Reducing Software Architecture Models Complexity: A Slicing and Abstraction Approach

Daniela Colangelo[1], Daniele Compare[1],
Paola Inverardi[2], and Patrizio Pelliccione[2]

[1] Selex Communications, L'Aquila, Italy
{daniela.colangelo, daniele.compare}@selex-comms.com
[2] University of L'Aquila, Computer Science Department
Via Vetoio, 67010 L'Aquila, Italy
{inverard, pellicci}@di.univaq.it

Abstract. Software architectures (SA) represents a critical design level for software systems. Architectural choices need to be analyzed and verified to achieve a better software quality while reducing the time and cost of production. Model-checking is one of the most promising verification techniques, however its use for very large systems is not always possible due to the *state explosion problem*. In this paper we propose an approach that *slices* and *abstracts* the SA of a system in order to reduce the model complexity without compromising the verification validity. This approach exploits the characteristics of the SA model and the structure of the property of interest. It is applied to an industrial telecommunication system of the Selex Communications company.

1 Introduction

Recently, Software Architectures (SA) [1,2] have been largely accepted as a well suited tool to achieve better software quality while reducing time and cost of production. SA provide both a high-level behavioral abstraction of components and of their interactions (connectors) and, a description of the static structure of the system. The aim of SA descriptions is twofold: on one side they force the designer to separate architectural concerns from other design ones, thus abstracting away many details. On the other side, they allow for analysis and verification of architectural choices, both behavioral and quantitative, in order to obtain better software quality in an increasingly shorter time-to-market development scenario [3].

Formal Architectural Description Languages (ADL) have been employed to specify SA in a formal and rigorous way. They are the basis for many methods and tools for analysis and verification of software architectures, both behavioral and quantitative [3]. One of the most promising verification technique is model-checking since is fully automated and its use requires no supervision or formal methods expertise. Due to these reasons, in recent years model checking has gained popularity and it is increasingly used also in industrial contexts [4,5]. However the application of model checking techniques is still prevented by the

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 243–258, 2006.

state explosion problem. As remarked by Gerald Holzmann in [6] no paper has been published on reachability analysis techniques without a serious discussion of this problem. State explosion occurs either in systems composed of (not too) many interacting components, or in systems where data structures assume many different values. The number of global states easily becomes enormous and intractable. To solve this problem, many methods have been developed by exploiting different approaches [7]. They can be logically classified into two disjoint sets [8]. The first set, that we call *Internal Methods*, considers algorithms and techniques used internally to the model checker in order to efficiently represent transition relations between concurrent processes, such as *Binary Decision Diagrams* [9] (used for synchronous processes) and *Partial Order Reduction* [10] techniques (used for asynchronous processes). The second set, that we call *External Methods* includes techniques that operate on the input of the model checker (models), and can be used in conjunction with Internal Methods. In this set there are *Abstraction* [11], *Symmetry* [12], *Compositional Reasoning* [13,14,8], and *Slicing* [15,16].

In this paper we propose an *architectural slicing* and *abstraction* approach which exploits the characteristic of the SA model and the structure of the property of interest for reducing the model complexity without compromising the verification validity. *Program slicing* [17] is a technique which attempts to decompose the system by extracting elements that are related to a particular computation. It is defined for conventional programming languages and therefore it is based on the basic elements of a program, i.e. variables and statements. *Architectural slicing* is the result of applying the slicing idea to SA [18,19]. Thus the basic elements on which is based the *Architectural slicing* are components, connectors, ports, roles, and messages exchanged between components. An architectural slicing can be considered a subset of the behaviour of a software architecture with the attempt to isolate its parts that are involved in the slicing criterion. In the approach that we are proposing the slicing criterion is the property we want to check on the SA. Thanks to the architectural slicing we are able to extract the parts of the system that play a role on the behavior implied by the property of interested.

Our approach makes use of TeStor [20], an algorithm that, taking in input state machines and scenarios expressed in terms of Property Sequence Charts (PSC) [21,22], generates test sequences in the form of scenarios. TeStor generates all traces containing the messages expressed in the input PSC and in the same order defined in the PSC by suitably abstracting with respect to message repetitions and loops. In this way it generates a final number of traces. Thus, given in input the state machines defining the behavior of the components composing the system and the property of interest (expressed in PSC notation), TeStor identifies all dependencies in the state machines and can be used as basis for the architectural slicing. In this work, we propose to extend TeStor in order to colorize the states of the components state machines that are involved on the considered property. When this step is done we can cut off from the SA the states that are not colored, thus obtaining a reduced and sliced SA.

After the slicing is performed some *architectural abstraction* criteria can be furtherly used to abstract parts of the system that are implied by the property, but that are not directly involved in its formulation. Finally, the reduced system model can be model checked by using the CHARMY [23,24] tool. The efficacy of this approach strictly depends on the characteristic of the SA. However it can be completely automatized and for some systems offers a good reduction, as in the industrial case study presented in Section 5. Through the case study, we show how the traditional approach fails with the used hardware resources, and, contrariwise, how the system can be successfully verified following this approach.

After an analysis of related work in Section 2, in Section 3 we introduce the notions and the instruments required to understand the approach. The approach is detailed in Section 4, and put in practice in Section 5, by presenting an industrial case study is presented. Finally, in Section 6 we present conclusion and future work.

2 Related Work

Program slicing was firstly introduced in [25] and later extended in other works [26,27]. For the sake of brevity, we report here only relevant works at the software architecture level.

In [19] the authors propose a dependence analysis technique called chaining to support software architectures testing maintenance and so on. Links in chaining reflect the dependence relationships that are in the architecture description. The relationships are been both structural and behavioral and based on components ports. A similar approach is proposed in [16] where is proposed a dependence analysis based on three different kinds of analysis based respectively on: relationships between a component and a connector; relationships between a connector and a component; and relationships inside a connector or a component. In this work and in [18] the author suggests to use the system dependence net to slice architectural descriptions written in the ACME ADL, and in the WRIGHT ADL respectively. This method produces a reduced textual architectural description just containing the ADL code lines associated with a particular slicing criterion. The works [19] and [16] are very similar in the main goal; however [19] does not focus on the description of the components themselves, but rather on the more abstract nature of the components and the connections. Our work builds on these prior works and it is based on a well detailed description of the component itself. Contrary to these works that give an abstract description of the components, by introducing only a dependence relationship between two different ports of a component or between two different roles of a connector, our work is based on a component description in terms of state machines that give a detailed description of the component behavior.

The works introduced above present static slice and dependence analysis techniques. In [15] authors propose a dynamic slicing, determined according to the input at run time. This kind of technique gives a slice that is smaller in size than the static one, and helps to isolate a specific execution path. Our work, although

is not performed at run time, is strongly related to this work. In fact we are interested in identifying the execution paths that are implied by the property that we want to verify on the system. This property is represented, as already explained, as a PSC diagram and represents the slicing criterion in our approach. The slicing criterion of the approach presented in [15] contains the event to be observed, in addition our slicing criterion contains a set of events to be observed and temporal relationships between them.

3 Background

3.1 Charmy: A Tool for SA Designing and Model-Checking

Charmy [23,24] is a project whose goal is to easy the application of model-checking techniques to validate the SA conformance to certain properties. In Charmy the SA is specified through state diagrams used to describe how architectural components behave. Starting from the SA description Charmy synthesizes, through a suitable translation into Promela (the specification language of the SPIN [5] model checker) an actual SA complete model that can be executed and verified in SPIN. This model can be validated with respect to a set of properties, e.g., deadlock, correctness of properties, starvation, etc., expressed in Linear-time Temporal Logic (LTL) [28] or in its Büchi Automata representation [29]. Charmy allows users to describe temporal properties by using an extension of UML 2.0 sequence diagrams, called Property Sequence Charts (PSC) [21,22], that are successively translated into a temporal property representation for SPIN. The model checker SPIN, is a widely distributed software package that supports the formal verification of concurrent systems permitting to analyze their logical consistency by on-the-fly checks, i.e., without the need of constructing a global state graph, thus reducing the complexity of the check. It is the core engine of Charmy and it is not directly accessible by a Charmy user.

The state machine-based formalism used by Charmy is an extended subset of UML state diagrams: labels on arcs uniquely identify the architectural communication channels, and a channel allows the communication only between a pair of components. The labels are structured as follows: '['*guard*']'*event*'('*parameter_list*')' '/'op_1';'op_2';'$\cdots$';'op_n where *guard* is a boolean condition that denotes the transition activation, an *event* can be a message sent or received (denoted by an exclamation mark "!" or a question mark "?", respectively), or an internal operation (τ) (i.e., an event that does not require synchronization between state machines). Both sent and received messages are performed over defined channels *ch*, i.e., connectors. An event can have several parameters as defined in the parameters list. $op_1, op_2, \cdots, op_n$ are the operations performed when the transition fires.

3.2 Property Sequence Charts (PSC)

PSC [21,22] is a diagrammatic formalism for specifying temporal properties in a user-friendly fashion. It is a scenario-based visual language that is an extended

graphical notation of a subset of UML2.0 Sequence Diagrams. PSC can express a useful set of both *liveness* and *safety* properties in terms of messages exchanged among the components forming a system. Finally, an algorithm, called PSC2BA, translates PSC into Büchi automata.

PSC uses a UML notation, stereotyped so that: (i) each rectangular box represents an architectural component, (ii) each arrow defines a communication line (a channel) between two components. Between a pair of messages we can select if other messages can occur (loose relation) or not (strict relation). Message constraints are introduced to define a set of messages that must never occur in between the message containing the constraint and its predecessor or successor. Messages are typed as *regular messages* (optional messages), *required messages* (mandatory messages) and *fail messages* (messages representing a fault).

An example of PSC is in Figure 4.

3.3 TEst Sequence generaTOR (TESTOR)

TESTOR [20] is an algorithm, which, taking in input state machines and scenarios, generates test sequences in the form of scenarios. The algorithm is based on the idea that scenarios are usually incomplete specifications and represent important and expected system interactions. Such incomplete specifications may be "completed" by recovering, from state machines, the missing information. The output of the algorithm is a set of sequence diagrams (outSD) containing the sequence of messages expressed by the input sequence diagram (inSD), enhanced and completed with information gathered by the components' state machines.

TESTOR, focussing on the first (not visited) message m in the selected inSD, and looking inside each state machine, searches a trace which permits to reach m, starting from the current state of the state machine. When such trace is found, TESTOR recursively moves to the next (not visited) message m' in inSD, and checks a trace which permits to reach m' starting from the current state. At the end of this process, TESTOR tries to merge together the different trace portions in a set of traces (the set outSD) which move from the initial state and covers any message in the inSD.

For more information on the TESTOR algorithm, please refer to [20].

4 The Approach

Our proposal makes use of TESTOR, the algorithm introduced in Section 3.3, which aims to extract test sequences from a SA specification (given in terms of state machines) following the trail suggested by a PSC property. We propose to use an extension of TESTOR, called DEPCOL, which, instead of returning a set of sequences, colors the state machines highlighting the parts of the SA model that are required to correctly verifying the property of interest. After this step is done the abstraction step can be performed. The idea is to compact, if and when is possible, some states of a component in only one abstract state. Since the transition from one state to another is made when a message is exchanged between a couple of components, this step is not trivial. When a final

reduced SA model is obtained, cutting off the parts of the system that can be removed and suitably abstracting the system, CHARMY can be used. CHARMY and TESTOR use the same representation for state machines. Since the notation used in CHARMY for expressing the property is PSC, the integration between this approach and CHARMY is straightforward. In the following we detail the approach step by step. Note that the whole approach can be fully automatized.

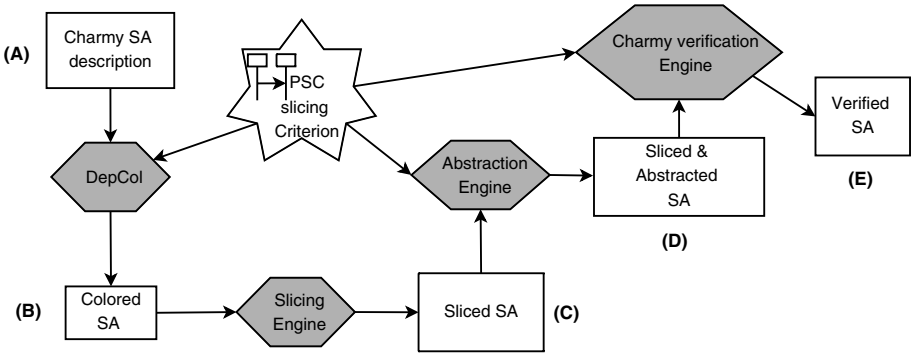

Fig. 1. The Approach

Figure 1 summarizes the approach: **(A)** we start from a CHARMY SA description, i.e. a SA described in terms of components and connectors with communicating state machines used to represent components and connectors behaviors. The PSC property is our slicing criterion. **(B)** DEPCOL, the extension of TESTOR that we propose, gets in input the CHARMY SA and the slicing criterion and returns a colored SA. **(C)** The colored SA contains information about the parts of the system that are necessary and the parts of the system that can be cut off. Thus, the slicing engine gets in input the colored SA, cuts off the unnecessary parts and returns a sliced SA. **(D)** The sliced SA is the input of the abstraction engine that returns a sliced and abstracted SA. **(E)** Finally, CHARMY can be used to check, through model checking techniques, if the reduced SA satisfies the property we want to verify, expressed as a PSC.

Section 4.1 explains the steps **(A)**, **(B)**, and **(C)**, while Section 4.2 details the step **(D)**. The step **(E)** is the standard use of CHARMY and it is explained in Section 3.1.

4.1 Architectural Slicing

The inputs of this step are the state machines representing the components behavior and the property of interest expressed as a PSC diagram.

Based on TESTOR we define the new algorithm that we call DEPCOL. This algorithm colors the parts of the state machines that are required for the SA verification. Let M be the set of messages that are $arrowMSGs$ or $intraMSGs$ of the considered PSC. Each start or target state of a $m \in M$ in at least one

sequence generated by TeStor is colored. This modification of TeStor is very easy. Unfortunately it is not enough. In fact the DepCol state machines (the same used by Charmy) make use also of variables to synchronize and store the state machines computation state. These variables can be *local* to a state machine but can be also *shared* among different state machines. Thus, let v_l be a *local* variable contained in a transition that has a colored target state. For each occurrence of v_l in the same state machine, if it is contained in a transition that has a non colored target state s, then each path leading from the initial state to s is colored. Analogously, for each *shared* variable v_s contained in a transition that has a target colored state, every occurrence of v_s in each state machine is identified. Also in this case, if v_s is contained in a transition that has a non colored target state s, each path from the initial state of the component containing v_s leading s is colored.

While coloring the paths, new messages can be considered (messages that have both start state and target state as colored states). Since messages have a component sender and a component receiver, new parts of the state machines require to be colored. Doing this step new messages could be considered, and then the whole coloring process must be iterated. It is important to note that only one state machine at a time is considered while coloring, thus we do not have problems of states explosion.

At the end of this step we have the state machine colored. The following properties hold:

- each state playing a role in the property is colored;
- each state that is non colored does not play a role in the property;
- is not possible to have a non colored state in the middle of a path that starts with the initial state of a state machine and that ends with a colored state. This is assured by construction, since we start from a state and we color each state traversed in reaching the initial state.

Thus, for each state machine, every message that has a start state or a target state not colored is cut off. For each state machine, every state not colored is cut off. Note that, it is impossible with the cut to generate two or more not connected parts of a state machine.

4.2 Architectural Abstraction

The idea of this step is to reduce the complexity of the model by abstracting parts of the state machines without compromising the validity of the verification. In the following we refer to the state machine formalism used by Charmy and shortly described in Section 3.1. We introduce the following two abstraction rules:

R1: For each state machine that has only one state, each sent message $m \notin M$ could be deleted. In order to do it we have to analyze each reception of m (on other state machines). Let s_0 be the start state and s_1 be the target state of the message m ($s_0 - ?m \rightarrow s_1$). If $outdegree(s_0) == 1$[1] then for

[1] $Outdegree(s)$ is the number of messages that have s as start state.

each message m' that has s_0 as target state, s_1 becomes the new target state of m' and the state s_0 can be deleted.

If m has a guard, the guard is preserved while m can be deleted. If m has an operation *op*, and s_0 is the initial state of the state machine, then *op* is preserved and m is deleted; otherwise if s_0 is not the initial state of the state machine, then *op* is added to the operations of each message that has s_0 as target state. A state machine with only one state without messages can be deleted. The same rule applies for received messages.

R2: Let SM_1 be a state machine, for each pair of consecutive exchanged messages[2], $s_1 - m_1 \rightarrow s_2$ and $s_2 - m_2 \rightarrow s_3$ and with $m_1, m_2 \notin M$, if m_1 and m_2 are always exchanged consecutively and in the same order in any other state machine SM_2, $s'_1 - m_1 \rightarrow s'_2$, and $s'_2 - m_2 \rightarrow s'_3$, then they can be abstracted and s_1 and s_3 collapse in the same state inheriting all entering and exiting messages. The same holds for s'_1 and s'_3. Note that not necessary s_1 (s'_1), s_2 (s'_2), and s_3 (s'_3) must be different states.

This rule can be applied iff m_1 or m_2 are self transitions or the states s_2 and s'_2 have *degree* (i.e. the number of entering and exiting messages) equals to 2, i.e. s_2 (s'_2) has only one entering message, m_1 (m'_1) and only one exiting message m_2 (m'_2). In fact, we cannot abstract if the states s_2 or s'_2 are involved in other paths.

These two rules are applied until it is not possible to further abstract the system.

The algorithm operates separately on each state machine without requiring they parallel composition.

5 The Integrated Environment for Communication on Ship (IECS) Case Study

The Integrated Environment for Communication on Ship (IECS), a project developed by Selex Communications, operates in a naval communication environment. IECS provides heterogeneous services on board of the ship.

The purpose of the system is to fulfill the following main functionalities: *i)* provide voice, data and video communication modes; *ii)* prepare, elaborate, memorize, recovery and distribution of operative messages; *iii)* configuration of radio frequency, variable power control and modulation for transmission and reception over radio channel; *iv)* remote control and monitoring of the system for detection of equipment failures in the transmission/reception radio chain and for the management of system elements; *v)* data distribution service; *vi)* implement communication security techniques to the required level of evaluation and certification. The SA is composed of the IECS Management System (IECS-MS), CTS, and EQUIPMENT components as highlighted in Figure 2.

[2] Note that here we do not consider send and receive of messages because the rule is independent of the operations. Thus, if SM_1 sends m_1, SM_2 has to receive it in order to apply this rule and viceversa.

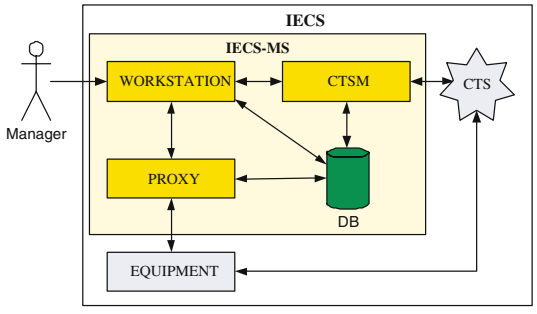

Fig. 2. IECS Software Configuration

In the following we focus on the IECS-MS, the more critical component since it coordinates different heterogeneous subsystems, both software and hardware. Indeed, it controls the IECS system providing both internal and external communications. The IECS-MS complexity and heterogeneity need the definition of a precise software architecture to express its coordination structure. The system involves several operational consoles that manage the heterogenous system equipment including the ATM based Communication Transfer System (CTS) through Proxy computers. For this reason the high level design is based on a manager-agent architecture that is summarized in Figure 2, where the Workstation (WS) component represents the management entity while the Proxy and the Communication Transfer System Manager (CTSM) components represent the interface to control the managed equipment and the CTS, respectively.

The functionalities of interest of the IECS-MS are: *i)* service activation; *ii)* service deactivation; *iii)* service reconfiguration; *iv)* equipment configuration; *v)* control equipment status; *vi)* fault CTS. A *service*, in this context, denotes a unit base of planning and the implementation of a logic channel of communication through the resources of communications on the ship. All the above described functionalities are "atomics", since it is not possible to execute two different functionalities at the same time on the system.

In this paper we focus on the *Service Activation* functionality for showing how we reduced the complexity of the SA for the verification of properties.

Service Activation Functionality: The Manager actor requests a service activation to the Workstation component that updates the information of the service that the Manager wants to activate on the DB component. If the CTSM is online, then the Workstation proceeds in parallel to create the chain of communication and configures the parameters of the equipments involved in the service. The DB component is finally updated.

5.1 IECS Case Study: System Modeling and Verification

In the previous sections we defined the static SA of the IECS-MS system. Now we extract from the specification, the state machines that describe the

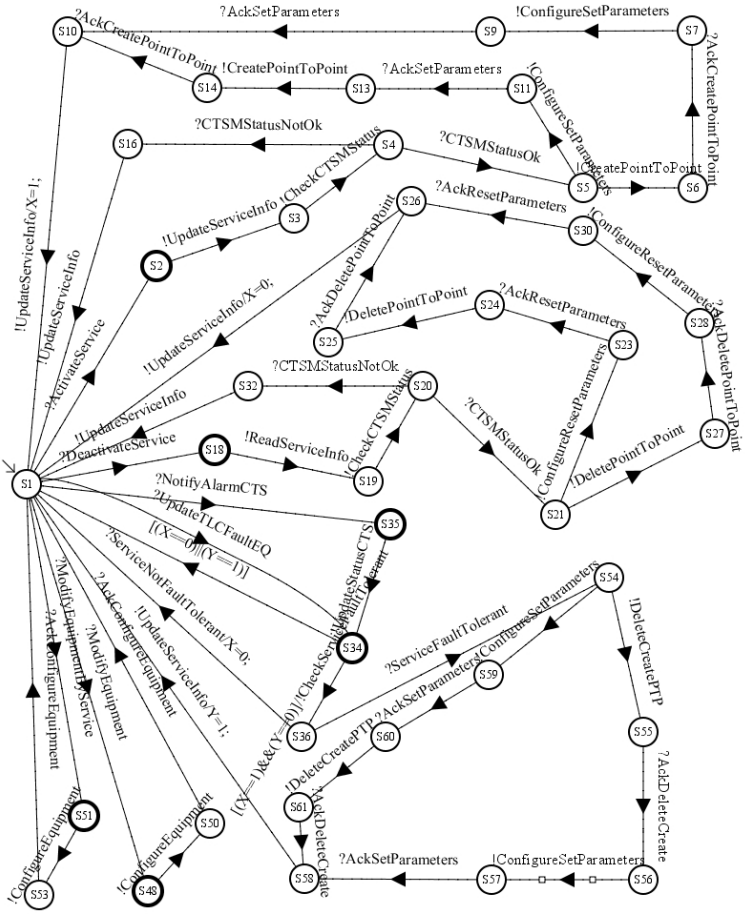

Fig. 3. Workstation internal behavior

internal behavior of the system components and the PSC scenarios that define the properties that the system has to satisfy.

Figure 3 shows the state machine for the WS component. This component has only one thread of execution. The actual size of the system does not permit to report in the paper details about the whole system. For this reason in the following we illustrate our approach only on significant excerpts of the system in order to give an idea of the modeling technique and of the analysis of the process followed.

The WS component coordinates and manages all the functionalities of the system. The access to each functionality is obtained through the reception of a message from the other components (e.g. USER, CTSM and Equipment); the reception of this message leads to a state that is the entry state of the functionality, represented in Figure 3 as a bold state. For example, when WS receives the message *Activate Service* it goes in the state *S2* to entry the path that manages the *service activation* functionality.

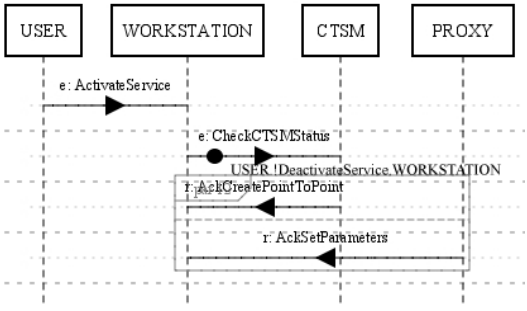

Fig. 4. Property: Service Activated

Furthermore, in the state machine is represented the "atomicity" of the functionalities of the system that are managed by the WS component. Every time a new functionality is required, if WS is ready for satisfying the request (i.e. a request can be satisfied only when previously requested functionalities are accomplished), this component goes from state *S1* to the path that manages this functionality. Finally, in the state machines are introduced two variables, one *shared*, `X`, for storing the service activation, and one *local*, `Y`, for storing the reconfiguration of the service in case of a fault of the CTS or Equipment components.

In order to check that the system correctly implements these functionalities, we define some properties that the SA should satisfy. The properties are modeled as PSC scenarios. Due to space reasons, in this paper we focus only on one property, that we use to show the approach. Figure 4 reports the considered property that concerns the service activation. The property is composed of two *regular messages* (the precondition of the property) that realize the service activation request. When the precondition is satisfied, if the USER does not deactivate the service (the constraint of the second message) the service must be activated (the last two *required* messages of the scenario).

When state machines and the PSC diagrams are modeled in CHARMY, the runnable Promela prototype of the system and the Büchi Automata of the property can be automatically generated. Through the use of the SPIN model checker we verified that the SA of our system is deadlock free, does not contain invalid end states, and does not contain unreachable parts (the standard verification of SPIN). The check is performed by using a PC with 3 Gb of RAM and with 4 processors of 1 Ghz. The size of the model is the following: $States = 1.27e + 08$ $Transitions = 6.15646e + 08$ $Memory = 2037.528Mb$.

Unlikely, when we tried to verify the properties of interest, we run into the *state explosion problem*. Thus, the next step is to apply the approach presented in Section 4 trying to reduce the complexity of the system.

Slice and Abstraction Applied to the Case Study. From the CHARMY SA description and the property *Service Activated* is now possible to obtain a colored SA, through the use of the algorithm DEPCOL. This colored SA, represented in Figure 5, highlights all paths required by the property.

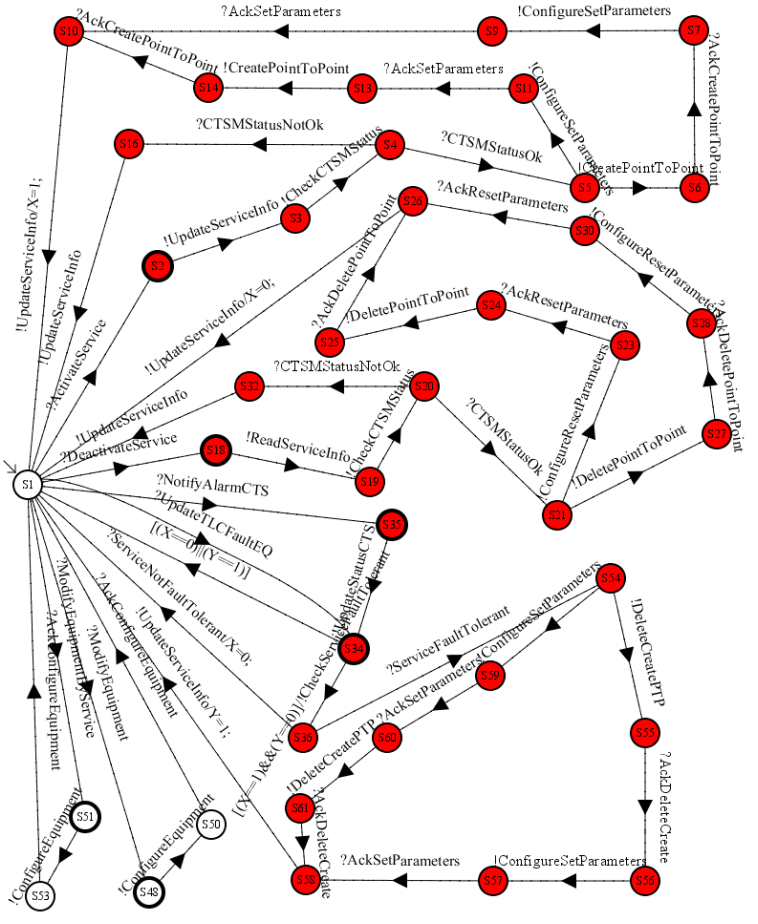

Fig. 5. Colored SA on WS state machines

The path that starts with the state $S2$ identifies the functionality *Service Activation* that is useful for the property we want verify. Instead, the message *DeactivateService* conducts to the path that manages the functionality *Service Deactivation*, while the messages *NotifyAlarmCTS* or *UpdateTLCFaultEQ* represent the entry in the path that manages the *Service Reconfiguration* when there is a fault in the components CTS or Equipment, respectively. The last two path are colored since they contain operations with *shared* and *local* variables. The states not colored in Figure 5 are then deleted to obtain the sliced SA.

Then, we proceed with the abstraction on the sliced SA, following the rules of abstraction presented in Section 4.2. Thanks to the first rule, in the case study we can delete all the state machines with only one state and containing messages that do not belong to the property. In the case study we deleted one *Thread* with only one state of the component DB, and consequently we deleted the relative

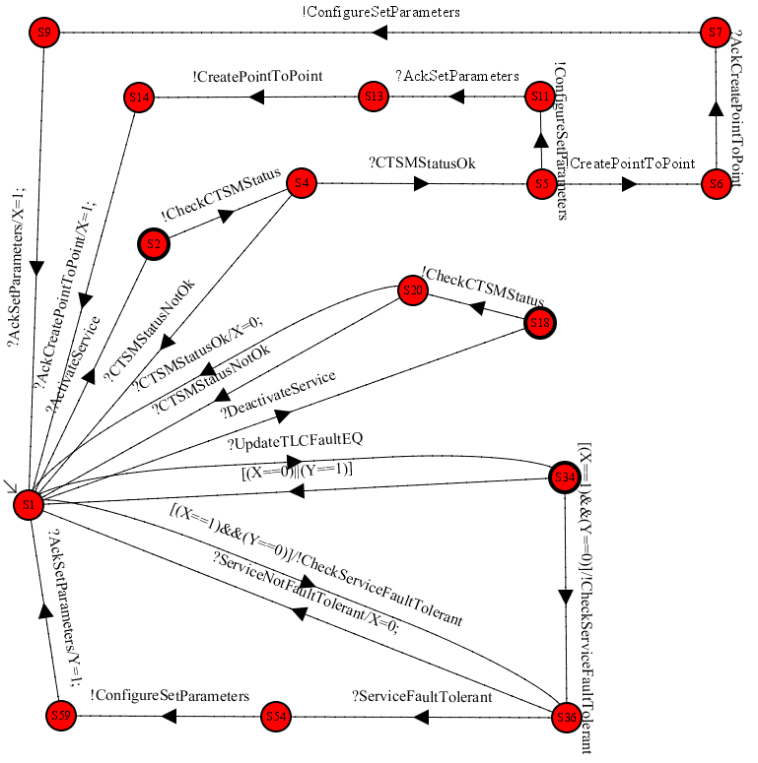

Fig. 6. Sliced and abstracted SA on WS state machines

send messages on the other components, e.g. considering the WS components, all the *UpdateServiceInfo* and *ReadServiceInfo* messages are deleted (refer to Figure 6).

Furthermore, still considering the WS component, the send message *UpdateServiceInfo* contains some operations; so, when the message is deleted the

Table 1. Resources used for verification of the properties

Property	System	Depth	Memory (Mb)	States	Transitions
Activate Service	full	> 8766	> 3034.084	$> 1.51433e+08$	$> 1.20706e+09$
	reduced	108.829	239.614	$1.6719e+007$	$2.8475e+006$
Deactivate Service	full	> 8766	> 3034.084	$> 1.51433e+08$	$> 1.07926e+09$
	reduced	9253	186.264	$2.11009e+006$	$1.18284e+007$
Reconfiguration Service	full	> 8802	> 3034.084	$> 1.51433e+08$	$> 1.50446e+09$
	reduced	2619	153.598	$1.65737e+006$	$9.73146e+006$
Modify Equipment	full	> 8678	> 3034.084	$> 1.51433e+08$	$> 9.32346e+08$
	reduced	265	34.302	741	2063
Modify Equipment by Service	full	> 8678	> 3034.084	$> 1.51433e+08$	$> 9.32346e+08$
	reduced	471	34.302	741	2063

operations are added to the messages that happen before it. The second rule allows us to delete the message *DeleteCreatePtP* and *AckDeleteCreate*. The approach is iterated until it is not further possible to abstract the system. The obtained model for the WS component is presented in Figure 6.

In Table 1 we report the result of the verification for all properties, comparing the resources used for verifying the reduced and the full model. Since for each property the result of the verification of the full model was *out of memory*, the information reported are the last result before the error. As can be noted, the verifications of the reduced SA are obtained by using very lower resources.

6 Conclusion and Future Work

In this paper we presented a slicing and abstraction approach for reducing the complexity of a SA description. The approach is composed of several steps: the SA description, in terms of Charmy state machines, is colored by an algorithm called DepCol that highlights the parts of the system that are required for the property verification. A slicing engine cuts off the unnecessary parts of the system. An abstraction engine further reduces the complexity of the model abstracting parts of the state machines without compromising the validity of the verification. Thus, the reduced SA model can be model checked with respect to the property of interest. The property of interest is expressed in a graphical formalism called PSC and it represents the slicing and abstraction criterion. The approach has been applied on a Selex Communications company case study in order to validate its efficacy.

On the future work side we plan to fully automatize the approach and to try to use it in other case studies. It is also planned to investigate other abstraction and slicing rules that could further reduce the system SA. For instance the abstraction rule R1, presented in Section 4.2, can be successfully instantiated for internal messages.

Acknowledgements

This work is partially supported by the PLASTIC project: Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication. Sixth Framework Programme. http://www.ist-plastic.org.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Massachusetts (1998)
2. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. In: SIGSOFT Software Engineering Notes. Volume 17. (1992) 40–52
3. Bernardo, M., Inverardi, P.: Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods. SFM-03:SA Lectures, LNCS 2804 (2003)

4. Compare, D., Inverardi, P., Pelliccione, P., Sebastiani, A.: Integrating model-checking architectural analysis and validation in a real software life-cycle. In: the 12th International Formal Methods Europe Symposium (FME 2003). number 2805 in LNCS, Springer (2003)
5. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison Wesley (2003)
6. Holzmann, G.J.: The logic of bugs. In: FSE 2002, Foundations of Software Engineering, Charleston, SC, USA (2002)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2001)
8. Caporuscio, M., Inverardi, P., Pelliccione, P.: Compositional verification of middleware-based software architecture descriptions. In: Proceedings of the International Conference on Software Engineering (ICSE 2004), Edimburgh (2004.)
9. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transaction on Computers **C-35**(8) (1986) 677–691
10. Katz, S., Peled, D.: An efficient verification method for parallel and distributed programs. In: Workshop on Linear Time, Branching Time and Partial Order Logics and Models for Concurrency. Volume 354 of LNCS., Springer (1988) 489–507
11. Frantz, F.K.: A taxonomy of model abstraction techniques. In: WSC '95: Proceedings of the 27th conference on Winter simulation, New York, NY, USA, ACM Press (1995) 1413–1420
12. Emerson, F.A., Sistla, A.P.: Symmetry and Model Checking. Formal Methods in System Design: An International Journal **9**(1/2) (1996) 105–131
13. Francez, N.: The Analysis of Cyclic Programs. PhD thesis, The Weizmann Institute of Science (1976)
14. Pnueli, A.: In transition from global to modular temporal reasoning about programs. Logics and Models of Concurrent Systems, sub-series F: Computer and System Science (1985) 123–144 Springer-Verlag.
15. Kim, T., Song, Y.T., Chung, L., Huynh, D.T.: Software architecture analysis: A dynamic slicing approach. Journal of Computer & Information Science (2) (2000) 91–103
16. Zhao, J.: Using dependence analysis to support software architecture understanding. in M. Li (Ed.), New Technologies on Computer Software, International Academic Publishers (1997) 135–142
17. Tip, F.: A survey of program slicing techniques. Journal of programming languages **3** (1995) 121–189
18. Zhao, J.: Applying slicing technique to software architectures. In: Proceedings of 4th IEEE International Conference on Engineering of Complex Computer Systems. (1998) 87–98
19. Stafford, J.A., Wolf, A.L.: Architecture-level dependence analysis for software systems. International Journal of Software Engineering and Knowledge Engineering **11**(4) (2001) 431–451
20. Pelliccione, P., Muccini, H., Bucchiarone, A., Facchini, F.: Deriving Test Sequences from Model-based Specifications. In: Proc. Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005). Lecture Notes in Computer Science, LNCS 3489, St. Louis, Missouri (USA) (2005) 267–282
21. Autili, M., Inverardi, P., Pelliccione, P.: A scenario based notation for specifying temporal properties. In: 5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06). (Shanghai, China, May 27, 2006.)
22. PSC home page: http://www.di.univaq.it/psc2ba: (2005)

23. Charmy Project: Charmy web site. http://www.di.univaq.it/charmy (2004)
24. Inverardi, P., Muccini, H., Pelliccione, P.: Charmy: an extensible tool for architectural analysis. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2005) 111–114
25. Weiser, M.: Program slicing. In: ICSE '81: Proceedings of the 5th international conference on Software engineering, Piscataway, NJ, USA, IEEE Press (1981) 439–449
26. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation. Volume 25., White Plains, NY (1990) 246–256
27. Korel, B., Laski, J.: Dynamic slicing of computer programs. J. Syst. Softw. **13**(3) (1990) 187–195
28. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc. (1992)
29. Buchi, R.: On a decision method in restricted second order arithmetic. In Press, S.U., ed.: Proc. of the Int. Congress of Logic, Methodology and Philosophy of Science. (1960) 1–11