2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems

Formal Verification of Distributed Transaction Management in a SOA Based Control System

Ivana Popovic, Vladislav Vrtunski
Telvent DMS DOO Novi Sad,
Novi Sad, Serbia,
ivana.popovic@telventdms.com
vladislav.vrtunski@telventdms.com

Miroslav Popovic
Faculty of Technical Sciences,
University of Novi Sad,
Novi Sad, Serbia
miroslav.popovic@rt-rk.com

Abstract — **In large scale, heavy workload systems, managing distributed transactions on multiple datasets becomes challenging and error prone task. Software systems based on service oriented architecture principles that manage critical infrastructures are typical environments where robust transaction management is one of the essential goals to achieve. The aim of this paper is to provide a formal description of the solution for transaction management and individual service component behavior in a SOA-based control system, and prove the correctness of the proposed design with the SMV formal verification tool. Atomic commitment protocol is used as a basis for solving distributed transaction management problem. SMV language and verification tool are utilized for formal description of the problem and verification of the necessary properties. The case study describes an application of the proposed approach in commercial software system for electrical power distribution management. Verification of given model properties has shown that suggested solution is suitable for the described class of SOA-based systems.**

Keywords – distributed transactions, model checking, service oriented architecture, two phase commit protocol

I. Introduction

There is a variety of challenges to be dealt with when designing software solutions for management of critical infrastructures, such as oil, gas and electricity distribution systems. As a primary task, these systems have to provide continuous, near real-time data monitoring, processing and control, while fulfilling highest availability, reliability and security standards. On the other hand, numerous economic, serviceability, and maintainability aspects regarding different operational, day to day activities, must be addressed as well. When combined together, all these requirements are typically satisfied by a suite of complex business processes and workflows, thus guiding the system designer to the set of necessary data and functionalities that need to be simultaneously served by the components of the system.

One of the first steps to take when designing enterprise system architecture is to recognize data architecture aspects. When defining data architecture for SOA based system, it is of the utmost importance to take distributed transactional mechanisms into consideration at the early design stages. Motivation to do so comes from the necessity to have complex dependencies in the system, which may impose numerous constraints on transactional system behavior, even on the design of service components themselves. Another important aspect to be taken into account is the need for the system change and evolvement during the time, by adding new and modifying existing service components. This is especially important for commercial products that need to be designed to easily adjust to new requirements of many different customers, while satisfying system scalability requirements that impose additional layer of design complexity. The logical consequence of all of this is the need to define architectural guidelines for individual component behavior, as well as rules and policies for transactional behavior. Failure to do any of those two may result in serious design mistakes, which are usually the most expensive ones to resolve, especially if discovered late in, or even after, the implementation phase.

The work presented in this paper illustrates how three different aspects are combined to solve distributed transaction management (DTM) problem in the target environment. As a first step, an approach for DTM based on atomic commitment protocol is tailored to satisfy specific requirements of the described class of the systems. After that, individual service behavior policies are tackled as another very important but often omitted aspect in the context of transactions in SOA systems. And finally, proposed solutions are modeled and verified by using automated model checker tool.

The content of this paper is organized as follows. The related work and description of target class of software systems that are addressed by the proposed solution are presented in the subsections A. and B., respectively. Proposed methodology for DTM and individual service behavior policies are described in Section 2. Description of model verification aspects is given is Section 3, and the case study is presented in Section 4. Conclusions and further research directions are given in Section 5.

A. *Related work*

Management of distributed transactions in heterogeneous environments is well known problem which has been around for a long time. It has been addressed in many researches and commercial case studies. For example, in early 1990's authors of [5] have proposed a distributed transaction management scheme and manual verification process to validate the required properties of their solution. It is obvious that for a today's large scale commercial software system, automated model checker tools make verification, change and reevaluation of existing formal models much more efficient.

Transactional behavior is the essence of many modern Web based SOA systems, such as those for e-banking, supply chain management, web shopping, and other. In [3], general aspects for coordination of services and transactional processing concepts for Web Service Systems environments are addressed. Another approach for event-driven verification of transactional behavior in Web Service systems based on Event Calculus formalism is shown in [4]. Transactional behavior defined in the scope of this paper is similar to the one presented in [11]. In this paper, Web Service Atomic Transaction (WS-AT) protocol is presented, modeled and verified using TLA+ formalism. Standard Two Phase Commit (2PC) protocol is expanded with the registration procedure, and support for volatile and durable participants. Our approach is similar, but in addition to transactional behavior we are describing individual service behavior policies, and we use SMV formalism.

Very important topic is discussed in [14], where differences between the SOA aspects in the context of Web services and those in industrial control systems are addressed. As explained, lower level control systems typically contain legacy backend components to deal with time critical control tasks, and SOA characteristics are different in that case. Another solution for DTM described in [2] also addresses the environments which are in nature more heterogeneous. A concept is proposed for distributed transaction processing within the middleware layer, for enterprise systems, which employ both object-oriented middleware (OOM) and message oriented middleware (MOM). Scope of the research presented in this paper is limited to class of systems described in the next subsection; however, the need to extend transaction scope to some external, OOM or MOM based system may arise in the future. In that case, middleware based DTM solution like Dependency Spheres described in [2] may be an interesting option for expanding and upgrading our DTM solution.

For performance improvement purposes, industrial system presented in the case study is architected to achieve high level of parallelism, as described in [15, 16, 17] by some of the authors of this paper. In such environments, errors in the design of DTM can have very serious implications. Formal verification and model checking techniques have been successfully applied in our previous work described in [18, 19], which led to the research described in this paper.

B. *Target environment*

Typically, physical infrastructure of an industrial control system consists of specific purpose equipment and smart devices, such as Intelligent Electronic Devices (IEDs) and Remote Terminal Units (RTUs), which are used for monitoring and control of an industrial process. Today, almost every large scale industrial system uses Supervisory Control and Data Acquisition (SCADA) system as a front-end for communication with different types of control devices.

Focus of the research in this paper is separate and independent environment which can be laid on top of any SCADA system. This layer, called Intelligent Control System (ICS), encompasses necessary control logic and process related intelligence, which can be very complex. It provides adequate user interfaces for interaction with system engineers and other qualified personnel, and implements needed business processes and workflows. Another important aspect of such a system is a capability to integrate with other enterprise systems, such as Geographic Information Systems (GIS), which provides the coordinates of the equipment and objects. Being an autonomous system, ICS has to be built in a way to allow standards based integration with other systems, and this can be done by exposing adequate Web services and standards based messages.

The structure of ICS is shown in Fig. 1. Seen from the SOA standpoint, system exposes master service that holds the logically consistent dataset which describes existing elements of the system infrastructure and its properties. Besides the Master Data Service, other services used for enterprise integrations may be exposed, depending on ICS's role and purpose..

Master Data Service communicates with other service components (business objects). Each of the service components in ICS is responsible to manage exactly one aspect of the overall system functionality, such as dynamic data, performing necessary calculations, providing graphical representation of infrastructure elements, and other. For example, there may be a dataset which describes calculation model of the system. This dataset has to be correct, e.g. if ICS manages water management system it has to satisfy hydrodynamics laws. If ICS is a power distribution system, at minimum it has to satisfy Kirchhoff's laws.

From architectural standpoint, every individual service component in the system can be seen as a set of reader and writer threads, which operate on the underlying dataset. In the scope of this paper, a *dataset* is a general term used to describe any collection of data or a data model, which can either be persisted on disk or cached in memory. Interaction between the system user and the services is provided through thin client application. Usually, there is no demanding data processing inside the client application itself, its only responsibility is to obtain data from particular service in the system periodically, or on user demand.

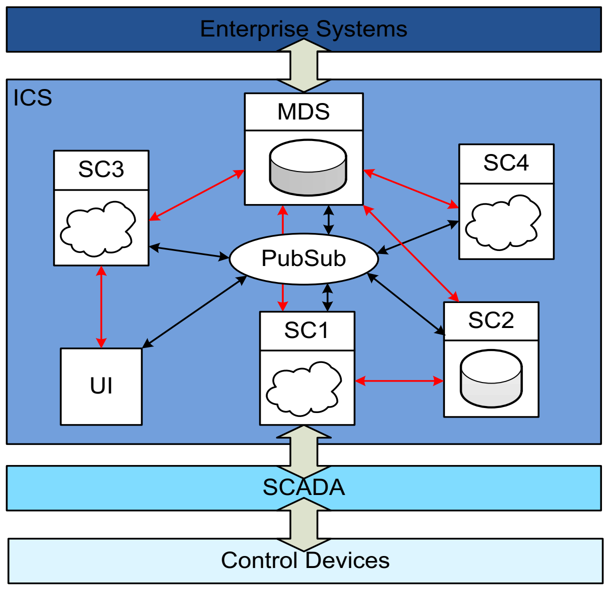

Figure 1. Target Environment

When the system evolvement aspects are taken into the consideration (i.e. the need to add new services and service components), one of the most important design goals is to provide the plug-and-play like integration capabilities. In order to provide loose coupling between service components and services, main internal communication backbone is designed in accordance with the publisher/subscriber paradigm. For synchronous, point-to-point calls (in Fig. 1 represented with red arrows), interfaces are provided to allow data access for the internal clients (other services and service components) and the external UI client, but the communication within the system is predominantly asynchronous, based on publishing and subscribing on different message topics (in Fig. 1 represented with black arrows). Important aspect regarding the communication in the system is the fact that all the datasets describe the current infrastructural state of the system (in the further text referred to as master dataset), which changes with the system during the time.

II. Distributed Transaction considerations

Master dataset changes (i.e. updates of the system) may be done automatically, through import process from some external system, or manually, through UI client application, by the system user. In both cases, errors are possible and the system must protect itself from invalid updates, which may severely jeopardize system stability. In the worst case scenarios, invalid changes may cause catastrophic, even life threatening consequences. For providing consistent and correct representation of the system state at any time, every change of master dataset must be propagated throughout the whole system as a transaction, because it is necessary to allow each service to apply its validation rules on the given change, and accept or reject update of its dataset. Master dataset is changed whenever new infrastructural elements are added, removed or significantly modified in the system. Significant modification of the element means that either: (1) new graphical data, calculation results, or other data is now needed to describe the infrastructural element state completely, or (2) part of the data is no longer needed for the particular modified infrastructural element.

Depending on systems nature and many other factors, frequency of dataset updates may vary from once a year, month or week, up to dozens times a day. Since the thin UI client application only serves as the user front end, and does not perform any significant and semantically important work, it does not participate in the dataset update transaction. Instead of that, when the update transaction on the server side is completed, and the system is updated to the new version of master dataset, the client application gets the update notification and simply requests changed datasets (blocks) from the services. During the update process, user experience of system engineers and other employees that concurrently work with the system must be the best possible. Ideally, users that are not working on the part of the system affected by dataset update should not experience any difficulties. Similarly as the UI client, the SCADA system also does not participate in the update transaction, but all dynamic data changes that are coming from it should be processed in a proper way, regardless of update transaction.

The aims of the work presented in this paper are divided in three stages:

1. provide general DTM policy for the target system,
2. provide general guidelines for transaction behavior of individual services in the system, and
3. verify the proposed solution by using formal model checking tool, in particular by verification of ACID transactional properties

A. *Distributed Transaction Management*

As basis for the proposed solution we used standard approach for DTM, atomic commitment protocol. Two Phase Commit (2PC) protocol [6] is simple and elegant atomic commitment protocol which treats occurrence of any failure in the system during the transaction as a global transaction failure. Initiation and orchestration of distributed transaction is typically performed by the component called Transaction Coordinator (TC). Traditionally, two types of distributed transaction execution models are used: sequential and parallel [7].

As shown in Fig. 2.a and 2.b, the proposed transactional model uses hybrid approach to take advantage of performance gain obtained by parallel execution, by ensuring that initial, critical part of transaction is done

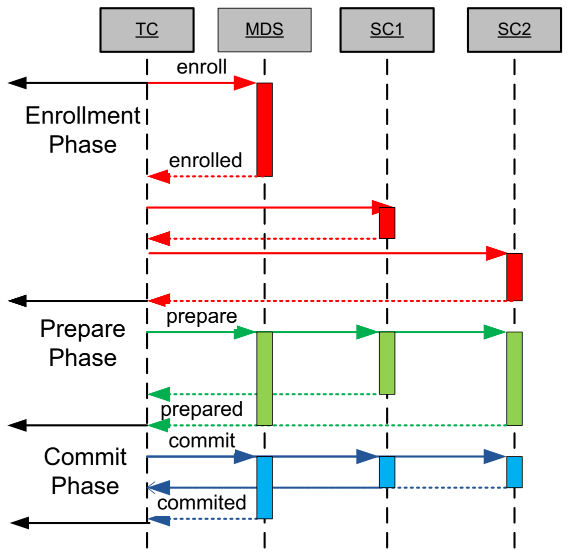

Figure 2.a Proposed Solution

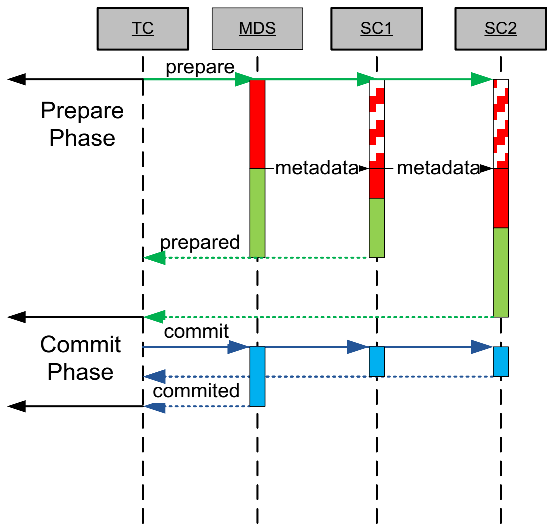

Figure 2.b Two Phase Commit Protocol

sequentially in respect to rest of the system. To achieve desired behavior, transaction is divided in three phases: Enrollment, Prepare and Commit. This approach is similar to classical 2PC, with the addition of the Enrollment phase which is needed to provide the highest possible level of system serviceability during the transaction.

As stated earlier in the text, there exists a master dataset that needs to be updated before any other dataset in the system. The master data service is the data owner, and it is also the entry point for all data model updates. Verification of data updates within the master service is performed by applying comprehensive validation rules as the first step of distributed transaction. If verification of model updates fails in the master service, it is pointless to initiate the transaction on other datasets in the system, since the master service is also the provider for model metadata, which is the necessary input for model update for other services in the system. Therefore, Enrollment phase is divided in two steps.

In the first step, TC sends *enroll* request to the master service. Master service tries to validate the updates, and if validation fails, update is rejected and the transaction is not started on other services in the system. If validation succeeds, in the second step TC sends *enroll* requests to all other services. During Enrollment phase, all services should serve and process requests on the current model version, while building and validating new, updated model in parallel. It is assumed that the most of time consuming work is done in Enrollment phase. If any of the services fails to Enroll, *rollback* request is sent to all transaction participants and the transaction is aborted.

During the Prepare phase, the system should start preparing for transition to the new model version. Therefore, individual components need to start queuing of all incoming requests and waiting for all the threads that are currently accessing data model to finish their tasks (drain). After the model is drained, additional merge between the current and the new model version might be needed to transfer the changes made on the current model between Enrollment and Prepare phase, to the new model. At the end of the Prepare phase, services are notifying TC whether they are ready to start working on the new model version. If any of the services votes that it is not ready to accept the new model version (e.g. because merge of current and new model has failed), all the transaction participants are getting *rollback* request from TC and transaction is aborted.

Work that needs to be done during the Commit phase is very simple. Since the current model was locked and drained from any reader or writer threads during the Prepare phase, and all the incoming requests are queued and wait to be executed, individual components only need to discard the current dataset and start working on the new dataset. Commit phase for the recoverable services in the system should include flushing of the transaction's logs to the non-volatile storage.

Proposed DTM solution actually extends the duration of the transaction compared to standard 2PC protocol (because of the sequential Enrollment phase). However, since the transaction duration is not important as long as the system interruption is minimal, proposed solution is better because there is only a minimal interruption during the Enrollment phase. Standard 2PC protocol would require that during the entire Prepare phase all service components and services remain locked (i.e. all incoming messages are queued and synchronous calls return exceptions). However, since service components can update the datasets only after the master dataset was successfully updated, this window of time while the system is non responsive would be much longer than in the hybrid solution. Attempt to avoid lengthy locks would require complex synchronization logic that would add additional layer of complexity and error proneness.

B. *Individual Service Behavior Policies*

The 2PC algorithm described in previous section is not sufficient to fulfill correctness and performance criteria described in Section 2. There are windows of opportunity when cascading calls from one thread that locked the dataset, to another service (which dataset is also locked), may cause system deadlock. Additionally, the asynchronous communication mechanism may cause message delays, so it might happen that the message arrives at the destination but the model version was changed in the meanwhile. These two issues are addressed by defining additional development rules and policies that need to be followed within the each service in the system.

1) *Message Handling Algorithm*

As explained earlier in the text, most of the communication between services and clients in the system is asynchronous, and message oriented. All datasets are describing current version of system infrastructure, master dataset version (MDV). Essential pieces of information contained in all messages in the system are therefore described with triplet (*topic, MDV, elemID*). An issue with the delayed messages is identified when a service (or any client) receives a message that was sent before the model was updated, but the recipient got it after updating its dataset. This situation may occur both at the external UI client (that is outside of the transaction) and at the internal client (one of the services). Early messages may appear when one of the services in the system has finished its update and started publishing messages with new *MDV*, but the complete transaction is not yet finished (some of the services are still in Commit phase). In this situation, the external UI client is still unaware of the model update that is happening in the system, but receives the message with *MDV* that is different than its datasets *MDV*.
In order to resolve the above mentioned issue, for all asynchronous, message oriented communication in the system, simple message handling algorithm (MHA) should be applied when processing the messages both on client and server side. It consists of the following steps:

1. check message *MDV* – if it is the same as the *MDV* in current dataset, process the message without additional checks; otherwise, proceed to the next step
2. check message *elemID* – if the element with the received *elemID* exists in the dataset, process the message as usual; otherwise, proceed to the next step
3. if (message *MDV*) < (current *MDV*), the message is late and should be dropped because the target element does not exist in the model anymore
4. *this step is needed only on external UI client:* if (message *MDV*) > (current *MDV*), the message arrived early, and should be queued and applied after the client is updated

For synchronous, request/response calls, same MHA rules are applied where possible, with the following difference: in the cases when the dataset is significantly modified during the model update and service is unable to provide the requested functionality, a response containing proper failure description must be returned to the client.

2) *Dataset Locking Policy*

Deadlock prevention mechanism in a distributed environment is traditionally a difficult challenge to deal with. Concurrent and asynchronous nature of the proposed solution requires strict dataset locking policies. Parallel execution of the transaction phases on different datasets violates commitment ordering (i.e. serializability) property, thus leaving the open space for occurrence of the deadlocks. One of the possible solutions for this issue in SOA environments would be to impose service components dependency graph restrictions (e.g. to allow only unidirectional, acyclic service graph dependencies), but such a decision might have repercussion on business processes, and would potentially compromise adding of new services, service components and functionalities in the environment. The approach for defining strict dataset locking policies to be used in every service is chosen instead. Proposed locking strategy is based on readers-writer pattern and pessimistic concurrency control algorithm, Two Phase Locking protocol (2PL) [12]. Multiple reader threads, or one writer thread, are allowed to access the service dataset. The writer must wait for all readers to finish, and the writer starvation must be avoided by blocking of all the incoming readers. The thread that holds a lock must complete its operation in a timely manner; I/O calls, such as calls to other services, must not prologue the locking time. If I/O or blocking call is needed after acquiring the data from locked dataset, sequence of operations should be as follows:

1. Lock the dataset
2. Gather the information needed to perform I/O or blocking call
3. Unlock the dataset
4. Perform I/O or blocking call
5. Re-lock the dataset
6. Update the internal dataset appropriately
7. Unlock the dataset
8. Return the results

It might happen that the dataset was changed between steps 3 and 5, so all the data gathered in 2 must be reassessed in accordance with message handling rules. If the dataset is changed significantly and the thread cannot finish its task, an adequate exception must be propagated to the caller.

The readers-writer lock mechanism may be implemented and verified as shown in [9], or some of the existing, practice proven implementations may be leveraged. After choosing one of these two approaches, the same readers-writers lock mechanism can be used in all services in the system.

III. MODEL VERIFICATION Model Checking

Verification methodology used in the proposed solution is based on Cadence SMV language and model checker tool [1]. Although primarily intended to be used as a verification tool for hardware designs, SMV can efficiently be used for checking that any software or hardware finite state system satisfies specifications given in a branching time temporal logic. SMV implicitly represents finite state systems as binary decision diagrams (BDDs), which make the model checking algorithm simple and efficient [1]. In the proposed verification model, the set of state and input sequences (paths) that guarantees coverage of the transition space of interest is defined, and necessary temporal properties are verified by traversing model state space.

Even when initial system design is given, it is hard to decide which criteria should be taken into account when evaluating the proposed solution. Potential weaknesses of the solution might sometimes be perceived in advance, but it is of the utmost importance to provide comprehensive evaluation strategy. For the assessment of the distributed transaction reliability, computer science theory defines the same four basic properties that need to be validated to prove individual transaction reliability. These four properties are atomicity, consistency, isolation and durability (common acronym for all properties is ACID). In the subsections given below, each of these properties will be discussed in the context of target environment described in Section 1.1., and process of formal verification of some of the individual properties will be explained.

A. SMV Formal Model

Language definitions in the SMV language can be divided in three different categories. First category is the definitional language declaration, and it is used to describe signals in the system and the relationships between them. Assignments and type definitions also belong to the definitional language category. Next SMV language category contains structural definitions, which provide language constructs for defining modules and structured data types in the system by combining definitional components, and instantiating them. Structural part also provides constructor loops and the set of conditional structures that make the definitions of state transitions easier. The third category of language definitions is the language of expressions. Expressions in SMV language are similar to the expressions in other programming and hardware description languages, and are considered to be self explanatory [1].

Each of the individual services and service components, as well as the Transaction Coordinator, are modeled as the SMV structural modules, formally represented as Mealy machines with a 6-tuple (S, $S0$, Σ, Λ, T, G), consisting of the following:

1. a finite set of transactional states (S),
2. a start state (also called initial state) $S0$ which is an element of (S),
3. a finite set called the input alphabet (Σ),
4. a finite set called the output alphabet (Λ),
5. a transition function ($T : S \times \Sigma \rightarrow S$) mapping a state and the input alphabet to the next state, and
6. an output function ($G : S \times \Sigma \rightarrow \Lambda$) mapping each state and the input alphabet to the output alphabet.

In reality, input and output signals (Σ and Λ sets) are different for each state machine which represents the particular service in the system.

As stated earlier in the text, communication in the system is mainly asynchronous, which makes building of consistent and comprehensive formal very difficult. To avoid state explosion problem, compositional reasoning and abstraction techniques are used [13].

Abstraction technique is applied to data and signals in the system. Minimal set of necessary signals consists of following categories:

- Transactional messages
- Messages that relate to the current dataset version (before data update)
- Messages that relate to the new dataset version (after data update)

Besides these signals, additional internal variables are used to describe necessary aspects of each module (Mealy machine), where needed.
Compositional reasoning was applied on top of the abstracted model. It led to the definition of two separate verification models:

- Transaction behavior model, and
- Individual service behavior model

Transaction behavior model contains set of modules representing services and service components that exist in the system, and the Transaction Coordinator component. All of these modules are represented as Mealy machines. For services and service components, input signals are used to model transactional commands which are coming from Transaction Coordinator: *Enroll, Prepare, Commit, and*

Rollback. Similarly, output signals are responses to transactional commands, and represent inputs to Transaction Coordinator: *Enrolled, Prepared, Committed, and Failed.* Transaction Coordinator has an additional input signal, *Update*, which indicates that the transaction should be initiated. Transactional states and signals are defined individually per each module in the system, but are semantically the same.

Individual service behavior model is used to check responsiveness of each service in the system. This model contains only one module which is a simple state machine that implements logic of Message Handling Algorithm described in section 2, and models service behavior. Input signals are messages related to old (i.e. before the update) or new (i.e. updated) version of the dataset. The symbolic names for these input messages are: *msg_MDV_old_elem_exists, msg_MDV_old_elem_deleted and msg_MDV_new*. The output signals are symbolizing action taken upon reception of the message: *msg_processed, msg_queued, msg_discarded, msg_lost.* The transactional states are modeled using the internal module variables (*Idle, Enrolled, Prepared*). In this way, behavior of any service or service component may be modeled in accordance with the expected response required during each transaction phase.

B. *Verification of Properties*

In this section, we will discuss atomicity, consistency, isolation and durability properties in the context of the verification of the proposed design for transactional behavior model. Standard distributed transaction theories usually discuss these properties in the context of distributed databases. The transactional update process described in this paper addresses the distributed datasets, and dataset is a more general term and may denote different types of data, as defined in Section 1. In accordance with the above stated, some of the ACID properties will be interpreted in a different manner.

Atomicity property of the distributed transaction means that if the transaction fails on any dataset in the system, it fails on the entire system (i.e. on all of the services), and state of the system remains unchanged. The distributed transaction atomicity in the proposed solution is ensured by using atomic commitment protocol. In the terms of formal verification, atomicity property is verified by checking coverage of the transition space of interest. In order to make temporal logic statements and formulas able to express evolvement of signals over time, besides the traditional Boolean operators (and, or, not and implies), SMV uses additional operators to specify relationships in time ("globally", "future", "until" and "next time" operators) [1]. To properly verify described property, an additional control signal (*dds_Failed*) is added to the one of the components in the system to simulate failure of validation process. Simplified code given below and collaboration diagram shown in Figure 3 illustrate verification of atomicity property in the system with Transaction Coordinator (TC), Master Data Service (MDS), Dynamic Data Service (DDS) and Calculation Data Service (CDS):

```
trans_UpdPassed:      assert G (s_tc=TC_IDLE &
tc_in=tc_Update -> s_tc=TC_IN_TRANSACTION &
tc_out0=mds_Enroll) -> (tc_in=mds_Enrolled &
s_tc=TC_IN_TRANSACTION -> tc_out1=dds_Enroll )->
(tc_in=dds_Enrolled -> tc_out2=cds_Enroll) ->
(tc_in=cds_Enrolled -> tc_out0=mds_Prepare &
tc_out1=dds_Prepare & tc_out2=cds_Prepare) ->
(s_dds=DDS_ENROLLED & dds_failed=0 ->
tc_in=mds_Prepared & tc_in=dds_Prepared &
tc_in=cds_Prepared)-> (tc_out0=mds_Commit &
tc_out1=dds_Commit & tc_out2=cds_Commit);

trans_UpdFailed:     assert G (s_tc=TC_IDLE &
tc_in=tc_Update -> s_tc=TC_IN_TRANSACTION &
tc_out0=sds_Enroll) ->  (tc_in=mds_Enrolled &
s_tc=TC_IN_TRANSACTION -> tc_out1=dds_Enroll )->
(tc_in=dds_Enrolled -> tc_out2=cds_Enroll) ->
(tc_in=cds_Enrolled -> tc_out0=mds_Prepare &
tc_out1=dds_Prepare & tc_out2=cds_)->
(s_dds=DDS_ENROLLED & dds_failed=1 ->
tc_in=dds_Failed)->(mds_in=sds_Rollback &
dds_in=dds_Rollback & cds_in=cds_Rollback);
```

Consistency property must ensure that the dataset remains in the consistent state after the transaction. In other words, this means that the every individual transaction (update process) applied on each of the datasets in the system must take the dataset from one consistent state to another. This property is tightly coupled with the structure of the individual datasets and validation rules that are applied to verify every update of the each dataset in the system individually. Structure of the individual datasets and verification of the properties of each individual transaction on every dataset in the system, as well as the semantics of the data in the system in general, was not in the scope of the work described here. However, from the system design and transactional behavior perspective, it can be said that consistency violation is taken into account as atomicity violation criteria. In the properties given in the previous subsection, the additional control

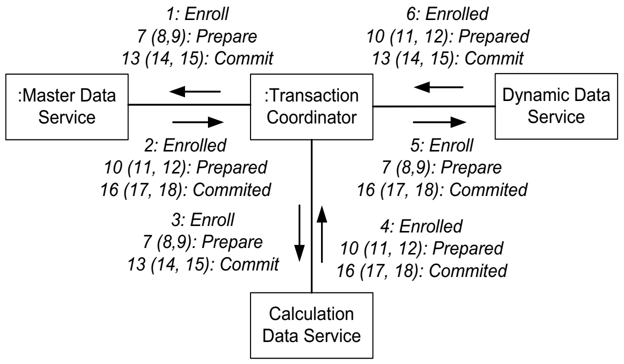

Figure 3. Atomicity property – collaboration diagram

signal is added to simulate failure of the validation process at one of the components in the system, thus causing the failure of the whole transaction.

Isolation property imposes the requirement to prevent the conflicts between concurrent transactions (reads or writes) on the same dataset. In order to ensure this, well known concept called readers-writer pattern is used a synchronization barrier above each dataset in the system (as stated in section 2). In [9], an implementation and verification of readers-writer pattern is described using Uppaal model checker tool.

Durability property guarantees the permanent survival of the datasets that have successfully completed the update process, meaning that even in the case of the overall system failure updates remain saved. Depending on the dataset purpose and nature, this property might not need to be verified for all the datasets in the system. For the recoverable services that need to provide transaction durability, transaction logs are flushed to non-volatile storage during the Commit phase. Durability property is modeled as a part of individual service behavior model, where applicable.

Individual service behavior verification can be formulated in a way that during the transaction, all input messages have to be adequately processed. An example of how this property can be modeled in general case is as follows:

```
mha_check: assert G
  (((msg_in=msg_MDV_old_el_exist) ||
(msg_in=msg_MDV_old_el_deleted) ||
(msg_in=msg_MDV_new)) ->
(msg_out=msg_processed) ||
(msg_out=msg_queued) ||
(msg_out=msg_discarded)));
```

Verification of properties and mapping to adequate formal models is summarized in the Table I below.

TABLE I. VERIFICATION OF PROPERTIES - SUMMARY

Property	Formal Model
Atomicity	Transaction Behavior
Consistency	Out of the scope of this paper, but violation detected in Transaction Behavior
Isolation	RW lock – not modeled
Durability	Service Behavior
System Responsiveness	Service Behavior

IV. CASE STUDY

The approach for managing distributed transactions that is presented in this paper is applied in Distribution Management System (DMS). DMS is a commercial software system used in electricity distribution utilities. It provides efficient execution of all technical and analytical tasks, such as monitoring, analysis, control, operational planning, network development planning, and other. In architectural context, DMS is integrated, heterogeneous environment developed by SOA guidelines. Each of the service components in the DMS system has well defined functionality, and is responsible to manage a certain aspect of the system. Master IEC CIM standard compliant network data model is managed by a service which is the first one to apply validation rules in the Enrollment phase (please see Figure 2). Other services manage different aspects of the system, such as graphical presentation of network elements, calculation of network topology based on current equipment statuses, scheduling of DMS calculations, building and managing calculation data model, and other. It is important to emphasize that for the performance improvement purposes, all datasets in the system are semantically divided into data blocks. Data blocks affected by changes are identified and processed individually, thus minimizing interruptions during the update transaction.

Many practice proven solutions for transaction coordination are available, such as widely used, open standards based X\Open [10]. Since the DMS system is built using predominantly .NET technology, Microsoft's Distributed Transaction Coordinator (DTC) facility is used for transaction orchestration. DTC is also selected because it allows sequential enrollment of services, thus minimizing user interruption in the case when an update is invalid.

Parameters of the verification models for transaction behavior and individual service responsiveness are given in the Table 2 below. The model of the DMS system included 5 service components and transaction coordinator. As stated in previous section, individual service responsiveness property was verified as a separate model, and parameters given in the Table 2 below are given for the simple generic service behavior model (described in previous sections).

TABLE II. PROPERTY PARAMETERS

Parameters	Models	
	Transaction Behavior	***Service Behavior***
BDD Size	652324	109
Lines of Code	453	66
Time[s]	10.7656	0.015625
Input Signals	26	6
Output Signals	51	5
Number of States	18	0 (3 internal)

V. CONCLUSION

Application of model checking techniques to design of software solutions is a promising trend, and it can significantly contribute to overall correctness of software designs. Although model checking might not be applicable to all domains of software solution design, it can efficiently address particular fields and architectural aspects. A priori verification of critical design points in the system can significantly contribute to system design quality, but employment of automated model checker tools is not a common practice in commercial software industry today. One of the reasons might be found in the fact that aggressive commercial schedules often require rapid development of software projects, thus imposing agile and adaptive development methods rather than predictive, more structured methodologies (such as waterfall, etc.).

In this paper we have shown an approach to formal verification and management of the distributed transactions on multiple datasets in a class of integrated, SOA based industrial systems. A priori verification of the proposed solution based on SMV language and model checking tool was done to verify properties that provide reliability of the proposed distributed transaction mechanism. Individual services in the system are modeled as finite state machines. The set of transactional states is defined for every individual service in the system, and the set of signals is used to simulate state transitions and synchronous and asynchronous communication in the system. The concise architectural guidelines for implementation of the important mechanisms are given, thus reducing the risk for inconsistencies and failures when adding new services in the environment. The technique described in the scope of this paper requires in depth knowledge of the software system architecture, and does not require advanced knowledge of formal verification and modeling. Therefore, it can successfully be applied by system designers and architects.

In software development, many functional problems arise from the gaps between the proposed designs and implemented solutions. Therefore, a priori verification of the proposed design is just a half of the work needed for final success. Comprehensive and fully automated verification of model properties, when the model is extracted from the existing code, would be extremely useful tool for every complex software system. Without that, even with the perfect design final solution can still be incorrect. For future work, tools for automated generation of verification models are considered. An approach for automated model generation may include backward engineering of statechart diagrams from the source code, as a first step. As a second step, SMV model would be generated from the statechart diagrams. Finally, manually defined properties would be verified on an automatically generated model.

UML is recognized as widely used standard for describing software solutions. UML formalism is often used by architects at the design phase, but there are also many tools which can "reverse engineer" the existing code and extract UML description from it. Being a common point between a system design phase and already implemented systems, formal modeling and verification of UML descriptions is seen as a very interesting topic for future research.

ACKNOWLEDGMENTS

The authors would like to thank to Mr. Collin Roth, and members of the Telvent DMS v3 Architecture and Development Teams for their valuable assistance and contribution to the results presented in this paper.

This work has been partly supported by the Serbian Ministry of Science, through grants No. III 44009 and TR 32031, 2011.

REFERENCES

[1] McMillan, K.L., The SMV Language, Cadence Berkeley Labs, 1-49 (1999)

[2] Stefan Tai, Thomas A. Mikalsen, Isabelle Rouvellou, Stanley M. Sutton Jr., "Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages," Enterprise Distributed Object Computing Conference, IEEE International, pp. 0105, Fifth IEEE International Enterprise Distributed Object Computing Conference, 2001

[3] Peter Hrastnik, Werner Winiwarter, "Coordination in Service Oriented Architectures Using Transaction Processing Concepts" Proceedings of the 18th International Conference on Database and Expert Systems Applications table of contents, Pages: 855-860, 2007, ISBN ~ ISSN:1529-4188 , 0-7695-2932-1

[4] Walid Gaaloul, Sami Bhiri, Mohsen Rouached, "Event-Based Design and Runtime Verification of Composite Service Transactional Behavior," IEEE Transactions on Services Computing, vol. 99, no. 2, pp. 32-45, , 5555.

[5] Ye, Xinfeng; Keane, John A. "Distributed transaction management scheme for multidatabase systems", Proceedings of the 1994 IEEE Region 10's 9th Annual International Conference (TENCON'94). Part 1 (of 2), Singapore, Singapore, 397-401. (1995)

[6] B.S. Boutros, B.C. Desai, "A two-phase commit protocol and its performance", September 09-September 13 ISBN: 0-8186-7662-0

[7] Udai Shanker , Manoj Misra, Anil K. Sarje: "Distributed real time database systems: background and literature review", Journal Distributed and Parallel Databases , Issue Volume 23, Number 2 / April, 2008

[8] "An Enterprise Information System Data Architecture Guide", October 2001,TECHNICAL REPORT,CMU/SEI-2001-TR-018,ESC-TR-2001-018

[9] B. van Gastel, Leonard Lensink :"Reentrant Readers-Writers: A Case Study Combining Model Checking with Theorem Proving", Lecture Notes in Computer Science, ISSN 0302-9743 (Print) 1611-3349 (Online), 2009

[10] X/Open, Distributed Transaction Processing: Reference Model, version 3, X/Open, 1996.

[11] James E. Johnson, David E. Langworthy, Leslie Lamport, Friedrich H. Vogt: "Formal specification of a Web services protocol", Electronic Notes in Theoretical Computer Science Volume 105, 10 December 2004, 147-158, Proceedings of the First International Workshop on Web Services and Formal Methods (WSFM 2004)

[12] N. B. Al-Jumaha, H. S. Hassanein, and M. El-Sharkawi "Implementation and modeling of two-phase locking concurrency control—a performance study", Information and Software Technology, Volume 42, Issue 4, 1 March 2000, 257-273

[13] Edmund M. Clarke Jr., Orna Grumberg, Doron A. Peled: "Model Checking", MIT Press, ISBN 0262032708 / 9780262032704 / 0-262-03270-8

[14] Komoda, N, "Service Oriented Architecture (SOA) in Industrial Systems", IEEE International Conference on Industrial Informatics, Issue Date: 16-18 Aug. 2006, Pages: 1 - 5
DOI: 10.1109/INDIN.2006.275708

[15] Trivunovic, B. Popovic, M. Vrtunski, V. „An Application Level Parallelization of Complex Real-Time Software", 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS), UK, 2010.

[16] M. Popovic, I. Basicevic, and V. Vrtunski, „A Task Tree Executor: New Runtime for Parallelized Legacy Software", Engineering ofComputer Based System Conference, San Francisco, USA, April 2009.

[17] I. Basicevic, S. Jovanovic, B. Drapsin, M. Popovic, and V. Vrtunski, "An Approach to Parallelization of Legacy Software", IEEE ECBS-EERC, Novi Sad, Serbia, Sept. 2009.

[18] Velikic, I., Popovic, M., Kovacevic, V.: A Concept of an Integrated Development Environment for Reactive Systems. Proc. of IEEE ECBS, 233-240 (2004)

[19] Popovic, M., Kovacevic, V., Velikic, I.: A Formal Software Verification Concept Based on Automated Theorem Proving and Reverse Engineering. Proc. of IEEE ECBS, 59-66 (2002)