Formal Verification at System Level

Silvia Mazzini, Stefano Puri
INTECS
Via E. Giannessi 5
Loc. Ospedaletto
I-56121 Pisa, Italy
{Stefano.Puri,Silvia.Mazzini}@intecs.it

Federico Mari, Igor Melatti, Enrico Tronci
Computer Science Department
La Sapienza University of Roma
Via Salaria 113
I-00198 Roma, Italy
{mari,melatti,tronci}@di.uniroma1.it

Abstract

System Level Analysis calls for a language comprehensible to experts with different background and yet precise enough to support meaningful analyses. SysML is emerging as an effective balance between such conflicting goals. In this paper we outline some the results obtained as for SysML based system level functional formal verification by an ESA/ESTEC study, with a collaboration among INTECS and La Sapienza University of Roma. The study focuses on SysML based system level functional requirements techniques.

1. Introduction

System Level Functional Analysis encompasses many activities among which (system level) V&V, namely: *Validation* (answering the question: *are we building the right system?*) and *Verification* (answering the question: *are we building the system right?*). Functional Analysis techniques (e.g. testing and simulation) are geared towards showing presence of errors; accordingly they can only provide evidence for a negative answer to V&V questions. On the other hand, *Formal Functional Analysis* techniques (e.g. model checking) are geared towards showing absence of errors; accordingly they can provide evidence for a positive answer to V&V questions.

The classes of system models handled by the above mentioned analysis techniques are also different. In fact, testing and simulation can handle quite detailed models as long as all inputs and parameters are defined. On the other hand, formal techniques can handle models with undefined parameters and inputs (e.g., modelling faults or disturbances) as long as such models have a *moderate* size. Thus, formal techniques can be used for a *worst case analysis* returning as output the *worst case scenario* (i.e. inputs and parameters) for the system under analysis.

The above state of affairs suggests using formal techniques as early as possible in the system design activities, namely: as soon as some system model (even qualitative) is available. In fact, this allows early detection of errors in system or subsystem specifications. For example, as for space software development, the above considerations suggests using formal techniques towards the end of phase A or at the beginning of phase B.

However, harvesting the promise of formal techniques requires a system engineering formalism that can describe in a precise, yet comprehensible, way all subsystems (components) forming the system being designed. In particular, to design and analyze software subsystems we need formalisms that can somehow describe *hybrid systems* [HybridSys], that is systems consisting of software as well as hardware (physical) components modelling the environment the software components interact with. SysML

[SysML] answers this need by providing a graphical language to define system requirements, system structure and behaviour of components (both physical as well as software). This suggests investigating the possibility of carrying out system level functional requirements formal verification starting from SysML models and translating them to the input language of a suitable model checker for hybrid systems (e.g. HyTech [HybridSys] or CMurphi [CMurphi, CMurphi-HS]).

1.1. Our Contribution

In this paper, with the help of a small yet meaningful example, we outline how SysML can be used to define models that can be translated into the input language of a model checker (HyTech in out study) and how SysML and model checking can be used to support *proof continuity*. Our contributions can be summarized as follows.

In *Section* 2 we describe SysML usage in Space System Engineering. In *Section* 3, in order to make our paper self-contained, we give some background on model. In *Section* 4 we describe our running example: the oven temperature control system (TCS). In *Section* 5 we outline how TCS can be modelled using SysML. In *Section* 6, using TCS, we describe how a SysML model can be translated into a HyTech model. In Section 7 we show our experimental results on using HyTech to verify TCS requirements. Finally, in *Section* 8 we discuss how system level formal validation results can be linked to subsystem formal verification results (*proof continuity*).

1.2. Related Works

We note that model checking of SysML models has been also studied in different contexts. Here are some examples. Model checking of SysML models by mapping them to Petri Nets has been studied in [SYsML-PetriNets-ETFA-2007]. Note that no hybrid dynamics is considered in this work. Model checking of probabilistic SysML models, by mapping them to Markov Chains, has been studied [MC-SysML-ECSB07]. Note that this work only considers finite state Markov Chains, thus hybrid dynamics cannot be handled using this approach. Mapping from SysML to System C for designing purposes has been studied in [SysML2SystemC].

2. Using SysML for Space System Engineering

SysML is a UML Profile for System Engineering intended to support modelling of a broad range of systems, which may include hardware, software, data, personnel, procedures, and facilities.

Developed by a large number of industry, government, academia and tool vendors, SysML support the analysis, specification, design, and verification of complex systems by taking advantage of significant system engineering and UML support and experience and it results in a wide accepted language that well fits to support the principles of the reference system processes defined by the ISO 15288 [ISO15288].

Modelling with SysML may support the system engineering processes according to the ECSS-ST-E10C [ECSS- ST-E10C], starting from mission requirements (mainly the mission oriented requirement document - MRD) to the definition of the whole real and operational system. Mission requirements of Phase 0/A are refined into system

requirements in Phase A, and then refined by several iterations into technical requirements, including software mainly in Phase B and Phase C.
The use of SysML as a modelling language allows to gather all the information on requirements, which may be distributed also in other different formalisms and models, that today may be used in the European Space sector, and to offer at the highest level a synthesis of the requirement facilitating their evaluation of compliance.

The SysML diagram in Figure 2.1 shows an abstraction of the kind of SysML models and the relationships among them, that should be defined in order to support the E10 processes, In the figure the high level system engineering model is partitioned into a collection of related sub-models each representing the outputs of one of the main areas of the space system engineering processes.

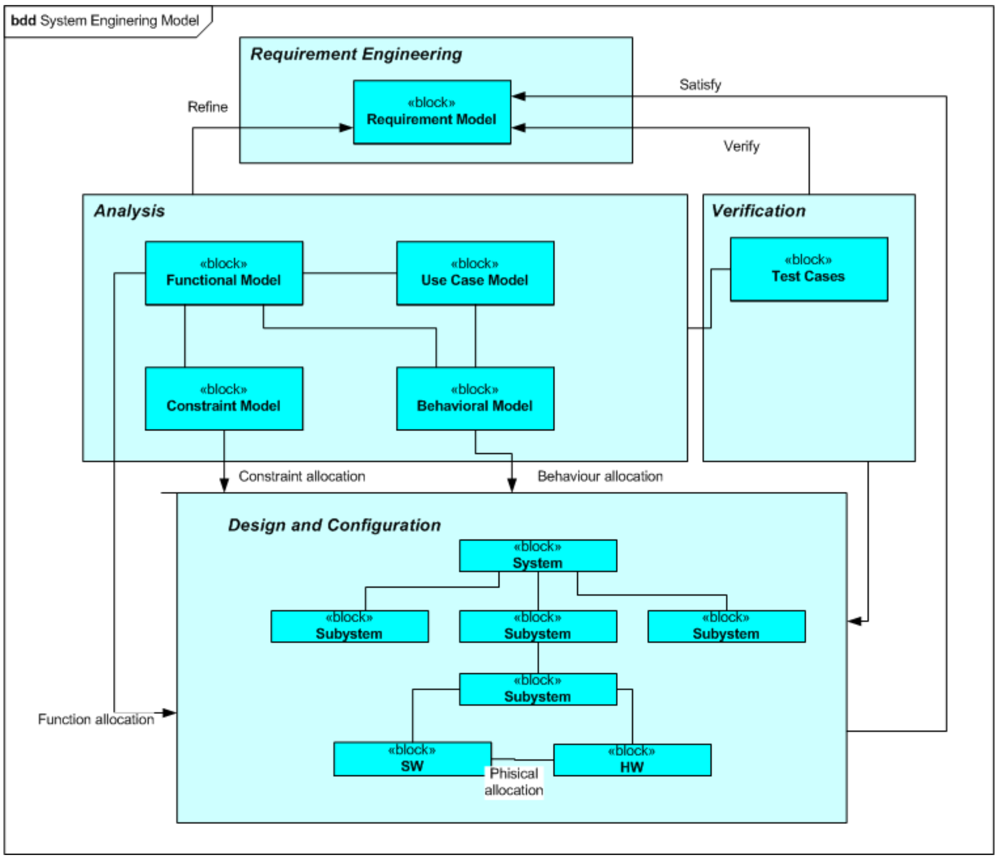

Figure 2.1. *The SysML model entities partitioned into the SE process areas*

A SysML model-based methodology to support system-software requirement capturing is currently investigated and experimented in the context of the ESA/ESTEC *System and Software Functional Requirements Techniques (SSFRT)* project, with consortium led by Intecs.

3. Background on Model Checking

A *model checker* [Model-Checking-McM, Model-Checking-Sur] takes as input a system description (using a suitable programming language) and a system property (typically defined by means of the same language used to define the system or by means of a suitable temporal logic such as CTL [Model-Checking-Sur]) and returns PASS if the system satisfies the given property, FAIL otherwise. In the latter case a model checker also shows a system run (counterexample) falsifying the given property. Figure 3.1 summarizes the above scenario.

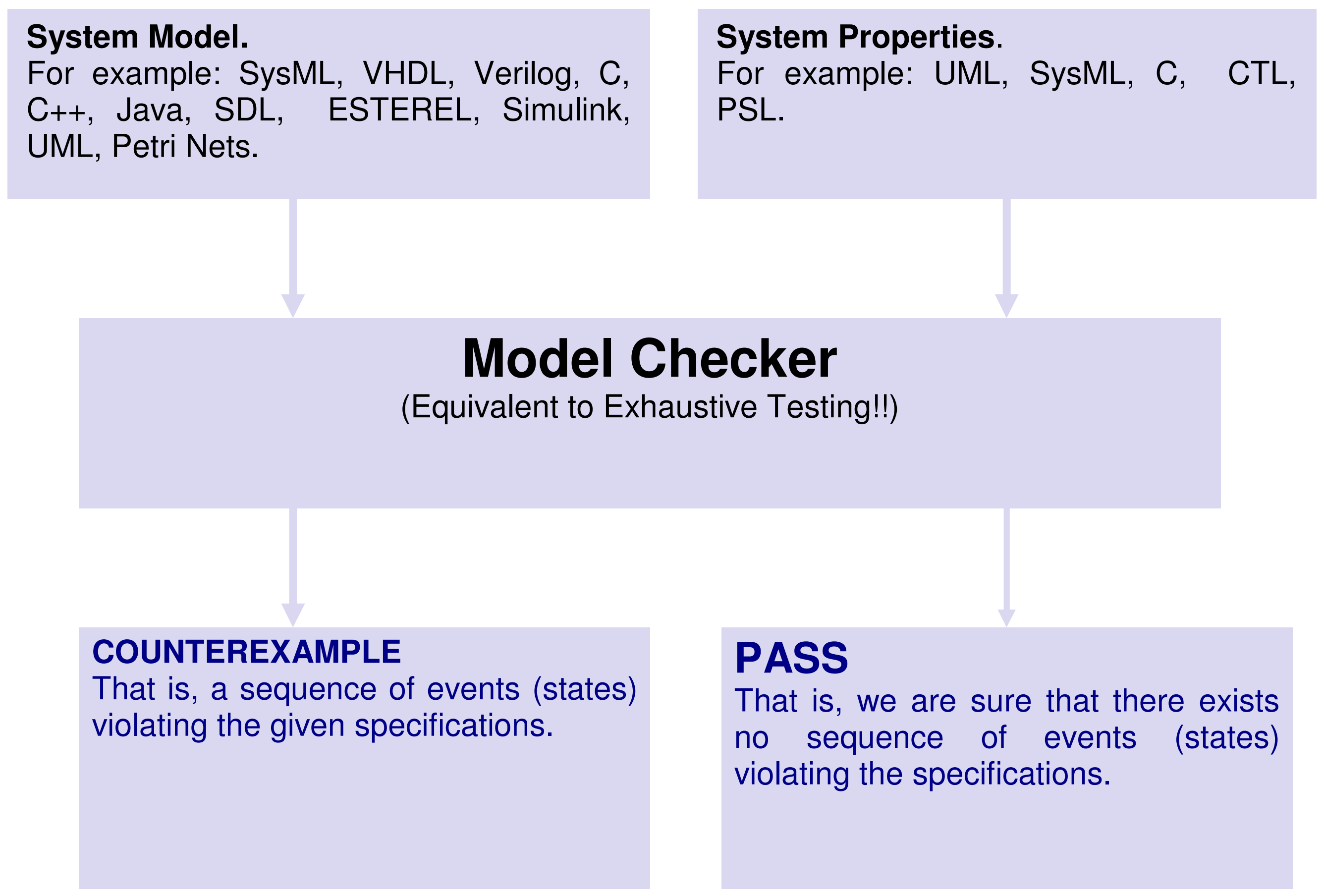
System Model.
For example: SysML, VHDL, Verilog, C, C++, Java, SDL, ESTEREL, Simulink, UML, Petri Nets.
System Properties.
For example: UML, SysML, C, CTL, PSL.
Model Checker
(Equivalent to Exhaustive Testing!!)
COUNTEREXAMPLE
That is, a sequence of events (states) violating the given specifications.
PASS
That is, we are sure that there exists no sequence of events (states) violating the specifications.

Figure 3.1. **Model Checking**

The main difference between model checking and testing is that while testing can only show presence of errors (by exhibiting an error trace) a model checker can also show absence of errors (when it returns PASS). On the other hand, testing is computationally much lighter than model checking. Indeed, the main obstruction to model checking is the *state explosion* problem, that is the fact that the number of states of a system may be exponential in the size of the description of the system itself. For example a procedure with just one 32-bit integer variable may generate 2^{32} possible states. Analogously an array of size n of integer variables yields 2^{32n} states. The success of model checking rests on the fact that efficient techniques have been devised to counteract the state explosion problem (which cannot be eliminated being the reachability problem PSPACE complete).

Many model checking algorithms and tools are available. Depending on the application domain one approach will work better than another. For this reason many model checkers are available each targeting a particular class of systems. System level analysis requires considering models describing software behaviour (*discrete state changes*) as well as models describing the behaviour of physical devices (*continuous state changes*). This calls for model checkers for hybrid systems such as HyTech [HybridSys], UPPAAL [UPPAAL], CMurphi [CMurphi, CMurphi-HS], HYSDEL [HYSDEL], HSMV [HSMV]. Since the systems we consider here can be modelled using Linear Hybrid Automata [HybridSys] we will use HyTech in this study.

4. Description of our Running Example: A Temperature Control System (TCS)

To clarify the forthcoming discussion we will use as a running example the (system level) modelling of a simple oven temperature regulator. This is indeed a typical on-board equipment.

We are given a heater that can be ON or OFF. The heater is used to keep an oven (for example for on-board instrumentations) always in a certain interval of temperature. For our purposes we can assume that the oven temperature τ follows (approximately) the following equations:

$a_1 \leq d\tau/dt \leq a_2$	(Heater ON) and ($\tau < \tau_{MAX}$)	(Eq. 1)
$d\tau/dt = 0$	($\tau \geq \tau_{MAX}$) or ($\tau \leq \tau_{MIN}$)	(Eq. 2)
$b_1 \leq d\tau/dt \leq b_2$	(Heater OFF) and ($\tau > \tau_{MIN}$)	(Eq. 3)

where:

- Variable $\tau(t)$ is the oven temperature at time t,
- Constants a_1 and a_2 give, respectively, the lower and upper bounds for the oven heating speed.
- Constants b_1 and b_2 give, respectively, the lower and upper bounds for the oven cooling speed.
- T_{MIN} is the min temperature the oven will reach if the heater is kept OFF.
- T_{MAX} is the max temperature the oven will reach if the heater is kept ON.

Intuitively, the speed of variation of the oven temperature ($d\tau/dt$) is bounded as shown in Equations 1, 2, 3. The oven temperature increases with a speed in $[a_1, a_2]$ ($[b_1, b_2]$) when the heater is on (off). Since b_1 and b_2 are negative we have that when the heater is off the temperature actually decreases. Furthermore, from Eq. 2 we see that the temperature cannot rise above T_{MAX} when the heater is on and cannot drop below T_{MIN} when the heater is off.

The goal of the temperature regulator is to keep the oven temperature always in the interval $[T_1, T_2]$ with $T_1 = 13$℃ and $T_2 = 17$℃. To this end the temperature regulator will measure the oven temperature and will turn the heater ON whenever the temperature falls below $T_{low} = 14$℃ and will turn the heater OFF whenever the temperature rises above $T_{high} = 16$℃.

The following table summarizes the value for the system parameters as well as their meaning.

Parameter Name	**Value**	**Short Description**
a_1	0.1 ℃/s	Min heating speed
a_2	0.2 ℃/s	Max heating speed
b_1	-0.02 ℃/s	Value $\|b_1\|$ gives the max cooling speed
b_2	-0.01 ℃/s	Value $\|b_1\|$ gives the min cooling speed
T_{MIN}	-50 ℃	Min temperature achievable
T_{MAX}	30 ℃	Max temperature achievable
T_1	13 ℃	Min safe temperature
T_2	17 ℃	Max safe temperature
T_{low}	14 ℃	Turn on temperature for the heater
T_{high}	16 ℃	Turn off temperature for the heater
SNSR_TOL	To be defined	Max error in sensor reading
DLY	To be defined	Max delay in heater controller

Table 4.1. System Parameters

The values for the parameters SNSR_TOL and DLY will be an output of the system level analysis activity. In fact, in order to save on hardware, we would like to make SNSR_TOL and DLY as large as possible. However, if SNSR_TOL or DLY are too large the controller may not be able to keep the oven temperature within the desired range. We would like to get admissible values for SNSR_TOL and DLY from the formal system level analysis activity.

5. SysML Model for the Temperature Control System

Our *Temperature Control System* (TCS) consists of a sensor measuring the oven temperature, an oven heater and a controller that turns the oven heater ON or OFF. The controller itself is a program running on a CPU on which other processes are also running. In the following we outline a SysML model for TCS and its requirements.

The SysML bdd in Figure 5.1 defines the system requirements (with the block `<<requirement>>`). The `requirement` block states that the oven temperature `temp` should be between 13° C and 17° C (T_1 and T_2 from Table 4.1). Figure 5.2 shows TCS structure. From such figure we see that TCS consists of 3 subsystems: a sensor, a controller and an oven.

Block `Sensor` variables are: `sensorMode` (ranging on STM `Sensor` states, i.e. SensorModeType) and `temp` (ranging on oven temperature). Block `Sensor` signals are `LowTemp` and `HighTemp`. Block `Ctr` variables are: `ctrMode` (ranging on STM `Ctr` states, i.e. CtrModeType) and `timer` measuring sojourn times in STM `Ctr` states. Block `Ctr` signals are `HeaterOn` and `HeaterOff`. Block `Oven` variables are: `ovenMode` (ranging on STM `Oven` states, i.e. OvenModeType) and `temp` (ranging on oven temperature).

Figure 5.2 shows the internal block structure for TCS. It is a classical feedback control system where the controller (`Ctr`) turns the oven heater (`Oven`) *on* or *off* depending on the oven temperature readings (`Sensor`). Note that for easy of presentation, our sensor reads the oven temperature and also sends signals triggering the controller.

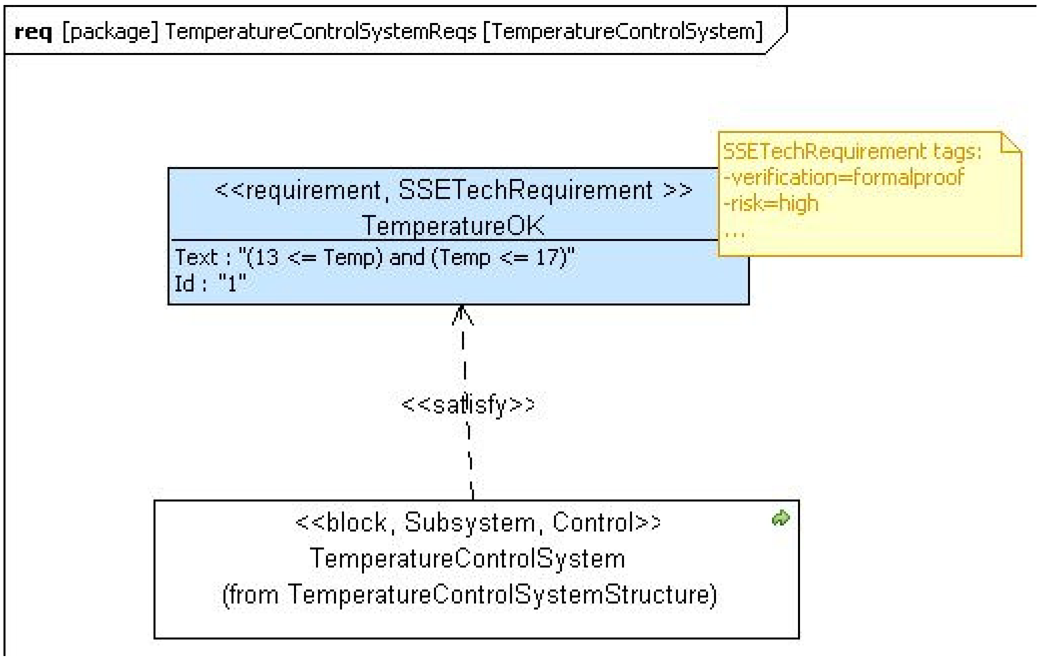
req [package] TemperatureControlSystemReqs [TemperatureControlSystem]
<<requirement, SSETechRequirement >>
TemperatureOK
Text : "(13 <= Temp) and (Temp <= 17)"
Id : "1"
SSETechRequirement tags:
-verification=formalproof
-risk=high
...
<<satisfy>>
<<block, Subsystem, Control>>
TemperatureControlSystem
(from TemperatureControlSystemStructure)

Figure 5.1. *SysML TCS Requirements*

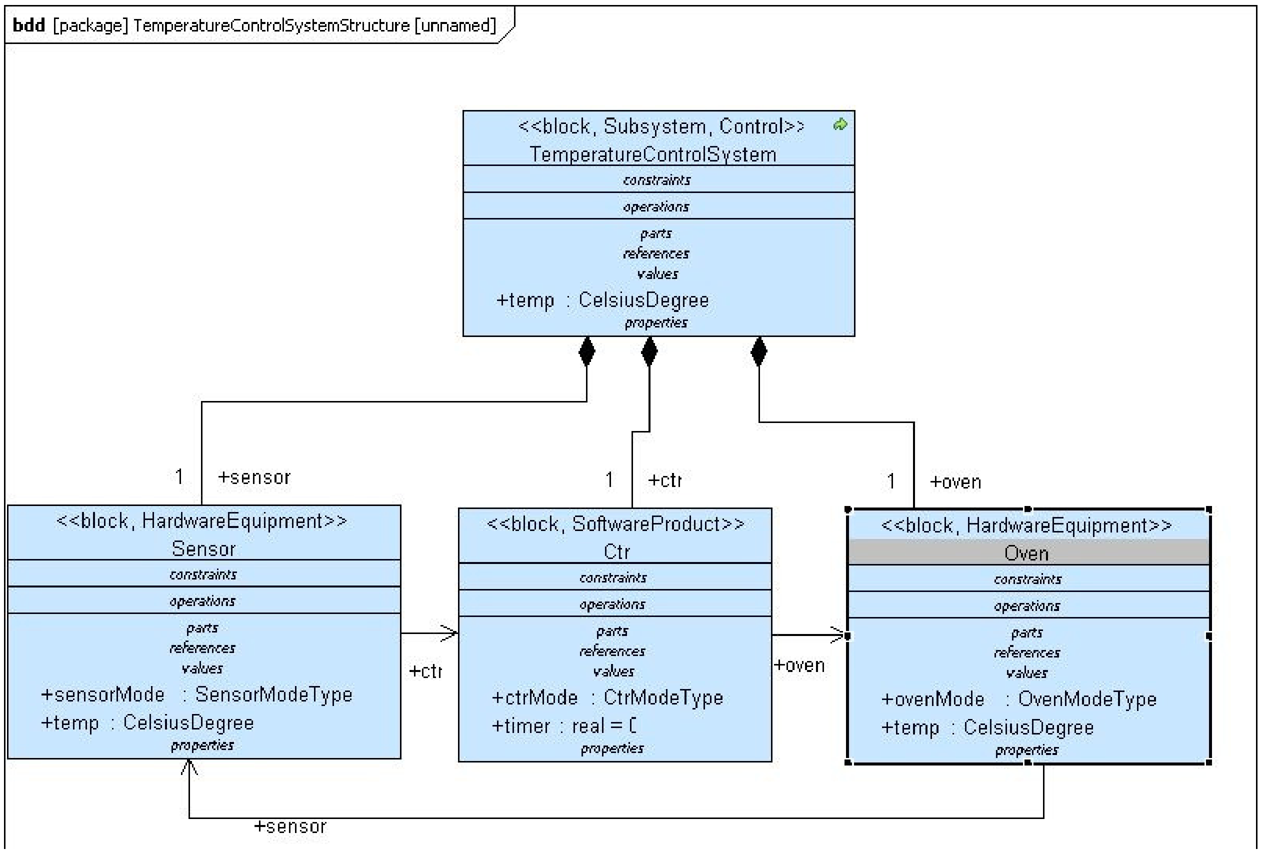

Figure 5.2. *SysML TCS Subsystems*

In the following we outline our SysML model for `Oven`. SysML models for `Sensor` and `Ctr` are obtained along the same line of reasoning.

Figure 5.3 shows block `Oven` constraints, Figure 5.4 shows the `Oven` state machine.

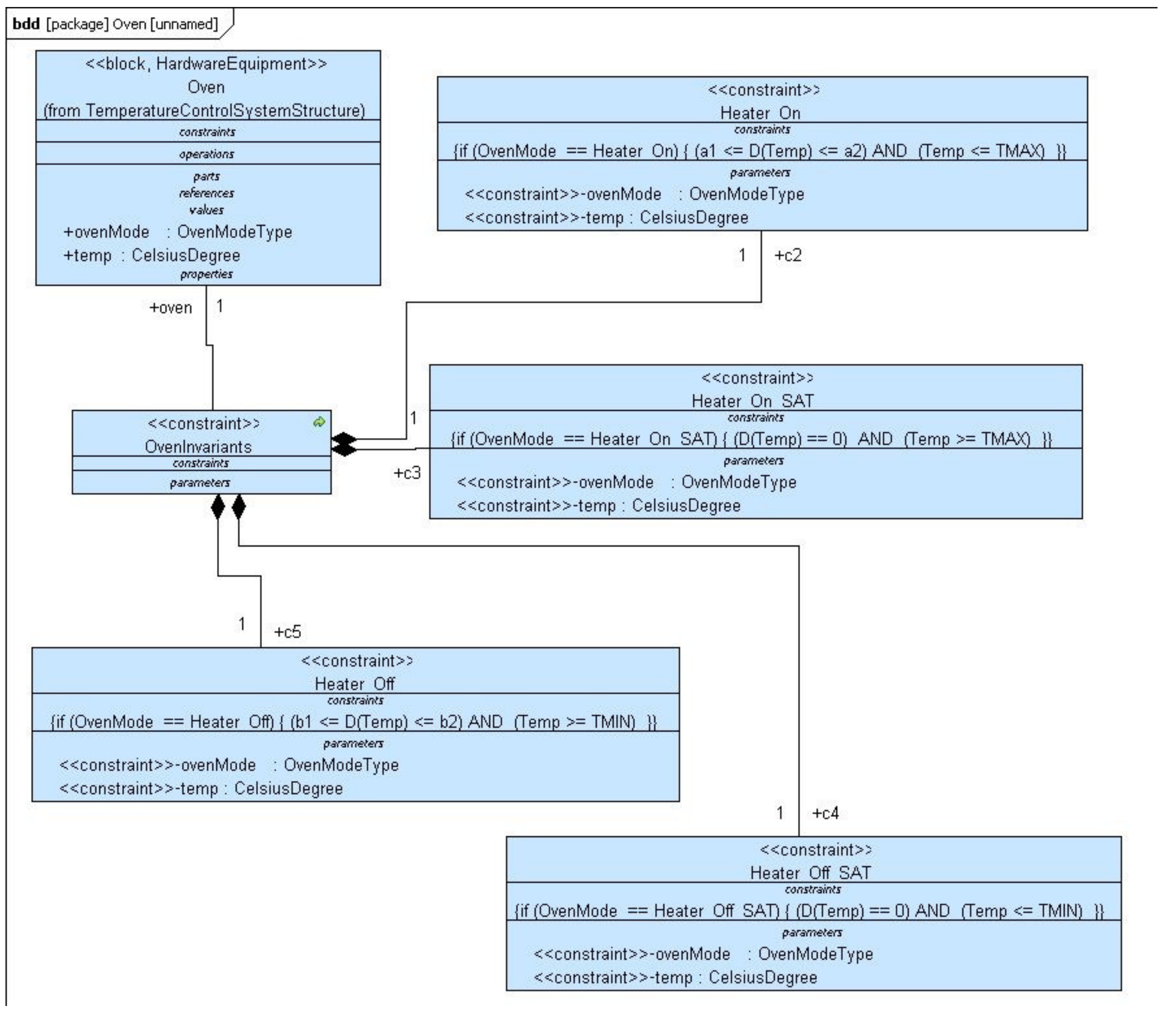
bdd [package] Oven [unnamed]
<<block, HardwareEquipment>>
Oven
(from TemperatureControlSystemStructure)
constraints
operations
parts
references
values
+ovenMode : OvenModeType
+temp : CelsiusDegree
properties
+oven
1
<<constraint>>
Heater_On
constraints
{if (OvenMode == Heater_On) { (a1 <= D(Temp) <= a2) AND (Temp <= TMAX) }}
parameters
<<constraint>>-ovenMode : OvenModeType
<<constraint>>-temp : CelsiusDegree
1
+c2
<<constraint>>
OvenInvariants
constraints
parameters
<<constraint>>
Heater_On_SAT
constraints
{if (OvenMode == Heater_On_SAT) { (D(Temp) == 0) AND (Temp >= TMAX) }}
parameters
<<constraint>>-ovenMode : OvenModeType
<<constraint>>-temp : CelsiusDegree
1
+c3
1
+c5
<<constraint>>
Heater_Off
constraints
{if (OvenMode == Heater_Off) { (b1 <= D(Temp) <= b2) AND (Temp >= TMIN) }}
parameters
<<constraint>>-ovenMode : OvenModeType
<<constraint>>-temp : CelsiusDegree
1
+c4
<<constraint>>
Heater_Off_SAT
constraints
{if (OvenMode == Heater_Off_SAT) { (D(Temp) == 0) AND (Temp <= TMIN) }}
parameters
<<constraint>>-ovenMode : OvenModeType
<<constraint>>-temp : CelsiusDegree

Figure 5.3. *SysML Oven Constraints*

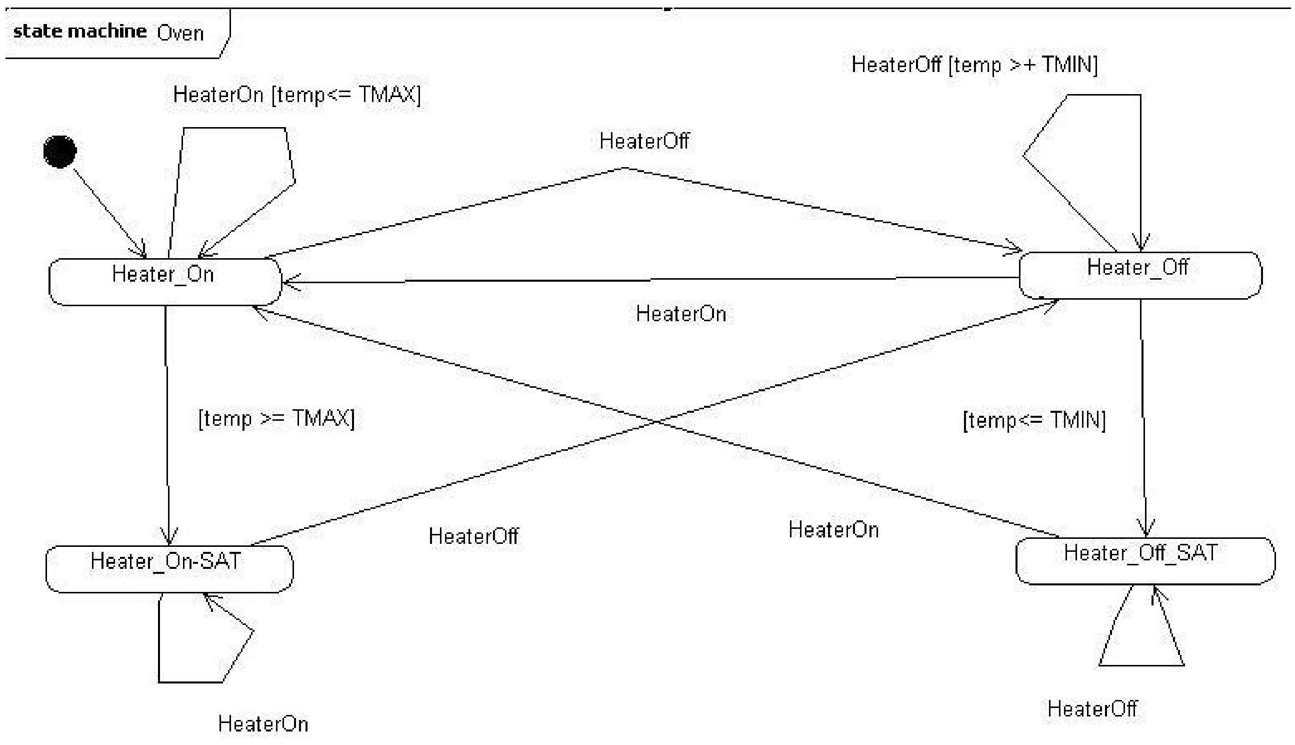

Figure 5.4. *SysML Oven STM*

State machine `Oven` in Figure 5.4 defines the possible oven modes: the heater is on (`Heater_On`), the heater is off (`Heater_Off`), the maximum temperature has been reached (`Heater_On_SAT`), the minimum temperature has been reached (`Heater_Off_SAT`).

Constraint `Heater_On` in Figure 5.3 defines the invariant for state `Heater_On` (i.e. `temp <= TMAX`) as well as the oven temperature `temp` dynamics (i.e. `a1 <= D(temp) <= a2`) accordingly to Eq. 1 in Section 4.

Constraint `Heater_Off` in Figure 5.3 defines the invariant for state `Heater_Off` (i.e. `temp >= TMIN`) as well as the oven temperature `temp` dynamics (i.e. `b1 <= D(temp) <= b2`) accordingly to Eq. 3 in Section 4.

Constraint `Heater_On_SAT` in Figure 5.3 defines the invariant for state `Heater_On_SAT` (i.e. `temp >= TMAX`) as well as the oven temperature `temp` dynamics (i.e. `D(temp) = 0`) accordingly to Eq. 2 in Section 4.

Constraint `Heater_Off_SAT` in Figure 5.3 defines the invariant for state `Heater_Off_SAT` (i.e. `temp <= TMIN`) as well as the oven temperature `temp` dynamics (i.e. `D(temp) = 0`) accordingly to Eq. 2 in Section 4.

When `Oven` in Figure 5.4 receives a signal `HeaterOn` (`HeaterOff`) from the controller (`Ctr`) it turns the heater on (off) thus moving to state `Heater_On` or `Heater_On_SAT` (`Heater_Off` or `Heater_Off_SAT`).
Note how using SysML constraint blocks and state machines we can easily model systems with different operational modes.

6. Mapping SysML Models to HyTech

As discussed in Section 3, all model checkers have essentially two inputs: a *formal model* and a *formal specification*. The (formal) model defines the behaviour of the system at hand whereas the (formal) specification defines a property that the model is supposed to satisfy. The model checker will then check if indeed the model does satisfy the given specification and, if not, it will return a counterexample, that is, a model execution trace falsifying the given specification.

From the above it is clear that in order to map SysML models into the input language for a model checker we need to build a formal model and a formal specification starting from the given SysML model. The natural approach is to map SysML blocks defining requirements to formal specifications and SysML blocks defining behaviours to formal models. Of course, the way this is actually done, depends on how the target model checker defines models and specifications.

For example, if we use a model checker like SMV [SMV] or SPIN [SPIN] then we have a language to define (temporal logic) formal specifications and another different language to define the system formal model. On the other hand, many model checkers for hybrid systems (including HyTech) use the same language to define both the formal model and the formal specifications. This is due to the fact that such model checkers only handle safety properties that, in turn, can be easily defined as set of (safe) system states. Set of states can be easily defined with a *monitor* returning *true* if a state is in the set and *false* otherwise. Such monitor can be easily defined using the model checker modelling language which is then also used to define safety specifications. In such cases the model checking problem comes down to check that no unsafe state can be reached from an initial state during the system evolution. Accordingly, when using such kind of model checkers, both SysML requirement blocks and behavioural blocks are mapped into the model checker system model with requirement blocks being used to define the set of safe (or unsafe) states.

Model checkers expect a formal definition of the system behaviour. Thus, to enable translation from SysML to HyTech a formal semantics for SysML diagrams describing behaviours has to be given. This can be done by suitably refining SysML semantics. In the following we show how we can build a HyTech model for the `TemperatureControlSystem` SysML model in Section 5. In Section 7 we will present some system level formal validation result using our HyTech model.

6.1. HyTech Model for TCS

Formal validation of SysML models enables system level requirement formal validation. This, in turn, allows detection of errors since from the first system engineering activities (such as definition of system requirements). In order to support automatic formal validation of SysML models in this section we show how mapping from SysML models to HyTech can be done. Of course, the actual translation strategy depends on how SysML models are defined (for example the language used to define the continuous dynamics) as well as on the target model checker. To give concreteness to our presentation we will present the general principles underlying the mapping strategy together with a running example showing how the general mapping strategy is to be applied on a small yet meaningful example. HyTech models systems as a set of *Linear Hybrid Automata* (LHA). An LHA consists of a finite set of *locations* and a finite set of *real valued variables*. A state *s* of an LHA is a pair (*loc, val*) where *loc* is a location and *val* is an assignment of values to the LHA real variables.

Time is continuous and only elapses when the LHA does not change location. On the other hand, transitions form a location to a different one are assumed instantaneous. An LHA can only stay in a location when the *invariant* for that location is true. An LHA moves from a location to another by making *transitions*. LHA transitions are *guarded*, thus a transition can be taken only if its guard is true. LHA variables may be assigned a value (*reset condition*) when taking a transition.

LHA define the dynamics for the continuous variables by defining linear bounds for their first time derivatives (*rate condition*). Each LHA location has its rate condition (as well as its invariant).

Transitions may have *synchronization labels*. A transition with synchronization label *a* can only be taken together with another transition (in a different LHA) labelled with also with *a*.

As for most model checker, HyTech input is textual. However LHA have a helpful graphical representation that we will use to illustrate our mapping from SysML to HyTech. When using a graphical representation an LHA is seen as a labelled directed graph which vertices represent LHA locations and edges LHA transitions. Accordingly, each vertex is labelled with the location name, its invariant and its rate condition whereas each transition is labelled with its guard, reset condition and synchronization label, if any.

6.2. Oven Model

Figure 6.1 shows our LHA for the oven model from Section 5. We see that locations, transition and transition guards in Figure 6.1 are those in the STM in Figure 5.4 of Section 5. State invariants and rate conditions in Figure 6.1 are obtained from the constraints in Figure 5.3 in Section 5. In Figure 6.1 we have denoted the fist time derivative of variable *x* with *D(x).*

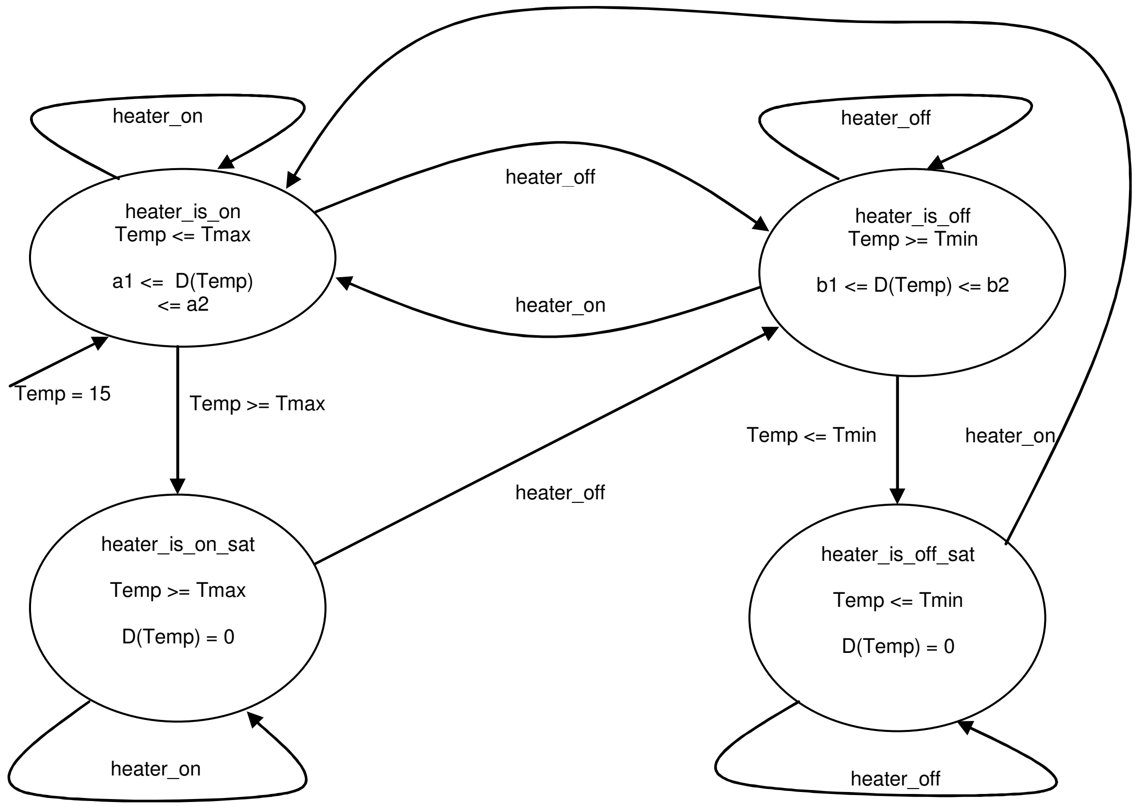

Figure 6.1. *Oven LHA*

7. System Level Validation Results

HyTech, as most model checkers for hybrid systems, only handles safety properties. This is done by defining a set of unsafe states and then carrying out a reachability analysis to check that no unsafe state is reachable from an initial state. For HyTech this is done by defining the set of *error states*, that is the set of states that violates the safety requirements, and then asking HyTech to check if there exists an error state reachable from an initial state.

As for TCS, from the `<<requirement>>` block in Figure 5.1 in Section 5 we see that the set of error state is defined as the set of states satisfying the constraint: "`(Temp <= 13) or (Temp >= 17)`". Running HyTech returns the conditions under which the safety property defined in the `<<requirement>>` block in Figure 5.1 in Section 5 is violated. The results are in Figure 7.1.

Conditions under which TCS violates safety requirements (HyTech output).	(SNSR_TOL > 1) OR (5*SNSR_TOL + DLY > 5)

Conditions under which TCS meets safety requirements.	(SNSR_TOL <= 1) AND (5*SNSR_TOL + DLY <= 5)

Figure 7.1. *Validation Results*

Figure 7.2 gives examples of safe and unsafe settings for our system (SNSR_TOL) and software (DLY) parameters.

Setting ID	SNSR_TOL	DLY	System Level Formal Validation Result
1	1.5	2	FAIL
2	0.5	2	PASS
3	0.5	2.4	PASS
4	0.5	3	FAIL

Figure 7.2. *Safe and Unsafe Settings*

8. Linking SysML Formal Validation to Subsystem Formal Verification

Typically subsystems are not defined using SysML but rather using specialized languages. For example: a subsystem implemented with digital hardware may be defined using Verilog, VHDL, SystemC, Esterel; a subsystem implementing a control system may be defined using Simulink; a subsystem implemented using software may be defined using SDL, C or Java; a subsystem detailed specification may be defined using Petri Nets. However, in order to guarantee *proof continuity* a formal link must be established between such subsystem definitions and the SysML system level model. Failing of establishing such link will make our system model a throw-away model without any formal link to the following design steps. As a result, we may formally validate a system level design that however may have nothing to do with the actual system built.

We can support *proof continuity* in our framework by using an *assume-guarantee* verification approach to establish a formal link between system level validation and subsystem verification. To this end we proceed as follows.

First, resting on the *assumption* that subsystems (software or hardware) meet their requirements, we verify that the overall system meets its system level requirements. This verification activity indeed validates the subsystems requirements since it shows that subsystem requirements are correct (i.e. we are building the *right* subsystems). For example, the results in Section 7 verify (*guarantee*) the TCS satisfies its requirements (in Section 5) *assuming* that the implementations of TCS subsystems (i.e. sensor, controller, oven) behave accordingly to their specifications. Note that the requirements being validated here are those of the subsystems.

Second, we *guarantee* that indeed the subsystems (software or hardware) meet their requirements. This is done by showing, for example by using model checking techniques, that the implementation of each subsystem satisfies the specifications defined using its SysML model. Of course, depending on how the subsystem is implemented, a suitable model checker will be used. For example, a hardware subsystem can be formally verified using a hardware model checker (e.g. [Cadence]) whereas a control system defined using Simulink could be formally verified using Lustre [Simulink2Lustre].

Note that, as for model checking purposes, in the system level validation activity we map SysML requirements into formal specifications and SysML behavioural models into formal models. On the other hand, when using model checking tools to link subsystems formal verification to system level validation, we map SysML model for subsystems into formal specifications and implementations for subsystems to formal models. Figure 8.1 summarizes the outlined approach.

Assume-Guarantee Verification Approach within SysML

System Level Validation:
SysML Requirements → Model Checker Specification,
SysML models → Model Checker Model

Subsytem Verification:
SysML models → Model Checker Specification,
Subsystem implementation (or model) → Model Checker Model

Figure 8.1. *Assume-Guarantee Verification Approach*

9. Conclusions

Our investigation shows that, with some semantic refinements, SysML can effectively be used to define system requirements and behavioural models suitable for formal verification via model checking. We note that system level formal verification is indeed a validation for the subsystem models since it shows that if the subsystems behave accordingly to their models then the system requirements will be satisfied. Accordingly, subsystem models can then be used as specifications for the implementation of subsystems. This, in turn, defines a formal link between system level requirements analysis and design of subsystems (*proof continuity*).

References

[Cadence] Cadence Incisive Platform: http://www.cadence.com/products/functional_ver

[CMurphi] Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci, Marisa Venturini Zilli: *Exploiting transition locality in automatic verification of finite-state concurrent systems*, International Journal on Software Tools for Technology (STTT) 6(4): 320-341 (2004)

[CMurphi-HS] G. Della Penna, B. Intrigila, I. Melatti, M. Minichino, E. Ciancamerla, A. Parisse, E. Tronci, and M. Venturini Zilli. *Automatic verification of a turbogas control system with the murphi verifier*. In Proc of: Hybrid Systems: Computation and Control (HSCC) Prague, Czech Republic, Lecture Notes in Computer Science, Vol. 2623, Springer, 2003.

[CMurphi] Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci, Marisa Venturini Zilli: *Exploiting transition locality in automatic verification of finite-state concurrent systems*, International Journal on Software Tools for Technology (STTT) 6(4): 320-341 (2004)

[ContDyn-SysML-08] Thomas Johnson, Christiaan Paredis, Roger Burkhart, Integrating Models and Simulations of Continuous Dynamics into SysML, Technical report, Jan 2008

[ECSS-ST-E10C] ECSS E-ST-10C, Space Engineering, System Engineering General Requirements, 6 March 2009.

[HSMV] Federico Mari, Enrico Tronci. *CEGAR Based Bounded Model Checking of Discrete Time Hybrid Systems*. 10th International Conference on “Hybrid Systems: Computation and Control” (HSCC), 3-5 Aprile 2007, Pisa, Italy, LNCS, Springer.

[HybridSys] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, *Automatic symbolic verification of embedded systems*, IEEE Transactions on Software Engineering 22:181-201, 1996.

[HYSDEL] F. D. Torrisi, A. Bemporad, *HYSDEL - a tool for generatine computation hybrid models*, IEEE Trans. On Control Systems Technology, 12(2):235-249, March 2004

[ISO15288] ISO/IEC 15288, System engineering, System life cycle processes, 2008.

[Model-Checking-McM] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.

[Model-Checking-Sur] Edmund M. Clarke, Orna Grumberg, Doron A. Peled, *Model Checking*, Cambridge, Mass., MIT Press, 1999.

[MC-SysML-ECSB07] Yosr Jarraya, Andrei Soeanu, Mourad Debbabi, Fawzi Hassaıne, Automatic Verification and Performance Analysis of Time-Constrained SysML Activity Diagrams, Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)

[Simulink2Lustre] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, F. Maraninchi, *Defining and Translating a “Safe” Subset of Simulink/Stateflow into Lustre*, EMSOFT’04, September 27–29, 2004, Pisa, Italy, ACM

[SPIN] Gerard J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional (September 4, 2003)

[SysML] OMG Systems Modelling Language(OMG SysML), V. 1.1, downloadable from http://www.omg.org, May 2008.

[SYsML-PetriNets-ETFA-2007] Marcos V. Linhares, Rˆomulo S. de Oliveira, Jean-Marie Farines, Francois Vernadat, Introducing the Modeling and Verification process in SysML, ETFA'2007 - 12th IEEE Int. Conf. on Emerging Technologies and Factory Automation.

[SysML2SystemC] Mauro Prevostini, Elena Zamsa, SysML Profile for SoC Design and SystemC Transformation, Technical Report, Univ. of Lugano, http://www.prevostini.ch/publications.htm

[SMV] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. *Symbolic model checking: 10^20 states and beyond*. Inf. Comput., 98(2):142-170, 1992.

[UPPAAL] Kim G. Larsen, Paul Pettersson, and Wang Yi. *Uppaal: Status and developments*. In Orna Grumberg, editor, Computer Aided Verification, 9th International Conference, CAV ’97, Haifa, Israel, June 22-25, 1997, Proceedings, volume 1254 of Lecture Notes in Computer Science, pages 456–459. Springer, 1997.