Debugger for Declarative DSL for Telecommunication Product Line

Abstract

Development of telecommunication product lines is still a very labor intensive task, involving a great amount of human resources and producing a large number of development artifacts — code, models, tests, etc. Declarative domain-specific languages (DSLs) may reasonably simplify this process by increasing the level of abstraction. We use the term "declarative" implying that such a DSL does not enable the development of a closed software application, but rather supports creation, generation and maintenance of various kind of software assets — product database, events and event handlers, target code data structures, etc. At the same time, such a DSL may have some executable semantic, but it could be very specific and have many environment-wise requirements. Thus, execution and debugging of such DSL specifications is a meaningful task, which has no common solution due to the unique executable semantic. Consequently, it is not possible to use debug facilities of known DSL environments, such as XText, MPS, etc. for such a case. In the current paper, we present a debugger for DDD — a declarative DSL intended for support device management in software development in the context of a router product line by XXX company. We clarify executable semantic for DDD, making it possible to execute DDD specifications in an isolated environment, i.e. in simulation mode, without generation of target code. We use a graphic model-based notation to depict every step of execution. Finally, we implement and integrate the debugger in the DDD IDE, using Debug Adapter Protocol and language server architecture combined with the Eclipse Xtext/EMF tool chain.

Keywords

Product Lines, Telecommunications, Declarative DSL, Debugging, IDE

1. Introduction

Nowadays, it is typical for large companies to develop not a single software product but a number of products with varying features and functionality, providing upgrades, etc. All of these products and corresponding development infrastructure form a product line [1]. This approach expands the market capacities of a company and provides reuse of various development assets, e.g. code, models, requirements, tests, etc.

Following the trend, the XXX company is developing a product line of network routers. The product line contains about fifty different products, hundreds of unique boards, several hundred thousand C files, and more than ten million lines of source code. One of the problems of a product line is the development of the Device Management layer. This layer focuses on hardware drivers and network interfaces of the router being provided to network management layer. The problem is in a large range of hardware, complicated hardware connections (in particular, it is possible to insert various cards into the motherboard of the router) and various configurations of one product depending on demands of customers.

To meet these problems, a special declarative DSL was developed [2]. This language provides the ability to specify hardware structure of the product that is visible to software. Furthermore, it can also specify the behaviour of a product in an event-based manner. It provides abstractions to define various product information, supporting generation of product configuration, network data, events and event handlers, target code data structure, etc. A special IDE that fully supports the proposed DSL was developed. Finally, a debugger was needed to improve maintenance of DSL programs [3].

Leading DSL environments such as Xtext [4], GEMOC Studio [5], and MPS [6] support a two-level debug model [7] that is not suitable for declarative DSLs. Moreover, debug development facilities that are provided within these environments are deeply integrated with them, and their transfer to other runtime platforms is highly limited. Microsoft Visual Studio Code supports the Debug Adapter Protocol that provides a standard for the debugger user interface rather than technologies for development. Thus, DSL debugging for declarative languages is a pressing problem.

There is a number of research papers concerning DSL debugging [3, 8], but they do not deal with event-based behaviour DSLs. Event-based debugging is implemented in a series of model-based development toolsets such as YAKINDU [9], Rhapsody [10], but these tools are oriented at the UML-based system structure (components, interfaces, ports, channels, etc.). In the case of DDD DSL, we have both a specific system structure model.

We may conclude that existing research and development tools do not provide any significant basis for developing a debugger for event-based declarative DSLs. Thus, creating it is going to be research-intensive.

The novel contributions of our paper are as follows:

- Scenario-oriented debugging concept for DDD;
- use cases of the debugger;

© 2023 Author:Pleasefillinthe\copyrightclause macro

CEUR Workshop Proceedings (CEUR-WS.org)

- An extension of DDD for configuration and initialization of system developed for debugging;
- Graphical model-based notation for visualization of debug execution trace,
- Implementation of the debugger with the support of Debug Adapter Protocol and integration into the DDD IDE.

This article is organized as follows. Section 2 provides some background of the research. Section 3 presents scenario-oriented debugging concept for DDD and use cases of the debugger. Section 4 describes extension of DDD for specifying debug configuration of the product. Section 5 introduces graphical model-based notation for visualization of debug execution trace. Section 6 describes debugger implementation issues. Section 7 contains an overview of related work, and finally, section 8 provides the conclusions of the paper.

2. Background

The software part of the router in the considered product line consists of two main components: network management and device management. The latter encompasses hardware drivers and a network agent that provides an intermediate level between the drivers and the network management component. It implements a set of rules that determine the router's reaction to various network management events.

The domain-specific language DDD is intended for describing the Device Management subsystem. DDD consists of the following parts:

- *Composition model* aims at describing hardware part of the router that is visible for drivers and network management. It consists of a set of boards and cards. The latter are a special type of boards and can be inserted into boards' or other cards' special slots, extending the functionality of the parent device. Actually, DDD-specification of the product describes a set of board and card types (moduleTypes). A real configuration of the product delivery depends on customer requirements — that is, similar to the variability of hardware units in a laptop, when the customer just specifies type of the storage, volume of RAM, etc. during their purchase. Thus, facilities for creating target product configurations are outside of the DDD due to including not only device management level information. Some features of DDD for creation of debug configurations (debug model) will be described later.
- *Inheritance Model* addresses to specifying network management attributes of hardware elements.
- *Behaviour Model* focuses on event-driven behaviour of the network agent.

Let us consider the behaviour model in more detail. Specification of the network agent behaviour consists of a set of rules. Each rule includes the event that the network agent is subscribed to. The event triggers the action sequence if the logical condition attached to the event is true. The following kinds of actions are allowed: create an alarm event, log information, restart the network agent, change attributes of the hardware elements of the router, as well as, possibly, other elements on the network.

It should be noted, DDDL was designed to describe router hardware structure and special data structures including various configuration information. DDD does not actually let the user specify software's control flow, whereas DDD specification is not a closed executable specification although it includes some behaviour facilities. Moreover, various parts of DDD specification generate various assets, including data for the router database, C data structures and function signatures, etc. But generated C code is not closed and ready to be executed. A significant part of device management code is implemented manually.

Thus, it can be said the DDD is a *declarative* domain-specific language. It should be stressed we do not imply logical programming facilities, but take into account to the fact that system code generated on DSL is not closed and consequently executed. A lot of other code is needed to execute it, and this additional code is developed outside the suggested DSL.

Nevertheless declarative DSL could contain some part, which have executable semantic and may be launched in some simulation environment. This simulation (debugging is a special case of such simulation) may have a sense for DSL users helping to clarify dark corners of the DSL specification or finding errors.

The complete grammar of DDDL is an Extended Backus-Naur Form (EBNF), which was created via XText [4]. Based on this grammar, an IDE language server is generated. DDD language server is integrated to Visual Studio Code, where an IDE interface is implemented. Visual Studio Code as a target environment is an external requirement to DDD.

3. Debug Concept and Debbuger Use Cases

In our case, we need a way to execute an event-based specification for a single component — that is, the device management agent. The behaviour of this agent is set using the behaviour model defined for the product with DDD tools. The device management agent receives

events from outside — as in, from the network, as well as from the hardware of its router. In addition, the agent can create events for itself and process them itself too.

Being dependent on the environment, the device management agent must correctly process events received from it. It is this aspect that is interesting from the point of view of the debugger, since the processing of one external event is a purely internal matter of the device management agent, and it does not require any additional data from outside. Thus, emulation of receiving such an event could be the start of a debug section run by the developer in order to test the agent's handling of it. It is important to understand that the agent can be in different states, in each of which it must correctly process such an event. For example, it can receive a request from the network for reconfiguration and router restart either in a normal, regular state, or in a state of reduced bandwidth. Accordingly, two different rules are required to process the same event, and they correspond to different specifications of the initial state of the agent and different debug sessions.

During the processing of a single external event, the device management agent can activate more than one rule. This happens via the mechanism of the agent creating events for itself, searching for a suitable rule and executing it. Accordingly, the debug session ends when all rules are executed, and the device management agent message queue is empty.

Let us explain why the device management agent generates events for itself. It is due to the fact that the behaviour model is composite: different rules are created at different levels of the product's decomposition, for example, at the level of chips included in the board, or at the level of ports. Specifying chips and ports, it is important to determine how the processing of various events addressed to them takes place. At the same time, the exact origin of these events is not considered — be it the network or the top level of the device management agent. These rules can also be created by different developers responsible for managing different hardware units of the router. Moreover, the same rule can participate in various scenarios, and in this case rules are used for behaviour decomposition and reuse.

Note also that the behaviour model may differ for different configurations of the product, since they may include different types of equipment.

We have identified the following DDD debug use cases:

- Exploring the product configurations for a specific customer without a target platform, i.e. on a DDD developer workstation.
- Considering a subset of product configurations during DDD development to detect possible bugs. It is important to find bugs exactly on the development level they are made on. If these bugs are detected on the following development levels, the cost of bug detection will increase.
- Analyzing a specific product configuration in the situation when some bug occurs. It could be possible that the reason for the bug is contained in the DDD specification. If it is not so, the next development level should be explored.

4. Debug Model

In order to run a debugger on a behaviour model of the product, it is required that the user precisely defines the debug scenario: product configuration, current state, and debug event. This is done with the DDD language, which has been suitably extended for this purpose.

In order to define the hardware product configuration used in this debug scenario, the appropriate moduleTypes defined in the main DDD product specification are instantiated and the relationships between these instances are specified. The latter means that cards are inserted into appropriate slots of boards and possibly other cards. By this means, a tree of real devices of the product is built. All necessary attributes of each device from this tree are then set — DDD has also been extended for this purpose.

State of product configuration refers to setting values attributes, specifying the required current state of the product configuration.

A debug event specifies the start event that triggers the debug scenario.

Below is a simplified example of a debug scenario for the case of "restarting the router when the voltage in the system drops". This scenario is described in the special debug_scenario1 package, which imports the core package of this product, containing the definitions of the main moduleTypes of the product.

The composition section describes the product configuration, which consists of the main_board1 and card1 inserted into the main_board1 in a slot called card_slot1. Note that the voltage sensor is installed on the card, as follows from the type description of this card in the main DDD specification of the product. Further, it is indicated that there is one external 100 Gbit port port1, into which the split4_25 optical converter is inserted, splitting this port into four 25 Gbit ports.

Further, in the attributes section, the state of the specified product configuration is set: main_board1, card1, sensorT have the "ready for operation" status, and card_slot1 is connected to power; sensorT also has a valid value of 12; the first of the 25 gigabit ports is activated (i.e. through it, the router communicates with the network).

Finally, in the event section, the event that triggers this debug scenario is set: the voltage measured by sensorT

becomes invalid (of value 9, but interval allowed is from 12 to 15).

The behaviour model has a rule which is activated when the voltage is below 12, see Figure fig:graph. It is triggered by the changing sensor's attribute from 12 to 9. In the context of this rule an alarm "Low voltage" is exposed and another event is created. The last is done by changing the attribute card1.port1.port25GE.IS_AVAILABLE from 1 to 0, meaning the active port is disabled. The second rule create alarm "Port is down".

```
package degug_scenario1 {
  import core;

  composition {
    BoardHardType main_board1;
    CardHardType card1;
    main_board1.card_slot1 <- card1;
    card1.port1 = split4_25;
  }
  attributes {
    main_board1 {
      IS_AVAILABLE = 1;
    }
    card1 {
      IS_AVAILABLE = 1;
    }
    main_board1.card_slot1 {
      POWER_STATUS = 1;
    }
    card1.sensorT {
      IS_AVAILABLE = 1;
      VLT_CURR_VALUE = 12;
    }
  }
  override attributes {
    card1.port1.port25GE {
      IS_AVAILABLE = 1;
    }
  }

  event {
    modify card1.sensorT
           : VLT_CURR_VALUE = 9;
  }
}
```

5. Visualization debug results

Let us now consider the graphical model-based notation for visualization of the debug execution trace. As mentioned above, such a trace visualizes the step-by-step execution of the rules involved in the debug scenario. Figure 1 shows an example of such a diagram. It starts with a Start symbol (double circle filled in blue inside).

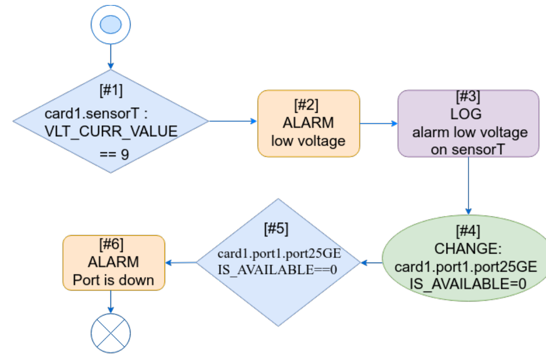

Figure 1: An example of graphical model-based notation for visualization of debug execution trace

It is followed by the first event that triggered this scenario. Note that events in the DDD behaviour model are changes of the attributes of the device database on the router. The corresponding router devices are subscribed to changes of certain attributes, therefore, these devices have rules that start with this event. Device management agent combines all of these rules to whole behaviour model as described above. There can be multiple rules for handling the same event, but then they must differ in conditions that immediately follow the event. An event is denoted by a blue diamond.

Further, the brown rectangle denotes an alarm, the lilac one — logging, and the green oval indicates network device attribute changes. These changes, in turn, can cause further events to be fired for which a suitable rule is found. After the execution of the last rule, the end symbol of the debug scenario is drawn — a circle with crossed lines. At the top of each graphical symbol, except for the start and end, the step number is indicated. The user executes the debug scenario step by step, and as a result of each step, the corresponding graphic element is drawn in the diagram.

6. Debugger Implementation

The debugger implementation scheme is shown in Figure 2. The debugger is divided into two parts: the Debugger Back End, which performs debugging and is integrated into the DDD language server, and the Debugger Front End, which implements the user interface and is integrated into the Visual Studio Code DDD plugin. These parts interact via the standard Debug Adapter Protocol, which passes debug commands from user to debug back end and debug information (attribute values. etc.) from back end to user the user to view.

The main difficulty was the implementation of the Debugger Back End. It consists of the following compo-

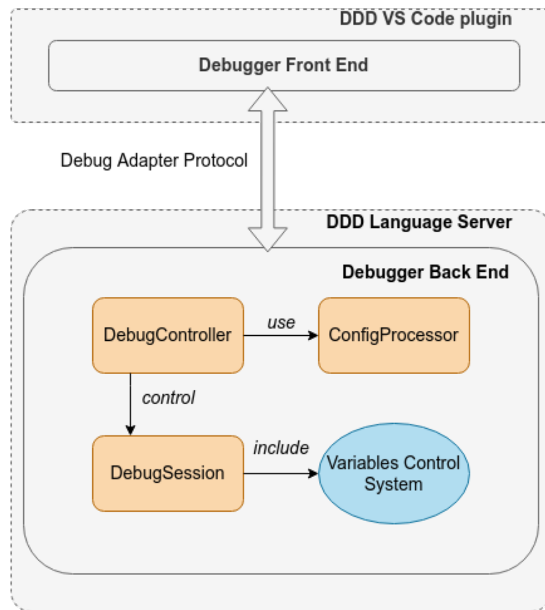

Figure 2: Debugger implementation schema

nents: ConfigProcessor, DebugController, DebugSession, Variables Control System.

The *ConfigProcessor* component processes DDD-specification of the debug scenario DDD specification or the whole product, transforming them into a convenient representation: namely, the device tree of a given product configuration based on hardware connections. This abstraction provides a structure that uniquely defines the "parent-child" relationship, which is important for searching in the behaviour model.

The *DebugController* component connects the Debugger Front End and Debugger Back End, providing an API to initialize the debug session. When a request is received to start a debug scenario, the DebugController processes the incoming debug configuration using the ConfigProcessor, and creates an instance of the DebugSession based on the received data. Next, the controller redirects the request received from the front end to the DebugSession, and upon completion of the action sends the result back to the Debugger Front End side.

The *DebugSession* component is the main debug engine. It implements various debugging steps, and also provides control over the storage and updating of data that is relevant for each step. Unlike general-purpose languages, where the program, as a rule, is executed on some hardware device, DebugSession simulates the entire execution process. Thus it is easy to support the rollback of steps, which is a difficult task in the general case.

The *Variables Control System* component is a collection of classes responsible for storing, processing and transforming debugging information. The tasks of this component are the following: ensuring correct persistent storage of values and attributes of the router; splitting data into stack frames corresponding to the debug state at a certain step; serialization of objects into a representation that specifies the nodes of the debug graph. Thus, the component acts as a universal delegate for working with data stored during debugging.

7. Related work

The need for debugger development tools for DSLs is recognized by the community. Due to this, XText [4], GEMOC Studio [5], and MPS [6] as well as other DSL environments support meta-debug facilities. However, these facilities are oriented at executable DSLs, which have strict executable semantics and can be generated into Java and other industrial programming languages. Very often in this case, a two-level debug model is used [7]. It means that real debug is performed for generated DSL code, and special tools just raise debug information to the DSL level and accept the corresponding user commands from there. This approach is not suitable for our case of various program assets being generated according to the DSL specification, as they do not form a closed executable application.

There are studies on creating meta debug facilities for more complex cases by declaratively specifying executable semantics of the DSLs [3, 8]. However, these studies are at their pilot stages and cannot be employed in the industry. In addition, using this approach, it is difficult to express event-oriented executable semantics, which is important for our case.

Event-oriented debugging is implemented in a series of model-based development toolsets for real-time systems such as YAKINDU [9] and Rhapsody [10]. Such toolsets support UML statecharts and provide facilities for debug statecharts inside of the modeling environment. But, first, these solutions are deeply integrated into the toolsets and can not be reused. Second, they are oriented at the UML-based system structure (components, interfaces, ports, channels, etc.). In practice, they provide execution and debug for a set of communicated components including statecharts. This execution model is redundant for our case, since we are executing a fragment of one component. In addition, we have a significantly different structure model.

So, we can conclude that creation of debuggers for declarative industrial DSLs is an open task that does not have a ready-made solution. Separate tools can be used for solving it, for example, the Debug Adapter Proto-

col and templates for creating the debugger front end. But the majority of work is in specifying the executable semantics for that part of the DSL that makes sense to debug, as well as support the corresponding executable environment in the DSL IDE.

8. Conclusions

In this paper, we have proposed a debugger for the DDD declarative language, which is intended for the development of device management components of a router product line of the XXX company. As a continuation of this work, we plan to focus on increasing the number of actions used in the rules, as well as adding support for new features of the behaviour model that will be introduced in the future.

References

[1] P. Clements, L. M. Northrop, Software product lines - practices and patterns, SEI series in software engineering, Addison-Wesley, 2002.
[2] The reference is hidden for the purpose of anonymization, 2021.
[3] R. T. Lindeman, L. C. L. Kats, E. Visser, Declaratively defining domain-specific language debuggers, in: E. Denney, U. P. Schultz (Eds.), Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011, ACM, 2011, pp. 127–136.
[4] Eclipse Project, XText, 2022. URL: https://www.eclipse.org/Xtext/.
[5] GEMOC, 2022. URL: https://gemoc.org.
[6] MPS: Meta Programming System, 2022. URL: https://www.jetbrains.com/mps/.
[7] M. Kartashov, Two-level debugging, System Programming 1 (2005) 348–365(In Russian).
[8] A. Chis, M. Denker, T. Gîrba, O. Nierstrasz, Practical domain-specific debuggers using the moldable debugger framework, Comput. Lang. Syst. Struct. 44 (2015) 89–113.
[9] Itemis AG, YAKINDU, 2022. URL: https://github.com/Yakindu.
[10] IBM, Rhapsody, 2022. URL: https://www.ibm.com/docs/en/rhapsody.