actually implemented, since removing flaws in the design phase saves cost and time.

This has motivated a significant amount of research into using formal methods for dependable systems development. However, part of the difficulty of critical systems development is that correctness is often in conflict to cost. Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided.

The Unified Modeling Language (UML, [RJB99, UML01], the de facto industry-standard in object-oriented modeling) offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context.

- As the de facto standard in industrial modeling, a large number of developers is trained in UML.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined.

Here we use a fragment of UML called UML$^{\text{light}}$ defined by a textual program notation together with a formal semantics, because this is sufficient to demonstrate our ideas and because for space-restrictions we cannot consider the complete UML, whose definition in [UML01] consists of about 800 pages altogether. We emphasize that we view it merely as a convenient means to transport our ideas concerning certain aspects of

[1] See http://news.bbc.co.uk/2/hi/science/nature/2569115.stm .

UML. A similar approach is taken in [BLP01] with the notation SMDL. Note, however, that here our goal is not to generate code, and not to be in any way complete in our description of UML beyond the aspects relevant for the ideas we want to present here. This allows our notation to be more abstract and compact (for example, there are no labels). For a more comprehensive fragment of UML with a formal semantics, see [Jür02b, Jür03]. A prototype analysis tool for UML$^{\text{light}}$ is being developed in student projects [Sha02, Men03] (currently focused on security aspects).

To support using UML for dependable systems development, we define some stereotypes capturing dependability requirements and related physical properties. This way we encapsulate knowledge on prudent dependability engineering and thereby make it available to developers which may not be specialized in dependability. One can also go further by checking whether the constraints associated with the stereotypes are fulfilled in a given specification, if desired by performing a formal analysis.

Some of the ideas reported here were or will be presented in an invited talk at FDL'02 [Jür02a] and tutorials on formal development of critical systems with UML at Safecomp 2002, Software Engineering 2003, and ETAPS 2003 (each unpublished).

After presenting some background on dependability and on UML extension mechanisms in the following subsections, we define the UML$^{\text{light}}$ notation used in this paper to formally evaluate UML specifications for dependability requirements in Section 2. We introduce the stereotypes suggested for safe concurrent systems development in Section 3, together with examples of their use. After pointing to related work, we indicate future work and end with a conclusion.

Dependability Reliability goals for safety-critical systems are often expressed quantitatively via the maximum allowed failure rate. For example, critical services of the Advanced Automation System (AAS, providing Air Traffic Control services) should be unavailable at most 3 seconds a year [CDD90]. To prevent any single catastrophic failure in any aircraft of a given type during its entire life-time one estimates that the maximum admissible failure rate for each failure condition is about 10^{-9} per hour [LT82, p.37]. Since 10^9 hours amounts to over 100,000 years, one may not achieve confidence that a system has such a degree of dependability just by testing.

This motivates the use of formal methods. Faced with feasability aspects, one often abstracts from probabilities by assuming that failures are masked perfectly, in order to keep the model as simple as possible. Then probabilistic behaviour is factored out into fault-tolerance components. [Jür01] gives conditions under which it is justified to abstract from failure probabilities.

We use the following examples for failure semantics in this paper.

- crash/performance failure semantics means that a component may crash or may deliver the requested data only after the specified time limit, but it is assumed to be partially correct.
- value failure semantics means that a component may deliver incorrect values (represented by the error message $\perp$).

UML extension mechanisms The three main "lightweight" extension mechanisms are stereotypes, tagged values and constraints. Stereotypes, in double angle brackets, define new types of modeling elements extending the semantics of existing types in the UML metamodel. A tagged value is a name-value pair in curly brackets associating data with model elements. Constraints may also be attached.

For the stereotypes suggested here, we give validation rules evaluating a model against included dependability requirements. For this we extend a formal semantics for the used fragment of UML in a modular way with a formal notion of *failures*.

2 UML$^{\mathbf{light}}$

We briefly sketch the necessary foundations for formally analysing UML specifications in the context of concurrent system design. For some of the constraints one needs to check we need to refer to a precisely defined semantics of behavioral aspects.

Here we only need a behavioural semantics for a simplified fragment of UML statecharts (defined here using the UML$^{\text{light}}$ notation, building on the work in [Jür02b, Jür03]).

In UML the objects or components communicate through messages received in their input queues and released to their output queues. Thus for each component C of a given system, our semantics below defines a process p_C which iteratively reads input from the input queues and adds output to the output queues. The behavioral semantics of this process models the run-to-completion semantics of UML statecharts. To evaluate the safety of the system with respect to a given failure semantics, the processes modelling the system components are composed with failure

$E ::=$	expression
$\perp$	error value
x	variable ($x \in \mathbf{Var}$)
d	data value ($d \in \mathbf{Data}$)
$E_1 :: E_2$	list concatenation
$\mathbf{head}(E)$	head of list
$\mathbf{tail}(E)$	tail of list

Fig. 1. Expressions.

processes with the specified failure semantics defined from the stereotypes at the communication links in the deployment and class diagrams, as explained in Section 3.

Specification language We define the UML$^{\text{light}}$ notation for simplified UML statecharts.

Processes communicate by sending messages to other processes, which are held in a queue until received by the receipient (thus communication is asynchronous). Processes are defined by programs that describe the output at a given point in time given the received input. Local state can be maintained through the use of local variables, and used for iteration (for instance, for coding *while* loops) which can be defined using CCS-style guarded recursive equations (defined below).

We assume sets **Op** of messsage names (including the completion message), **Var** of variables and **Data** of data values to be used as arguments of messages. The values communicated over channels are formal expressions built from the error value $\perp$, variables, and data values using concatenation as defined in Figure 1 (with the usual equations for concatenation, **head**(), and **tail**()).

UML$^{\text{light}}$ statecharts are defined inductively in Figure 2. Here $k \in \{y, t, x\}$ is the *kind* of the action, representing *entry*, *transition-bound*, and *exit*, resp. . $exp \in \mathbf{Exp}$ is an expression, $msg = op(exp)$ a message, $var \in \mathbf{Var}$ a variable, and $bexp$ is a Boolean expressions over $(\mathbf{Exp}, =)$. In the $op(var)$ expression, the incoming value is assigned to the variable.[2]

One can specify iteration (corresponding to transition loops in the diagrammatic presentation of statecharts) by using *guarded recursion* of the form $A \overset{\text{def}}{=} E(A)$ where A occurs in E only within subexpressions of the form $a.F$ as in CCS. Guardedness ensure that such equations have unique solutions.

[2] Note that our usage of ";" is different from that in [BLP01].

$act ::=$	actions
$out_k(msg)$	output of expression
$var :=_k exp$	assignment
$trs ::=$	transitions
$op(var)[bexp]act_1 \dots act_n$	
$p ::=$	programs
$trs.p$	firing of transition
$p_1 \| p_2$	parallel composition
$p_1 + p_2$	nondeterminism
0	final state
$p_1 ; p_2$	submachine composition

Fig. 2. UML$^{\text{light}}$ statecharts.

Here we follow some of the reductions regarding statechart constructs proposed in [RACH00, Cav00]:

- entry and exit actions are factored into the transitions (which thus need to allow more than one action).
- This way the distinction between internal and external transitions becomes implicit.
- Synch states can be modelled using other constructs.

Furthermore we omit the following features because they are orthogonal to the ideas we would like to present here.

- history states and deferred events,
- type hierarchies,
- creation and deletion of state machines,
- transitions crossing state borders.

Internal activities are themselves again modelled by statechart expressions.

The structural operational semantics of the programs is given in Figure 3. It is of the form $(s,p)\overset{op(exp);out(\boldsymbol{msg})}{\rightarrow}(s',p')$ where p,p' are programs, s,s' are valuations of the variables appearing in p,p' (called "states"), $op(exp)$ specifies that the transition will be fired at reception of the message op with argument exp, and $\boldsymbol{msg}$ is a list of messages send out when the transition is fired. In the first rule, $op(exp)$ is the message expected in trs, $bexp(s)$ the condition in trs evaluated at state s, $\boldsymbol{msg}$ the list of messages send out by the actions in trs, and $\boldsymbol{var}$ is the list of variables assigned the list of values $\boldsymbol{exp}$ by the actions in trs. Note that the ordering in the case of the assignments means that variables may be

$$(s, trs.p) \overset{op(exp);out(\boldsymbol{msg})}{\rightarrow} (s[\boldsymbol{var} \mapsto \boldsymbol{exp}], p) \quad bexp(s) = true$$

$$\frac{(s, p_1) \overset{op(exp);out(\boldsymbol{msg_1})}{\rightarrow} (s_1, p_1'), (s, p_2) \overset{op(exp);out(\boldsymbol{msg_1})}{\rightarrow} (s_2, p_2')}{(s, p_1 || p_2) \overset{op(exp)out(\boldsymbol{msg_1} \bowtie \boldsymbol{msg_2})}{\rightarrow} (s_1 \bowtie s_2, p_1' || p_2')}$$

$$\frac{(s, p_1) \overset{op(exp);out(\boldsymbol{msg_1})}{\rightarrow} (s, p_1'), \neg\exists (s, p_2) \overset{op(exp);out(\boldsymbol{msg_2})}{\rightarrow}}{(s, p_1 || p_2) \overset{op(exp);out(\boldsymbol{msg_1})}{\rightarrow} (s, p_1' || p_2)} \quad \text{and symmetric}$$

$$\frac{(s, p_1) \overset{trs}{\rightarrow} (s', p')}{(s, p_1 + p_2) \overset{trs}{\rightarrow} (s', p')} \quad \text{and symmetric}$$

$$\frac{(s, p_2) \overset{op_2(exp_2);out(\boldsymbol{msg_2})}{\rightarrow} (s_2', p'), (s, p_1) \overset{op_1(exp_1);out(\boldsymbol{msg_1})}{\rightarrow}_x (s_1', p_1'), \neg\exists (s, p_1) \overset{op_2(exp_2);out(\boldsymbol{msg'})}{\rightarrow}}{(s, p_1; p_2) \overset{op_2(exp_2);out(\boldsymbol{msg_2} \bowtie \boldsymbol{msg_1})}{\rightarrow} (s_2' \bowtie s_1', p')}$$

Fig. 3. Structural Operational Semantics.

assigned several different values, and the last value then remains to be assigned. In the second rule, $msg_1 \bowtie msg_2$ is a non-deterministic merge of the two lists msg_1 and msg_2 and $s_1 \bowtie s_2$ a non-deterministic merge of the two states s_1, s_2 (which means that at variables where there is a conflict in the assignments, one of the two values is chosen non-deterministically). More precisely, this rule represents a set of rules for each possibility to resolve the mentioned non-deteminism. In the third rule, $\neg\exists(s,p)\overset{trs}{\rightarrow}$ means that there is no transition labelled trs from (s,p) and "symmetric" means that there is an analogous rule obtained by swapping the subscripts 1 and 2. In the last rule $\overset{trs}{\rightarrow}_x$ is defined as $\overset{trs}{\rightarrow}$, but only referring to the *exit* actions among the actions of trs (including *out* actions and variable assignments).

Given a sequence $\boldsymbol{i}$ of input messages and a process p, we write $[\![p]\!](\boldsymbol{i})$ for the set of sequences $\boldsymbol{o}$ of output messages such that there are transitions $(p_n, s_n) \overset{op_n(exp_n);out(\boldsymbol{msg_n})}{\rightarrow} (p_{n+1}, s_{n+1})$ for $n = 0, \ldots, k$ with $p_0 = p$, s_0 is the state where all variables are evaluated at $\perp$, and such that $\boldsymbol{i} = (op_n(exp_n))_n$ and $\boldsymbol{o}$ is the concatenation of the sequences $\boldsymbol{msg_n}$ (each n). This is the set of possible sequences of output messages of p given the sequence of input messages $\boldsymbol{i}$.

3 Stereotypes for safety analysis

In Figure 4 we give the suggested stereotypes, together with their tags and constraints. The constraints, which in the table are only named briefly, are formulated and explained in the remainder of the section. Figure 5 gives the corresponding tags. Note that some of the stereotypes on subsystems refer to stereotypes on model elements contained in the subsystems. For example, the constraint of the «containment» stereotype refers to contained objects stereotyped «critical» (which in turn have tags {level}). The relations between the elements of the tables are explained below in detail.

Stereotype	Base Class	Tags	Constraints	Description
risk	link, node	failure		risks
crash/performance	link, node			crash/performance failure semantics
value	link, node			value failure semantics
guarantee	link, node	goal		guarantees
redundancy	dependency, component	model		redundancy model
safe links	subsystem		dependency safety matched by links	enforces safe communication links
secrecy	dependency			assumes secrecy
safe dependency	subsystem		«call», «send» respect data safety	structural interaction data safety
critical	object	secret		critical object
containment	subsystem		prevents down-flow	information flow

Fig. 4. Stereotypes

Tag	Stereotype	Type	Multipl.	Description
failure	risk	$\mathcal{P}$({delay, corruption, loss})	*	specifies risks
goal	guarantee	$\mathcal{P}$({immediate, correct, eventual})	*	specifies guarantees
model	redundancy	{none, majority, fastest}	*	redundancy model

Fig. 5. Tags

Well-formedness rules We explain the stereotypes and tags given in Figures 4 and 5 and give examples. By their nature, some of the constraints can be enforced at the level of abstract syntax (such as «safe links»),

Risk	$\mathsf{Failures}_{none}()$
Crash/performance	{loss,delay}
Value	{corruption}

Fig. 6. Failure semantics

while others refer to the formal definitions in Section 2 (such as «containment»). Note that even checking the latter can be mechanized given appropriate tool-support.

Redundancy This stereotype of dependencies and components and its associated tag {model} can be used to describe the redundancy model that should be implemented for the communication along the dependency or the values computed by the component. Here we consider the redundancy models *none, majority, fastest* meaning that there is no redundancy, there is replication with majority vote, or replication where the fastest result is taken (but of course there are others, which can easily be incorporated in our approach).

Risk, crash/performance, value With the stereotype «physical risk» on links and nodes in deployment diagrams one can describe the risks arising when using these links or nodes, using the associated tag {failure}, which may have any subset of $\{delay, corruption, loss\}$ as its value. In the case of nodes, these concern the respective communication links connected with the node. Alternatively, one may use the stereotypes «crash/performance» or «value», which describe specific failure semantics (by giving the relevant subset of $\{delay, corruption, loss\}$): For each redundancy model R, we have a function $\mathsf{Failures}_R(s)$ from a given stereotype $s \in \{$«crash/performance»,«value»$\}$ to a set of strings $\mathsf{Failures}_R(s) \subseteq \{delay, corruption, loss\}$.

If there are several such stereotypes relevant to a given link (possibly arising from a node connected to it), the union of the relevant failure sets is considered.

This way we can evaluate UML specifications. We make use of this for the constraints of the remaining stereotypes.

As an example for a failures function, Figure 5 gives the one for the absence of any redundancy mechanism ($R = none$).

guarantee «call» or «send» dependencies in object or component diagrams stereotyped «guarantee» are supposed to provide the goals described in the associated tag {goal} for the data that is sent along them as arguments or return values of operations or signals. The goals may be any subset of $\{immediate, correct, eventual\}$. This stereotype is used in the constraint for the stereotype «safe links».

safe links This stereotype, which may label subsystems, is used to ensure that safety requirements on the communication are met by the physical layer. More precisely, the constraint enforces that for each dependency d with redundancy model R stereotyped «guarantee» between subsystems or objects on different nodes n, m, we have a communication link l between n and m with stereotype s such that

- if {goal} has *immediate* as one of its values then $\mathsf{delay} \notin \mathsf{Failures}_R(s)$,
- if {goal} has *correct* as one of its values then $\mathsf{corruption} \notin \mathsf{Failures}_R(s)$, and
- if {goal} has *eventual* as one of its values then $\mathsf{loss} \notin \mathsf{Failures}_R(s)$.

Example In Figure 7, given the redundancy model $R = none$, the constraint for the stereotype «safe links» is violated: The model does not provide the goal *immediate* given $R = none$, because the Internet communication link between web-server and client does not provide the needed safety guarantee according to the $\mathsf{Failures}_{none}(crash/performance)$ scenario.

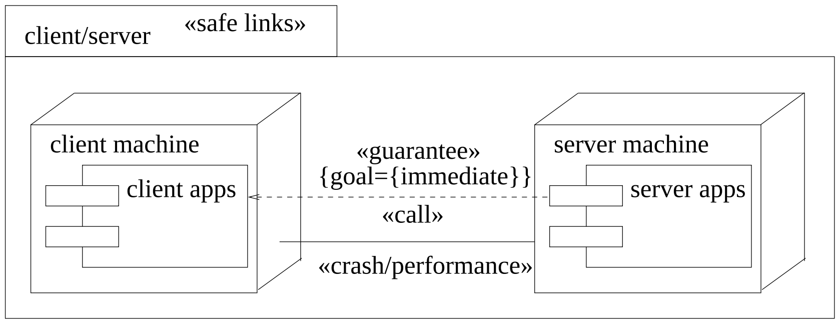

Fig. 7. Example *safe links* usage

critical We assume that we are given an ordered set *Levels* of *safety levels*. Then this stereotype labels objects whose instances are critical in some way, as specified by the associated tags {level} (for each level $level \in Levels$), the values of which are data values or attributes of the current object with the required to be protected by the given safety level. This protection is enforced by the constraints of the stereotypes «safe dependency» and «containment» which label subsystems that contain «critical» objects.

safe dependency This stereotype, used to label subsystems containing object diagrams or static structure diagrams, ensures that the «call» and «send» dependencies between objects or subsystems respect the safety requirements on the data that may be communicated along them. More exactly, we assume that each $level \in Levels$ has an associated set of goals $goals(level) \subseteq \{immediate, correct, eventual\}$. Then the constraint enforced by this stereotype is that if there is a «call» or «send» dependency from an object (or subsystem) C to an object (or subsystem) D then the following conditions are fulfilled.

– For any message name n offered by D, the safety level of n is the same in C as in D.
– If a message name offered by D has safety level $level$ and $goal \in goals(level)$, then $goal$ is one of the goals provided by the dependency.

Example Figure 8 shows a sensor/controller subsystem stereotyped with the requirement «safe dependency». We assume that $immediate \in goals(realtime)$. The given specification violates the constraint for this stereotype, since Sensor and the «call» dependency do not provide the realtime goal *immediate* for measure() required by Controller.

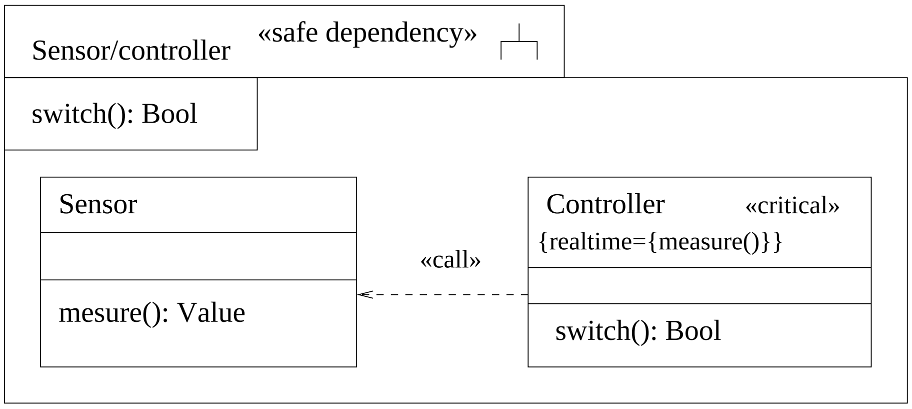

Fig. 8. Example *safe dependency* usage

Containment This stereotype of subsystems enforces safe containment following an approach proposed in [DS99] by making use of the associated safety levels. For this we define an ordering on the set $Levels$ as follows: For $l, l' \in Levels$ we have $l \leq l'$ if $goals(l) \subseteq goals(l')$.

Then the ≪containment≫ constraint is that in the stereotyped subsystem, the value of any data element of level l may only be influenced by data of the same or a higher safety level: Write $H(l)$ for the set of messages of level l or higher. Given a sequence $\boldsymbol{m}$ of messages, we write $\boldsymbol{m}\!\upharpoonright_{H(l)}$ for the sequence of messages derived from those in $\boldsymbol{m}$ by deleting all events the message names of which are not in $H(l)$. For a set M of sequences of messages, we define $M\!\upharpoonright_H \stackrel{\text{def}}{=} \{\boldsymbol{m}\!\upharpoonright_H : \boldsymbol{m} \in M\}$.

Definition 1. *Given a UML*^{light} *statechart* p *and a safety level* l*, we say that* p provides containment *with respect to* l *if for any two sequences* $\boldsymbol{\imath}, \boldsymbol{\jmath}$ *of input messages,* $\boldsymbol{\imath}\!\upharpoonright_{H(l)} = \boldsymbol{\jmath}\!\upharpoonright_{H(l)}$ *implies* $[\![p]\!]\boldsymbol{i}\!\upharpoonright_{H(l)} = [\![p]\!]\boldsymbol{j}\!\upharpoonright_{H(l)}$.

Intuitively, providing containment means that an output should in no way depend on inputs of a lower level.

Example The example in Figure 9 shows the diagrammatic representation of a Fuel Controller that computes the amount of used fuel of an airplane from the distance travelled so far. This is done (quite simplistically for the purpose of the example) by multiplying the distance with a constant (supposed to give the amount of fuel consumed per length unit). Because of different air resistance, this constant depends on the fact whether the wheels of the plane were pulled in-board or (mistakenly) left outside. This is modelled by having two states corresponding to the state of the wheels, and having different constants $c \neq d$. Now the result of the message $fuel$ is supposed to be of the level $safe$. However, the message $wheelsin$ giving the state of the wheels is not assigned any safety level. Therefore this example violates ≪containment≫, because a *safe* value depends on a value not at least of level *safe*. This can be checked using the textual representation of the UML$^{\text{light}}$ process p defined by:

$$\begin{aligned}
p &= fuel(x)[true]out(return(d.x)).p \\
&\quad +wheelsin(y)[y = false].p \\
&\quad +wheelsin(y)[y = true].p' \\
p' &= fuel(x)[true]out(return(c.x)).p \\
&\quad +wheelsin(y)[y = false].p \\
&\quad +wheelsin(y)[y = true].p'
\end{aligned}$$

Then considering the sequences $\boldsymbol{\imath} = (wheelsin(true), fuel(1))$ and $\boldsymbol{\jmath} = (wheelsin(false), fuel(1))$, and the safety level $l = safe$, we have

$\mathbf{i}\!\restriction_{H(l)} = \mathbf{j}\!\restriction_{H(l)}$, but $[\![p]\!]\boldsymbol{i}\!\restriction_{H(l)} = \{return(c)\} \neq \{return(d)\} = [\![p]\!]\boldsymbol{j}\!\restriction_{H(l)}$ since $c \neq d$ by assumption on c, d.

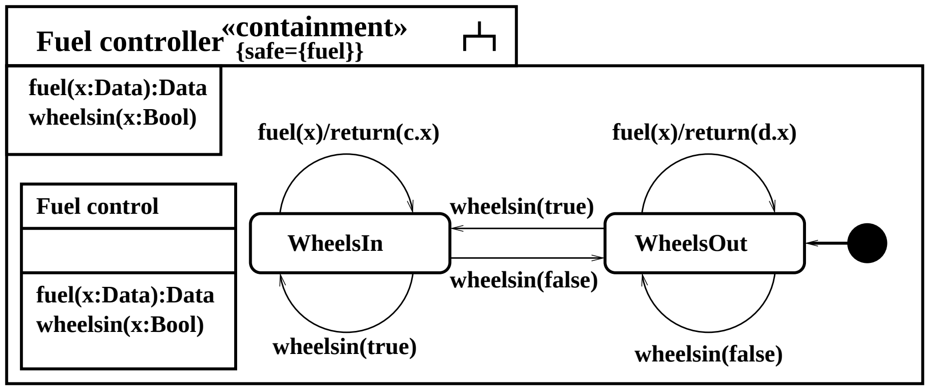

Fig. 9. Example *containment* usage

Related Work To our knowledge, this is the first work proposing to use UML for the formal development of safety-critical systems. Some of the ideas reported here were or will be presented in an invited talk at FDL'02 [Jür02a] and tutorials on formal development of critical systems with UML at Safecomp 2002, Software Engineering 2003, and ETAPS 2003 (each unpublished). [Jür02c, Jür03] proposes to use UML for developing security-critical systems. See also [JCF$^+$02] for approaches relating to other criticality requirements or for approaches to safety-critical development without a formal basis.

Also relevant is the work towards a formal semantics of UML including [LP99, KER99, RACH00, AM00, BLMF00, GI01], and notably [BLP01] which is the approach most similar to the one here (but closer to the concrete UML syntax for example by including state labels).

Research on the analysis of UML model for non-functional properties includes [LL99, DMY02].

4 Conclusion and Future Work

We proposed to use a formal fragment of UML statecharts, called UML$^{\text{light}}$, to aid development of safety-critical systems. Given the current state of dependable systems in practice, with many failures reported continually, this seems to be a useful line of research, since it enables developers without a background in dependability to make use of dependability engineering knowledge encapsulated in a widely used design notation. Since the behavioral parts of UMLsafe are considered with a formal semantics, this allows a formal evaluation (parts of which may be mechanized). Thus even dependability experts undertaking a formal evaluation for certification purposes may profit from the possibility of using a specification language that may be perceived to be more easily employed than some traditional formal methods. Since UML specifications may already exist independently from the formal evaluation, this should reduce cost of certification.

Note that one may use our approach without having to refer to a formal semantics for UML. In that case, the constraints for the safety requirements would have to be checked by a CASE tool and explained to the user informally. It is however beneficial to have a formal reference that tool providers can refer to if necessary; this is why we provide a formal semantics for the used fragment of UML; a larger fragment is given in [Jür02b].

For this line of research to be of practical value it is important to develop tool support, for example by analysing the diagram data exported from UML tools in XMI (a UML-specific XML dialect). This is currently being done for the application domain of security [Sha02, Men03], an extension to safety is planned.

In our presentation here we remained in the non-probabilistic situation to keep it easily accessible. Sometimes, in safety-critical systems, one is concerned with probabilities of system failures, although in many cases one can abstract from concrete numbers, as shown in [Jür01]. These ideas can be incorporated in our context here, as initial attempts have shown [Jür02a], which would be interesting to see worked out in detail.

References

[AM00] J. Araújo and A. Moreira. Specifying the behaviour of UML collaborations using Object-Z. In *Americas Conference on Information Systems (AMCIS)*. Association for Information Systems, 2000.

[BLMF00] J.-Michel Bruel, J. Lilius, A. Moreira, and R.B. France. Defining Precise Semantics for UML. In *ECOOP'2000 Workshop Reader*, volume 1964 of *LNCS*. Springer, 2000.

[BLP01] D. Björklund, J. Lilius, and I. Porres. Towards efficient code synthesis from statecharts. In *Workshop of the pUML-Group* [GI01].

[Cav00] A. Cavarra. *Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method.* PhD thesis, DMI, Universitá di Catania, 2000.

[CDD90] F. Cristian, R. Dancey, and J. Dehn. High availability in the advanced automation system. In *Digest of Papers, The 20th International Symposium on FaultTolerant Computing*, Newcastle-UK, June 1990. IEEE.

[DMY02] A. David, O. Möller, and Wang Yi. Formal verification of UML statecharts with real time extensions. In *FASE 2002*, volume 2306 of *lncs*, pages 218–232, 2002.

[DS99] Bruno Dutertre and Victoria Stavridou. A model of noninterference for integrating mixed-criticality software components. In *DCCA-7, Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, San Jose, CA, January 1999.

[FR99] R. France and B. Rumpe, editors. *Second International Conference on the Unified Modeling Language - UML'99*, volume 1723 of *LNCS*. Springer, 1999.

[GI01] GI. *Workshop of the pUML-Group*, Lecture Notes in Informatics, 2001.

[JCF$^+$02] J. Jürjens, V. Cengarle, E. Fernandez, B. Rumpe, and R. Sandner, editors. *Critical Systems Development with UML*, number TUM-I0208 in TUM technical report, 2002. UML'02 satellite workshop proceedings.

[Jür01] J. Jürjens. Abstracting from failure probabilities. In *Second International Conference on Application of Concurrency to System Design (ACSD 2001)*, pages 53–64. IEEE Computer Society, 2001.

[Jür02a] J. Jürjens. Critical Systems Development with UML. In *Forum on Design Languages*, Marseille, Sept. 24–27 2002. European Electronic Chips & Systems design Initiative (ECSI). Invited talk.

[Jür02b] J. Jürjens. *Principles for Secure Systems Design*. PhD thesis, Oxford University Computing Laboratory, Trinity Term 2002. Submitted.

[Jür02c] J. Jürjens. UMLsec: Extending UML for secure systems development. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 – The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425, Dresden, Sept. 30 – Oct. 4 2002. Springer. 5th International Conference.

[Jür03] J. Jürjens. *Secure Systems Development with UML*. Springer, 2003. To be published.

[KER99] S. Kent, A. Evans, and B. Rumpe. UML Semantics FAQ. In *ECOOP'99 Workshop Reader*, volume 1743 of *LNCS*. Springer, 1999.

[Lap92] J.C. Laprie. Dependability: basic concepts and terminology. *Dependable Computing and Fault-Tolerant Systems*, 5, 1992.

[LL99] Xuandong Li and J. Lilius. Timing analysis of UML sequence diagrams. In France and Rumpe [FR99], pages 661–674.

[LP99] J. Lilius and I. Porres. Formalising UML state machines for model checking. In France and Rumpe [FR99], pages 430–445.

[LT82] E. Lloyd and W. Tye. Systematic safety: Safety assessment of aircraft systems, 1982. Reprinted 1992.

[Men03] S. Meng. Secure database design with uml. Master's thesis, Munich University of Technology, 2003. In preparation.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Hußmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE2000)*, volume 1783 of *LNCS*, pages 127–146. Springer, 2000.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[Rus94] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[Sha02] P. Shabalin. Design and possibilities for automated processing of UMLsec models, 2002. Study project.

[UML01] UML Revision Task Force. OMG UML Specification v. 1.4. OMG Document ad/01-09-67. Available at http : //www.omg.org/uml, 2001.