Complementary Verification of Embedded Software using ASD and Uppaal

Richard Doornbos
Embedded Systems Institute
Eindhoven
The Netherlands

Jozef Hooman
Embedded Systems Institute
Radboud University Nijmegen
The Netherlands

Bernard van Vlimmeren
FEI Company
Eindhoven
The Netherlands

***Abstract*—To increase the confidence in the correctness of software components, we investigated the use of two complementary formal methods in industrial software development. We combine a commercial refinement checker, the ASD:Suite of the company Verum, with the academic verification tool Uppaal to encompass a larger range of verification possibilities. Wheras the ASD:Suite is based on the compositional verification of a single component with respect to its interface, Uppaal concentrates on the global verification of a closed system. Another difference is that ASD:Suite includes code generation from formal models, whereas Uppaal allows model simulation. The combination of the two tools has been applied in industry on a case study of a camera protection system.**

I. Introduction

We show how a combination of complementary formal methods can be used to increase the confidence in embedded software. The approach emerged from the usage of a formal approach in industry - namely at FEI Company - and experiments with additional tooling.

A long standing industrial problem in the field of embedded systems concerns the correctness of the embedded software. How can we increase confidence in our software designs, and how can we detect errors early? This is extremely advantageous as it prevents long test and integration times, and by that limits cost.

An approach to this problem is the use of formal methods. At FEI Company the Analytical Software Design (ASD) approach of the company Verum is used to develop control software. ASD [1], [2] is based on a formalization of the Cleanroom [3], [4] in CSP [5]. Verum's commercial tool ASD:Suite hides the formal CSP details for the user. It provides a tabular notation to specify state machines which are used to define interface models as well as design models. For industrial users, the most important reason for using ASD is that code can be generated from verified design models, shifting most of the effort from plain coding to the modeling level.

The ASD:Suite uses the FDR2 refinement checker [6] to verify that a design model conforms to a specification model. Scalability is obtained by restricting the modeling and verification to data-independent control decisions and the use of a compositional approach [7]. This means that the verification of a component is based only on the interfaces of the components it uses. These used components can be developed and verified separately according to their interfaces.

To obtain a user-friendly scalable tool, the ASD approach only checks a fixed set of properties. This approach certainly helps to detect errors early. Inspired by this success and the needs expressed by industrial users for extended verification capabilities, we experimented with additional support using the Uppaal tool to check other properties and increase the range of faults that can be detected early.

The Uppaal tool was chosen because of its nice and understandable user interface and simulation possibilities, which appeared to be attractive for industrial users. Another reason is that the tool was already known to some of the industrial users from post-graduate courses.

Uppaal is an integrated environment for modeling, validation and verification of systems modeled as networks of automata. The tool uses timed automata with synchronous communication along channels, extended with data types (bounded integers, arrays, etc.). For a thorough treatment of Uppaal see [8], [9].

We observed that ASD and Uppaal are complementary in many respects. The main differences are listed in Table I.

TABLE I
Comparison of ASD and Uppaal

ASD	Uppaal
tabular representation of state machines	visual representation of timed automata
refinement check - standard properties - limited checks - focus on traces	property check - user defined properties - expressive specification language - focus on sequences of states
specific execution model	generic system modelling
code generation	no code generation
no simulation	simulation
single component	multiple components
compositional, scalable - lower risk of state space explosion	requires closed model - larger risk of state space explosion
no timing	timing

Most relevant for the study described here is the complementary nature of the verification approaches of both methods.

978-1-4673-1101-4/12/$31.00 ©2012 IEEE

ASD performs refinement checks with respect to the interface, in contrast to user-defined properties by Uppaal. We want to investigate how to exploit both verification capabilities by combining them such that we get more confidence that the generated code actually delivers the intended system properties.

As far as we know, the coupling of the industrial software tool ASD and a formal property verification tool such as Uppaal is new. Uppaal has been combined with other techniques, such as a tool for modeling and performance simulation [10]. Related to our combination is work on a translation from FDR2 models to Uppaal [11]; whereas that work concentrates more on the semantics of the translation, our main interest is to investigate industrial applicability.

This paper is structured as follows. Section II introduces the case study. The ASD approach is presented in Section III. Section IV describes the translation to Uppaal. The Uppaal verification techniques are applied to the case study in Section V. Finally, Section VI contains concluding remarks.

II. Case Study

The ASD approach is used in a pilot project at FEI Company, a company building electron microscopes, for a camera safety system. This system under development should guarantee that a particular expensive and very sensitive camera is protected against a too high dose of electrons, at all times. An important part of this system is the software that keeps track of the location of the camera, the blocking of the electron beam by other components and the intensity of the electron beam.

In the case study described here, we selected a crucial part of the large set of ASD models that are used to specify and design the software for the camera protection. The selected models are shown in Figure 1, expressed in UML. The clients of the dose protection system are allowed to call the methods described in the interface IDoseProtector. The design model DDoseProtector implements this interface, using the interfaces of two other components, IBlankerShutter and ISafetyList. The SafetyList component calculates whether the intensity of the beam is safe for the camera. If not, the BlankerShutter is asked to block the beam, which is called ”blanking”.

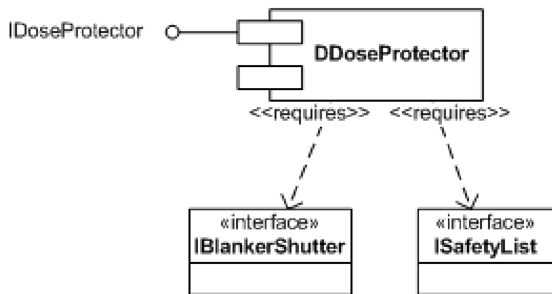

Fig. 1. Component diagram of the Dose Protector

III. ASD Models

We illustrate the ASD approach by the case study described in the previous section. Interface models are described in Section III-A and design models in III-B. Formal verification is addressed in Section III-C. Because of space limitations we only describe the ASD features used in the current case study and refer to [12] and the User Manuals available on [13] for additional explanation and more details on other aspects such as asynchronous callbacks and the use of variables.

A. Interface models of ASD

In ASD, an interface is not just a set of function calls, but it also includes a state machine which specifies the allowed traces of calls and responses. It represents a protocol between client and server, similar to the use of protocol state machines of UML [14].

The interface model of the dose protector is shown in Figure 2. It represents a state machine with three states: Created, Off, and On. The model contains two groups of function calls: sub interface Lifecycle with calls Initialize and Terminate, and sub interface Test with calls Activate and Deactivate. The ”+” behind the name indicates that they are valued, i.e., they return a value upon completion.

	Interface	Event	Guard	Actions	riable I	Target State
1	**Created <>**					
3	Lifecycle	Initialize+		Lifecycle.OK		Off
4	Lifecycle	Terminate+		Lifecycle.NotAllowed		Created
5	Test	Activate+		Test.NotAllowed		Created
6	Test	Deactivate+		Test.NotAllowed		Created
7	**Off <Lifecycle.Initialize+>**					
9	Lifecycle	Initialize+		Lifecycle.NotAllowed		Off
10	Lifecycle	Terminate+		Lifecycle.OK		Created
11	Test	Activate+		Test.OK		On
12	Test	Deactivate+		Test.NotAllowed		Off
13	**On <Lifecycle.Initialize+, Test.Activate+>**					
15	Lifecycle	Initialize+		Lifecycle.NotAllowed		On
16	Lifecycle	Terminate+		Lifecycle.OK		Created
17	Lifecycle	Terminate+		Lifecycle.Failed		On
18	Test	Activate+		Test.NotAllowed		On
19	Test	Deactivate+		Test.OK		Off
20	Test	Deactivate+		Test.Failed		On

Fig. 2. Interface model IDoseProtector

The model must be complete, in the sense that in each state the response to all calls has to be defined. In this model the actions define the return value and the next state is defined.

Observe that the state machine is non-deterministic; in state On there are two possible return values for the calls Terminate and Deactivate. Finally note that rules 2, 8 and 19 are not shown; they are used for invariants which are not discussed in this paper.

Similarly, the used interfaces are defined. Figure 3 shows the interface of the BlankerShutter component. Observe that on lines 9 and 22 the response ”Illegal” is used which means

that the client should not call the corresponding function in the current state.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**Initial <>**					
3	IBS	SetForcedBlankOn+		IBS.Ok		ForcedBlankOn
4	IBS	SetForcedBlankOn+		IBS.Failed		Error
5	IBS	SetForcedBlankOff+		IBS.Ok		ForcedBlankOff
6	IBS	SetForcedBlankOff+		IBS.Failed		Error
7	**ForcedBlankOn <IBS.SetForcedBlankOn+>**					
9	IBS	SetForcedBlankOn+		Illegal		-
10	IBS	SetForcedBlankOff+		IBS.Ok		ForcedBlankOff
11	IBS	SetForcedBlankOff+		IBS.Failed		Error
12	**Error <IBS.SetForcedBlankOn+>**					
14	IBS	SetForcedBlankOn+		IBS.Ok		ForcedBlankOn
15	IBS	SetForcedBlankOn+		IBS.Failed		Error
16	IBS	SetForcedBlankOff+		IBS.Ok		ForcedBlankOff
17	IBS	SetForcedBlankOff+		IBS.Failed		Error
18	**ForcedBlankOff <IBS.SetForcedBlankOff+>**					
20	IBS	SetForcedBlankOn+		IBS.Ok		ForcedBlankOn
21	IBS	SetForcedBlankOn+		IBS.Failed		Error
22	IBS	SetForcedBlankOff+		Illegal		-

Fig. 3. Interface model IBlankerShutter

B. Design models of ASD

Design models of ASD are similar to interface models with a few differences. Design models must be deterministic and they have a number of associated interface models: an implemented interface model and a number of used interface models. Figure 1 shows that the design model of the dose protector should implement the IDoseProtector interface model of Figure 2 and uses the IBlankerShutter model of Figure 3 and the interface model of the SafetyList component (not shown here).

Figure 4 shows a part of the design model of the dose protector. Partly, it equals the interface model of Figure 2. A difference can be observed on line 18 of Figure 4 where function CheckChanges of a sub state machine called HandleChanges is called. This sub state machine, depicted in Figure 5, is accessible by means of the "viaHandleChanges" interface. After calling CheckChanges, the top-level state machine enters state Activating. In this state the response to all possible exit events of the sub state machine is defined.

The HandleChanges sub state machine calls the used components. For instance, if it receives the CheckChanges event (line 11 of Figure 5) it calls CheckDose of the SafetyList component and waits for the return value in the CheckDose state. Based on this return value and the value of the "IsBlanked" variable, the BlankerShutter component might be used. E.g., line 45 performs a call to block the beam if exposure is unsafe and blanking is not already on.

The ASD:Suite allows code generation from design models for a number of programming languages (C, C++, C#, Java), which is the key feature for industrial users.

C. ASD Verification

The ASD:Suite contains a fixed number of verification conditions that can be checked automatically. Figure 6 shows a screenshot of the successful verification of the dose protector.

	Interface	Event	Guard	Actions	able	Target State
1	**Created <>**					
3	Lifecycle	Initialize+		Lifecycle.OK		Off
4	Lifecycle	Terminate+		Lifecycle.NotAllowed		Created
5	Test	Activate+		Test.NotAllowed		Created
6	Test	Deactivate+		Test.NotAllowed		Created
14	**Off <Lifecycle.Initialize+>**					
16	Lifecycle	Initialize+		Lifecycle.NotAllowed		Off
17	Lifecycle	Terminate+		Lifecycle.OK		Created
18	Test	Activate+		viaHandleChanges.CheckChanges+		Activating
19	Test	Deactivate+		Test.NotAllowed		Off
27	**Activating <Lifecycle.Initialize+, Test.Activate+>**					
33	viaHandleChanges	UnBlanked		Test.OK		SafeExposure
34	viaHandleChanges	Error		Test.OK		Error
35	viaHandleChanges	Blanked		Test.OK		ForcedBlankOn
40	**SafeExposure <Lifecycle.Initialize+, Test.Activate+, viaHandleChanges.UnBlanked>**					
42	Lifecycle	Initialize+		Lifecycle.NotAllowed		SafeExposure
43	Lifecycle	Terminate+		Lifecycle.OK		Created
44	Test	Activate+		Test.NotAllowed		SafeExposure
45	Test	Deactivate+		Test.OK		Off
53	**Error <Lifecycle.Initialize+, Test.Activate+, viaHandleChanges.Error>**					
55	Lifecycle	Initialize+		Lifecycle.NotAllowed		Error
56	Lifecycle	Terminate+		viaHandleChanges.UnBlank+		Terminating
57	Test	Activate+		Test.NotAllowed		Error
58	Test	Deactivate+		viaHandleChanges.UnBlank+		Deactivating

Fig. 4. Part of Design Model DDoseProtector

	erfa	Event	Guard	Actions	te Variable Upda	Target State
1	**Initial <>**					
7	viaH	Blank+	IsBlanked	viaHandleChanges.Blanked		Initial
8	viaH	Blank+	otherwise	BlankerShutter:IBS.SetForcedBlankOn+	IsBlanked=true	CheckBlanker
9	viaH	UnBlank+	IsBlanked	BlankerShutter:IBS.SetForcedBlankOff+	IsBlanked=false	CheckBlanker
10	viaH	UnBlank+	otherwise	viaHandleChanges.UnBlanked		Initial
11	viaH	CheckChanges+		SafetyList:ISL.CheckDose(>>SafetyResult)+		CheckDose
16	**CheckBlanker <viaHandleChanges.Blank+>**					
25	Blanl	Ok	IsBlanked	viaHandleChanges.Blanked		Initial
26	Blanl	Ok	otherwise	viaHandleChanges.UnBlanked		Initial
27	Blanl	Failed	IsBlanked	viaHandleChanges.Error	IsBlanked=false	Initial
28	Blanl	Failed	otherwise	viaHandleChanges.Error	IsBlanked=true	Initial
31	**CheckDose <viaHandleChanges.CheckChanges+>**					
42	Safe	SafeExposure	IsBlanked	BlankerShutter:IBS.SetForcedBlankOff+	IsBlanked=false	CheckBlanker
43	Safe	SafeExposure	otherwise	viaHandleChanges.UnBlanked		Initial
44	Safe	UnsafeExposure	IsBlanked	viaHandleChanges.Blanked		Initial
45	Safe	UnsafeExposure	otherwise	BlankerShutter:IBS.SetForcedBlankOn+	IsBlanked=true	CheckBlanker

Fig. 5. Sub state machine Handle Changes

The pre-defined conditions include checks on the absence of livelock and deadlock in all interface models and design models. Most important is the Interface Compliance check on the design model which verifies that the design model conforms to the interface model. Conformance has been defined formally in the failures-divergence model of CSP [15] and is checked with the underlying FDR 2 model checker [6]. This check also verifies that no illegal calls are performed on the used interfaces.

When a property is not satisfied, ASD:Suite shows an error trace in the form of a nice graphical representation of a sequence diagram which illustrates the problem. This is convenient for fast debugging and error correction.

IV. Translating ASD Models to Uppaal

To extend the verification described in Section III-C, we translate the ASD models to Uppaal. The translation of ASD used interface models and design models is described in Sections IV-A and IV-B, respectively. We concentrate on the ASD constructs needed for our case study and show the application to this case study. In Section IV-C we explain how a closed system is obtained which is needed for Uppaal

DoseProtector (completed)			
[C:\Doc...odelsVersion4\DoseProtector.dm]			
Start time: 19 Aug @ 23:07	End time: 19 Aug @ 23:07		
DoseProtector			
Modelling Error check	Passed	15	< 1m
Livelock check	Passed	15	< 1m
Deadlock check	Passed	15	< 1m
BlankerShutter			
Modelling Error check	Passed	12	< 1m
Livelock check	Passed	12	< 1m
SafetyList			
Modelling Error check	Passed	4	< 1m
Livelock check	Passed	4	< 1m
DoseProtector			
Deterministic check	Passed	27	< 1m
Modelling Error check	Passed	53	< 1m
Deadlock check	Passed	53	< 1m
Interface Compliance check	Passed	55	< 1m
Livelock check	Passed	55	< 1m

Fig. 6. ASD Verification of the Dose Protector

verification.

Currently, the translation has been done by hand, but in view of possible automation later, the aim is to follow a simple standard procedure as much as possible. Hence we translate an ASD model line-by-line. Figure 7 shows a template of a line in an ASD model.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**State1** <>					
3	I	E	G	I.R	U	State2

Fig. 7. Template of line in ASD model

A function call "E" is translated to a communication over synchronous channel E in Uppaal. The caller will perform a send command, written E! in Uppaal, and the callee a receive command E?. Communication only takes place if both sender and receiver are ready to execute the matching commands. Similarly, reply "R" is also represented by a synchronous communication in Uppaal.

Calls where the Action field contains "Illegal" are not translated. In this way, any illegal call will lead to a deadlock which will be detected by the Uppaal verifier.

ASD states are translated to Uppaal locations. The translation of guards and actions is straightforward. For brevity, sub interface "I" is only used to avoid confusion between events with the same name.

A. Translation of used interface models

For a used interface, the ASD line of Figure 7 is translated into the Uppaal fragment of Figure 8. Since two synchronous communications (for the function call and the reply) are not allowed on a single transition in Uppaal, two transitions are used with a so-called "committed" location in between to ensure that the two transitions are executed as an atomic step.

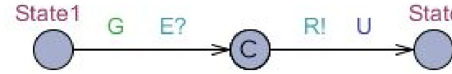

Fig. 8. Uppaal translation of ASD rule for used interface

Applying this translation to the ASD model of the BlankerShutter interface of Figure 3 we obtain the Uppaal model of Figure 9.

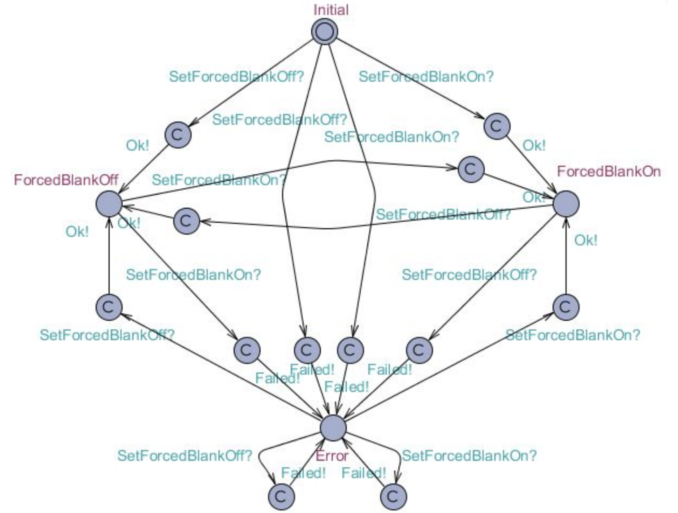

Fig. 9. Uppaal model of BlankerShutter interface

B. Translation of design models

A similar translation is applied to ASD design models. In such models the Event "E" might additionally be a return value that is received from a used component. Moreover, the Action part might contain calls to used components or sub state machines. All these events and actions are translated to a communication over a synchronous channel. As an illustrative case, we show in Figure 10 a part of the Uppaal model resulting from the translation of the design model for the DoseProtector of Figure 4. Note that in location "Off" the receipt of an Activate message leads to a CheckChanges message which triggers sub state machine HandleChanges.

Sub state machine HandleChanges (Figure 5) is modelled as a separate parallel automaton in Uppaal. Part of this Uppaal model is shown in Figure 11. Observe that after receiving trigger CheckChanges in location Initial from the automaton of Figure 10, it calls CheckDoseCommand of the SafetyList (this automaton is shown later in Figure 13) and depending on the received return value and boolean "isBlanked" it might call SetForcedBlankOff of the BlankerShutter.

C. Adding client model to obtain closed system

Simulation and verification in Uppaal requires a closed set of models which includes the environment of the component(s)

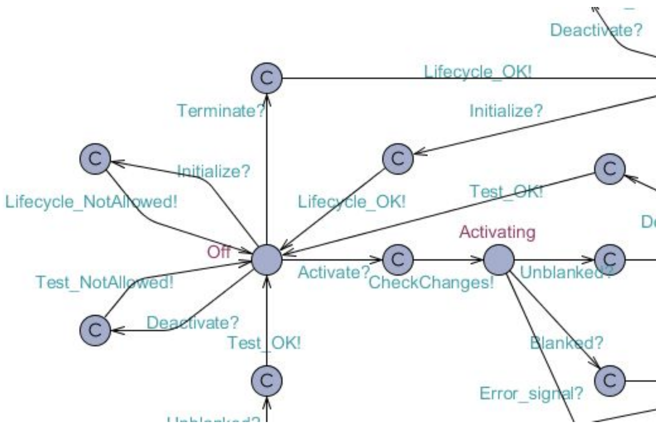

Fig. 10. Part of Dose Protector in Uppaal

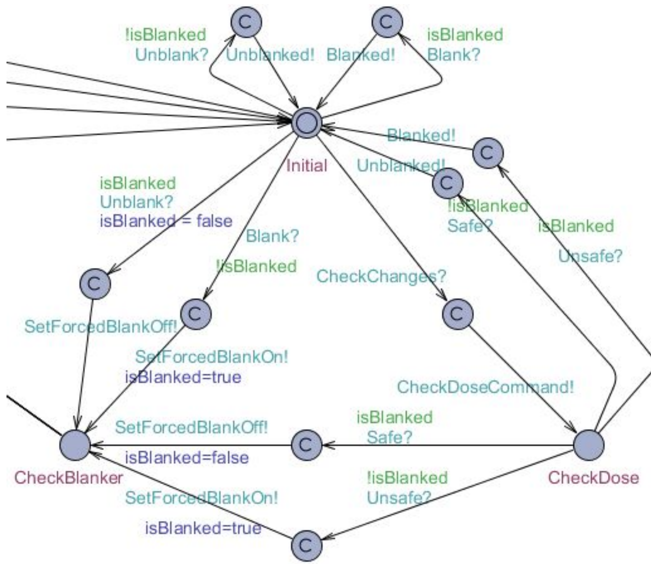

Fig. 11. Part of HandleChanges in Uppaal

under study. In our case, we can easily obtain a closed system using the implemented interface of an ASD design model. We translate this model into an Uppaal model of a client of the component. This is similar to the translation of used interface models, however the direction of sending and receiving is mirrored. Hence, the template of Figure 7 is translated to Figure 12.

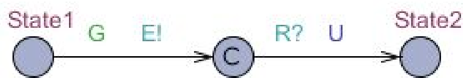

Fig. 12. Uppaal translation of implemented interface

V. Verification of the Case Study

The result of the translation described in the previous section is a complete Uppaal model that can be simulated and verified. Since the code is generated from ASD models, the Uppaal model is a truthful representation of the software.

The possibilities to simulate the Uppaal models are greatly appreciated in our industrial context and provide additional insight in the collaboration of the ASD models, since the ASD:Suite does not provide simulation.

Uppaal allows the verification of user-defined properties which have to be expressed in a version of temporal logic. Both safety properties (nothing bad happens) and liveness properties (eventually something good happens) can be expressed. In general, we concentrate on properties that have not yet been verified by the standard checks of ASD:Suite. In this paper, we illustrate the approach by means of a few safety properties.

The properties to be verified have been defined in cooperation with the software architect, supervising the creators of the system design, and the system architect. From these verification efforts two major issues were found. These issues could not be found in the ASD approach since it does not allow this type of verification. We show two verification expressions that illustrate how one of these issues was found.

A. Example verification using temporal logic

As a first step we extended the Uppaal automaton of the SafetyList component with a variable ”safe” which records the result of the safety computations, as depicted in Figure 13.

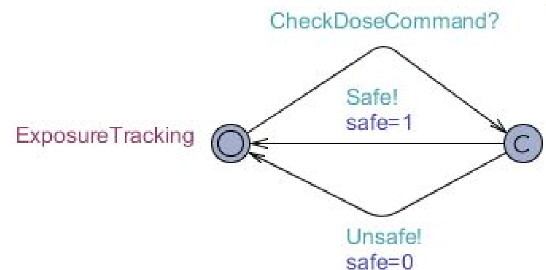

Fig. 13. Extended Uppaal model of SafetyList

First we expressed the property “when the dose protector design is in a safe state (state SafeExposure) this should always imply that the beam intensity is safe” as follows:

A[] DoseProtector.SafeExposure imply (safe==1)

The Uppaal verifier shows that this property holds.

Next we checked the property “when the dose protector interface is in a safe state this should always imply that the intensity is safe and the BlankerShutter should not be in an error state”. This is expressed as:

A[] DoseProtector.SafeExposure imply
(safe==1 and !BlankerShutter.Error)

This property does not hold, as shown by a trace of 20 steps. This failure was not expected by the designers. We used the diagnostic trace and the simulation capability of Uppaal to get insight into the detected problem, which led to the conclusion that this issue can be solved straightforwardly.

B. Verification by means of observers

Disadvantage of above approach is that the temporal logic expressions are not easy to read and construct by industrial users. Moreover, it is annoying that some information had to be added to record the occurrence of certain transitions.

As an alternative, we experimented with the use of observers, similar to [16]. An observer is an additional parallel automaton which observes the communication between the other automata and enters a location named Error when an incorrect trace is observed. Figure 14 shows an example.

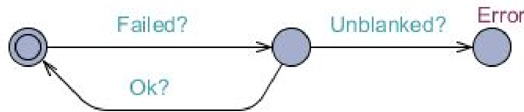

Fig. 14. Observer to detect bad traces

To be able to observe the relevant channels, they are declared as broadcast channels in Uppaal. Figure 14 expresses that after a Failed command of the BlankerShutter there should not be an Unblanked event. The verification reduces now to the verification of the very simple expression:

A[] not Observer.Error

Comparing both methods of verification, we agree with [16] that the observer method is easier to understand and very convenient in an industrial setting.

VI. Concluding Remarks

We are convinced by the positive results of our pilot project that the combination of the proposed complementary tools provides a promising and valuable new direction for industrial software development.

On beforehand, we expected that a large effort would be needed to obtain some industrially relevant results, but we were able to select a small, essential set of components fairly quickly. Next, it took a non-expert in ASD and Uppaal only a few days to get the first significant results.

Since our first concern was to get quick feedback about the usefulness of the combined techniques, the translation from ASD models to Uppaal has been constructed manually. Given the promising results, future work will include automated support for such a translation. When needed, also aspects of ASD that have not been covered yet (such as asynchronous callbacks) will be included. Moreover, it will be investigated whether the timing aspects of Uppaal provide additional value in our context. An important aspect of future applications is scalability; when applying the approach to a larger set of components, there is a danger that the Uppaal model cannot be checked anymore. To avoid this, we expect to use the compositional structure of the ASD models; main difficulty would then be to split an end-to-end property in a sequence of properties for the individual components.

The acceptance in industry of this combination of tools depends strongly on the way it is integrated, the consequences for the software development process, and the interest by the engineers. We think substantial support should be given in the form of automated translation between ASD and Uppaal, systematic methods to generate verification expressions or (preferably) observers, and expert support during the introduction phase. It is our expectation that designing and reasoning at a higher abstraction level than code, will help to improve confidence and speed in creating software systems.

Acknowledgment

This work has been carried out as a part of the Condor project at FEI Company under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program. The second author was supported by the Dutch national program COMMIT.

References

[1] G. Broadfoot and P. Broadfoot, “Academia and industry meet: Some experiences of formal methods in practice,” in *APSEC ’03: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*. IEEE Computer Society, 2003, pp. 49–58.

[2] P. Hopcroft and G. Broadfoot, “Combining the box structure development method and CSP for software development,” *Electronic Notes in Theoretical Computer Science*, vol. 128, no. 6, pp. 127–144, 2005.

[3] H. Mills, M. Dyer, and R. Linger, “Cleanroom software engineering,” *IEEE Software*, vol. 4, no. 5, pp. 19–25, 1987.

[4] S. Prowell, C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering - Technology and Process*. Addison-Wesley, 1998.

[5] C. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[6] *FDR2 model checker*, Formal Systems (Europe) Ltd, 2011, http://www.fsel.com/.

[7] J. Hooman, *Specification and Compositional Verification of Real-Time Systems*, ser. Lecture Notes in Computer Science. Springer, 1991, vol. 558.

[8] *Uppaal*, UP4ALL, Uppsala, 2011, http://www.uppaal.com/.

[9] G. Behrmann, A. David, and K. Larsen, “A tutorial on Uppaal,” in *Formal Methods for the Design of Real-Time Systems*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 3185, pp. 33–35.

[10] J. S. Xing, B. D. Theelen, R. Langerak, J. C. van de Pol, G. J. Tretmans, and J. P. M. Voeten, “From POOSL to UPPAAL: Transformation and quantitative analysis,” in *Application of Concurrency to System Design, Tenth International Conference on*. IEEE Computer Society, 2010, pp. 47–56.

[11] J. Hoenicke and E.-R. Olderog, “Combining specification techniques for processes, data and time,” in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2335, pp. 245–266.

[12] J. Hooman, R. Huis in ’t Veld, and M. Schuts, “Experiences with a compositional model checker in the healthcare domain,” in *Foundations of Health Information Engineering and Systems (FHIES 2011)*, ser. Lecture Notes in Computer Science. Springer, 2012, to be published.

[13] *ASD:Suite*, Verum, 2011, http://www.verum.com/.

[14] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language user guide - the ultimate tutorial to the UML from the original designers*, ser. Addison-Wesley object technology series. Addison-Wesley-Longman, 1999.

[15] A. Roscoe, *The theory and practice of concurrency*. Prentice Hall, 1998.

[16] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, “The IF toolset,” in *Formal Methods for the Design of Real-Time Systems*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 3185, pp. 131–132.