SDL to Fiacre translation

Subeer Rangra
ESIEE Paris, France
subeer.rangra@edu.esiee.fr

Emmanuel Gaudin
PragmaDev, France
emmanuel.gaudin@pragmadev.com

Abstract

The translation of a system model to an intermediate format is an important step towards formal proofs of properties. This paper presents the formulation of rules to enable the translation of SDL standard models to Fiacre language. The translated model is then provided as input to a suitable toolset that performs model checking and other analysis. The formulated translation rules are discussed in view of the correctness of translation and ease of implementation. Implementations of some of the translation rules in an industrial tool (PragmaDev RTDS), and a proof of the concept on a simple SDL model are presented to ascertain the validity of the proposed translation.

Keywords: Model Checking, Formal Methods, SDL, MSC, Fiacre, TINA

1 Introduction

Validation of software is a major concern in the domain of software engineering. Currently, industries engage their efforts in testing and simulation processes. The coverage of the test sets decreases gradually as modern day embedded systems increase in complexity. Under such circumstance, it becomes necessary to use new methods to ensure the reliability of software.

Over the last two decades, model checking has emerged as an attractive approach for ascertaining the correctness of control computer based systems. Model checking is a formal verification technique which allows for desired behavioral properties of a given system to be verified on the basis of a suitable model of the system through systematic inspection of all states of the model. [1]

The approach of formal verification is not limited to any domain and can be used in multitude of scenarios where a formal proof or validation of a system is required.

In order to formally verify a design, it must first be converted into an appropriate format. This can be accomplished after needed transformations from a user defined model described in a precise modeling language such as SDL. In the context of this work, we focus on porting SDL models to a pivot language called Fiacre. This can then be used by various toolsets to perform various analyses or verify the system.

All through the development cycle of the Fiacre language there have been various translations from many DSMLs (Domain Specific Modeling Language). One such translation to SDL has been done at LAAS-CNRS [2], with an aim to translate SDL to Fiacre. The present paper differs on many levels. First, the work reported in [2] was performed in the very early life cycle of Fiacre language and one of the aims of the translation was to build Fiacre language itself. Second, it did not aim to enable the formal verification of the model in SDL. The focus was to translate the language schematics, which is one of the two main goals of present work. We also aim for the translation to be able to verify the system and for the results to be easily interpretable. Further, the implementation of such a verification tool chain in an industrial tool like PragmaDev Real Time Developer Studio will enable industries to use formal methods in real world scenario.

The translation and subsequent formal verification is a rather complex problem in itself and may cause issues ranging from the state space explosion problem, inability to conserve the semantics and incomprehensibility of verification results due to different formalisms used. Therefore, for the sake of correctness and simplicity a fresh approach is necessary. Also, in this domain of formal verification of SDL with the use of Fiacre, published literature is very limited. Similar model transformations from other languages to Fiacre do exist but this particular translation opens another bridge to formal verification of SDL models.

In this paper, brief introduction to SDL and Fiacre is given in Section 2. Section 3 presents various translation rules concerning the selected notions of SDL. Section 4 briefly describes the encoding of these translation rules in PragmaDev RTDS. Section 5 covers the proof of concept, accomplishing a two part goal, first that the proposed translation works and produces an accurate translation of the system, and second that the system can be correctly verified by the use of a verification tool.

2 SDL and Fiacre

Specification and Description Language (SDL) [3] is a formal language standardized by the International Telecommunication Union (ITU-T) as the Z.100 recommendation. It can be used easily in combination with other languages, such as SysML or AADL. It is an object-oriented language, created to describe complex, event-driven and interactive applications involving many concurrent activities and communicating with each other through messages. SDL is often used in combination with MSC (Message Sequence Charts), which is a graphical and textual notation for the description and specification of the interactions between system components. It helps in better visualization of the messages that are being exchanged in the system internally and with the environment. The objectives of SDL include a language that is easy to learn, use and interpret, while implementing a precise system specification and description. The five main aspects to formally model the system e.g. structure, behavior, communication, data, and inheritance make SDL a favorable candidate amongst the various DSMLs currently used in system modeling to describe an executable model. While SDL former versions presented difficulties in dealing with some of the Real-Time embedded concepts, SDL 2010, the latest release, inspired by SDL-RT [4], provides the required balance between the high level concepts of SDL and technical effectiveness of C code in dealing with such systems.

FIACRE [5,6] stands for "Format Intermédiaire pour les Architectures de Composants Répartis Embarqués", French for "Intermediate Format for the Embedded Distributed Component Architectures". It is an intermediate format language formally defined for representing both the behavioral and timing aspects of embedded and distributed systems for formal verification and simulation purposes. It was defined as a pivot language to provide gateway for the Domain Specific Modeling Language (SDL, SysML, AADL etc) into the domain of formal verification supported by tools such as TINA, CADP and OBP.

Fiacre was designed and developed in the framework of several projects, involving industry and academic partners, such as TOPCASED [7], the VASY project of INRIA [8], VerTICS Team [9] of LAAS-CNRS, and the ACADIE team [10] of IRIT. This multi-team development has led to the creation of Fiacre as a pivot language. Thus giving it a clear advantage of being able to easily translate, and then provide formal verifications in a variety of different model checking tools [11], developed by these teams.

Some modified versions of Fiacre have been proposed to help writing generic libraries of protocols, for example communication and scheduling protocols used by the AADL runtime [12]. Examples of translation and verification such as AADL to Fiacre [13], done under the framework and context of TOPCASED project, are available. Attempts have also been made to make translations from UML to Fiacre [14]. Projects such as Quarteft [15] aim to improve the usability of Fiacre by proposing real time extensions among other things. Extensions like various real-time constructs and real-time verification patterns have been proposed in [16]. Methods have also been proposed to make Fiacre more useful in itself, for example the feedback of verification results from model checking tools [17].

There are three toolsets that accept a Fiacre description and are briefly mentioned as follows. The very development life cycle of Fiacre ensured that these tools can be used with a Fiacre file as input and the required transformations and analysis be carried out.

TIme petri Net Analyzer (TINA) [18] is a toolbox for editing and analyzing Timed Petri Nets [19]. TINA works with Timed Transition Systems (TTS) [20], a generalization of Time Petri Nets with data variables and priorities. TINA employs a State/Event LTL model checker [21] in the form of a tool : *selt*. Construction and Analysis of Distributed Processes (CADP) [22] is a toolset that accepts high-level protocol descriptions written in the ISO language LOTOS (Language Of Temporal Ordering Specification, ISO 8807) [23]. Observer-Based Prover (OBP) Toolset [24] is an implementation of CDL (Context Description Language) [25, 26] developed at ENSTA-Bretagne. It is a model-checker based on observers and context descriptions.

Advantages regarding the usage of Fiacre lie in its ease of translation to a format or language used by these verification tools. Binaries can be used (viz. flac (CADP), frac (TINA)) or there are meta-models available in Eclipse environment [27,28] to perform the required model transformation.

3 Formulating the rules

The disadvantage of any intermediate or pivot language is that each step of retranslation introduces possible mistakes and ambiguities. Using a pivot language involves at least two translation steps, one from the model to the pivot language and another from the pivot to the final abstraction. A simple and fresh approach is hence necessitated.

The translation rules we present here aim to satisfy the following main points:

- The semantics of SDL are preserved.
- The verification of the system using one or more toolsets mentioned above is possible.
- The verification results are observable to the very least.

To comply with the above points, a full translation of SDL is not mandatory; focusing on some selected SDL semantics help in getting initial results faster. Once the initial proof is accomplished the addition of more exhaustive rules is comparatively an easier task.

3.1 Architecture

SDL supports a hierarchical structure with blocks and processes connected with channels. Blocks are structuring elements that do not imply any physical implementation on the target, rather are used to represent hierarchical structures; processes are used to describe their functionality.

As there are no constructs except the process and component in Fiacre, a few propositions are made in this work to retain some structural aspects of SDL, which can be obtained from the configuration of Fiacre data structures and component. Fiacre components allow for a hierarchical structure of the system to be defined and the processes describe the behavior of sequential components. Processes cannot contain other processes while components can contain other components enabling recursive specifications.

A few propositions which use components to give a hierarchical notion to a system in Fiacre are given in Table 1.

Table 1. SDL architecture to Fiacre component

SDL	Fiacre

SYSTEM system_name**;**	`component sys-`
	`tem name is`

BLOCK block_name;	`component`
	`block_name is`
	`par * in`
	`.`
	`end`
	`end Block1`

PROCESS process_name;	`process_name()`

In this translation rule the SDL system name forms the name of the main Fiacre component and is called first when a Fiacre file is compiled.

Components can declare local variables to be passed to sub-processes (or sub-components). All the shared variables and labels must be declared and initialized inside the component that they are being used in. This structure is used to define the internal structure of a single block or even a system in which the processes or procedures are all communicating among themselves. The `par...end` structure is the composition operator which describes what labels are to be synchronized with among the enclosed elements. This along with the right use of component can be used to represent a hierarchical structure in Fiacre. As explained by the use of a hierarchical SDL structure given in Figure 1.

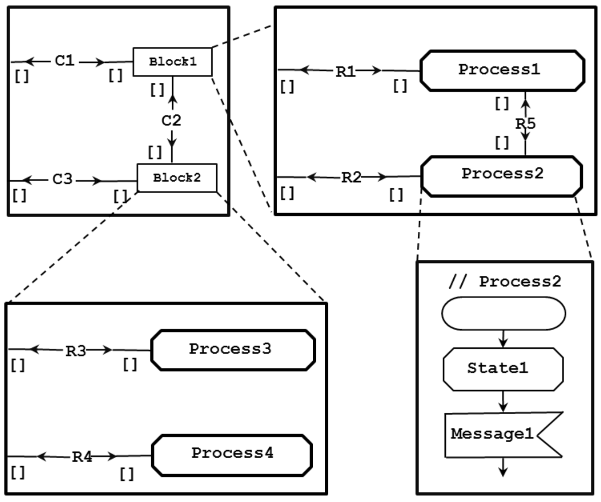

Fig. 1. A hierarchical structure in SDL

The above SDL hierarchy represents relation between agents and procedures in a system. A structure representing an SDL system is illustrated in top left of Figure 1 (let's call it `SystemExample`), the rest are the internal hierarchical structure represented by the use of blocks (`Block1`, `Block2`) and processes inside those blocks. The textual description below (`Process2`) is a process definition that is written in Fiacre when defining a process. The component structure is just used to call them with appropriate arguments.

```
component SystemExample is
  component Block1 is
      par * in
        // PROCESS1 IS CALLED
        // PROCESS2 IS CALLED
      end
  end Block1
  component Block2
      par * in
        // PROCESS3 IS CALLED
      end
      par * in
      // PROCESS4 IS CALLED
      end
  end Block2
//SystemExample IS CALLED
SystemExample
```

Using * denotes universal label sets in Fiacre, such that all the labels in use are shared between the processes, this can be changed to selectively synchronize different SDL processes.

Unlike SDL there is no special provision for an Environment variable in Fiacre. An accurate definition of the system is needed to have an exhaustive and mathematically correct verification. This presents a problem as to whether make a custom translation and build an environment process automatically, by looking at all the communications of the processes with the Environment; or on the other hand ask the user to model a process which accomplishes what the user needs the environment to do (generally send initial signals to start the system, input or output messages). The conclusion was drawn that for the translation to remain true to the nature of the original system, it is better to ask the user to model an environment process, rather than modeling one automatically in the translation. This way the user can be certain of the system being correct to his specifications and we get a closed system for the needed abstraction.

3.2 Communication

Messages, processes and their associated message queues are fundamental SDL constructs to enable communication. Fiacre has ports/labels as modes of communication which offer synchronous communication, however these are not used, as SDL communication is asynchronous. Instead, in this context shared variables are used as a user defined data type to keep communication asynchronous in Fiacre. A shared variable also gives a way to work around the Fiacre's single communication rule which states that at the most only one interaction label can be used along any execution path.

The idea of message passing among processes is translated and implemented as part of the proof of concept. Each process has a message queue of its own `processQueue` which is initialized at the beginning of the program. The message information is stored in the message queue as an enumerated type of all the possible message names in the system. All the queues are stored in an array `messageQueueArray`. The array contains each of the process queues (indexed by a constant integer value associated with each process). This array is a globally shared variable. It can be accessed by the processes which in turn can access the queue of the relevant process to send messages and also their own queue to read messages. This queues and array construct is shared as references because Fiacre variables can only be shared among several processes if passed by reference. This method is valid in the present context, for the limited scope of this translation it is an acceptable compromise to reduce complexity.

Translation of the communication aspects of SDL are presented through a simple example which has two processes: `pProcess1` and `pProcess2`, communicating by passing messages `message1`, `message2` between each other as given below and shown in Figure 2.

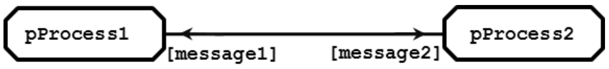

Fig. 2. Message passing in SDL

```
//INITIALISATION
const NUMBER_OF_PROCESSES: nat is 2
const PROCESS_QUEUE_SIZE: nat is 2
//initializing a random int value to
//reference the queues
const pProcess1 : int is 1
const pProcess2 : int is 2

type messageName is union M_message1
| M_message2
end
type processQueue is queue
PROCESS_QUEUE_SIZE of messageName
type messageQueueArray is array
NUMBER_OF_PROCESSES of processQueue
//INITIALISATION END

//BODY OF PROCESS 1
process pProcess1 (&MESSAGEQueue:
read write messageQueueArray)
var self : int := pProcess1, re-
ceivedMessage : messageName
// RECEVING MESSAGE, BY READING FIRST
ELEMENT AND THEN REMOVING IT FROM THE
QUEUE
receivedMessage:=
first(MESSAGEQueue[$self]);
MESSAGEQueue[$self]:=
dequeue(MESSAGEQueue[$self]);

// SENDING MESSAGE, BY ADDING IT TO
RECEIVER PROCESS QUEUE
MESSAGEQueue[$pProcess2]:=
enqueue (MESSAGEQueue[$pProcess2],
M_message2);
// BODY OF PROCESS 1 END

// BODY OF PROCESS 2
process pProcess2 (&MESSAGEQueue:
read write messageQueueArray)
var self : int := pProcess2, re-
ceivedMessage : messageName

// RECEVING MESSAGE
```

```
receivedMessage :=
first(MESSAGEQueue[$self]);
MESSAGEQueue[$self]:=
dequeue(MESSAGEQueue[$self]);
// SENDING MESSAGE
MESSAGEQueue[$pProcess1] :=
enqueue (MESSAGEQueue[$pProcess1],
M_message1);
```

Signal parameters in SDL are represented in the following proposed manner. This associates with each message a record type that contains the message parameters.

```
type messageName is union M_message1
| M_message2
end
type message_param_type is int
type messageRecord is record
  message_param : message_param_type,
  message_N : messageName
end
type processQueue is queue
PROCESS_QUEUE_SIZE of messageRecord
type messageQueueArray is array
NUMBER_OF_PROCESSES of processQueue
```

3.3 State Machines

The dynamic behavior of the system in SDL is described in the processes through extended FSMs (Finite State Machines), communicating with messages. A process state determines what behavior the process will have upon reception of a specific message.

Fiacre allows for simple state declarations and we have used a prefix to denote the start state and differentiate it from other state declarations. Stop state of SDL is not translated because such a state will inevitably lead to a deadlock preventing an exhaustive analysis of the system.

States are declared inside a process as `states RTDS_start, state1`. Transitions from one state to another in Fiacre is a simple structure of `from <initial_state>` followed by the operations and conditions ending with `to <final_state>`.

```
from state1
// TRANSITION BODY
to state2
```

The start state in SDL is translated into a simple state (`RTDS_start`) in Fiacre with no conditions in the transition. This allows for automatic initialization of the process transitions and aids in translating concept of modeling the environment process as explained in Section 3.1.

Fiacre has two types of decision statements:

- non-blocking (`if-else`), if the optional else branch is omitted the control is passed to the statement following the condition. Table 2 gives a simple decision statement translation into if-else condition in Fiacre.
- blocking (`select, case, on, where`) in which if no match is found, then the construction is blocking: no transition is possible through that statement.

Table 2. Condition statements translation to Fiacre

SDL	Fiacre

DECISION cond;
(true):
/* action1 */
(false):
/*action2*/
ENDDECISION;
	`if cond then`
	`/*action1*/`
	`else`
	`/* action2*/`
	`end`

3.4 Data types

Fiacre only supports three standard data types (integer, natural and boolean) but complex data types if needed can be created with records and unions. Such elaborate data type declaration constructs allow for easy translation of predefined data types in SDL.

In SDL, the index type for an array can be of any type. However in Fiacre, only numeric indices between 0 and an integer value are supported. Thus SDL arrays only based on an integer index type with 0 or a positive value as lower bound can be translated. The translation rules for various data types that are used in SDL are given as in the Table 3.

Table 3. Data types translation rules

SDL	Fiacre
SYNONYM maxCount integer = 3;	`const maxCount : int is 3`

DCL
v **integer**,
b **boolean**,
	`var v int,`
	`var b bool`
	`//Can be declared and initialized as`
	`var b : bool:= false;`

NEWTYPE nouveau
LITERALS dial, ring, hangup;
ENDNEWTYPE;
	`type nouveau is union`
	`  dial|ring|hangup`
	`end`

NEWTYPE nouveau
STRUCT
field1 integer;
field2 boolean;
ENDNEWTYPE;
	`type nouveau is record`
	`  field1 : int,`
	`  field2 : bool`
	`end`

SYNTYPE interv = integer **CONSTANT** 0:4 **ENDSYNTYPE**;	`type interv is 0..4`

NEWTYPE intTable
ARRAY(interv, integer)
ENDNEWTYPE;
	`type intTable is array 4 of int`

4 Implementing translation rules in PragmaDev RTDS

PragmaDev Real Time Developer Studio employs an Abstract Syntax Tree (AST) to handle information exchanged internally in a machine and programming language-independent way.

RTDS parses the SDL model and generates an Abstract Syntax Tree. All export mechanism and code generators in the tool are based on this Python AST. The above mentioned rules were written in a Python class. This allows for automatic and accurate generation of a Fiacre file that can either be exported for manual use or coupled with a shell script to be given as input to the desired verification toolset. The verification using TINA performed in the proof of concept uses the shell script that can be customized to use the various tools of TINA on the generated Fiacre file.

5 Proof of concept

Using TINA and the translation rules coded in the PragmaDev RTDS, a simple SDL example given as follows is verified against some properties.

This example system in SDL comprises of four processes and six signals namely `go`, `m2` (message2), `m3` (message3), `m2f` (message2Forward), `m3f` (message3Forward) and `done`. The environment process initiates the system to begin by sending message `go` to `pProcess1`. `pProcess1` upon receiving `go`, sends out `m2` and `m3` to `pProcess2` and `pProcess3` respectively. `pProcess3` on receiving `m2` forwards a message `m2f` to `pProcess4`. Similarly `pProcess3` upon receiving `m3` sends out `m3f` to the `pProcess4`. Now `pProcess4` is where we have two messages arriving namely `m2f` and `m3f`. Only when the messages are received in this order, `pProcess4` forwards a message `done` to the environment process. In other cases we have the system not emitting `done` even after a `go` was sent.

Figure 3 is an MSC diagram of the system described above that shows the sequential flow of messages between the environment process and the four processes described above.

Expectations. One important aspect of SDL i.e. asynchronous message passing is tested in this example. The goal is to check that every time the messages *go* is sent by the environment process, it is not guaranteed to receive the message *done*. We expect the property stating as such to be false when verified. It also allows one to observe the translation and see if the schematics of SDL were preserved and there are no errors in the automatically generated Fiacre file.

The MSC trace of this system in Figure 3 is inconclusive as to whether the properties that we are verifying are true or false in all possible executions of system.

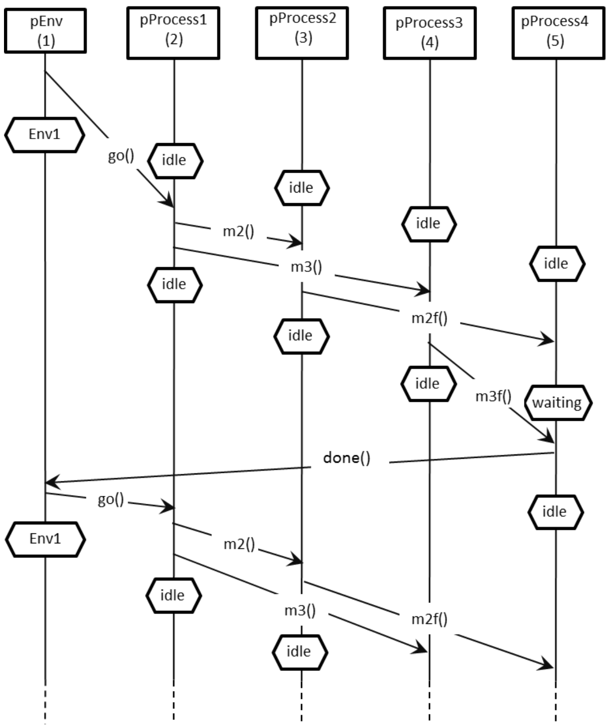

Fig. 3. MSC of our example system

Once generated, the path we take here to verify our Fiacre system description with TINA is as follows:

1. *frac* analyzes a Fiacre description and compiles it in a Time Transition System description (extension .tts) suitable for analysis with the TINA tools.
2. *tina* (tool) constructs the reachability graphs by generating the state space of a TTS in the ktz format, which is a compressed, binary format for Kripke transition systems.
3. *selt* which is a State/Event LTL model checker checks the Kripke transition systems built in the previous step, against S/E LTL formulae.

All of these tools are called by an external shell script that runs these commands in order to give the correct files as inputs to the appropriate tools.

5.1 Properties for verification

To perform model checking using TINA we need to write properties. All Fiacre items are observable (*states*, *local variables*, *events*); all the information of a Fiacre description is available for model checking by writing properties in LTL.

One way to write properties is inside the Fiacre file and another is in the command line of *selt* (for the TINA tool used in this case). We chose to add the properties to the generated Fiacre file for this demonstration as it is easy to understand and write from the user's perspective.

The properties are written in Fiacre format [29]. Properties are defined from observables (or probes); the languages of properties include LTL properties, Dwyer et al. patterns [30].

The following property checks if there are any deadlocks in the system.

```
property ddlf is deadlockfree
assert ddlf
```

The second property says that receiving `go` in `Process1`, leads to `done` received in `pEnv`, which behaves as the environment process that we have modeled.

```
property goLeadsToDoneState is
sSystem/3/state idle leadsto
sSystem/1/state idle
assert goLeadsToDoneState
```

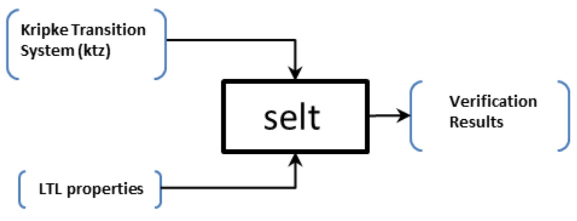

Fig. 4. selt tool of TINA

The properties and observables given as input to *selt* must be in the format of place or transition identifiers. If one choses to write properties in the Fiacre format (as in this case) they are automatically converted to appropriate format for use in *selt* by the *frac* compiler.

5.2 Results of the verification

The *selt* tool generates the following output which is a transitional description (*scn* format) leading to the case where the properties are being violated.

```
<selt>...
Selt version 3.1.0 -- 01/07/13 --
LAAS/CNRS
ktz loaded, 60 states, 135 transi-
tions
0.002s
.
operator ddlf : prop
0.000s
# [] - dead |- FALSE
pEnv 1 t0 pProcess1 1 t0 pPro-
cess1 1 t1 pProcess2 1 t0 pPro-
cess2 1 t1 pProcess3 1 t0 pPro-
cess3 1 t1 pProcess4 1 t0 pPro-
cess4 1 t3 pProcess4 1 t1 pEnv 1 t1
pProcess1 1 t1 pProcess3 1 t1 pPro-
cess2 1 t1 pProcess4_1_t4 pPro-
cess4 1 t3
# accepting all
0.000s
```

The operator `ddlf` is the property where we verified the system to be deadlock free. The result is FALSE. This signifies the existence of a deadlock in the system.

```
operator goLeadsToDoneState : prop
0.000s
# [] (pProcess1 1 sidle => <>
pProcess4 1 sidle) |- FALSE
pEnv 1 t0 pProcess1 1 t0 pPro-
cess1 1 t1 pProcess2 1 t0 pPro-
cess2 1 t1 pProcess3 1 t0 pPro-
cess3 1 t1 pProcess4 1 t0 pPro-
cess4 1 t3 pProcess4 1 t1 pEnv 1 t1
pProcess1 1 t1 pProcess3 1 t1 pPro-
cess2 1 t1 pProcess4_1_t4 pPro-
cess4 1 t3
# loop accepting
0.000s
```

The result for the second property namely `goLeadsToDoneState` is also shown here to be FALSE, as expected. Because of asynchronous message passing in SDL, the messages might not maintain their order.

This output not only gives the TRUE/FALSE result of verification, but also prints out a counter example of the transitions in the system (TTS description), which is the series of transitions taken that led to the aforementioned property being violated. On closer observation of these series of events, it was found coherent with our expectations; the conflict arose because of the order of messages (received by `pProcess4`), not being maintained as required.

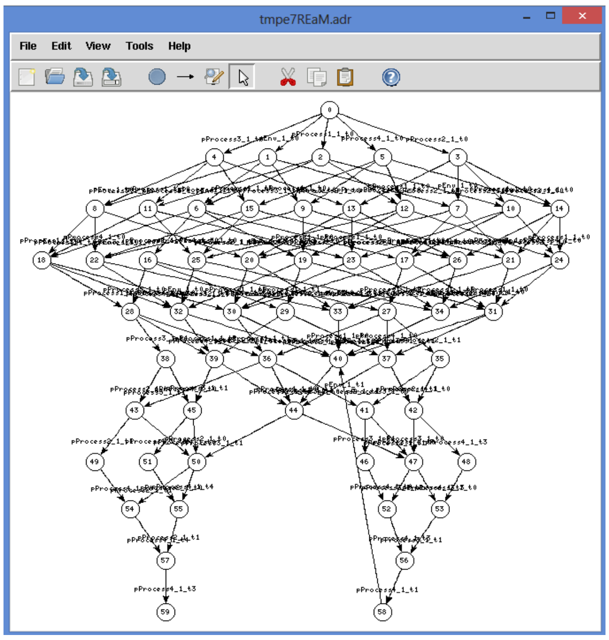

Fig. 5. Graphical view of Kripke Transition system

Figure 5 is the graphical view of the transition system. It is generated from the description format of the Aldebaran tool of the CADP toolset, using *nd* editor supplied inside TINA. The last transition on the left-bottom is a deadlock and also the second property that we verified is being falsified here. The transition sending a `done` message back from `pProcess4` is absent here hence the deadlock. The above results not only prove the correctness (although in limited context) of the translation rules, but also established that TINA is capable of adequately perform model checking with our automatically generated Fiacre file.

6 Discussion

A custom *Start* state was added in the Fiacre description as an empty transition to translate the SDL's semantics appropriately. In the process of this change it was discovered that only by adding a simple state which simply points to the first state of the process; the total number of states in the transition system changes drastically causing a state space explosion problem. This kind of problem is rather expected while using a formal method like model checking. Hence, for any future work this needs to be addressed in an appropriate way.

A loopback for analysis of the verification results has not been implemented as part of this paper. It was originally proposed to have the verification results generated in a graphical representation, preferably in the form of an MSC, as it is often used with SDL. This would be an added advantage, as the user would not need to know the inner working of the verification tool chain; the results shall be displayed in close representation to the original SDL model making them easier to understand.

The problem seems to be that the output of the verification results have gone through too many translation themselves that it was difficult to understand what the counter examples generated by *selt* represented in SDL or even in Fiacre terms.

Manual parsing allowed verifying that the results were in fact correct. But parsing them line by line, referencing back not only through the generated Fiacre file but the TTS format; was not only a complex task and needed more time, but the risk of introducing errors also ran high. This was due to the fact that the counter-examples produced by *selt* are at TTS level rather than at Fiacre level, representing transitions from one state space to the other. The messages that are pushed on to the message queues also have a different representation in the TTS description.

Such problems are a part of using a complex tool chain in which multiple transformations of the system description take place not only in terms of the language used but semantic changes as well. The major importance of messages in SDL does not translate well when dealing with transition systems impacting the ability to observe, verify and write accurate properties for verification. Any future work hence needs to focus on such issues and address them carefully, in order not only to have a feedback of some sorts but also to ensure an accurate translation.

7 Conclusion

In this paper we discussed the rules to translate standard SDL models to the Fiacre language. Their implementation in PragmaDev RTDS has also been achieved as part of an internal version of RTDS due to the very limited scope of this translation. With some of the propositions presented here, it can very well be released as part of PragmaDev RTDS to those interested.

A fresh and simple translation scheme has been adopted in this work to keep the original semantics and easy interpretation of transformations. However, further additions if required can easily be done to further enhance the SDL support. Verification with TINA done as part of proof of concept generated the results as per expectations establishing evidence that the translation does not add any unnecessary ambiguities and is correct. The whole chain of model checking the system with TINA works satisfactorily.

The feedback of the verification results was investigated but the semantic difference between different translations the system went through would require a substantial amount of work to implement. There is ample scope to investigate further with TINA or other toolsets to understand various formalisms they embody, properties they use and the results that they generate to arrive at a robust verification tool chain for system models defined in SDL.

The proof of concept presented in this paper has a flash demonstration with the internal build of RTDS, which is available on PragmaDev website. It is a screen-capture of the whole process from the SDL model in RTDS down to verification results using TINA.
[http://www.pragmadev.com/product/viewlet/FIACRE/fiacre_viewlet_swf.html]

ACKNOWLEDGEMENTS

We acknowledge the support from ESIEE especially Associate Prof. Yasmina Abdeddaïm for proofreading and help in placement at PragmaDev. Help from Mr. Bernard Berthomieu (LAAS CNRS) regarding literature and discussion on Fiacre and TINA were crucial to this work. Inputs from Mr. Philippe Dhaussy (ENSTA-Bretagne) are also greatly appreciated.

REFERENCES

1. Christel Baier, Joost-Pieter Katoen. Principles of Model Checking. The MIT Press.
2. Rodrigo Saad. Schema de Traduction SDL vers Fiacre. LAAS report 07580, October 2007.
3. Specification and Description Language – Overview of SDL-2010. Recommendation ITU-T Z.100.
4. Specifications and Development Language-Real Time. http://www.sdl-rt.org/
5. B.Berthomieu, J.P.Bodeveix, M.Filali, H.Garavel, F.Lang, F.Peres, R.Saad, J.Stoecker, F.Vernadat. The syntax and semantics of Fiacre. Draft Version 3.0. http://projects.laas.fr/fiacre/papers.php
6. B.Berthomieu, J.P.Bodeveix, P.Farail, M.Filali, H.Garavel, P.Gaufillet, F.Lang, F.Vernadat. Fiacre: an Intermediate Language for Model Verification in the TOPCASED Environment.
7. TOPCASED: "Toolkit in OPen-source for Critical Applications and SystEms Development" http://www.TOPCASED.org.
8. VASY Validation of systems. http://vasy.inria.fr/
9. « VERification des Systèmes Temporisés CritiqueS » http://www.laas.fr/VERTICS/.
10. « Assistance à la Certification d’Applications DIstribuées et Embarquées » http://www.irit.fr/-Equipe-ACADIE
11. Tools and projects to work with Fiacre. http://projects.laas.fr/fiacre/projects.php
12. Jean-Paul Bodeveix, Mamoun Filali, Manuel Garnacho, Regis Spadotti, Zhibin Yang. On the Mechanization of an AADL Subset.
13. B. Berthomieu, J.P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, P. Gaufillet, S. Heimk, F. Vernadat. Formal Verification of AADL models with Fiacre and Tina.
14. Heng Sotharith. Transformation de modèles UML vers des programmes Fiacre. ENSTA Bretagne. (dumas-00725248).
15. QUARTEFT project http://quarteft.loria.fr
16. Nouha Abid Silvano, Dal Zilio. Real-time Extensions for the Fiacre modeling language
17. Faiez Zalila, Xavier Crégut, Marc Pantel. Verification results feedback for Fiacre intermediate language.
18. B. Berthomieu, P.O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. International Journal of Production Research, 42(14), 2004.
19. P. M. Merlin and D. J. Farber, “Recoverability of communication protocols: Implications of a theoretical study.” IEEE Tr. Comm., vol. 24, no. 9, pp. 1036–1043, Sept. 1976.
20. T.A. Henzinger, Z. Manna, and A. Pnueli. Timed Transition Systems. In REX Workshop, pages 226–251, 1991.
21. Chaki, Sagar; Clarke, Edmund M.; Ouaknine, Joël; Sharygina, Natasha; and Sinha, Nishant, "State/Event-Based Software Model Checking" (2004). Computer Science Department.Paper 409.
22. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, Mihaela Sighireanu CADP A Protocol Validation and Verification Toolbox.
23. Information processing systems -- Open Systems Interconnection -- LOTOS -- A formal description technique based on the temporal ordering of observational behavior. ISO 8807:1989.
24. OBP-CDL, Lab STICC, ENSTA-Bretagne. http://www.obpcdl.org/doku.php.
25. Philippe Dhaussy, Pierre-Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, Benoit Baudry. Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation.
26. Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux, “Improving Model Checking with Context Modeling” Advances in Software Engineering, vol. 2012, Article ID 547157, 13 pages, 2012.
27. Signal metamodel to Fiacre metamodel. {http://www.TOPCASED.org/index.php?idd_projet_pere=107}
28. ATL Use Case - Compiling a new formal verification language to LOTOS (ISO 8807). http://www.eclipse.org/atl/usecases/FIACRE2LOTOS/
29. Frac properties http://projects.laas.fr/fiacre/properties.html
30. Matthew B. Dwyer , George S. Avrunin , James C. Corbett. Property Specification Patterns for Finite-state Verification.