

Automated Check of Architectural Models Consistency using SPIN

Paola Inverardi, Henry Muccini & Patrizio Pelliccione
Dipartimento di Informatica
Università dell'Aquila - Via Vetoio, 1
67100 L'Aquila, Italy
{inverard, muccini, pellicci}@univaq.it

Abstract

Architectural description techniques are a key element of the development of large systems. In recent years several formalisms and models have been introduced to represent the software architectural level of description. These models represent SA from different viewpoints, handling different aspects of the system separately and allowing for analysis and validation of the architectural choices, both behavioral and quantitative. Behavioral views are recognized to be one of the most attractive feature in the SA description and in practical contexts, state diagrams and scenarios are the most used tools to model this view.

Although very expressive this approach has two drawbacks with respect to analysis and validation: system specification incompleteness and view consistency. Our work can be put in this context with the aim to manage incompleteness and to check views conformance: we suppose to have state diagrams and scenarios models representing the system dynamics at the architectural level; they can be incomplete and we want to prove that they describe, from different viewpoints, the same system behavior. To reach this goal, we are using the SPIN model checker and we are implementing a tool to manage the translation of architectural models in Promela and LTL.

1. Introduction

Software developers are more and more often asked to produce increasingly complex software, with shorter time-to-market and better quality. Software Architectures are recognized to be a powerful tool in meeting these requirements.

Software Architectures (SAs) emerged in the '90 to structure complex software systems, exploiting commonalities in specific domains organizations (pipeline, client-server, pipe and filter) and providing an high-level system description. Nowadays SA is an autonomous discipline

with the originality of focusing on the overall organization of a large software system using *abstractions* to express the logical coordination structure of complex distributed systems. The emphasis in SA specification is in capturing the *system structure* (i.e., the architecture topology) by identifying architectural *components* and *connectors*. Initially, the SA descriptions mainly relied on informal box-and-line diagrams and was usually impossible to analyze an architectural description for consistency or to infer properties about it.

During the years, the informal architectural diagrams have been replaced/complemented by *formal* architectural descriptions. Tools and architectural languages have been introduced in order to make rigorous specification and analysis possible [5, 33, 1].

More recently, researchers in industry and academia have integrated the SA description in their software development processes, e.g. [29, 10]. Putting SA in practice, software architects have learnt some lessons: i) despite the high level of abstraction, notably in industrial contexts, the SA can be *too complex* to be described and managed; ii) SA production and management is, in general, an expensive task. Therefore the effort is justified if the SA artifacts are extensively used for multiple purposes. Typical use of SA is as a high level design blueprint of the system to be used during the system development and later on for maintenance and reuse. At the same time SA can be used in itself in order to analyze and validate architectural choices, both behavioral and quantitative. For this reasons architectural descriptions encompass not only static information but also dynamic (behavioral) aspects.

To tackle system complexity several viewpoints [19, 10, 24] have been introduced. The use of views allows to handle different aspects of the system separately and to manage the description of real systems. Several slightly different views (logical, module, deployment and so on) are defined in different approaches [19, 10, 2]; different notations, generally graphical, have been introduced for their representation.

All the approaches agree that *dynamic aspects* are or-

thogonal to all the views and they allow for analysis and validation of architectural choices. They are recognized to be one of the most attractive feature in the SA description and several works are using these dynamic descriptions for testing [3, 28], analysis and model checking [1, 4, 8, 18]. Many specification languages [12, 5, 1, 23] have been introduced to specify and model the *system behavior* at the architectural level but there is not a standardized way to represent SA dynamics. In the current industrial practice *State-based machines* and *scenarios* are the most common used tools to model behavioral aspects: state diagrams describe components behavior while scenarios (Message Sequence Charts or Sequence diagrams) identify how they interact. Although very expressive, this approach has two drawbacks with respect to analysis and validation.

The first one deals with system specification *model incompleteness*: notably in industrial contexts, these models are not completely specifiable or become practically intractable due to state explosion. Thus we can in general rely on a finite set of scenarios that complement (sub-)systems state models.

The second is a problem of *view consistency*: the use of several views allows to handle different aspects of the system separately but also facilitate the growing of inconsistency between views representing different aspects of the same system [25, 27]. Thus state diagrams and scenarios provide alternative views of the system: these views are not independent and can erroneously specify contradictory or inconsistent behaviors.

The approach we present in the paper addresses the two above problems. First we put ourselves in a practical industrial setting, thus assuming to deal with multiple views SA descriptions. Second we only deal with dynamic views of SA descriptions. In this setting we propose a framework in which it is possible to obtain, starting from the (incomplete) dynamic views to synthesize, through a suitable translation, an actual SA complete model in the SPIN model checker. Then we validate the obtained model with respect to a set of selected scenarios. This gives us a first feedback on the multiple views consistency side, since we are able to check whether the state and the scenarios views coherently integrate. Once the validation step has been performed we have a SA model at disposal that can be subsequently checked by using the SPIN capabilities with respect to a wide range of behavioral properties.

The paper is organized as follows. In the next section we summarize all the hypothesis our framework is based on. These are very general and constitute the assumptions we make on standard SA design processes and consider as input to our methodology. Section 3 details our approach describing how the input SA artifacts are suitable translated in the SPIN model checker [31] and the views consistency check is performed. Section 4 illustrates the architecture of

our framework, which is composed besides SPIN of other tools components, and presents its use on a couple of case studies. Section 5 compares our approach with a wide range of proposals existing in the literature outlining commonalities and differences. The last section summarizes our approach and presents future research goals.

2. Setting the context

Let us now define our assumptions on what an SA description should provide in order to integrate in our framework. Our general assumption is that the architectural description consists of components, connectors and channels: a component is a computational or storage unit, a connector is a particular kind of component, responsible for coordination and a channel is the way components and connectors interact. Components dynamics is expressed by state machines, connectors may be explicitly modeled by scenarios and a set of scenarios describe how components and connectors interacts. State diagrams and scenarios are defined using a notation general enough to encompass those generated by the architectural description languages used in the current practice and rich enough to allow for analysis. Thus, in the following sections, we will use the CCS notation to describe state diagrams and a stereotyped UML notation¹ to refer to scenarios. Figure 1 illustrates the notation we will use for state diagrams and scenario: the state diagrams (Fig. 1.a) are described using a Labeled Transition System (LTS) notation in which i) labels uniquely identifies the architectural communication channels, i.e., a channel is denoted by its label and can be used only by a pair of components; ii) for each element $l \in \mathcal{L}$, a $\bar{l} \in \bar{\mathcal{L}}$ may be defined, following the CCS rules. Notice that the assumption in i) above is completely general since it is always possible to uniquely label a communication between two components.

Scenarios (Fig. 1.b) are described using a UML notation in which i) each rectangular box represents a component, ii) each arrow defines a communication line between the left and the right boxes. There are several types of arrows, each one representing a particular kind of connection: the asynchronous (the half stick arrow), the synchronous (the filled solid arrow) and the generic (the stick arrow). The asynchronous communication can be represented either by an horizontal arrow (the first arrow in Fig. 1.b) when we only express that a message has been sent or by a slanted arrow² (labeled with “m4” in the figure) that models an asynchronous communication with an upper bound on the reception of the message. The second kind of communication is called *deferred synchronous*.

¹each rectangular box represents a component, each arrow defines a communication line between the left and the right boxes

²this notation is not so usual in UML scenarios but it is allowed, as described in [26].

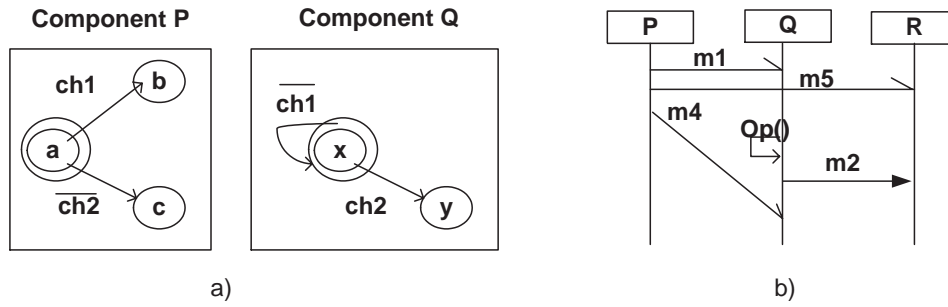


Figure 1. State diagrams and Sequence formalisms

The SPIN model checker [31], defines the model by using the Promela language [31] while the properties to be checked are defined by using Linear-time Temporal Logic (LTL) formulae [31]. Promela is a verification modeling language for modeling concurrent systems communicating via either messages or shared variables. It introduces a notation to model processes that can communicate via synchronous and asynchronous channels. The key elements used in the language are processes (proctypes), message channels and variables: processes are global objects while message channels and variables can be declared either globally or locally within a process.

In the following we will then assume to start from SA descriptions expressed in our notations. We want to stress again that the notations we are using are general enough to easy map into.

3. The approach

Our approach is composed of three steps: in the former, architectural models, describing the components behavior are translated into a Promela specification. This specification can be enriched by adding information on the communication type that can be obtained from the scenarios. In the second step scenarios are translated into LTL formulae. Finally the SPIN model-checker checks if the LTL formulae are verified on the obtained global model.

In the following subsections we will summarize the three steps, without giving too many details on how the translation algorithms work. The technical details may be found in [15].

3.1. Step one: generating the Promela model

In this step our goal is to generate a Promela specification of the architectural state diagrams. Architectural components are translated in Promela proctypes and the behaviors modeled by the components state diagrams are expressed through Promela statements. This kind of mapping, pre-

cisely described in [15], is similar to many others (as outlined in the related works) but its nature is mainly architectural. It means that, in our translation, we look at the architectural elements (components, connectors, channels, messages exchanged over the channels, and so on) and at the components state diagrams and translate them into Promela. Following this idea, the translation algorithm:

i) maps component state diagrams into proctypes. As shown in Figure 2 each proctype will model one state diagram behavior. Each state diagram state will be mapped into a label inside the <proctype description> field and how one state can reach the others is described in the <state description> field;

ii) extract connectors from scenarios and express these connectors by Promela proctypes. The idea is that the software architect can use some scenarios to explicitly identify connector behaviors or connectors may be synthesized analyzing the arrows type among different components in the architectural scenarios (synchronous, asynchronous, deferred). The first scenario in Figure 2 shows a connector modeling an asynchronous communication while the other shows a synchronous communication. The *Op()* item can be used to define the operation we would like to apply on the message during its delivery (encryption, decryption, compression, decompression);

iii) maps architectural channels into Input and Output channels. If component Q needs to send a message to P using the "ch1" channel and a synchronous communication (like in Figure 2, first connector), then Q sends the message to the ch1 Output channel, the synchronous connector receives the message, performs some operations on it, and sends it to the "ch1" Input channel, implementing the desired communication typology (synchronous in this example);

iv) defines variables (located in the "a" and "b" parts in Figure 2) used during the SPIN execution to store information on *when* send and receive operations are performed over the defined channels. These variables will be used in Subsection 3.3 to analyze if the Promela model behavior

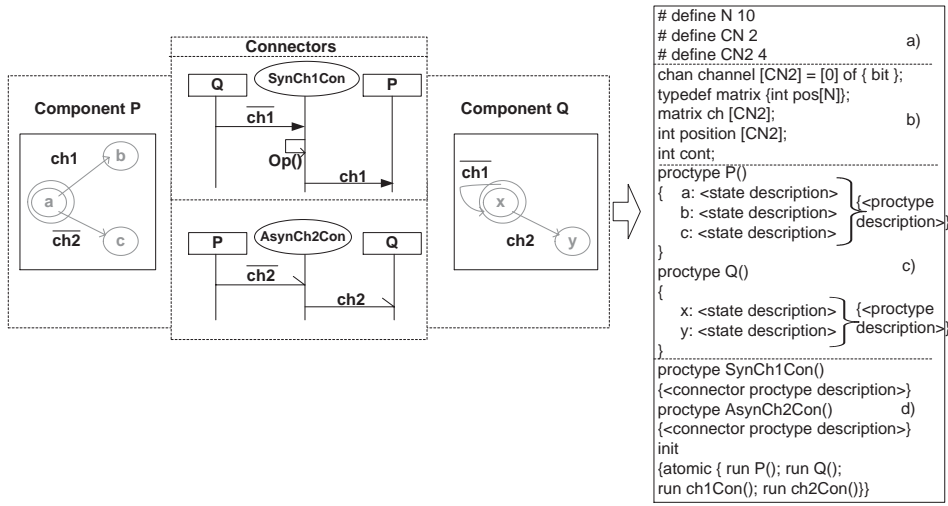


Figure 2. Translating state diagrams in Promela

conforms to the scenarios description.

The mapping will structure the resulting Promela specification reflecting the separation among components and connectors that we have in the architectural specification. Moreover, the connectors are defined in a *parameterized* way to enable the creation of a standard library of reusable connectors and to separate the computational units (the components) from the coordination one (the connectors). It means that, in our Promela specification, the component sends a message on the desired channel and does not have any idea on what components will receive it. The connectors manage the coordination among different components, receiving and forwarding messages in a parameterized way.

3.2. Step two

In this subsection we explain how scenarios are translated into LTL formulae. Scenarios define a temporal ordered sequence of events, i.e., they define the order in which messages are sent and received through channels. Given a selected scenario, the algorithm we propose analyzes the order in which arrows are defined in the scenario and the arrows type (synchronous, asynchronous or deferred) to identify when a send or a receive is performed over each channel.

Let's briefly describe an informal mapping between scenarios and temporal logic formulae by using the scenario in Figure 1. The notation " $Sm1 < Sm2$ " will be used to say that the send of message $m1$ happens before the send of message $m2$. Rm_i will be used to identify the reception of a message.

- Initially, P sends the message $m1$ to Q using an asynchronous connector. It means that $Sm1 < Rm1$ but we

cannot define precisely when $Rm1$ will be received. We suppose it will be received before the end of the scenario (END).

- P sends $m5$ to R in an asynchronous way. It means that $Sm1 < Sm5$ and $Sm1 < Rm1 < END$ and $Sm5 < Rm5 < END$;
- P sends $m4$ to Q using an asynchronous with bound communication. It stands for $Sm1 < Sm5 < Sm4$ and $Sm1 < Rm1 < END$ and $Sm5 < Rm5 < END$ and $Sm4 < Rm2 < Rm4$. The last rule identifies the notion of "deferred" communication.
- Q synchronously sends $m2$ to R. The informal temporal logic formula now becomes $Sm1 < Sm5 < Sm4 < Rm2$ and $Sm1 < Rm1 < END$ and $Sm5 < Rm5 < END$ and $Sm4 < Rm2 < Rm4$. In the synchronous communication we decided to evaluate only the reception of the message, considering send and receive as atomic operations.

How the informal formula defined in the last bullet is formalized in the LTL formalism will be defined in the next subsection.

3.3. Step three

The main goal of this approach is to check the conformance between scenarios and state diagrams. In other words, *we want to check if the temporal ordered sequence of events the scenarios describe are also exhibited by some paths in the complete model generated by SPIN.*

To gain this objective, we run the global model obtained from the Promela specification and model check its

<pre> [] (!(a && b && c && d && e && f)) #define a (ch[m1_S].pos[0]==1) #define b (ch[m5_S].pos[0]==2) #define c (ch[m4_S].pos[0]==3) #define d (ch[op_R].pos[0]==5) #define e (ch[m2_R].pos[0]==7) #define f (ch[m4_R].pos[0]==8) </pre> <p style="text-align: center;">Exists & Strict Check</p> <pre> [] ((a && b && c && d && e && f) (g)) #define a (ch[m1_S].pos[0]==1) #define b (ch[m5_S].pos[0]==2) #define c (ch[m4_S].pos[0]==3) #define d (ch[op_R].pos[0]==5) #define e (ch[m2_R].pos[0]==7) #define f (ch[m4_R].pos[0]==8) #define g (ch[m4_R].pos[0]==0) </pre> <p style="text-align: center;">All & Strict Check</p>	<pre> [] (!(a && b && c && d && e)) #define a (ch[m1_S].pos[0] < ch[m5_S].pos[0]) #define b (ch[m5_S].pos[0] < ch[m4_S].pos[0]) #define c (ch[m4_S].pos[0] < ch[op_R].pos[0]) #define d (ch[op_R].pos[0] < ch[m2_R].pos[0]) #define e (ch[m2_R].pos[0] < ch[m4_R].pos[0]) </pre> <p style="text-align: center;">Exists & Loose Check</p> <pre> [] ((a && b && c && d && e) (f)) #define a (ch[m1_S].pos[0] < ch[m5_S].pos[0]) #define b (ch[m5_S].pos[0] < ch[m4_S].pos[0]) #define c (ch[m4_S].pos[0] < ch[op_R].pos[0]) #define d (ch[op_R].pos[0] < ch[m2_R].pos[0]) #define e (ch[m2_R].pos[0] < ch[m4_R].pos[0]) #define f (ch[m4_R].pos[0] == 0) </pre> <p style="text-align: center;">All & Loose Check</p>
--	--

Figure 3. Writing LTL formulae

execution with respect to the LTL formulae. When the Promela specification runs, the variables we described in our statediagram-to-Promela algorithm store information on *when* send and receive operations are performed over the defined channels. These information is compared (by the model checker) with the defined LTL formulae and behavioral errors trace are expressed by scenarios.

Now there can be several ways of mapping the informal temporal formula obtained in the previous subsection into a LTL formula. Each one has slightly different semantics as detailed below.

We can run the model check operation with some options, each one corresponding to a different LTL formula:

i) we can check if there *exists* at least an architectural behavior that conforms to our informal formula or if *all* the architectural paths conform to the selected scenarios. This information is embedded in the final LTL formula.

ii) we can check the formula imposing that the only actions allowed are those specified by the scenario (*strict check*) or *other actions are allowed* (loose check) as long as the fixed temporal ordering is respected. In the strict check, the formula is written imposing that $Sm1 < Sm2$ and $Sm1$ is the first operation and $Sm2$ is the second. In the loose check, we only impose that $Sm1 < Sm2$ that implies that something may happen before or after $Sm1$.

This types of checks are embedded in the way the LTL formula is written. Different check options result in different analysis we are interested in. For example, the *exists & loose* options check that at least one state machine path “loosely” behaves as the selected scenario while the *all & strict* options check if all the Promela model paths “strictly” behaves as the scenario. To verify that a scenario is true in

all the state diagram paths (*all & strict* option) is usually not an interesting job; more informative is the *exists & loose* option. The *all & loose* option could be used to prove that the system conforms to the scenario but there are some slight differences, captured by the loose option.

In Figure 3 different implementations of the informal formula proposed in the previous section is shown. Notice that the “strict” and “loose” options could be combined, generating a more complex formula.

4. The Charmy Tool: an easy way to practice the approach

To support our approach we are developing a tool, called Charmy (CHecking ARchitecture Models consistency), to help the software architect to perform the consistency checking of its architectural models. It enables the software engineer to draw architectural state diagrams and scenarios (using the ScenariosEditor and StatechartsEditor components) and to automatically translate these models in Promela and LTL. A prototype version is now available and is mainly composed of three components:

- a *graphical editor*, to draw the SA topology, the components state diagrams and the scenarios. It is composed of the SAEditor&ActionManager, the ScenariosEditor and the StatechartsEditor sub-components;
- a *translator utility*, to convert enriched state diagrams and scenarios in Promela statements and LTL formulae respectively. This component is composed of the GeneratePromela and the Generate LTL sub-components;

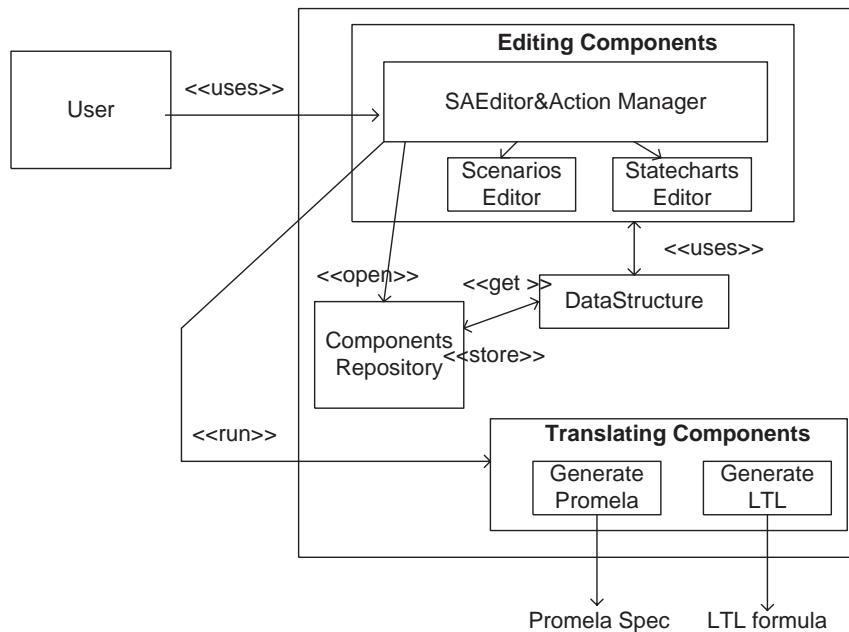


Figure 4. The Charmy SA topology

- a *repository* in which models may be stored to be reused. The “DataStructure” component temporarily stores information into internal data structures.

The Charmy tool architecture is shown in Figure 4.

The system requirements we imposed may be summarized as follows: i) extensibility: new model elements may be added without modifying the tool architecture or design. We have chosen a state machine formalism as presented in Section 2 for our convenience, but we do not want to be constrained by this choice. In future work we will extend the tool to enable users to use stereotyped UML diagrams to represent also state diagrams at the architectural level or Message Sequence Charts formalisms for scenarios; ii) new translation algorithms may be added in a easy way to translate our models in different languages. For example, it could be interesting to integrate the algorithms proposed in [22] in our tool to check the consistency of UML models. iii) The user has to be able to easily select the options about the LTL formulae generation. To implement this requirement we added a button to the translation utility to generate the LTL formulae conforming to the identified options.

To experiment the approach in practice and to test the tool we analyzed the software architecture models of two different systems: the Teleservices and Remote Medical Care System (TRMCS) and the Charmy tool SA itself.

The TRMCS [13, 14] case study provides monitoring and assistance to disabled or elderly people and the typical service is to send relevant information to a local phone-center. The TRMCS SA consists of multiple User and

Router components and a Server component. The SA dynamics may be informally defined as follows: the User sends Alarm or Check messages to its Router component; the Router waits for Alarms and Checks, forwards Alarms to the Server and internally evaluates the Check msgs; finally, the Server processes the Alarm messages and sends back an ack to the User, through the Router³.

In [16] we applied our approach to this case study. We first identified some architectural level scenarios representing some behaviors of interest. We obtained the Promela generated specification from the SA description, composed of state diagrams for the architectural components and scenarios to identify components interactions, then we generated some LTL formulae corresponding to scenarios. In the first scenario, we wanted to check if the same User can send two successive Alarms before receiving any acknowledgment. SPIN proves that the state diagram model does not allow this behavior. Scenario2 aimed at checking whether two different Users can send Alarms before receiving any ack, and it proved to be true. Finally, the third scenario asks to SPIN if a specific component, CheckCoordinator, may receive two consecutive checks from different Users. This scenario allows to put in evidence an architectural model error, because the CheckCoordinator component, as it was modeled, cannot receive more than one check if it does not forwards the first one to the Router component.

In Figure 5 the Charmy graphical user interface is shown. The components behavior is described as follows:

³More information may be found in [16].

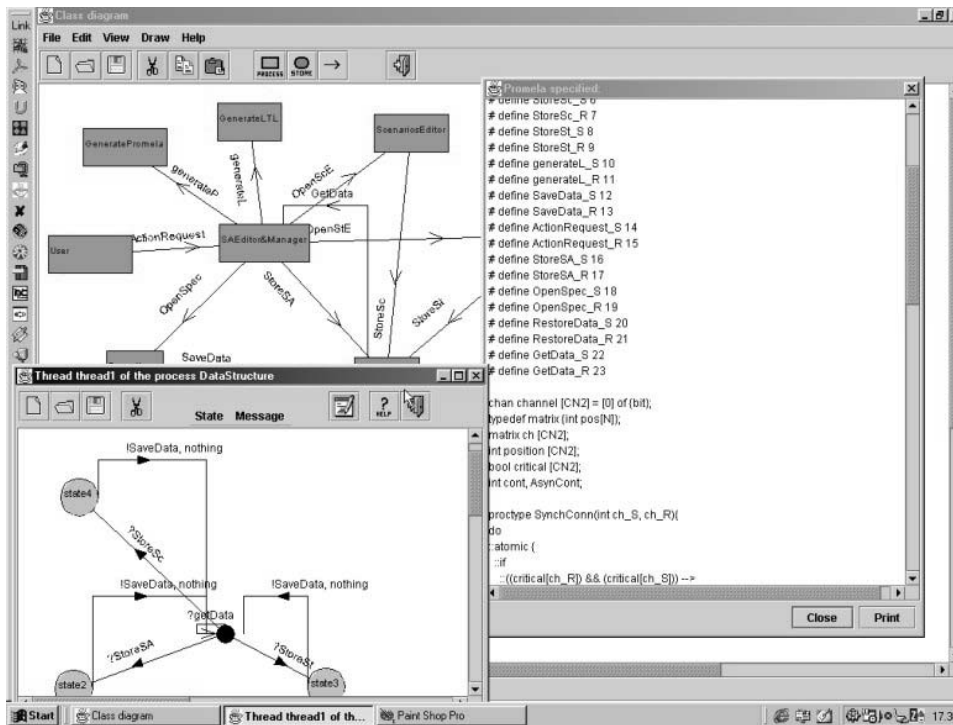


Figure 5. The Charmy SA graphical interface

- the *User* makes a request to the *SAEditor&ActionManager* component. He can use this component to "Draw the SA topology", to "Edit Scenarios", to "Edit State diagrams" or to "Open" an existing SA description;
- the *SAEditor&ActionManager* component has two main activities: it is an editor for SA components and connectors and it gives the *User* the opportunity to open other editors or to open an existing SA description. The *ScenariosEditor* enables the *User* to draw scenarios while the *StatechartsEditor* may be used to draw architectural state diagrams;
- when an SA topology or a scenario or a statechart is drawn, the data structure of these diagrams are temporarily stored into the *DataStructure* while the *ComponentsRepository* component allows to permanently save the diagrams, to be reused afterwards;
- finally, the *GeneratePromela* and *GenerateLTL* components apply the defined algorithms to the state diagrams and scenarios diagrams to build the Promela and the LTL specifications.

Although this software architecture could seem not very interesting, it is to be noted that there are some interesting architectural choices and constraints: the *SAEditor&ActionManager*

may be enabled to open the *ScenariosEditor* and the *StatechartsEditor* whenever the *User* wants or only if some architectural components have been already drawn. The editing components may concurrently or sequentially be run; in the former case, they can concurrently access the *DataStructure* component and inconsistency problems may arise. When the *SAEditor&ActionManager* is called to translate the specifications, it recalls the information from the *DataStructure* and sends it to the *TranslatingComponents* component. An alternative choice should propose that the *SAEditor&ActionManager* calls the *TranslatingComponents* modules that load information from the *DataStructure* component.

For each component, a statechart model has been identified. Scenarios have been extracted from the requirements, someone putting in evidence the architectural constraints already presented. Using the tool we draw the architectural state diagrams and the scenarios generating the Promela and LTL specifications, as partially shown in Figure 5. The Promela model has been checked with respect to the scenarios shown in Figure 6 with the following results:

Scenario a): the *User* sends the open *ScenariosEditor* request to the *SAEditor&ActionManager*, it forwards the message to the *ScenariosEditor*. It opens the editor and saves the information into the *DataStructure*. The *User* sends the open *StatechartsEditor* request to the *SAEditor&ActionManager*. The SPIN model-checker proves that

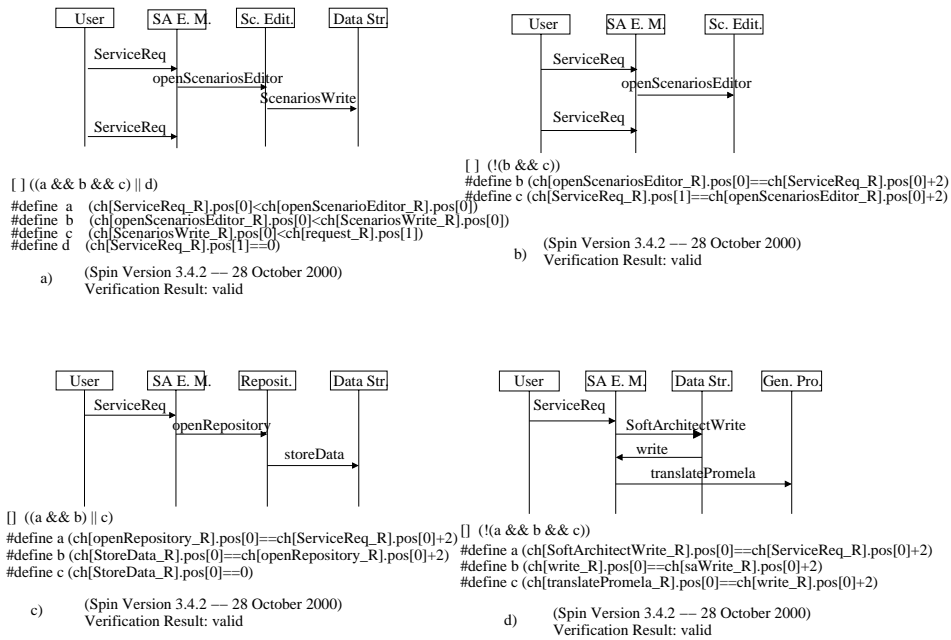


Figure 6. Some scenarios model check

it is a valid behavior. It means that the ScenariosEditor and Statechartseditors may be run before any architectural component has been drawn (and stored) using the SAEditor&Manager.

Scenario b): this scenario is similar to the previous one but we ask to SPIN if it is possible to open another editor (the StatechartsEditor in this example) before the ScenariosEditor has stored information into the DataStructure component. SPIN proves that our model does not allow this behavior. Notice that the model-checking result, shown in Figure 6.b, is “valid” because we preferred to check the formula negation.

Scenario c): this scenario shows how the open Repository request may be served by the system. The User asks to open a stored specification, the SAEditor&ActionManager asks to the Repository to copy its information into the DataStructure component. SPIN shows that this scenario is correctly implemented by the statechart model.

Scenario d): how the translator utilities receive data for the translation? The possible solutions are that i) the DataStructure sends information back to the SAEditor&Manager that forwards this information to the translators or ii) the DataStructure sends information directly to the translator utilities. The scenario depicted in Figure 6.d represents the solution i). SPIN proves that this behavior is not implemented in the statecharts and put in evidence an architectural error: the Charmy SA topology, in fact, does not allow a direct communication between the DataStructure and the TranslatingComponents components. It means that the SA topology is not conform to the SA dynamics.

5. Related works

The general themes of system (in)completeness and (in)consistency have received a lot of attention in the last few years, showing the growing interest on this area, see e.g. [32, 30].

As outlined in [6] consistency checking among multi-perspective or multi-language specifications is not a young field of research. There are different papers proposing different approaches to handle consistency. Some of these are used for checking model consistency:

- In [7] the authors propose an approach to design a method for multiple view consistency checking. The SA specification is defined in terms of a collection of diagrams each one representing a software view without a mathematical basis. To check the view consistency, they provide a formal definition of the views and their relationships, they define their semantics as sets of graphs and apply an algorithm to check the consistency of diagrams.
- in [6] the authors present how inconsistencies may be handled using the *ViewPoints* framework and a logic-based consistency handling. Their approach is based on the assumption that to check specifications consistency among view points is equivalent to an inconsistency handling in distributed logical databases: views specified in the ViewPoints framework are translated into classical logic formulae and stored in logi-

cal databases checked for consistency using temporal logic. .

Comparing these approaches with our own we can say that there are some commonalities and some relevant differences: all the approaches translate models in other formalisms (Promela or LTL in our case, graphs in [7], logical databases in [6]) and compare them using some strategy (Buchi automata, disequation systems on graphs [7] or temporal logic properties among databases [6]). In [7] the focus is to check views structural properties while our is mainly devoted to check dynamic properties, like communication conformance among models. In [6] the proposed approach reveals and handle the inconsistency while we just reveal them, identifying the unexpected behavior. We adapt existing tools (SPIN) while they define "ad hoc" techniques.

In [11] the Hugo tool is presented. It has been designed to automatically verify whether the interactions expressed by a collaboration diagram can indeed be realized by a set of state machines. State machines are compiled into Promela, collaborations are translated into set of Buchi automata (LTL formulae) and SPIN is called upon to verify the model against the automata. This approach is really close to our, but the level of abstraction they works (design level) is different from our (architectural level) and their interest is mainly related to UML diagrams.

Another class of works that can be related to us is based on the idea of translating models in Promela or LTL and using SPIN to model check them. In [22] the authors translate UML State Diagrams in Promela to validate the model correctness with respect to the standard SPIN model checking (deadlock, constraint violation, livelock and so on). In [9, 20] a subset of UML State diagrams is translated into an intermediate representation of Hierarchical Automata and in [9] this model is translated into a semantic automata (that can be represented as an FC2 object) and is model-checked using the JACK verification environment. These approaches differ from us because they do not translate Scenarios, do not use LTL formulae and do not use this approach for consistency checking among different models.

In [21] the authors translate Message Sequence Charts (MSCs) and High Level MSCs to Promela; the model synthesized from the MSCs may be simulated and animated and liveness properties, under-specified in the MSC model, may be specified in LTL. With this approach they prove properties on the synthesized model but they do not check consistency with respect to other models.

In [34] the authors present an algorithm that synthesize a behavioral model describing the closest possible implementation for a specification based on MSCs. They describe a technique to detect, on the synthesized model, unexpected or undesirable scenarios (the so called implied scenarios). This approach may be considered complementary to our: they make the assumption that the synthesized state dia-

grams describe at least the MSCs scenarios information and detect the state diagrams behaviors not described by the MSCs. In our case, instead, the state machine is obtained independently from the scenarios and we check the model to verify if all the scenarios behaviors are contemplated in the state machine.

Finally, in [18] the authors detect UML state diagrams deadlock. They enrich this model to represent communication's type between two nodes (synchronous, asynchronous), they translate this model in the FSP language and use the LTSA tool to detect deadlock. LTSA property violation traces are expressed through UML Sequence diagrams. Common to our approach is the need to enrich the statechart and the detection of error traces.

6. Conclusions and future work

In this paper we presented an approach that starting from a SA incomplete model, represented through components state machines and scenarios possibly describing connectors, allows a SA complete Promela model to be obtained. Then architectural scenarios are selected and translated to LTL formulae in order to validate the consistency between these two dynamic architectural views. We also described the tool we have developed to help the software architect to apply the approach in practice. We have briefly shown the use of the tool by means of two real case systems. We are now applying the approach in a project dealing with the Java two Enterprise Edition (J2EE) architecture. In this project we are using the approach to prove that two architectural models behaves in the same way, with respect to selected scenarios. Moreover, the scenarios represent system properties, not architectural scenarios, as in [17]. It means that our approach can be not only used to show the inconsistency between state diagrams and scenarios but also to demonstrate some kind of "behavioral equivalence" between different architectural models. The problem in this case is that the two architectural models are not equivalent in general but can be shown to behave equivalently under selected scenarios and modulo a suitable abstraction.

Some ideas for future work are to consider enriched statecharts and scenarios to prove that the architectural model correctly behaves with respect to the expected behavior and that quantitative or temporal requirements are met by the specification and to prove UML statecharts and Sequence diagrams conformance, extending our algorithms.

As far as the tool is concerned, there is an interesting list of future works to consider:

- extend our tool to cope with UML State diagrams and Sequence Diagrams or Message Sequence Charts formalisms. Extending our drawing capabilities we would apply some other translation algorithms (e.g.,

starting from those described in [22]) to check the consistency of UML models.

- deal with automata explicitly is unpractical. Thus we will integrate the tool in such a way to input automata automatically from the architectural specification languages or already defined in some well known formalisms (e.g., FC2 or CADP formalisms). Inputs (and Outputs) should be also read (produced) in a XML format.
- The approach proposed in [34] and already summarized in Section 5 could be integrated with our. We check if scenarios are correctly modeled by state diagrams, but we do not check if state diagrams provide behaviors not described in the scenarios.
- A distributed version of the tool should be implemented, to enable several software architects to concurrently build and check the model.

Acknowledgments

The authors would like to acknowledge the Italian M.U.R.S.T. national project SALADIN that partly supported this work and Leonardo Quaglia that developed the Charmy tool.

References

- [1] Allen, R., and Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
- [2] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*. SEI Series, Addison-Wesley 1998.
- [3] Bertolino, A., Inverardi, P., and Muccini, H. An Explorative Journey from Architectural Tests Definition down to Code Tests Execution. On ACM Proceedings, ICSE2001, June 2001.
- [4] Compare, D., Inverardi, P., and Wolf, A.L. Uncovering Architectural Mismatch in Dynamic Behavior. *Science of Computer Programming*, 33(2):101–131, February 1999.
- [5] Darwin, an Architectural Description Language. Web site: <http://www.dse.doc.ic.ac.uk/research/darwin/darwin.html>.
- [6] Filkenstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. Inconsistency handling in multi-perspective specifications. In: *IEEE Trans. on Software Engineering*, Vol. 20, Number 8, pages 569–578, August 1994.
- [7] Fradet, P., Le Metayer, D., and Perin, M.: Consistency Checking for Multiple View Software Architectures. In: *Proc. European Software Engineering Conference (ESEC/FSE'99)*, pages 410–428, Toulouse, 1999.
- [8] Giannakopoulou, D., Kramer, J., and Cheung, S.C. Analysing the behaviour of distributed systems using tracta. *Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1):7–35, January 1999.
- [9] Gnesi, S., Latella, D., and Massink, M. Model Checking UML Statecharts Diagrams using JACK. in *Proc. Fourth IEEE International Symposium on High Assurance Systems Engineering*, IEEE Press, 1999.
- [10] Hofmeister, C., Nord, R. L., and Soni, D. *Applied Software Architecture*. Addison Wesley, 1999.
- [11] Schafer, T., Knapp, A., and Merz, S. Model Checking UML State Machines and Collaborations. On the Workshop on Software Model Checking, Paris, July 23 2001. To appear in ENTCS, volume 55 number 3.
- [12] Inverardi, P., and Wolf, A.L. Formal Specifications and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):100–114, April 1995.
- [13] Inverardi, P., and Muccini H. The Teleservices and Remote Medical Care System (TRMCS) Case study of the IWSSD-10 workshop, San Diego, California, November 2000.
- [14] Inverardi, P., and Muccini, H. Coordination models and Software Architectures in a Unified Software Development Process. In *Proc. 3rd Int'l Conf. on Coordination Languages and Models, LCNS 1906*, Springer Verlag, September 2000.
- [15] Inverardi, P., Muccini, H., and Pelliccione, P. Checking consistency between architectural models using SPIN. *IR 02/01, University of L'Aquila*, on-line at: <http://www.dm.univaq.it/~muccini/Page2.html>.
- [16] Inverardi, P., Muccini, H., and Pelliccione, P. Applying the Approach to the TRMCS case-study. On-line at: <http://www.dm.univaq.it/~muccini/Page2.html>.
- [17] Inverardi, P., Muccini, H., and Pelliccione, P. Checking consistency between architectural models using SPIN. On the *First Int. Workshop "From Software Requirements to Architectures"* (STRAW'01), Toronto, Canada, May 2001.

- [18] Kaveh, N., and Emmerich, W. Deadlock Detection in Distributed Object Systems. In Proc. of the Joint 7th European Software Engineering Conference and 9th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, Vienna, Austria. LNCS, Springer Verlag. 2001.
- [19] Kruchten, P.: Architectural Blueprints - The "4+1" View Model of Software Architecture. IEEE Software, 12(6) November 1995, pp. 42-50.
- [20] Latella, D., Majzik, I., and Massink, M. Automatic Verification of a behavioural subset of UML Statecharts Diagrams using the SPIN model-checker. in Formal Aspects of Computing 11(6): 637-664 (1999).
- [21] Leue, S., and Ladkin, P. B. Implementing and Verifying MSC Specifications Using PROMELA/XSPIN. in J-C. Gregoire, G. Holtzmann and D. Peled, Proceedings of the DIMACS Workshop, SPIN96.
- [22] Lilius, J., and Paltor, I.P. Formalising UML State Machines for Model Checking. In: UML'99 - The Unified Modeling Language. Beyond the Standard. Fort Collins, October 1999.
- [23] Magee, J., and Kramer, J. *Concurrency: State models & java programs*. Wiley publisher, April 1999.
- [24] Magee, J., Kramer, J., and Giannakopoulou, D. Behaviour Analysis of Software Architectures. In Proc. First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, Texas, 1999.
- [25] Pons, C., Giandini, R., and Baum, G. Dependency Relations Between Models in the Unified Process. in Proc. IWSSD 2000, November 2000.
- [26] Rational Corporation. Uml Resource Center. UML documentation, version 1.3, On-line at: <<http://www.rational.com/uml/index.jtmpl>>.
- [27] Reggio, G., Cerioli, M., and Astesiano, E. Towards a Rigorous Semantics of UML Supporting its Multi-view Approach. In Proc. FASE 2001, LNCS n. 2029, Berlin, Springer Verlag, 2001.
- [28] Richardson, D.J., and Wolf, A.L. Software Testing at the Architectural Level, Second International Software Architecture Workshop (ISAW-2), October 1996.
- [29] Robbins, J.E., Medvidovic, N., Redmiles, D.F., and Rosenblum, D.S. Integrating Architecture Description Languages with a Standard Design Method, *Proc. 20th Int'l Conf. on Software Engineering* Apr. 1998, pp. 209-218.
- [30] Second International Workshop on Living with Inconsistency. May 13, 2001 Toronto, Canada (Part of ICSE'01)
- [31] SPIN home page. On-line at: <<http://cm.bell-labs.com/cm/cs/what/spin/index.html>>
- [32] Tenth International Workshop on Software Specification and Design. Track on (In)completeness, (in)consistency and (sufficient) correctness. November 2000, San Diego, California.
- [33] The C2 Architectural Style On-line at <<http://www.ics.uci.edu/pub/arch/c2.html>>
- [34] Uchitel, S., Kramer, J., and Magee, J. Detecting Implied Scenarios in Message Sequence Chart Specifications. in Proc. European Software Engineering Conference (ESEC/FSE'01), Vienna 2001.