Modeling Soft State Protocols with SDL

Xiaoming Fu and Dieter Hogrefe

Telematics Group, University of Goettingen
{fu, hogrefe}@cs.uni-goettingen.de

Abstract. Soft state provides new services to packet-switching networks by introducing a type of state in the network nodes which is refreshed by periodical messages and otherwise expires. The operations of soft state protocols, which are being designed with ever greater complexity, need to be error-free and deadlock-free to avoid misusing network resources. Thus, verification, formal analysis and validation of these protocols become a vital task. In this paper we utilize formal techniques, specifically, Specification and Description Language (SDL) and Message Sequence Charts (MSCs), for modeling, analysis and validation of various soft state protocols. We propose a general architecture for state management systems and find employing these techniques can help identify and correct possible design errors, which may be caused by informal specifications.

1 Introduction

In communication networks, there is a need to maintain certain information ("*state*") in network nodes, associated with endpoint-generated sessions or calls. For example, ATM switches maintain information about VCs such as bandwidth allocation and VCI/VPI input-output mapping. The state maintained by the network can be categorized as *hard state* and *soft state*. Hard state is installed in nodes upon receipt of a setup message and is removed only upon receipt of an explicit removal message. It is vital for the state initiator to know when the state has been installed or removed, and ensure that installation and removal are performed only once. Furthermore, since hard state remains installed unless explicitly removed, there needs a mechanism to remove orphan state that appears once the state initiator has crashed or departed without removing state. In contrast, "soft state" refers to certain non-permanent state in network nodes which will expire unless refreshed. Since soft state will eventually expire, in principle this approach does not require explicit removal or a mechanism for removing orphan state. Several routing protocols (e.g., BGP, OSPF and RIP) and early Internet signaling protocol (ST-II) use hard state. Once the soft state paradigm was applied to RSVP [1], which allows endpoints to establish QoS reservation state in the network nodes along the path for their end-to-end communications, it has been adopted by many other protocols, such as RTCP, PIM, SIP and CASP [2].

By the use of state – either hard state or soft state – inside the network, protocols can provide certain enhanced services for end-to-end communications, such as QoS reservation setup or flow-coupled firewall configurations. Note both hard state and soft state can be installed either in intermediate nodes or end hosts only, or both. Due to the nature of state, unlike other types of protocols, state management protocols often require extremely complex and powerful mechanisms to ensure that the state is perfectly synchronized and up-to-date. With the informal IETF specifications, operations of these protocols tend to be error-prone. For example, a report [3] showed numerous problems or unexpected behaviors in the specification and implementation of TCP, the dominating end-to-end transport protocol which uses hard state in end hosts. With an ever-increasing number of soft state protocols and the increase in their complexity, unfortunately, the risk of design and implementation errors for soft state protocols increases. An example is the "auto-refresh" loop in RSVP, possibly keeping a state alive forever [1]. In general, it is vital to ensure the correctness of state management operations in protocol specifications.

The methods for studying protocol operations and correcting possible flaws can be classified into two basic groups. The natural, empirical ("trial-and-error") approach, is to study them with an actual implementation (or a prototype). Another possibility involves a model-based approach in which protocol behaviors may be studied using a model of the protocol. The empirical approach is only effective for examining standard, ordinary behaviors of a protocol, whereas model-based approaches, based on simulation or analytical models, can be more effective in determining possible errors in the design. Applying the latter approach to state management systems may reduce excessive problems (and costs!) in standards and implementations, unlike the first method of debugging and correcting these specifications. From our experience [4], real soft state protocols can be rather complex and difficult to be analyzed through implementations. This is partly due to the fact that soft state protocols must independently maintain several types of timers and soft states associated with a given end-to-end session in a distributed fashion. Moreover, they are generally specified informally and imprecisely. Therefore, model-based approaches are preferred.

In this paper we employ a model-based approach to study basic functionalities and liveness properties of general soft state systems, using the Specification and Description Language (SDL) [5] and the Message Sequence Charts (MSCs) [6]. As successfully demonstrated in other protocol modeling experiences (e.g., [7]), SDL and MSCs are efficient tools for such tedious tasks. Based on [8], we present a general architecture for state management systems and model such a system with SDL/MSC. Demonstrating the feasibility of applying this approach in modeling soft state protocols and analyzing their basic properties, we advocate a potential way of improving protocol descriptions. We focus on the functionality modeling and validation following [8], including verification of (the absence of) deadlocks and livelocks. Our goal here is not to model a real soft state protocol such as RSVP or CASP, or hard state protocols like ST-II or TCP, but rather to model general soft state protocols in order to capture and verify the essential concepts and general functionalities of interest. There are other, non-functional aspects of these protocols, such as complexity and performance, but they are beyond the scope of this paper.

The rest of this section discusses related works. Given the importance and difficulty in analyzing soft state protocols, Section 2 presents a general architecture covering all variants of soft state protocols, followed by SDL models for representative soft state management (Section 3) and a verification of the models (Section 4). Section 5 summarizes our experiences and outlines future work.

Previous Studies on Soft State Protocols: System designers argue soft state is "better" than hard state, and using soft state the handling of network condition changes is "easy" [9, 10]. However, these claims are more based on intuitive, high-level thoughts and explanations, instead of formal, exhaustive modeling and analysis. In contrast to original expectations, soft state protocols developed and being developed so far are still far from being simple, especially when coupled with multicast sessions or traffic control models.

There are two types of soft state protocols which have been developed so far: end-to-end protocols and hop-by-hop protocols. The former only involves certain types of state in an end-to-end way, without involving any other nodes in between; examples of this type include RTCP and SIP. Hop-by-hop protocols, such as RSVP and CASP, on the other hand, involve state in one or more router(s) in between in addition to state in the communicating ends. For the purpose of demonstration and general discussions of soft state operations, we have chosen to use the latter since it is more representative and comprehensive.

Given the particular importance of soft state protocols, recent studies have looked at issues regarding their modeling and analysis. Raman and McCanne [9] presented a model for the soft state notion based on Jackson queueing networks; a performance study of hard state and soft state signaling protocols was performed by Ji *et al.* [8]. Unfortunately, these studies lack more detailed formal modeling and validation. Bradley *et al.* [11] studied the correctness and interoperability issues with the HTTP protocol, and established that multi-stage interactions between the HTTP server and clients can be stateful and error-prone; they then verified these behaviors by using a formal checking tool SPIN [12]. However, their study is limited to application-layer hard state management between two endpoints and does not consider the soft state paradigm for packet-switching networks (which may involve multi-hop behaviors of state management systems).

A Brief Introduction to SDL and MSC: An SDL system is divided into building blocks that communicate using channels, whereas blocks are further composed of processes. Processes (within a block) are connected using signal routes. Each process is an extended finite state machine, which has its own infinite queue and is assumed to operate independently from all other processes. MSCs are another valuable description technique for visualizing and specifying inter-system, asynchronous component interaction. MSCs' strength lies in their ability to describe communication between cooperating processes. Each process is represented as an identifier and has a process life line that extends downward.

There are arrows representing messages passed from a sending to a receiving process. Messages not starting or ending at a process life line are exchanged with users, be they human or mechanical (the "*environment*"). Detailed descriptions of SDL and MSC can be found in [5, 6]. Modern SDL development tools like Telelogic Tau also support verification and validation based on developed models.

2 A General Architecture of State Management Systems

In contrast to hard state (HS) protocols, soft state (SS) protocols can be further classified into 4 variants: "pure" soft state (pSS), soft state with explicit removal (SS+ER), soft state with reliable trigger (SS+RT) and soft state with reliable trigger/removal (SS+RTR) [8]. In this section, we present a simple abstraction and typical operations for all these state management protocols (both soft state and hard state), as well as possible problems that may occur in their operations.

We begin our modeling with a generic architecture for state management systems as shown in Fig. 1, which covers two services and one protocol as below:

– The state message transport (ST) service, which transmits state messages over the lossy channel.
– The state management service (SMP service), which is the service rendered by the state management protocol to the state application (SA).
– The state management protocol (SMP) managing state in network nodes.

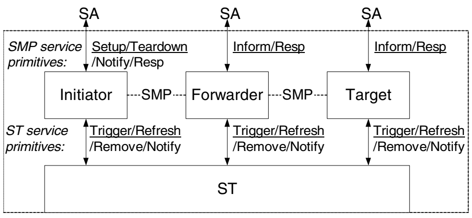
SA
SA
SA
SMP service primitives:
Setup/Teardown /Notify/Resp
Inform/Resp
Inform/Resp
Initiator
SMP
Forwarder
SMP
Target
ST service primitives:
Trigger/Refresh /Remove/Notify
Trigger/Refresh /Remove/Notify
Trigger/Refresh /Remove/Notify
ST

Fig. 1. A general architecture of state management systems

Expected Behavior of State Management Protocols: Behaviors of a state management protocol are determined by timers and state messages used by the three types of protocol entities, namely the state initiator, forwarder and target. An important feature of soft state systems is timers. In a soft state system, there can be three types of timers dealing with state management: the *state timer* which will expire the state unless refreshed, the *refresh timer* which triggers periodical refreshes, and the *retransmission timer* which triggers periodical retransmission of trigger or removal messages (SS+RT or SS+RTR). In HS systems there are only retransmission timers for state trigger and removal messages, while state timers and refresh timers are necessary for SS systems for operating

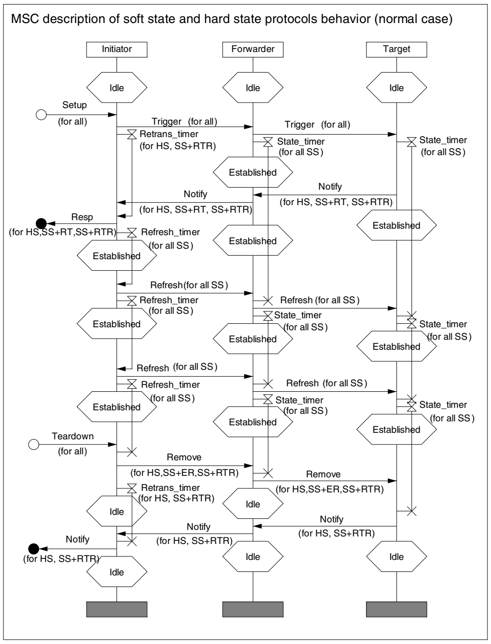
MSC description of soft state and hard state protocols behavior (normal case)
Initiator
Forwarder
Target
Idle
Setup (for all)
Trigger (for all)
Retrans_timer (for HS, SS+RTR)
State_timer (for all SS)
Established
Notify (for HS, SS+RT, SS+RTR)
Resp (for HS,SS+RT,SS+RTR)
Refresh_timer (for all SS)
Refresh (for all SS)
Teardown (for all)
Remove (for HS,SS+ER,SS+RTR)
Notify (for HS, SS+RTR)

Fig. 2. MSCs for typical operations in various state management protocols

state refresh messages. The existence or absence of these timers and different operations for state messages in a state management protocol determines which protocol type it belongs to.

The MSCs are shown in Fig. 2 to illustrate various sate management protocols. The communications between the three types of entities are realized through several service primitives (Setup, Trigger, Teardown, and optionally, Refresh, Resp, Notify and Remove). For simplicity purposes, we omit the Inform and Resp primitives since they generally do not change state information.

The protocol communication takes place in 3 possible, distinct phases:

– *State Setup:* This phase is initialized by SA in the Initiator with a *Setup*. Initiator can thereafter issue a *Trigger* towards Target. Upon the receipt of Trigger, every Forwarder creates a state (which is associated with a state timer for soft state protocols), and then forwards Trigger on. When Target receives Trigger, it creates a state (which is associated with a state timer for soft state protocols). If HS, SS+RT or SS+RTR is used, additionally Target issues back a *Notify* to Initiator when it receives a Trigger. In this case, Initiator starts a retransmission timer before it issues Trigger. If it does not receive a Notify after the Retransmission timer expires, Trigger is transmitted again. After repeating certain times, retransmission stops and Initiator becomes inactive again.

- *State Maintenance:* This phase is only used in soft state protocols. Upon the expiration of the refresh timer, Initiator sends a *Refresh*[1] towards Target and restarts the timer. Any Forwarder or the Target receives the Refresh checks whether corresponding state already exists: if yes, refreshes the state timer, otherwise recovers a state together with a state timer. If it is not the Target, the node forwards the Refresh onwards. If no Refresh is received in a Forwarder or Target before the state timer expires, state will be removed.
- *State Teardown:* In pSS or SS+RT, there is no such phase; Initiator remains inactive and states will expire in all the other nodes when their state timers time out. In other state management protocols, a *Remove* is issued by the Initiator towards the Target to remove all state and associated timers (should they exist). Additionally, if HS or SS+RTR is used, after Target receives Remove and removes its state (and timers), a *Notify* is sent back to the Initiator. Initiator in a HS or SS+RTR system retransmits Remove (no more than a maximal retrial counter) if no Notify is received within a given time.

Possible Problems in State Management Operations: Because the above general model captures the key concepts and operations of all the 5 possible types of state management protocols, we believe it can serve as the basis for studying actual behaviors of real state management systems. The most obvious problem that occurs in state management protocols is failure to install or remove state correctly. In addition, there are timing considerations to be taken account, since a protocol of this kind needs to be able to react to timer events and receipts of state messages appropriately. An installed state can become invalid either due to the receipt of a removal message or when the state timer expires. The latter can occur when a state refresh or notify message gets lost during its transmission, or when the state initiator crashes.

There are other potential problems with state management protocols, e.g., infinite state management loops, i.e., state messages enter a transmission circle somewhere between an initiator and a target. This can be either an endless loop consisting of a set of interacting states which cannot progress towards a next expected behavior (livelock), or a state without possibilities to enter another state (deadlock). A deadlock is most often caused by two processes waiting for a message from each other; the result is that both wait and nothing happens. In pSS, such loops are theoretically impossible, as it only allows refresh and expiration operations for a state, and there is no resource contention. However, we cannot exclude this possibility in more comprehensive state management protocols, due to more complex synchronization and/or notification mechanisms.

These are very undesirable since there appears to be a valid state management behavior in any individual node, but in reality it cannot further deliver other desired messages or jump out of certain running state(s). For example, the following operation could be possible in the original description of SS+RT [8].

[1] Note some SS protocols (e.g., RSVP) allow initiated by intermediate nodes; here we limit refreshes to be originated from Initiator for simplicity.

Example 1. A deadlock behavior in SS+RT can be briefly explained as follows:

1. Initiator initially sends a Trigger to Target through Forwarder. The link between Forwarder and Target then suddenly goes down.
2. Trigger is lost before reaching Target, and Initiator cannot receive a desired Notify which acknowledges the success of state installation along the path.
3. After retransmission timer expires, Initiator resends a Trigger.
4. Again the Trigger is lost, and Initiator still thinks it is just due to random loss and retransmits the Trigger.
5. If there is no specification about which conditions to stop sending Triggers, the result is a deadlock in which case Initiator is waiting for a Notify after sending out a Trigger while Responder is waiting for a Trigger forever.

This error is somewhat easy to detect and fix. However, sometimes such flaws can be very subtle, so that even senior designers cannot detect them in protocol specifications, particularly in IETF informal, text-based specifications for complex operations of state management protocols, which were designed typically without formal modeling, validation and verification. For example, the inadvert synchronization problem [13] was noticed as an important issue for periodical soft state systems. In real specifications of some soft state protocols, for example RTCP and RSVP, designers have tried to avoid this problem by setting the refresh timers to be varied randomly (e.g., over the range [0.5, 1.5] times the calculated interval). However, true randomness in a real implementation is hard to achieve and moreover, as these intervals are sometimes not mandatory requirements in protocol specifications (e.g., as a "should" requirement in RSVP), typically default fixed values (30 seconds in case of RSVP) are used instead in practice for implementation simplicity. All these contribute to the potential synchronization problem of soft state management.

3 Modeling Soft State Protocols with SDL

Key Modeling Issues and the System Model: Based on the above general architecture, we chose the most comprehensive state management protocol, SS+RTR in a hop-hop manner, as the target for modeling. Other variants can be easily derived from this model by removing certain messages, timers, state transitions and/or behaviors.

The address of each node is represented by a pre-assigned integer, 1, 2, 3, respectively. The state message content is also simplified as a simple character string. There are further issues vital for the modeling process:

– How do we model ST, so as to allow state messages (generally from Initiator to Target) be visible for Forwarder?
– How do we model the lossy channel, which can be of a given loss rate?
– How do we model the duration (start and end) of a session state? This also naturally reflects the system model, namely which information should be made inside the system, and which needs to be put in the environments.

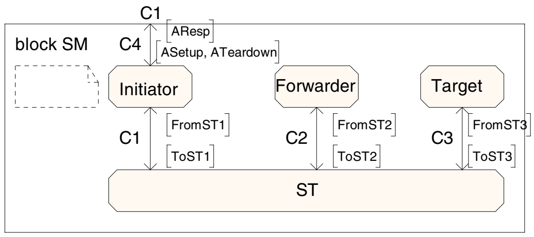

Fig. 3. SDL blocks of the SMP system

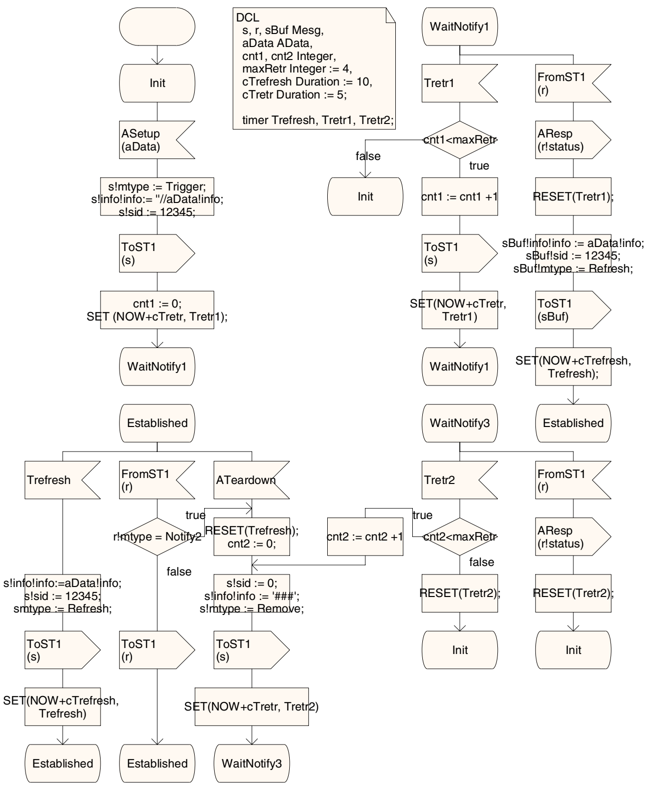

Fig. 4. SDL model for Initiator in the SMP system (SS+RTR)

Fig. 3 shows the SMP system model for SS+RTR. We assume the SMP system model to be composed of 3 nodes: Initiator, Forwarder and Target, each represented by a process. Furthermore, an additional process ST is used to transmit SS messages from Initiator towards Target, or reverse. These 4 processes form a single block *SM* of the system.

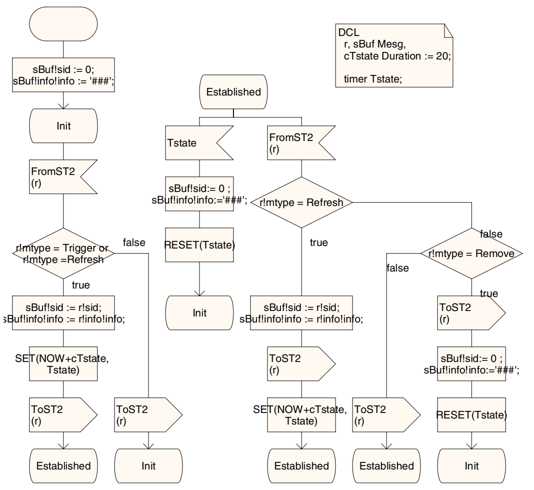

Fig. 5. SDL model for Forwarder in the SMP system (SS+RTR)

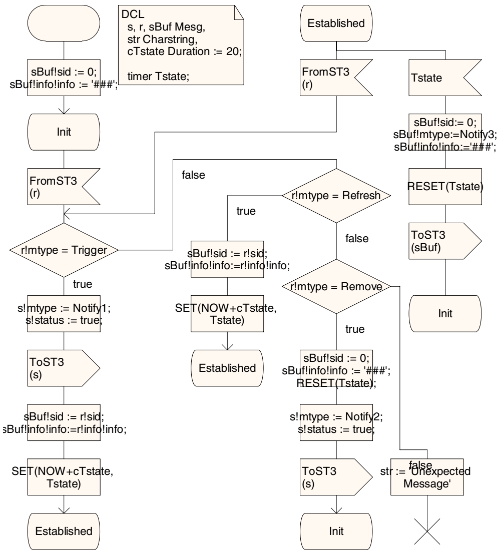

Fig. 6. SDL model for Target (SS+RTR)

ST: ST is modeled as Fig. 4 and can be used for all SMP variants. We use random Abstract Data Type (ADT) to simulate a given loss rate of the link. Note the transport service should determine the direction (forward or backward) according to the type of received messages It is easy to extend to form more comprehensive STs e.g., those having different loss rates in different links.

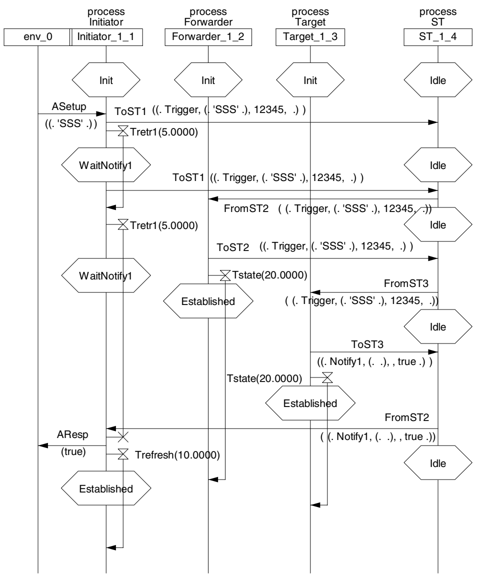

Fig. 7. MSCs for state setup phase (SS+RTR)

SMP Entities: The Initiator process (Fig. 5) communicates with environments through an SA-SMP interface. When it receives an ASetup with certain state information data, it assigns a new session identifier (sid) and installs a soft state locally, before going through the state setup and maintenance phases. When Initiator receives an ATeardown from the SA-SMP interface, it enters the teardown phase. For simplicity sid is currently set as a fixed value. To avoid confusion different notification messages have to be identified: Notify1 for notifying the success of state setup, Notify2 for notifying state timer expiration in Target, and Notify3 for notifying the success of state teardown.

The Forwarder process (Fig. 6) first listens to ST input. If a Trigger or Refresh message arrives, it installs a soft state locally and forwards the message on. Expiration of a local state in Forwarder only removes itself.

The Target process (Fig. 7) installs and (if it should exist) refreshes soft state locally upon receipt of a Trigger or Refresh message. When receiving a Trigger or Remove message, Target needs to notify Initiator about it. Expiration of a local state in Target also accompanies a Notify2 to trigger a teardown.

4 Model Verification and Validation

Since state management protocols involve distributed timers and message interaction, their correctness is hard to verify using an informal description. We use the Tau integrated package SDT 4.4 which includes support for SPIN, to verify

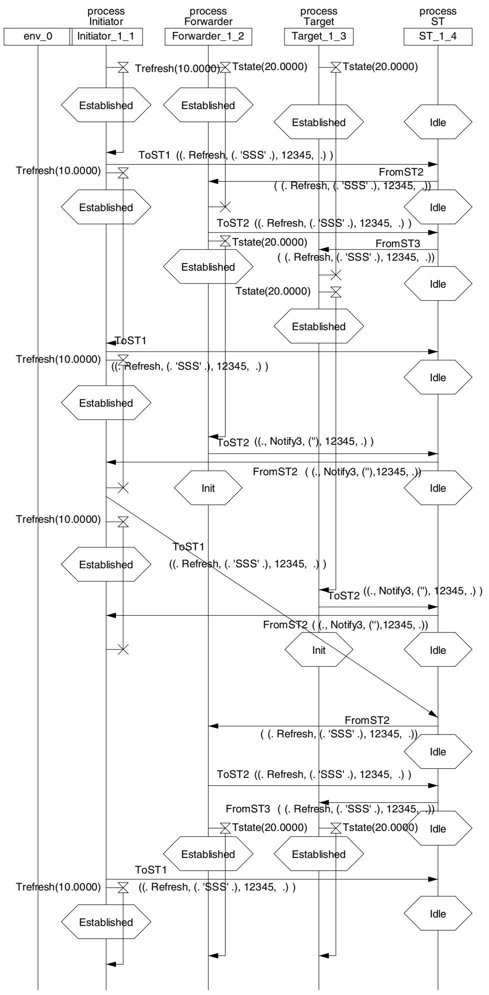
process Initiator
process Forwarder
process Target
process ST
env_0
Initiator_1_1
Forwarder_1_2
Target_1_3
ST_1_4
Trefresh(10.0000)
Tstate(20.0000)
Tstate(20.0000)
Established
Established
Established
Idle
ToST1 ((. Refresh, (. 'SSS' .), 12345, .))
Trefresh(10.0000)
FromST2
((. Refresh, (. 'SSS' .), 12345, .))
Established
Idle
ToST2 ((. Refresh, (. 'SSS' .), 12345, .))
Tstate(20.0000)
FromST3
((. Refresh, (. 'SSS' .), 12345, .))
Established
Idle
Tstate(20.0000)
Established
ToST1
Trefresh(10.0000)
((. Refresh, (. 'SSS' .), 12345, .))
Idle
Established
ToST2 ((., Notify3, (''), 12345, .))
FromST2 ((., Notify3, (''),12345, .))
Init
Idle
Trefresh(10.0000)
ToST1
((. Refresh, (. 'SSS' .), 12345, .))
Established
ToST2 ((., Notify3, (''), 12345, .))
FromST2 ((., Notify3, (''),12345, .))
Init
Idle
FromST2
((. Refresh, (. 'SSS' .), 12345, .))
Idle
ToST2 ((. Refresh, (. 'SSS' .), 12345, .))
FromST3 ((. Refresh, (. 'SSS' .), 12345, .))
Tstate(20.0000)
Tstate(20.0000)
Idle
Established
Established
ToST1
Trefresh(10.0000)
((. Refresh, (. 'SSS' .), 12345, .))
Idle
Established

Fig. 8. MSCs for state maintenance phase (SS+RTR)

the SDL model of the SS+RTR protocol against deadlocks, unspecified receptions, livelocks and unreachable states. In order to verify the model, we have chosen the following scenario case study to validate our design against some of the specific properties of the modeled SS+RTR protocol:

1. Establish a session between Initiator and Target.
2. Stop the state message transmission between Forwarder and Target.
3. Change the ST loss rate between Forwarder and Target to different values between 0 and 1.
4. Stop the Target process.
5. Let Initiator teardown the session.

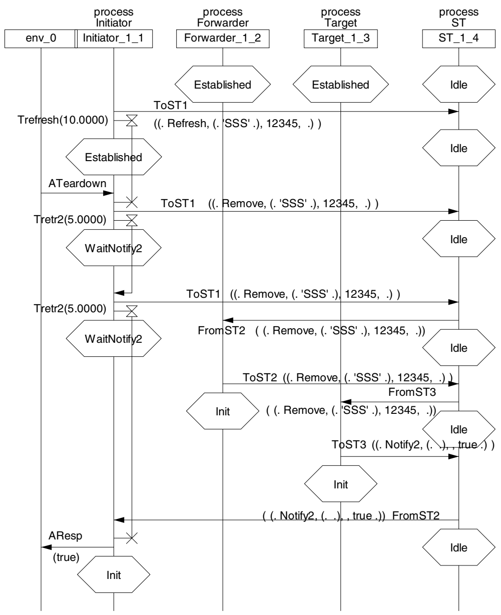
process Initiator
process Forwarder
process Target
process ST
env_0
Initiator_1_1
Forwarder_1_2
Target_1_3
ST_1_4
Established
Established
Idle
ToST1
Trefresh(10.0000)
((. Refresh, (. 'SSS' .), 12345, .))
Idle
Established
ATeardown
ToST1 ((. Remove, (. 'SSS' .), 12345, .))
Tretr2(5.0000)
Idle
WaitNotify2
ToST1 ((. Remove, (. 'SSS' .), 12345, .))
Tretr2(5.0000)
FromST2 ((. Remove, (. 'SSS' .), 12345, .))
WaitNotify2
Idle
ToST2 ((. Remove, (. 'SSS' .), 12345, .))
FromST3
Init
((. Remove, (. 'SSS' .), 12345, .))
Idle
ToST3 ((. Notify2, (. .), , true .))
Init
((. Notify2, (. .), , true .)) FromST2
AResp
(true)
Idle
Init

Fig. 9. MSCs for state teardown phase (SS+RTR)

With the above scenario, we have covered all ST service primitives and SMP service primitives as well as all important scenarios, but not all possible scenarios. Therefore, after checking the scenario, we have used Tau validator to validate all possible walk algorithms.

We generated a number of MSCs for this scenario case study to check the protocol functionality at each stage of the simulation. For the SDL models designed in Section 3, MSCs shown in Fig. 7-9 help us to identify the representative protocol behaviors: 1) Fig. 7 represents the message interactions for state setup phase (SS+RTR) upon the receipt of an external trigger (ASetup) is received by the system, completed by a response indicating successful state establishment from the system to the environment (AResp); 2) after that, the system goes into the internal state maintenance phase without interactions with the environment (shown in Fig. 8); 3) upon the receipt of an external trigger for state teardown (ATeardown), the system enters into the state teardown phase (Fig. 9). Through a comparison with the general description, we are able to refine the abstract system and make more concrete descriptions.

Through this procedure, we have found some modeling errors as well as flaws in the original description, including the missing default behavior for some message types, and unreachable states. We have found that an imprecise informal specification can result in deadlocks and livelocks (for example, the retransmission counters and refresh durations – either explicit or implicit – are missing in many text-based IETF specifications); through verification and validation of the SDL models, these could be avoided. As an example we corrected a flaw in the original description by adding description on retransmission counters.

Some other results show that in the first version of the models Notify1, Notify2 and Notify3 messages received by Forwarder were not processed but consumed; they need to be sent back to ST. Also, misused values of timers can exclude Initiator from entering the Established state. We also come to the following conclusion: by adding reliable trigger and explicit removal to soft state protocols, the usage of state (reflected as network resources especially memories) can be more efficient. This can be explained as, for example for the reliable explicit removal case, if a user tries to remove a state, but the teardown message is lost during transmission, the state will remain in place until it times out after a relatively long time. Since state typically implies enhanced services for end-to-end communications, maintaining a state incurs costs, thus users must pay for the extra time that has been spent waiting for the state expiration.

With the developed model, we have verified these aspects against the description of SS+RTR, and improved upon it.

5 Summary and Future Work

Given the fundamental importance of soft state protocols, it is vital that their behaviors are specified correctly. We have generalized and modeled behaviors for different variants of soft state communications with the aid of formal techniques SDL and MSC. On the other hand, we found that a weakness concerning inaccurate timing in the tool; a more realistic performance evaluation has to rely on tools with better real-time support. Two types of error sources are identified during the modeling processes: one due to technical flaws in the specifications and the other due to our modeling errors (which can also be caused by misunderstanding of imprecise specifications). Nevertheless, formal techniques have turned out to be of great help for efficient designing and engineering of soft state communication models, and in particular functional behaviors (through checking for possible deadlocks and livelocks). Currently we are modeling two realistic soft state protocols, CASP and RSVP. Based on that we will further study how soft state protocols handle node failure, mobility and route changes using formal techniques.

References

1. Braden, R., Zhang, L., Berson, S., Herzog, S., Jamin, S.: Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205 (1997)
2. Schulzrinne, H., Tschofenig, H., Fu, X., McDonald, A.: CASP – Cross-Application Signaling Protocol. Internet draft, work in progress (2003)
3. Paxson, V., Allman, M., et al.: Known TCP Implementation Problems. RFC 2525 (1999)

4. Fu, X., Hogrefe, D., Willert, S.: Implementation and Evaluation of the Cross-Application Signaling Protocol (CASP). In: Proc. of ICNP 2004, Berlin, Germany (2004)
5. ITU-T Recommendation Z.100 – Specification and Description Language (SDL). (1999)
6. ITU-T Recommendation Z.120 – Message Sequence Chart (MSC). (1999)
7. Schaible, P., Gotzhein, R.: View-based Animation of Communication Protocols in Design and in Operation. Computer Networks **40** (2002) 621–638
8. Ji, P., Ge, Z., Kurose, J., Towsley, D.: A Comparison of Hard-state and Soft-state Signaling Protocols. In: Proc. of SIGCOMM 2003, Karlsruhe, Germany (2003)
9. Raman, S., McCanne, S.: A Model, Analysis, and Protocol Framework for Soft State-based Communication. In: Proc. of SIGCOMM 1999, Cambridge, MA (1999)
10. Sharma, P., Estrin, D., Floyd, S., Jacobson, V.: Scalable Timers for Soft State Protocols. In: Proc. of INFOCOM'97, Kobe, Japan (1997)
11. Bradley, A., Bestavros, A., Kfoury, A.: Safe Composition of Web Communication Protocols for Extensible Edge Services. In: Proc. of Workshop on Web Content Caching and Distribution (WCW), Boulder, Colorado (2002)
12. Holzmann, G.: The Model Checker SPIN. IEEE Trans. on Softw. Engineering **23** (1997) 1–17
13. Floyd, S., Jacobson, V.: The Synchronization of Periodic Routing Messages. IEEE/ACM Trans. on Networking **2** (1994) 122–136