3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing

Using SDL to Model an Object-Oriented Real-Time Software Architectural Design

J. Jenny Li and J. Robert Horgan

Using SDL to Model an Object-Oriented Real-Time Software Architectural Design

J. Jenny Li and J. Robert Horgan
Telcordia Technologies (formerly Bellcore)
445 South Street
Morristown, NJ 07960-6438, USA
{jjli | jrh}@research.telcordia.com
phone: (973)829-4753; fax: (973)829-5981

Abstract

Specification and Description Language (SDL) is a formal object-oriented language for modelling real-time interactive systems. It is an International Telecommunication Union (ITU) standard. A software architecture is the structure of a program including a set of inter-communication components. These components are often independently executable super objects. We use the architecture design to answer questions such as how the super objects fit together and how to reuse them.

This paper investigates the feasibility and benefits of using SDL to represent the dynamic aspect of the software architectures. It includes a methodology and an accompanying tool, Workflow-to-SDL-Transformation (W2S), for deriving software architectures in SDL from an originally informal use case flow definitions. The focus of research is in telecom domain, many of which are real-time interactive systems. We applied our method to a partial telecom system. The experimental results are collected and presented. Our result shows the feasibility and benefits of representing the dynamic of software architecture in SDL.

1. Introduction

Many researchers believe that software engineering must go beyond object oriented methodologies to a new technology based on software architecture. Software architecture has many uses, including: 1) guiding the wiring-up of interacting modules on distributed systems; 2) avoiding system reengineering costs if modifications are required after the system is implemented or set up; 3) analyzing the behaviors and attributes of the system before the system is developed, which can be used to improve the quality of the architecture and to properly price the software; and 4) assisting customers to decide whether a component fits their architecture and how it would affect overall performance.

In order to verify and simulate software architecture, the architecture must be represented in a standardized language. Many architecture description languages (ADLs) have been proposed, varying depending on the properties that their proponents were interested in capturing about the system (e.g., object-oriented vs. procedural) or depending on their generality. One category of such language is called executable architecture definition language (EADL). It captures the dynamic aspects that are not obvious in object models. An example of such language is Rapide[2] developed in software engineering lab of Stanford university.

The focus of our research is in the telecommunications domain whose software systems typically include well over a million lines of code. The system configuration, that is, the components comprising the system and the sources of the components, typically varies for each customer. We are seeking a language that is most appropriate for representing architectures of such systems.

Based on the representation, we wanted to verify and simulate the architecture to answer questions about a particular system configuration. Examples of questions include: How do these components from different sources fit together? What is the behavior of the integrated system? What trade-offs in performance/reliability/cost are implied by alternative versions of a component?

We evaluated a number of existing ADLs, such as Unified Modeling Language (UML), and Rapide. UML embodies a domain analysis approach that combines elements of object-oriented modeling, feature modeling and case-based reasoning. It also includes state charts for dynamic behavior representation, which does not have

formal semantics. Rapide uses an event-based execution model, which is similar to SDL. We found most ADLs inadequate for a number of reasons: (1) too formal; although a sound semantic basis for analysis is a worthy goal, the cost of producing such a specification is too much when a different system architecture may be required for each customer. (2) too informal; some only provide a set of notations without formal semantics, which makes simulation and analysis more difficult. (3) wrong abstraction level; components of such abstraction level can be small objects, modules, or functions which is too detailed for a large real-time interactive system with over one-million lines of codes.

While researchers are putting a lot of effort on defining new ADLs, we focus on investigating whether the existing specification language, such as SDL, is suitable for representing object-oriented software architectures.

This paper includes 5 sections. Section two describes the requirements for ADLs including various view points and general principles. It shows that SDL can be extended to meet all the requirements. Section three presents the Workflow-to-SDL-Transformation (W2S) method of specifying software architectures in SDL. Section four gives an example of specifying software architecture in SDL. Section five draws some conclusions.

2. SDL Meets Requirements of ADLs

In the past years, researchers have studied ADLs to come up with a list of requirements and view points of software architecture. An architectural description may include various views of a system [3]. Besides view points, there are general principles to be satisfied by ADLs[2].

Kruchten[3] proposed 4+1 View models of software architecture: logic, process, physical and development view of architecture plus scenario definitions. Architectures are captured in the first four concurrent views and validated by the fifth view.

First, the logical view describes the functions and services that the system provides to its end users. The main purpose of the SDL is to specify the behavior of systems, which may include the use of data if necessary. Behavior of systems is embedded in the functions and services provided to its users, which are specified in SDL processes. As an example, consider a small telephone exchange. The service provided to the users includes establishing and tearing down phone calls between telephones. Such behavior can be given in an SDL specification with phone processes, callers and callees, each of which defines behavior of a phone call handler. An SDL specification provides a specification and description of a system's software architecture in logical view point.

Secondly, the process view describes the network of independently executing processes that comprise the system, and how they interact and synchronize. SDL specification includes a set of hierarchical block interaction diagrams. Blocks communicate through delaying or non-delaying channels. Each block is defined with a set of SDL processes communicating through non-delaying signal-routes. Each SDL process is an independently executable entity. The SDL flattening function converts the hierarchical block diagrams into one diagram consisting of interacting SDL processes. This flattened SDL system specification diagram provides a process view of the entire system's software architecture. For example, suppose two phones are controlled by two independent processes. The SDL specification described previously allows the interact and synchronization between the two processes so that phone calls can be set up. An example of an interaction is that the caller sends a signal to check for the status of the callee. If the callee is busy, the caller will send the user a slow-busy tone, otherwise, it will send out a ring tone and send another signal to callee to ask it to ring.

Third, the physical view describes the mapping of the software components to the hardware. The software system is executed on a network of computers (processors). The components in the software architecture must be mapped to various computers (processors). Software architecture must define such mapping to instruct software development in the later stages. SDL dose not have notions of hardware equipments. Hardware equipment in this case consists of computers and their interconnection media. The SDL specification can define such mapping if some default rules are set in constructing SDL block interaction and process interaction diagrams. SDL specification has two types of communicating media. It also allows remote procedure calls (RPCs). If a procedure in a process is exported to the other process through RPC, these two processes are most likely to be installed on two different computers. If two processes inside two different blocks are communicating through delaying channels, these two processes are also most likely to be installed on two different computers. If the communication between two processes has no delay, the processes could be on the same computer or the same

domain where the actual delay may be very small and negligible. The rule for using SDL to specify software architecture in physical view is to put the processes of different computers on different blocks communicating through delaying channels. Export local procedure to processes on different computers if it is necessary. Continuing with the previous example, if the delaying in the communication between the caller and callee is significant, the two processes will be put in two different blocks communicating through delaying channels, otherwise, put them in the same block.

Fourth, the development/structural view describes the static modular structure of the software code. Object-oriented software systems include objects. Procedural systems usually include modules. The development view of software architecture is represented in object relationship diagrams or module relationship diagrams. Some research has been done on converting SDL to Object Modeling Technique (OMT) diagrams[4]. The reversal, from SDL to OMT, is possible. This reversal is very useful, not only for representing development view of the software architecture, but also for reengineering of procedural programs into more advanced object-oriented programs. If an OMT diagram can be derived from a SDL specification, the SDL specification can provide the development view of a software architecture. Deriving an object model diagram from SDL specification is an on-going research as reported in [4]. A set of major rules in the conversion is reported below: 1) Treat the SDL instances, such as block instance, channel instance and process instance, as objects; 2) treat the abstract data structures of processes as objects inheriting from the process objects; 3) represent state transitions and data structure manipulations as object method; 4) identify common methods between objects to derived a common object that the objects with common methods can be inherited from; and 5) represent the processes inside a block, or blocks inside a super block as a component of the high level object with aggregation relationship.

Lastly, the scenarios definitions that are used to verify that the elements of the first four views work together seamlessly. For each scenario, corresponding scripts (sequences of interactions between objects and between processes) are defined. Such sequences of interaction are reflected in the event sequences implied in SDL specification of processes, blocks and the system. For example, when an external input signal is sent to an SDL process, it will cause a chain reaction which triggers the interaction between processes until an external output is generated or an internal transformation is performed. The event sequence from the initial external input to the final output or transformation is a scenario defined by SDL specification.

Besides representing various view points of software architecture, SDL meets the requirements for executable ADLs. Luckham[2] proposed 7 requirements for ADLs: component abstraction, communication abstraction, communication integrity, dynamism, causality and time, hierarchical refinement, and relativity. SDL meets these requirements with some minor extension which is commonly used in practice.

Component abstraction: The SDL is able to describe the behavior of each component (processes) in a form allowing execution and analysis. It does not give the information of the facilities provided and the facilities required by a component. Such information is often included in SDL specification as complementary information. SDL does not include the attributes of each component, such as through-put, reliability, availability, price, etc. Such information is often given in performance specification of the system which is complementary to SDL behavioral specification. Overall, the SDL specification of software architecture with the addition of complementary information provides component abstraction capability for SDL.

Communication abstraction: Communication between components in SDL includes channels and signal-routes. The communication is asynchronous. Both channels and signal-routes are formally defined, which allows execution and analysis.

Communication integrity: SDL processes can communicate directly or indirectly. Indirectly and implicitly communication is allows between two processes inside one block. There is no necessary to have an explicit communication path for two processes to communicate.

Dynamism: SDL allows dynamic creation and termination of process instances and their corresponding communication paths during execution. SDL is capable of modelling architectures of dynamic systems in which the number of components and connectors may vary when the system is executed.

Causality and Time: SDL processes are independent concurrently executed components. A clock provides the center time. Timers can be set in each processes independently. Input signals and timer time-out signals trigger state transitions.

Hierarchical Refinement: Various abstraction levels of the SDL block diagram provide hierarchical refinement

for both components and connectors.

Relativity: Since the SDL can specify software architecture on various abstraction level, the relationship between the specifications of different abstract level is automatically defined. For example, a high abstraction level of software architecture in an SDL specification can be used to verify the conformance of a more detailed level of software architecture or even the implementation of the software system. This feature has been used in software observer and software supervisor to automatically detect failures of target software systems.[5]

In conclusion, SDL provides various views for specifying software architecture. It meets requirements for ADLs. Since the existing language, SDL, is able to specify software architecture, we believe that research on defining new architecture description languages is not always necessary. The next section describes a method of using SDL to represent software architectures given originally in diagrams and flows.

3. Workflow-to-SDL-Transformation (W2S)

SDL is capable of representing software architectures. In the applications discussed in this paper, we have architectural specifications that represent the combinations of the logical, process and physical views at a high abstraction level. We consider requirements, workflows, operational profiles, or regression tests the sources which define the behavior of a system. We use a set of "flows" to define system behavior and to capture the interaction of the system components for various scenarios. We call such semi-formal architecture description ArchFlow. We were able to develop a converter which can convert the semi-formal architecture documents in ArchFlow to SDL model of the system behavior with complementary performance specification. The SDL model can be used in a behavior and performance simulator for architectural analysis.

An *ArchFlow* description includes two parts: an architecture diagram, which describes the interconnection between components; and a *workflow* set, which describes the set of scenarios prescribed on the system behaviors. An architecture diagram is an undirected graph $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E})$, where $\boldsymbol{N}$ is the set of components and an edge, $(\boldsymbol{n}_i, \boldsymbol{n}_j)$, in $\boldsymbol{E}$ stands for a communication path between two components $\boldsymbol{n}_i$ and $\boldsymbol{n}_j$. The behavior of a component is informally described in English text.

A *workflow* set is denoted as $\boldsymbol{W} = \{\boldsymbol{V}_i = <\boldsymbol{v}_{i1}, \boldsymbol{v}_{i2},..., \boldsymbol{v}_{im}>\}$, where each $\boldsymbol{V}_i$ is called a *workflow* scenario of the system consisting of a sequence of steps $\boldsymbol{v}_{i1}$, $\boldsymbol{v}_{i2}$,..., $\boldsymbol{v}_{im}$. Each step $\boldsymbol{v}_{ij}$ describes the expected behaviors of a component in this scenario, such as receiving a message from another component, performing internal actions, and/ or sending out messages to other components. The description is given informally in English text.

Apart from the semi-formal nature of *ArchFlow* specifications, these specifications are often incomplete. For example, the communication paths among components, i.e., the edges in the architecture diagram, do not show the direction of message passing. So an edge may represent an unidirectional channel from one component to another, or two unidirectional channels between two components. The association between channels and signals are not explicit, either. Moreover, some of the communication paths maybe missing: there are message exchanges between two components in their respective behavior descriptions but there is no edge between them in the architecture diagram. Therefore, in order to perform architecture analysis through simulation to an *ArchFlow* specification, we must construct an SDL model that is complete with respect to the information given in the architecture diagram $\boldsymbol{G}$ and the associated *workflow* set $\boldsymbol{W}$.

The SDL model $\boldsymbol{P}$ of an *ArchFlow* specification $\boldsymbol{A}$ is a network of communicating extended finite state machines (CEFSM [6]). Each component $\boldsymbol{n}_i$ is modeled as an extended Finite-State Machine (EFSM) $\boldsymbol{P}_i$ that exchanges signals via channels with other components, i.e., other EFSM's. Initially, $\boldsymbol{P}_i$ is manually derived from the informal behavioral description of the component, based on domain expertise. In the course of the derivation, the architecture diagram $\boldsymbol{G}$ is modified accordingly as follows: 1) An undirected edge $(\boldsymbol{n}_i, \boldsymbol{n}_j)$ is replaced by unidirectional channel $\boldsymbol{C}_{ij}$ from $\boldsymbol{P}_i$ to $\boldsymbol{P}_j$ if $\boldsymbol{P}_i$ sends signals to $\boldsymbol{P}_j$, and/or channel $\boldsymbol{C}_{ji}$ from $\boldsymbol{P}_j$ to $\boldsymbol{P}_i$ if $\boldsymbol{P}_j$ sends signals to $\boldsymbol{P}_i$; 2) The message set $\boldsymbol{M}_{ij}$ on channel $\boldsymbol{C}_{ij}$ contains the set of signals sent from $\boldsymbol{P}_i$ to $\boldsymbol{P}_j$; and 3) Additional channels may be added to $\boldsymbol{P}$ if $\boldsymbol{P}_i$ sends signals to $\boldsymbol{P}_j$, even though there is no edge between components $\boldsymbol{n}_i$ and $\boldsymbol{n}_j$ in the original architecture diagram $\boldsymbol{G}$.

The next step is to manually transform each *workflow* $\boldsymbol{V}_i = <\boldsymbol{v}_{i1}, \boldsymbol{v}_{i2},..., \boldsymbol{v}_{im}>$ in $\boldsymbol{W}$ into an execution

sequence of $\boldsymbol{P}$, again based on domain expertise. Since *workflow*s do not always have discrete events to stimulate each action, it is important to abstract discrete events from the *workflow* descriptions in order to derive the signal flows in the corresponding CEFSM model. To this end, we reorganize the steps so that each step corresponds to exactly one component that carries out the step. The component executing the step may receive a signal from another component, perform a sequence of internal transitions, and/or send signals to other components. The behavior of each node now includes two parts: behavior defined originally in the node definition and behavior derived from the *workflow*s. After the addition of *workflow* behaviors, we need to add to the nodes triggering events to activate these behaviors.

To execute step $\boldsymbol{v}_{ij}$ of a *workflow* $\boldsymbol{V}_i$ by a node, an activating signal must be sent to the node to start the execution. After the current node has finished, activating signals must be automatically sent to the next node to keep the *workflow* going. Hence the actions performed by this node are expanded to include the receiving of the activating signal from the previous node or an outside environment. and the sending of the activating signal to the next node. For the last step, the component involved should send a signal to take itself back to its initial state. Therefore, at the end of the transformation, each *workflow* in $\boldsymbol{W}$ becomes an execution sequence that starts and ends in the initial global state of $\boldsymbol{P}$. The addition of signal sending and receiving must consider all *workflow*s. However, if a *workflow* scenario is included in the original definition of components, it is not necessary to be added again. Combined with the original architecture node definition, the input set of a node becomes $\boldsymbol{I}_a \cup \boldsymbol{I}_w$ and the output set becomes $\boldsymbol{O}_a \cup \boldsymbol{O}_w$, where $\boldsymbol{I}_a/\boldsymbol{O}_a$ is the original inputs/outputs and $\boldsymbol{I}_w/\boldsymbol{O}_w$ is the inputs/outputs implied in the *workflow*s.

In the meantime, the architecture diagram $\boldsymbol{G}$ needs to be updated accordingly: 1) New channels must be added if they are missing. 2) The message set of each channel must include those in the *workflow* definitions but not in the component descriptions and those that are added to synchronize steps between components in a *workflow*.

The last step is to incorporate the behaviors of each node derived from $\boldsymbol{W}$ into the state transition diagram of the corresponding EFSM of the node. Now, the behavior of each node includes two parts: behavior defined originally in the node definition and behavior derived from the *workflow*s.

Besides the behavioral model, the performance of a CEFSM system is determined by the performance of each EFSM and each channel. We identified two issues:

1) CEFSM requires performance specifications of the channels, which are not explicitly given in the architecture descriptions. To obtain such information, we reused the data collected from the existing system to come up with performance models of various channels. As a rough approximation, we classified channels into two types: ones whose performance is significant to the system and ones that are not significant, i.e., where the delay is negligible. The former are modelled as queueing systems with nondeterministic delays.

2) The performance of each component is implied in the flow definitions of *ArchFlow*. Each EFSM in CEFSM model can only specify the behavior of a component. We have to extend each EFSM to include performance information. The performance can be either behavior-dependent or behavior-independent, depending on the granularity of the performance analysis. For example, the performance can be defined by the processing time of the entire component or the processing time of each task in the component.

We now model each component in the architecture to be an ***EFSM*** in CEFSM model with the extension of performance model: $\boldsymbol{CP}$ = ($\boldsymbol{EFSM}$, $\boldsymbol{PF_{node}}$). The performance model can be given on various abstraction level. One simplest mathematics model of delay, for example, assumes that the processing time (flow delay) is f(t) = at for a software component and a constant 'r' for a hardware component.

We developed a tool that can read in the formal model of each node, the inter-communication diagram and the formal definition of *workflow*s to generate a more complete SDL model with complementary performance specification. A general algorithm for combining two formal models into one is very difficult. FIGURE 1 summarizes the simplified algorithm of translating *ArchFlow* to SDL and performance models.

We use an example in Figure 2 to illustrate this algorithm. Figure 2 (a) shows an example of formalized definition of the nodes with its inter-communication diagram. Figure 2 (b) shows an example *workflow* definition which is to be added to the original formal model given in Figure 2 (a). Our goal is to add the behavior defined in the *workflow* to the SDL model. We

```
Algorithm W2S-Translation (in: (N, E, W[][]), out: (EFSM, ch,
PFnode, PFflow))
{
EFSM = new (S, I, O Va);
ch = new (E);
for (i = 1; i <= NUM_WORKFLOWS; i++ ) {
for (j = 1; j <= NUM_STEPS; j++ ) {
if IO_action(W[i][j]) update(ch);
else {
ni = behavior_carrier(W[i][j]);
Behavior(ni) += Behavior(W[i][j]);
PFnode(ni) += Performance(W[i][j]);
I(ni) += activate_Wij;
O(ni) += activate_Wij+1;
ch += line_Wij_2_Wij+1;}}
PFflow = Performance(W[i]);}
return(EFSM, ch, PFnode, PFflow);}
```

FIGURE 1. A Simplified SDL and Performance Modelling Algorithm

are not concerned about the resulting model having new legitimate behavior due to combination. We assume that *workflow*s do not conflict with each other.

Figure 2 (c) shows the resulting CEFSM model in SDL. The *workflow* behavior not defined in the original CEFSM model is appended to the model. New branches are constructed using ANY construct to insure that the additional behavior does not intervene with the original behavior defined by the model. ANY is a nondeterministic construct, which allows the execution moves along any branch following the construct.

In summary, we can transform an architecture in ArchFlow as: *Architecture* => (***G***, ***W***) => (***N***, ***E***, ***W***) => (***EFSMs***, $\boldsymbol{E} \cup \boldsymbol{E}_w$) => *CEFSM model*. The CEFSM model is complemented with performance models. Sometimes, the *workflow* performance is only used to test the simulated architecture to see whether it achieves the performance requirements.

4. SDL Repre. of a Macro SW Architecture

The W2S method was applied to the description of the software architecture of a telecomsystem X. System X controls the flows of call routing, customer caring, addition of new features, and trouble reporting. It consists of four independent software packages, each of which may be provided by different vendors.

4.1 ArchFlow of System X

The architecture of system X was described in ArchFlow originally, including communication line-and-box diagram and the definition of workflows.

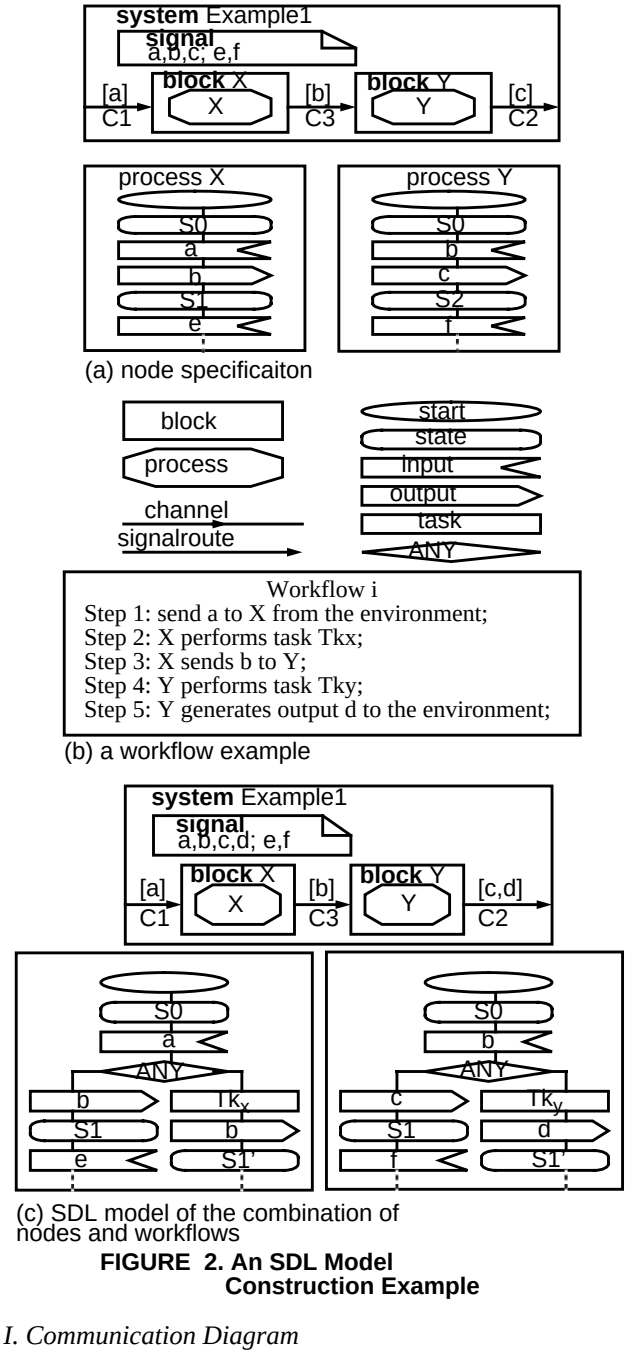

FIGURE 2. An SDL Model Construction Example

I. Communication Diagram

Figure 3 shows the communication diagram of the architecture. There are four major components in the system. We assume that database systems are outside the scope of the system, residing in the environment of the system. Components, CX, WX and MX, each has an attached database. RX is a control module. The behavior of each component and the behavior of the entire architecture is defined in tabular form.

II. Work Flow Definition

The behavior of the entire architecture is defined in

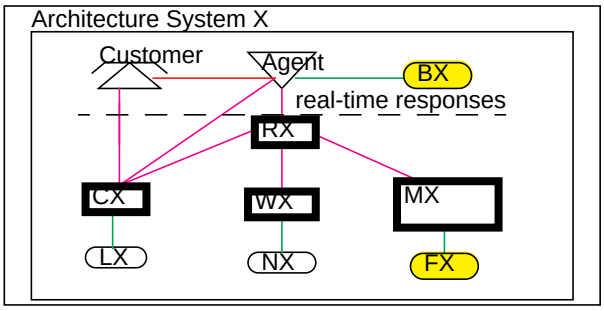

FIGURE 3. Communication Architecture Diagram

workflow scenarios in tables. A partial scenario example of the system X is given in Table 1. In this scenario, when the customer places a request to CX, CX uses its script to access the database for domain information. Once it derives the information from database LX, it passes on the information to RX to request for processing. RX routes the report to the desire destination, an agent in this case. Steps 4 and 5 are related to the database LX. It is treated as external signals based on the previous assumption.

4.2 SDL Spec. of System X Architecture

The SDL specification of the architecture was derived using the converter described in the previous section. The converter derives an SDL model from the ArchFlow description. First, it examines the architecture diagram and the workflows to identify the interactions between components. It then uses the information to convert the communication lines to be directional. The choice of delaying or non-delaying channels depends on whether the communication time between the two components is negligible. The initial system level diagram of System X is given in Figure 4.

Table 1. Scenario #1 -- Customer Initiated Report

flow name	Process Step	feat ures	refs
...	1. Customer: call placed > CX	...	...
	2. CX: access script; access LX to determine domain	...	...
	3. CX: inquiry > LX	...	...
	4. LX: accept inquiry from CX accept data based on info provide domain info back to CX	...	...
	5. LX: response > CX	...	...
	6.CX: accept message from LX; provides specific scripts to route call	...	...
	7. CX routes customer call > RX agent ...	...	...

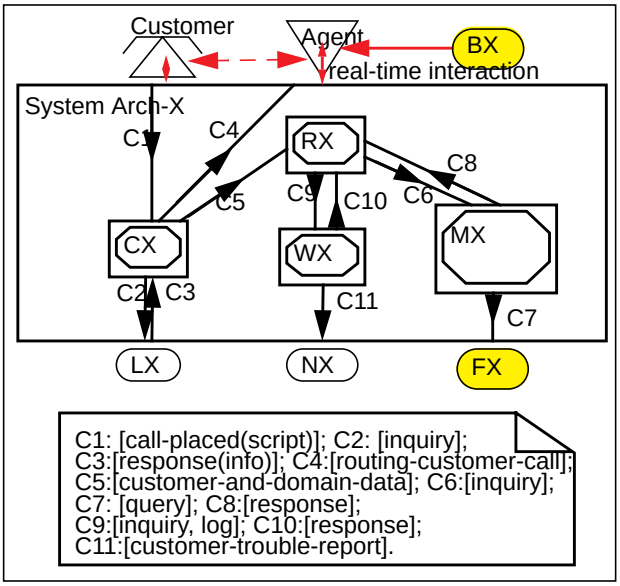

FIGURE 4. Initial SDL System Level Specification

The system level diagram is refined to include workflow activation signals. For example, if a scenario of work flow includes two subsequent steps on two different entities that do not communicate with each other. A connection is established between the two entities and a signal is sent from the entity of the first step to the entity of the second step. As for the System X, a connection, C12, and a signal, S2-3, is added in between CX and WX. S2-3 means the signal is used for flow No.2's step No.3. The final resultant system level diagram of System X is given in Figure 5.

The only difference between Figure 4 and Figure 5 is the addition of channel C12. This is because there is only one case that two subsequent steps do not communicate with each other.

The second major step is to specify each components in SDL processes. The specification is difficult to automatic. The observations show that the description of each component often include sending or receiving messages and the actions that are described in verbs. The verbs used in a particular system is a very small set. Based on this observation, a partial automatic conversion of the component description to SDL specifications was

conducted in W2S method.

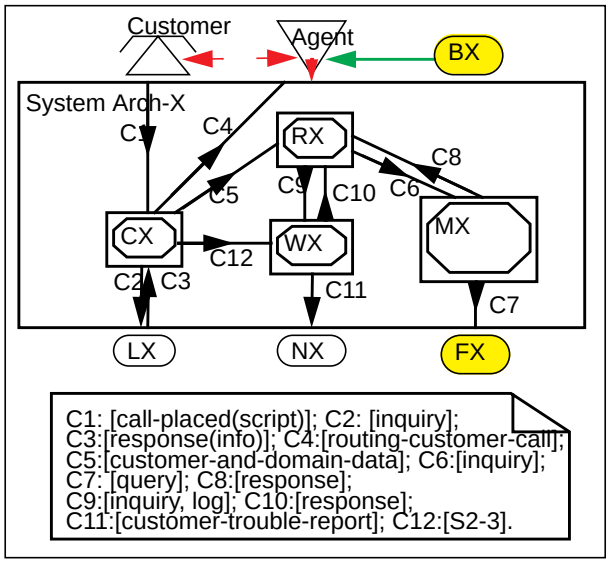

FIGURE 5. Resultant SDL System Level Specification

The third step is to validate the obtained SDL specification of the software architecture. The validation is to check whether the workflows are properly reflected in the SDL specification. The SDL processes must be enhanced to accommodate the addition of some flow step activation signals. For instance, if the component CX cannot handle the signal "call-placed(script)" in the workflow defined previously, such a signal and corresponding transition must be augmented to the component model of CX. The augmentation must be consistent in the way that the workflow can be smoothly triggered and no other new behavior is introduced. Figure 6 shows the addition of the new transition to component model CX.

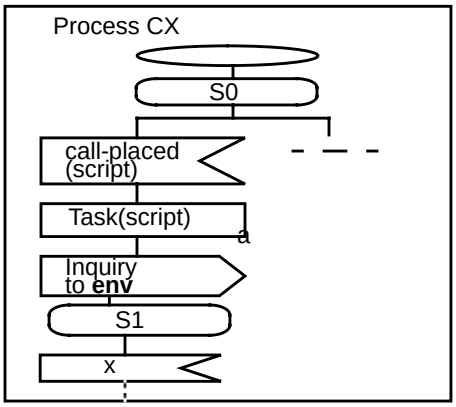

FIGURE 6. Component CX Partial Definition

5. Concluding Observations

This paper investigates the feasibility of using SDL as an executable Architecture Description Language (EADL) to specify software architectures. This feasibility was shown through a Bellcore system example. We evaluated a number of existing ADLs, but found them inadequate for a number of reasons such as too formal and wrong abstraction level. Furthermore, we found that creation of new architecture description language is not always necessary. Using existing specification languages such as SDL can save a lot of research efforts and allows the reuse of existing technologies.

The future research direction of this work is to apply other SDL technologies to software architecture analysis. For instance, the SDL simulation provides dynamic information for architectural analysis. We can also use SDL model to quantify software architecture properties such as reliability, dependability, performance, cost, etc.

References

[1]International-Telecommunication-Union-T Recommendation Z.100. *Specification and Description Language SDL*. Geneva: International Telecommunication Union, 1989.

[2]An Event-Based Architecture Definition Language, David C. Luckham, James Vera, IEEE Transactions on Software Engineering, Vol 21, No 9, pp.717-734. Sep. 1995

[3]P. B. Kruchten, "*The 4+1 View Model of Architecture*", IEEE Software, pp42-50, Nov. 1995.

[4]From OMT to SDL : a Prototype for transforming an Object-Oriented Analysis Model into a Formal Specification, by Herman Peeters Promoted by Prof. Dr. Viviane Jonckers, Faculty of Sciences Licence in Applied Computer Science (M.Sc.) Vrije Universiteit Brussel Academic year 1994 -1995

[5]J. Jenny Li and R. Seviora, "Automatic Failure Detection with Conditional Belief Supervisor," Proc. IEEE 7th International Symposium on Software Reliability Engineering (ISSRE96), pp.4-13, Nov. 1996.

[6]CEFSMD. Brand and P. Zafiropulo, "On Communicating Finite-State Machines", Journal of ACM, Vol. 30, No. 2, April 1983, pp. 323-342.

[7]J.Jenny Li, "Performance Prediction Based on Semi-Formal Software Architectural Description", Proc. 1998 IEEE International Performance, Computing, and Communications Conference (IPCCC'98), February 16-18, 1998.