A Transformation of SDL Specifications — a Step Towards the Verification (extended abstract)

Natalia Ioustinova[1] and Natalia Sidorova[2]

[1] Dept. of Computer Science, Institut of Technische Informatik, University of Rostock, Alb. Einstein Str. 21, D-18059 Rostock, Germany
fax +49(0)381 498 3440
ustin@informatik.uni-rostock.de

[2] Dept. of Math. and Computer Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands
fax +31(0)40 246 3992
n.sidorova@tue.nl

Abstract. Industrial-size specifications/models (whose state space is often infinite) can not be model checked in a direct way — a verification model of a system is model checked instead. Program transformation is a way to build a finite-state verification model that can be submitted to a model checker. Abstraction is another technique that can be used for the same purpose. This paper presents a transformation of SDL timers allowing to reduce the infinite domain of timer values to a finite one with preserving the behaviour of the system. A timer abstraction is proposed to further reduce the state space. We discuss the ideas behind these transformations and argue their correctness.

1 Introduction

Model checking [3] is one of the most popular and successful verification techniques accepted both in academia and in industry. One of the main reason of its success is its promise to automatically check a program against a logical specification, typically a formula of some temporal logic. A stumbling-block limiting the model-checking application area is the notorious state-space explosion. The major factors influencing the state space are, clearly, the size and the complexity of a specification. In many cases, abstractions and compositional techniques make it possible to cope with the state-space explosion and apply model checking to real-life industrial systems. However, besides size and complexity, there exists another factor cumbering verification.

Development of a specification language, its semantical concept are greatly affected by the intended mode of its use, its application domain. Often, the final objective is to get an executable specification/implementation. In that case, the specification language and its semantics should provide a framework for constructing faithful descriptions of systems. No wonder that specifications written in these implementation-oriented languages are harder to verify than the ones written in the languages developed as input languages for model checkers. In this paper, we concentrate on some aspects of modelling time in the implementation-oriented languages, taking SDL (Specification and Description Language) [10] as an instance of this class of languages.

SDL is a popular language for the specification of telecommunication software as well as aircraft and train control, medical and packaging systems. Timing aspects are very important for these kinds of systems. Therefore, behaviour of a system specified in SDL is scheduled with the help of timers involved into the specification. The model of SDL timers was induced by manners of implementation of timers in real systems. An SDL timer can be activated by setting it to a value (`NOW` + δ) where expression `NOW` provides an access to the current system time and δ is a delay after which this timer expires, i.e., the timer expires when a system time (system clock) reaches point (`NOW` + δ). Such an implementation of timers immediately means that the state space of SDL-specifications is infinite just due to the fact that timer variables take an infinite number of growing, during the system run, values. An inverse timer model is normally employed in the verification-oriented languages: a timer indicates a delay left until its expiration, i.e., a timer is set to value δ instead of (`NOW` + δ), and this value is decreased at every tick of the system clock. When the timer value reaches zero, the timer expires. This model of timers guarantees that every timer variable takes only a finite (and relatively small) number of values.

Another SDL peculiarity that adds to the complexity of verification is the manner the timers expire in SDL. SDL is based on the Communicating Extended State Machines; communication is organized via the message passing. For the uniformity of communication, timers are considered as a special kind of signals and a process learns about a timer expiration by dint of a signal with the name of the expired timer, inserted in the input port of the process. From the verification point of view it would be better if a timer expiration had been diagnosed by a simple check of the timer value.

Though formal verification of SDL-specifications is an area of rather active investigations [2, 7, 5, 9, 13], the time-concerned difficulties were being got round for a long time by means of abstracting out time and timers. Due to engineering rather than formal approaches to constructing abstractions, some of proposed abstractions turned out to be not safe (cf. [2]). In [2] a toolset and a methodology for the verification of time-dependent properties of SDL-specifications are described. The SDL-specifications are translated into DT Promela, the input language of the DT Spin (Discrete Time Spin) model checker [1], and then verified against LTL formulas. Some arguments are given in favour of a behavioural equivalence of the DT Promela translation to the original specification.

Here, we propose a transformation of SDL specification into SDL itself, where the new `timer` type is substituted for the traditional SDL `timer` type. The underlying idea is similar to the one in [2], but providing SDL to SDL transformation, we make the transformation principles independent of a particular model checker, and the formal proof of model equivalence substantiate that the transformed model can be safely used for the verification.

Admitting that in a number of cases timer abstractions are useful[1], we believe that the known safe abstraction (cf. [2]) of SDL timers does not yield a desirable result in a

[1] If a property is expected to hold independently of the settings of a timer, for example.

number of cases because it abstracts not just timers but time. Here, we propose a more flexible w.r.t. the refinement degree abstraction, for which the abstraction of [2] is a particular case.

The paper is organised as follows. In Section 2 we shortly survey the SDL time semantics. In Section 3 we present a behavioure-preserving transformation for SDL-specifications. In Section 4 we propose a timer abstraction. We conclude in Section 5 by evaluating the results.

2 SDL Time Semantics

SDL is a general purpose description language for communicating systems. The basis for the description of a system behaviour is Communicating Extended State Machines represented by processes. A process consists of a number of states and a number of transitions connecting the states. The input port of a process is an unbounded FIFO-queue. Communication is based on the signal exchange between the system and its environment and between processes within the system.

Two data types, `Time` and `Duration`, are used to specify time values. A variable of the `Time` type indicates some point of time. A variable of the `Duration` type represents a time interval. A process can access a current system time by means of the `NOW` operator. The concept of timers is employed to specify timing conditions imposed on a system. A timer is related to a process instance; it is either active (set to a value) or inactive (reset). Two operations are defined on the timers: `SET` and `RESET`. A timer is activated by setting it to a value (`NOW` + δ) where δ is in fact a delay after which this timer expires.

With each timer there are associated a pseudo-signal and an implicit transition, called a time-out transition. When a timer expires, its time-out transition becomes enabled and may be executed. The execution of the time-out transition adds the corresponding pseudo-signal to the process queue. The time-out transitions of timers expiring simultaneously can be executed in any order. A time-out signal is handled like an ordinary signal. The timer is active until a pseudo-signal is consumed from the queue. If a `SET` or `RESET` operation is performed on an expired timer while its time-out transition is still enabled, the time-out transition becomes disabled. If the timer is set or reset after adding its associated pseudo-signal to the process queue (before the signal is consumed from the queue) the pseudo-signal is removed from the queue.

Though there is no standardized time semantics in SDL, there exist two semantics accepted in the SDL-community [5]. According to one of them (the one, which is supported by the commercial SDL-design tools [14, 12] and the one we work with), the transitions of SDL processes are instantaneous (take zero time); time can only progress if at least one timer is active and the system is blocked: all the processes are waiting for further input signals (i.e., all input queues are empty, except for saved signals, and there is no `NONE` input enabled). Time progression amounts to performing a specific transition that makes time increment until an active timer expires. Later on, we refer to a segment of time separated by the time increment transitions as a *time slice*. (Note that time progression is discretised.) When the system time gets equal to the timer value, the time-out transition becomes enabled and it can be executed at any point of the time slice. The time slice always starts with a firing of one of the enabled time-out transitions. This action unblocks the system. In case several time-out transitions become enabled at the same time, one of them is taken (non-deterministically) to unblock the system and the rest are taken later at any point of the time slice since they have the same priority as normal transitions.

Though complicated, such a time semantics is suitable for implementation purposes [10]. It is natural to model a timer as a unit advancing from the current time derived by evaluation of `NOW` expression to the the time point specified by the expression `NOW` + δ, i.e., waiting for a point of the system time.

3 Timer Transformation

An SDL process (its state) is described by its current control state, the states of the timers belonging to the process, the values of the process variables and the content of the input queue. Since `NOW` gives an access to the current system time, each evaluation of operation `SET`(T, `NOW` + δ) on timer T gives a new state of the process. Moreover, a time-out transition can add a timer signal at any point of the time slice. This blow up the state space due to the number of possible interleaving sequences of events. Keeping a time-out signal in a process queue also adds to the length of the state vector.

To avoid the state-space explosion due to the interpretation of timers and the overhead caused by the management of time-out pseudo-signals, we substitute SDL concept of timers as a special kind of signals by a concept of timers as a special data type. Timer variable T represents a declared in the original specification timer T. The value of this variable shows the delay left until the timer expiration. Since delays are non-negative, we use -1 to represent inactive timers. Therefore, the `RESET` operation is transformed into the assignment of the -1 value to a timer variable, and the `SET`(`NOW`$+x, T$) operation is transformed into the assignment of $\max\{0, x\}$ to it (Fig. 1).

Fig. 1. Transforming `SET` (on the left) and `RESET` (on the right)

A timer whose value in the original system is equal to the current system time, can expire. The transformed system should demonstrate the same behaviour. Since we suppose that the value of the transformed timer is a delay left until its expiration, only the timers whose values are equal to 0 may expire. Therefore, we replace inputs of timer messages from the process queues by the enabling condition consisting of the `NONE` input guarded by the $T = 0$ condition (Fig. 2).

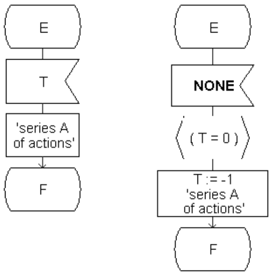

Fig. 2. Transforming the input of a time-out signal

Such a substitution does not give a straightforward reflection of the original system behaviour in the behaviour of the transformed system since the sending of the time-out signal to the process queue in the original system can be separated from its consumption from the queue by other actions. In the transformed system, we get the behaviour where sendings of time-out signals are projected out and the consumptions of these signals are mimicked by the consumptions of the correspondent `NONE` signals, whose enabling conditions are guaranteed to be true in this case. The projection is not harmful from the verification point of view because not the presence or absence of the time-out signal but the consumption of it and the resulted actions are important for the verification. The same concerns the process queue: saying that its content in the transformed system is the same as in the original one, we mean that the original queue from which the time-out signals are projected out, coincides with the process queue in the transformed system.

Now, it is easy too see that the sequence of actions in the original system where a time-out signal is sent to the process queue due to a firing of the time-out transition and then removed from it because the correspondent timer is reset before the time-out signal is consumed from the queue, is fairly modelled in the transformed system. Neither sending nor removing the signal are not presented there being projected out, and the enabling condition may not be taken since the timer is reset, hence the condition is not fulfilled.

System time is not present in the transformed system — one infinitely grown variable is enough to cause the state-space explosion. The time-increment transition is mimicked by the transition decreasing the values of all active timers by the value of the minimal of them. Like the time-increment transition, this transition can take place only if the system is blocked, and "blocked" has exactly the same meaning as for the original system.

The equivalence of two models can be shown by using the simulation technique. Introducing the simulation relation that consider projected out timeout transitions and invisible ones, one can prove that the systems simulate each other in a step-wise manner by induction.

Corollary 1. *The transformation of SDL specifications according to the schema above preserves the system behaviour.*

4 Timer abstraction

Abstraction, intuitively, means replacing one semantical model by an abstract, in general, simpler one. In addition to the requirement that an abstract (verification) model should have a smaller state space than the concrete (implementation) one, the abstraction needs to be *safe*, which means that every property checked to be true on the abstract model, holds for the concrete one as well[2]. This allows the transfer of positive verification results from the abstract model to the concrete one.

The concept of safe abstraction is well-developed within the *Abstract Interpretation* framework [4]. The requirement that Abstract Interpretation puts on the relation between the concrete model and its safe abstraction can be formalized as a requirement on the relation between the data operations of the concrete system and their abstract counterparts, as follows. Every value of the concrete state space is mapped by the *abstraction function* α into an abstract value which, intuitively, "describes" the concrete value. The requirement of mimicking is then formally phrased as:

$$\forall x : \alpha(f_{conc}(x)) \in f_{abs}(\alpha(x))$$

In the following we call this the *safety statement*.

Besides decreasing the state space of the system and safeness, the main requirement for an abstraction is that the abstract system behaviour should correctly reflect the behaviour of the original system with respect to a verification task in the sense that an abstraction captures all essential points in the system behaviour, i.e., it is not "too abstract". The safe abstraction of timers for the Promela translations of SDL-specification from [2] does not meet this requirement well enough.

The abstraction is based on a natural idea of allowing timers to expire at an arbitrary moment after they are set ([2]). A typical problem arising when one starts to apply this abstraction in practice is introducing "zero-time cycles" which are not present in the concrete model. A usual pattern for SDL-specifications is that a timer schedules some periodical activity; after a timer-out signal is consumed by the process, some actions are initiated, then the timer is set again and the process returns to the same control state where it was before the consumption of the timer signal. Since a timer is allowed to expire at any arbitrary moment after its setting (also immediately after it has been set), an undesirable cyclic behaviour can be introduced. The "timer input–setting–expiration" chain of transitions is now permanently enabled and all the other behavioural branches may be ignored forever. Therefore, the properties which hold for the concrete model and which are expected to hold independently of the timer settings, fails to hold for the abstract model since "independently" means in fact that the property holds for a concrete model whatever *positive* delay is assigned to a timer. Another problem arises in case

[2] A safe abstract system is, intuitively, a system whose behaviour (the set of all transitions) is a superset of the concrete system behaviour.

a timer serves as guard preventing from taking a transition too early. With abstracting time, this timer guard is broken.

We propose a safe abstraction for timers that keeps this guard delaying the timer expiration. A concrete timer that can be set only to delays exceeding some k is abstracted according to the patterns given in Figure 3. The setting of the concrete timer to a (NOW+x) value is replaced with setting it to NOW+k ($x \leq k$) and the timer is allowed to expire at any point of time after the delay of k time units. This transformation only adds the behaviour while preserving all the behaviour of the concrete system, which means that the abstraction is safe. (The formal proof of safeness is performed via proving that the safety statement is not violated.) Varying k, we can change the refinement degree of the abstraction. Taking k equal to 0, we get the most abstract version of it, which is a particular case when not just timers but time is abstracted. Taking k equal to the lower boundary of the timer delays, we get the most refined abstraction defined by this transformation schema.

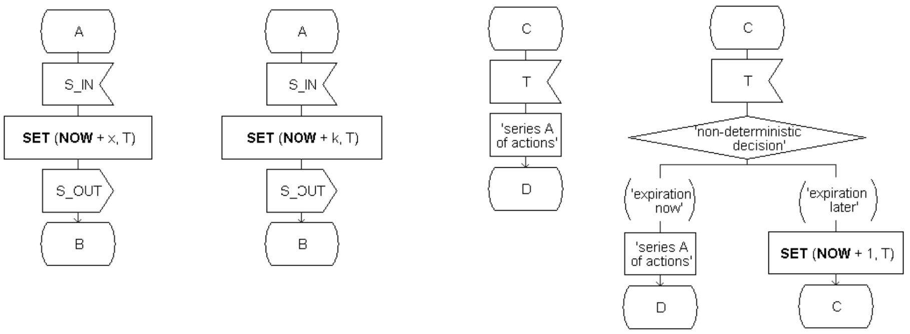

Fig. 3. Mimicking the SET operation (on the left) and the INPUT of a timer-signal (on the right)

The experiments shows that the smallest number of states is usually obtained when k is equal to 1, not 0. (We compare the number of states in the models, to which the transformation from Section 3 is applied.) The state space normally grows with increasing k, which is no wonder since the number of possible states of the timer itself grows in that case. Rather unexpected is the fact that 0 increases the state space and can even lead to the state-space explosion. The behaviour of the system with zero timer delay often has a different nature of regularity, the behaviour branches, excluded formerly by the timer guards, can be taken, which explains this phenomenon.

5 Conclusion

The proposed transformation of SDL-timers is a simple and cheap step that can be considered as a zero-phase of the translation of SDL-specifications to a model-checker input language. The transformation is aimed at reducing the state space to a finite domain. A side issue (though important on its own) is that the described transformation gives a better insight into the timer semantics. Our experience shows that the complicated time semantics of SDL can lead to errors due to its misunderstanding both in the design-phase and in the translation-phase (cf. [2, 11]). Treatment of timers as variables is simpler than treatment them as signals.

The abstraction given in Section 4 is aimed at the state space reduction. Due to its flexibility, it is applicable to a wider range of problems that the earlier used for SDL timer abstractions. Its practical application confirmed its efficiency for the real-life situations (cf. [11]).

Here, we gave only a sketchy description of the transformations of SDL-specifications and the intuitive ideas behind it. The formalization of the timer transformation and the timer abstraction will have to wait for the full paper, as well as the formal proof of the statements that the transformation does not change the specified system behaviour and that the abstraction is safe.

References

1. D. Bošnački, D. Dams, *Integrating Real Time into Spin: A Prototype Implementation*, S. Budkowski, A. Cavalli, E. Najm, editors, Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'98), Kluwer, 1998.
2. D. Bošnački, D. Dams, L. Holenderski, N. Sidorova, *Verifying SDL in Spin*, Tools and Algorithms for the Construction and Analysis of Systems TACAS 2000, LNCS 1785, pp. 363-377, Springer, 2000.
3. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
4. P. Cousot, R. Cousot, *Abstract Interpretaion: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, In the 4th POPL, Los Angeles, CA, ACM, January 1977.
5. U. Hinkel, *Verification of SDL Specifications on the Basis of Stream Semantics*, In Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC, Y. Lahav, A. Wolisz, J. Fischer, E. Holz (eds.), Humboldt-Universitaet zu Berlin.
6. G. J. Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991.
7. G.J. Holzmann, J. Patti, *Validating SDL Specification: an Experiment*, In E. Brinksma, G. Scollo, Ch.A. Vissers, editors, Protocol Specification, Testing and Verification, Enchede, The Netherlands, 6-9 June 1989, pp. 317-326, Amsterdam, North-Holland, 1990.
8. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, *Property Preserving Abstractions for the Verification of Concurrent Systems*, In Formal Methods in System Design, Kluwer Academic Publ., 6, 1-36, 1995.
9. F. Regensburger, A. Barnard, *Formal Verification of SDL Systems at the Siemens Mobile Phone Department*, In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98) 1998, LNCS 1384, pp. 439-455, Springer, 1998.
10. A. Olsen *et al.*, *System Engineering Using SDL-92*, Elsevier Science, North-Holland, 1997.
11. N. Sidorova, M. Steffen, *Verification of a Wireless ATM Medium-Access Protocol*, In Proc. of the 7th Asia-Pacific Software Engineering Conference (APSEC 2000), pp. 84-91, IEEE Computer Society, 2000.
12. Telelogic Malmö AB. *SDT 3.1 User Guide, SDT 3.1 Reference Manual*, Telelogic, 1997.
13. H. Tuominen, Embedding a Dialect of SDL in PROMELA, 6th Int. SPIN Workshop, LNCS 1680, pp. 245-260, Springer, 1999.
14. Verilog, *ObjectGEODE SDL Simulator - Reference Manual*, 1996.