Using Alloy and UML/OCL to Specify Run-Time Configuration Management: A Case Study

Geri Georg

Agilent Laboratories, Agilent Technologies, 4380 Zeigler Road,
Fort Collins, Colorado 80525
geri_georg@agilent.com

James Bieman

Computer Science Department, Colorado State University,
Fort Collins, Colorado 80523
bieman@cs.colostate.edu

Robert France

Computer Science Department, Colorado State University,
Fort Collins, Colorado 80523
france@cs.colostate.edu

Abstract: There are many different ways to specify the requirements of complex software systems, and the optimal methods often vary according to the problem domain. We apply and compare two languages, UML/OCL and Alloy, to specify a problem in one domain, the run-time configuration management of a loosely coupled distributed system, to determine which is more appropriate for this domain. The specific problem that we specify in the case study involves the run-time configuration management of an Asynchronous Transfer Mode / Internet Protocol (ATM/IP) Network Monitoring System. Neither Alloy nor UML/OCL supports the specification of key temporal aspects of the problem. This paper addresses the representation of requirements specification; continuing research will compare the usefulness of the specifications for modeling and design purposes.

1 Introduction

There are many different ways to specify the requirements of complex software systems, and the optimal methods often vary between different problem domains. We apply two languages, UML/OCL ([FK97], [OM01], [WK99]) and Alloy ([Ja99], [Ja00a], [Ja00b], [Ja01c], [JSS00]) to the domain of run-time configuration management of a loosely coupled distributed system.

Run-time configuration management refers to physical changes made to a loosely coupled distributed system once it has been deployed and is executing. From time to time in a large system, changes in computers, measurement devices, software applications, etc. must be made while the overall system continues to run. A common example occurs when a new application or service is created that needs to be deployed across the system and its execution begun while other parts of the system continue to execute properly. In a large distributed system changes like this must also be accomplished in the context of unexpected network errors that inhibit communication between computers or measurement devices.

The run-time configuration of loosely coupled distributed systems must be flexible and extensible. Flexible means that it is possible to logically or physically move portions of the system. Extensible means that it is possible to create a new entity by logically grouping resources, or to physically add an entity to a computer. Yet, in our industrial experience, the specification methods often used to describe system requirements are UML class diagrams (without accompanying OCL constraints) and/or structural box diagrams along with textual descriptions. The resulting specifications do not specify system run-time configuration management flexibility, extensibility, or any other system-wide behavior. As a result, many distributed systems do not meet their run-time configuration management flexibility and extensibility requirements, and these systems often require large amounts of engineering resources to maintain and enhance.

The notation used to specify run-time configuration management requirements of these systems must be expressive, yet easy to use for both a system architect developing the architectural models of the system, and system designers/implementers who are involved in the actual creation of an implementation of the system. This paper discusses the first part of this problem, choosing a notation that is expressive and easy for a system architect to use. The second part of the problem, ensuring that the notation is precise and comprehensible to system designers and implementers, is the subject of on-going research.

We use the run-time configuration management requirements of an existing Asynchronous Transfer Mode/Internet Protocol (ATM/IP) Hybrid Network Monitoring System to compare two specification languages. We use UML/OCL (initially as implemented in the USE tool from the University of Bremen ([RG00]), then subsequently the full UML/OCL language [OM01])[1] and Alloy (version 1.0 from MIT [Ja00b]; a newer version of Alloy has been proposed but is not yet available as part of a validation tool [Ja01a], [Ja01b]) to specify the run-time configuration management capabilities of the case study system. Both of these tools are currently research tools.

UML is a popular modeling language widely used in industry, however in our industrial experience, the typical models consist only of static class diagrams without OCL constraints. These diagrams do not contain enough information to completely specify run-time configuration management requirements. Constraints can be expressed using OCL, although, in our experience, formal constraints are rarely used in industry, except in domains where perfect operation is required (e.g. medical and aviation domains [Ha96]) and then formal specification and design methods are often used and automatically verified for proper behavior. We are interested in evaluating the usefulness of OCL to complete UML diagrams in an industrial software environment.

Alloy is a modeling language designed to provide precise semantics for specification and modeling purposes. We selected Alloy because of its reputed expressive power and

[1] Additional OCL tools have been brought to our attention since this research began, however the USE tool is the only one discussed in this paper.

because a tool is available that can validate its models. The Alloy tool provides syntactic as well as invariant and operation validation of an Alloy model. We are also interested in evaluating the usefulness of Alloy to complement UML diagrams in an industrial software environment.

The paper is structured as follows. Section 2 provides a high-level description of the Agilent Technologies ATM/IP monitoring system and its run-time configuration management requirements. Section 2 describes the problem domain context for interpretation of the models presented in subsequent sections. Section 3 describes the UML/OCL specification of a portion of the run-time configuration management needs of the monitoring system. This section also contains an explanation of the kind of UML models that are used as well as some explanation of the notations used in those models. Section 4 describes the corresponding Alloy specification. Section 5 compares the use of the two languages to specify the dynamic configuration management of the ATM network monitoring system. Finally Section 6 presents conclusions.

2 An Asynchronous Transfer Mode/Internet Protocol (ATM/IP) hybrid network monitoring system; the problem domain context for model interpretation

The Agilent Technologies *accessATM/IP Network Monitoring System* is an extension to a previous internet protocol–only network monitoring system (*accesSS7*). *AccessATM/IP* can monitor the behavior of a hybrid ATM/IP network. For example, it monitors quality of service (e.g. delay, lost packets, peak bandwidth, etc.) between various points of the network. Hardware probes are connected to local area networks (LANs) at various local monitoring sites, which also contain site server computers and routers. Data is consolidated at the geographical level at these local sites. Operators can access the monitored information that has been collected throughout the monitoring network from workstations, as well as direct the operation of the overall system. A simplified version of this entire system is shown in Figure 1.[2]

Hardware monitoring probes in the monitoring system must be located outside the core ATM portion of the network, due to the high speed of the core (up to OC192 speeds, approximately 10Gb). Network speed at the probes is approximately 155Mb, or OC3 speeds. (See [Za99] for a description of ATM networks.) Probes are connected locally to monitoring system site server computers that perform the analysis of the header information they obtain, and these local monitoring networks are further connected via wide area networks (WANs) to a central server. Workstations at the central site of the monitoring system control the observations of the probes and the analysis of header information at local sites. The monitoring system may be comprised of tens of thousands of probes with each local site having on the order of tens of probes. There is one probe for each link being monitored in the system.

[2] Graphic developed with the help of G. Pollock and D. Thompson of Agilent Laboratories.

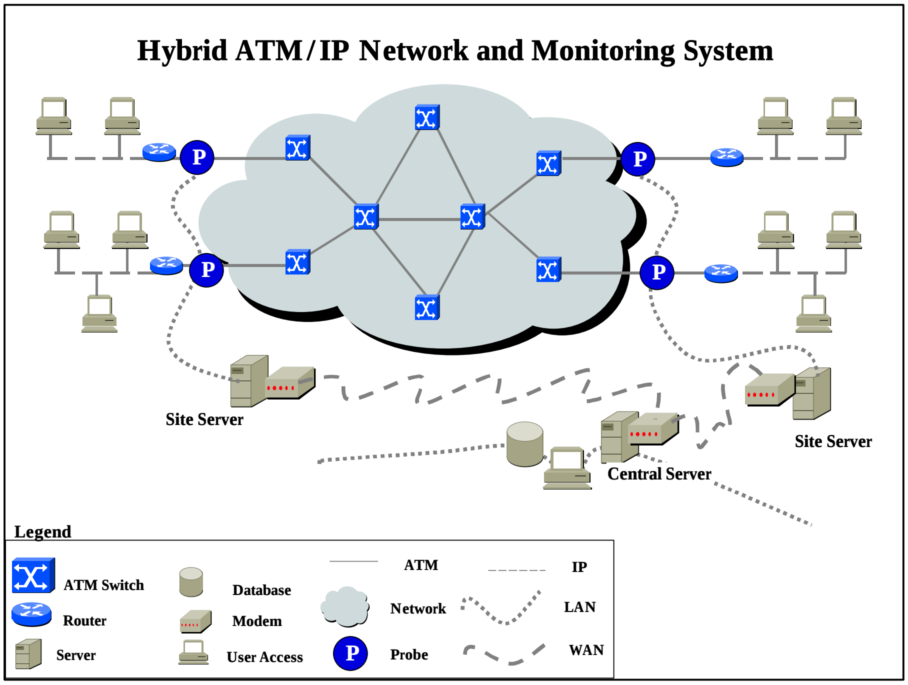

Figure 1. A simplified hybrid ATM/IP network and its monitoring system.

In the ATM/IP network monitoring system, probes that capture packet headers are physically attached to the network. Relevant information is stripped off and analyzed (looking for patterns) at site servers. The site servers are connected via LAN or WAN (depending on their geographical locations), to a central server computer where final analysis and decision making occur. Due to the huge amount of data in this system, a primary task is to perform as much processing as possible at the source of the data in order to cut down on the amount of information that has to pass among computers at different "levels" and geographical locations in the system. The probe data can be mined for many types of information, such as quality of service, connection information for billing purposes, and also patterns that indicate fraudulent use of the network. Data is static and also time-sensitive; after some relatively short period of time it has no value.

The system is very dynamic; it can be changed from one that monitors network performance to one that searches for fraudulent use by changing the recognition patterns used in the system. This transformation requires changing applications that are running on the various local site computers. Since the computing and probe system is very large (typically thousands of probes and hundreds of computers are involved over large geographical areas), the task of getting the correct version of the correct application to the correct computer presents a significant challenge. A related problem is getting the correct services/applications operating on the correct computers, in the correct sequence. For example, if a probe is to begin searching for a new pattern, the local computer must already be running the new correlation and analysis program before it can correctly analyze new data from the probe.

Another interesting aspect of the monitoring system is that since it is using many networks (including LAN and WAN) that are geographically dispersed, portions of the overall monitoring system may be operating or not, solely depending on network status. New probes/computers may also be added at any time, further complicating the situation. Whenever the monitoring system topology changes, the system must somehow determine its current configuration, determine what software needs to be in place on the changed parts of the system, and effect configuration changes to make all parts of the system run properly.

In order to focus this case study, we only specify run-time configuration management needs of software items. We focus on moving particular pieces or versions of software around the system, instantiating them on particular machines, placing them in particular states, etc. There are also temporal constraints on run-time configuration management. These are:

(1) An attempt to move toward a new desired configuration must not take place unless a minimum time has passed since the last configuration change began. (This allows the entire system to converge on a particular desired configuration.)

(2) Configuration changes must happen within a defined acceptable time.

3 The UML/OCL Specification

We use a *type* model to specify run-time software configuration management. The model is accompanied by a completely textual OCL specification of additional constraints and operations. Two portions of the model are presented and discussed in this paper. The entire model is too large to be included in this paper.

3.1 Model choice and notation

A *type* model consists of *concepts*, UML *types*, and the relations between them. In the UML, a "…Type is used to specify a domain of objects together with operations applicable to the objects without defining the physical implementation of those objects" ([OM01]). A *concept* is a domain of objects. (See [La98] for a discussion of *concepts*.) We use the problem domain to identify *concepts*. Once we identify operations related to a *concept*, it becomes a *type*. For this case study problem, we are only interested in partial operation signatures and pre- and post-conditions. We are not interested in specific return values of operations at this level of abstraction, so the operation signatures do not include return values.

We show behavior in two ways. First, operation signatures (excluding return value types) are shown in the <<*operations*>> section of the concept in the *type* model. Second, a state transition diagram is shown in the type's <<*state machine*>> section. These two sections show the operations as well as their effect on the state of the object. OCL descriptions of the operations use information from the state diagram, as shown in Figure 3.

3.2 UML/OCL Type Models

Figure 2 shows the configuration management *types*, *concepts*, and their relations. Figure 3 shows the operations and state transitions effected by the operations of *RunnableItem*.

There are five basic *types* that need to be managed in the model of configuration management software items. These are *SourceCode*, *ObjectCode* (created to run on a particular combination of a machine type and a compiler, called a *Platform*), *Object* (this is a concept of the problem domain, not an instance of a *type*), *Service* (a related set of operations that provide functionality to multiple applications), and *Application* (an *Application* makes use of services to achieve a functional goal). Depending on the particular application, not all of these software items may exist in the system, but in all cases there will at least be source code, object code, and some sort of executing code. All five *types* of software exist in the ATM network monitoring system. We show these *types* as specializations of *ConfigItem.*

Configuration changes are made to executing or executable applications, services, and objects in the ATM/IP network monitoring system. These are specializations of *RunnableItem*. *RunnableItem* contains the operations needed to perform these configuration changes (see Figure 3).

Figure 2 also shows a configuration service, *ConfigService*. *ConfigService* is a specialization of *Service*. Since it performs configuration management operations on software items, it has additional operations that must be implemented (e.g. *compare-change*, which is shown in Figure 3). Any particular *ConfigService* has an association with a single *LocalSet*. A *LocalSet* is a partition of the overall system that allows dynamic configuration management.

In order for an application, service, or object to be successfully executed in the ATM/IP network monitoring system, there are often other applications, services, or objects that must also execute. For example, an application may need to use a trading service to find a particular kind of object. In this case the application is dependent on a service as well as an object. This kind of dependence leads to the notion of a dependency set, and the *dependsOn* relation. This relation is used to define the dependency set for each kind of configuration management software item in the system in Figure 2. It is shown as a relation for *ConfigItem* (with the roles *CIDependsOn* and *depCI*), and is further defined in the <<*constraints*>> section of the other *types*. Specifically, the dependency relation is constrained so that *Object* types can only depend on other *Object* types, *Service* types can depend only on *Object* types and other *Service* types, and *Application* types can depend on *Object* types and *Service* types.

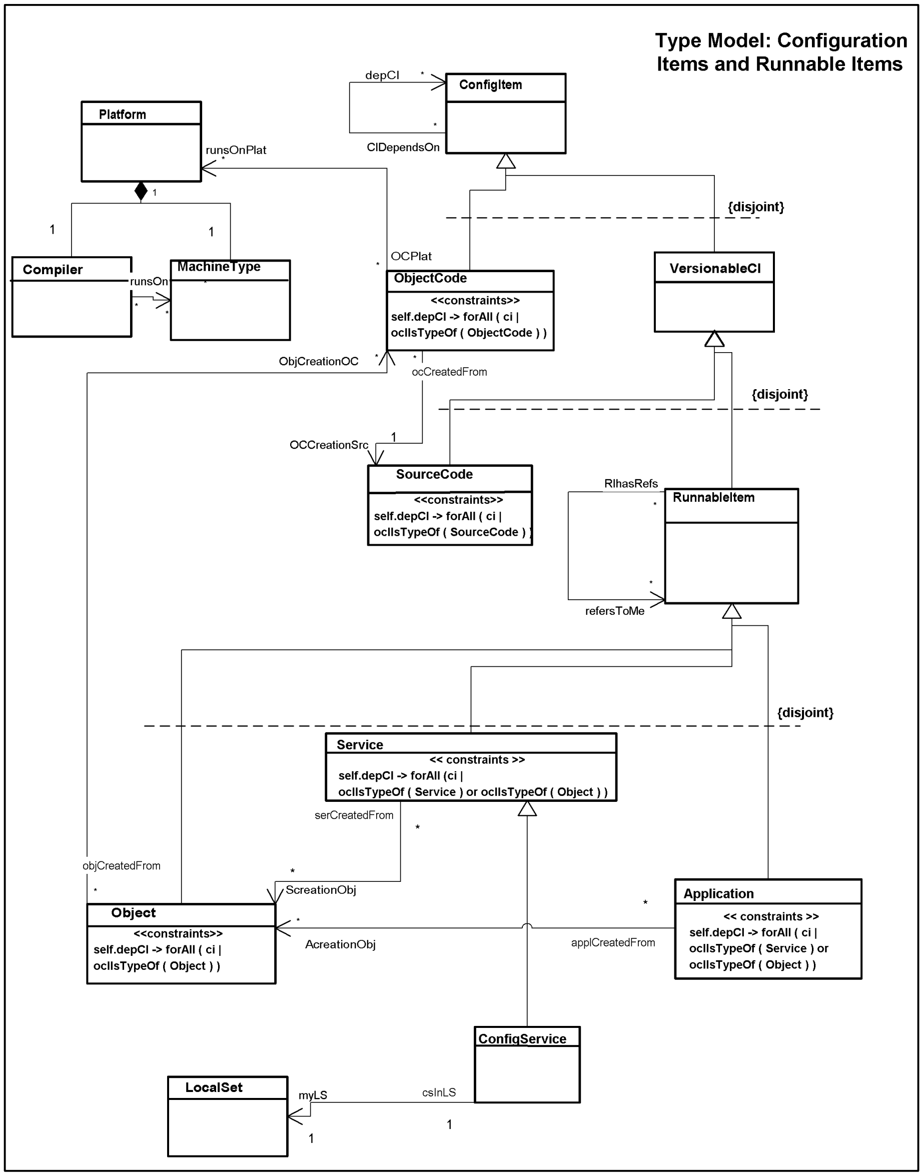

Figure 2. A *type* model containing *concepts,* UML *types,* and associations of the case study system that need to be dynamically configured.

Figure 3 shows the operations of *RunnableItem*, both in terms of calling arguments (<<*operations*>> section) and operation effect on state (<<*state machine*>> section).

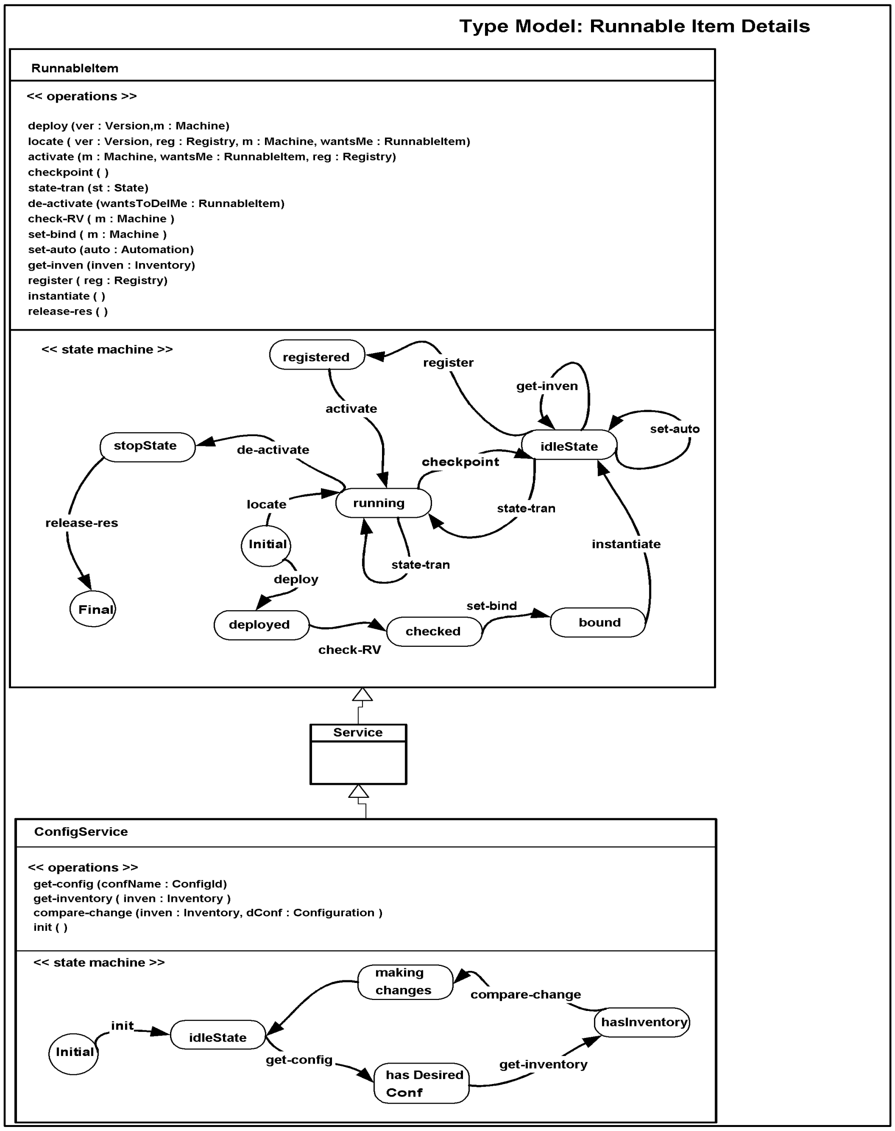

Figure 3. Operations provided by and state machine details of *RunnableItem* and *ConfigService*.

Figure 3 shows the operations needed by each *RunnableItem* that allow it to be dynamically configured. The <<*operations*>> section shows the calling signatures of the operations and the <<*state machine*>> section shows how each operation effects state. Transition lines are labeled with operation names that effect the state transition.

Figure 3 also shows the operations and state machine for *ConfigService*. *ConfigService* is a specialization of *Service*, and is responsible for the dynamic configuration management of the runnable items in the system. It therefore has additional operations that need to be defined. These operations use the other operations defined in the *RunnableItem* type to accomplish dynamic configuration management. The state machine section of *ConfigService* shows the *making changes* state, which is equivalent to the *running* state in *RunnableItem*.

4 The Alloy Specification

Alloy is a modeling language ([Ja99], [JSS00]), and is supported by an associated tool, that tests assertions, invariants, and operations of an Alloy model. Version 1.0 of Alloy is discussed in this case study. A second version has been proposed, but its associated tool is not yet available and so it is not considered in this case study ([Ja01a], [Ja01b]).

An Alloy model cannot contain a UML diagram. The entire Alloy model must be built textually using its associated tool. An Alloy model consists of several parts. A key part is an initial list of sets that are fundamental to the model (the *domain* statement). Other sets can then be defined in terms of these initial sets, and relations between sets can also be created (the *state* statement). Definitions of the relations are included in the model (*def* statements), as are invariants (*inv* statements) and assertions (*assert* statements) regarding the sets and their relationships. Finally, operations (*op* statements) that are performed on the sets are also defined.

Figure 4 shows a portion of the Alloy model for the dynamic software configuration management of the case study system. This figure shows the structure of an Alloy 1.0 model, as well as showing some example relations (e.g. *platformMT*, *platformC*, and *dependsOn*), invariants dealing with a *LocalSet*, and an operation (*deploy*). This subset model compiles and is shown to be consistent using the Alloy tool.[3]

The subset of the full Alloy model shown in Figure 4 highlights some of the structural components of Alloy. The *domain* and *state* sections are shown, along with definition (*def*) and invariant (*inv*) statements. An operation (*op*) is also shown. Notation used in the *state* section includes *partition* (subsequent names are subsets of the name after the colon, do not share any members, and are its only subsets), colon (the first name is a subset of the second name), and exclamation mark (singleton set). The model components shown include the *platformC* and *platformMT* relations (required to define the *Platform* tuple), the *dependsOn* relation (initially defined as existing between *ConfigItems* and then explicitly defined in a *def* statement), and invariants (*oneCSControls* and *oneCSControlsAll* which are used to specify independent configuration of local sets). In the operation definition, primed values of state components indicate the post state of the variables that are primed. The notation

[3] The Alloy tool creates a boolean satisfaction formula from an Alloy model, and assigns a scope to the formula. The analysis determines if an instance exists for the formula within the scope (the number of elements in each domain set). If one does exist, then the Alloy model is consistent. The tool can also be used to look for theorem (assertion) counterexamples that indicate model inconsistency. Failing to find an instance of a formula does not necessarily indicate that the model is inconsistent; it may simply have an instance at a higher scope. Similarly, failing to find a counter example to an assertion does not mean that the assertion is consistent; a counter example may exist at a higher scope.

+createdFrom indicates one or more uses of the *createdFrom* relation, in order to reach the associated *ObjectCode* set (which has the relation *developedToRunOn*). The *//* notation is used to begin a comment. This Alloy model defines a *LocalSet* based upon a partitioning of the machines on which *RunnableItems* are executing. The full Alloy model of the dynamic software configuration management of the ATM monitoring system is available on-line at http://www.cs.colostate.edu/~georg/alloy-atm.pdf.

```
model ATM-monitoring-CM-example {
domain { Machine, LocalSetId, Platform, Compiler, MachineType, ConfigItem, OpAttempted, ReturnValue,
Version }
state {
    partition SourceCode, ObjectCode, Object, Service, Application : ConfigItem
    NotOCode : ConfigItem
    RunnableItem : ConfigItem
    ConfigService : Service
    partition deployed, located, bound : OpAttempted
    OK : ReturnValue
    Error : ReturnValue
    controlledBy (~controls) : RunnableItem -> ConfigService !
    on  (~riOn) : RunnableItem  -> Machine !
    LocalSet (~lsRI) : RunnableItem -> LocalSetId !
    LocalIndex (~mach) : Machine -> LocalSetId !
    dependsOn : ConfigItem  -> ConfigItem
    createdFrom : ConfigItem  -> ConfigItem
    platformC : Platform -> Compiler !
    platformMT : Platform -> MachineType !
    FeedBack : OpAttempted -> ReturnValue
    hasMType : Machine -> MachineType !
    hasVersion : NotOCode -> Version !
    developedToRunOn : ObjectCode -> Platform }

 def NotOCode  { NotOCode = SourceCode + Object + Service + Application }
 def RunnableItem   { RunnableItem = Object + Service + Application }
 def LocalSet { all ri | ri.LocalSet = ri.on.LocalIndex }
 def controlledBy  { all ri : RunnableItem | one cs : ConfigService | ri.controlledBy = cs -> ri.on.LocalIndex =
cs.on.LocalIndex }
 def dependsOn { all ci | ( no ci1 : ConfigItem |ci.dependsOn  = ci1 ) }        // The complete definition is too
long to include in this figure.
 def createdFrom { all ci | ( ci in SourceCode -> ci.createdFrom = ci ) && ( ci in ObjectCode -> one src :
SourceCode | ci.createdFrom = src ) && ( ci in Object -> some oc : ObjectCode | ci.createdFrom = oc ) && ( ci
in Service -> some obj : Object | ci.createdFrom = obj )  && ( ci in Application -> some obj : Object |
ci.createdFrom = obj ) }
 inv oneCSControls  { all cs1, cs2 : ConfigService | no ri : RunnableItem |  ri in cs1.controls & cs2.controls
&& ! ( cs1 = cs2 ) }
 inv oneCSControlsAll { all lsid : LocalSetId | one cs : ConfigService | cs in lsid.lsRI && lsid.lsRI =
cs.controls }
 op deploy ( ri : RunnableItem !, ver : Version !, m : Machine ! ) {
    m.hasMType in ri.+createdFrom.developedToRunOn.platformMT
    ri.hasVersion = ver
    ri.on' = m
    deployed.FeedBack' in OK  }
}
```

Figure 4. A portion of the Alloy model of dynamic software configuration management for the ATM network monitoring system.

Some observations about the Alloy model include:

- The *type* model (excluding state machines and OCL expressions) must be duplicated in Alloy before the specification can be completed using Alloy. While this is time consuming, it does seem to be possible to follow a method to create an Alloy model from *type* models, which also implies that an automated tool could be developed to aid in this initial process.
- We developed an Alloy model from an earlier set of UML *type* models. These initial models did not include any OCL constraints nor state machine descriptions. Development of the Alloy model pointed out changes that needed to be made to the UML models. In addition, as the UML models were refined, changes that needed to be made to the Alloy model became clear. Thus, the effort of developing both models semi-concurrently was very synergistic, and resulted in better models.
- In modeling the case study system it is important to be able to declare relations and refine their definitions as the problem becomes clearer. Alloy allows relations to be declared, then to be precisely defined using the *def* notation. The *dependsOn* relation is initially declared as a relation between *ConfigItems*. The *def* (definition) notation is then used to precisely state the nature of this relation between applications, services, objects, object code and source code. We accomplish the same thing in the *type* model using the *dependsOn* relation with roles *CIDependsOn* and *depCI* for the ConfigItem generalization. OCL constraints precisely define this relation for applicable specializations.
- One problem with using version 1.0 of Alloy is that it requires one relation for each part of an aggregation. To model the *Platform* aggregation, two relations are added to the Alloy model. Thus, the relation *platformC* is included in the Alloy model to map between a *Platform* and a *Compiler*, and the relation *platformMT* is added to map between a *Platform* and the *Machine* type it represents. If Alloy supported tuples directly, these multiple relations could be replaced by a single relation.
- System-level constraints on behavior can be included in the Alloy model. For example, the notion that local sets can work independently, moving toward a new desired configuration can be included in the model as a set of invariants associated with local sets and configuration services.
- The temporal constraints of configuration management operations noted in Section 2 are not specified in the Alloy model since Alloy has no direct notion of time. (See Section 5 for a discussion of this problem with respect to both UML and Alloy.)

5 Comparing UML/OCL and Alloy for Run-Time Configuration Management Specifications

We compare UML/OCL and Alloy based on using them to specify the software run-time configuration management requirements of the ATM network monitoring system. This comparison does not include a deep analysis of the two languages. It also does not address open issues such as the understandability of specifications written in the languages, nor the ability of those specifications to lead through design and implementation to systems that exhibit the required behavior. These are topics for future research.

- OCL/UML allows development of models containing *concepts* and *types*, relations, and operations. It has built-in data types (e.g. Integer and Set) and operations on them (e.g. mathematical operations on Integers and operations such as union and iterate on Sets). These built in data types and their associated operations can cause a subtle shift in the viewpoint of the modeler to the more concrete aspects of a system, thus potentially turning specification into design. Operations like *iterate* exacerbate this problem. The shift from abstraction to concrete aspects is less likely to occur when the modeler only uses sets to model the system (as when using Alloy), although no doubt experience can help keep the specification process at an abstract level. We experienced this problem when writing OCL post conditions for operations like deploy. The deploy operation must ensure that the runnable item being deployed was built for the machine type where it will be running. This can be accomplished by iterating through its associated object code, however there are no doubt many ways to accomplish the check. An OCL expression that uses the *iterate* operation can be interpreted as the correct design, precluding other methods that accomplish the machine type check. A more abstract expression uses the *forAll* operation on the associated object code.
- Alloy does not include the notion of sequencing. Thus, it is difficult to specify that particular operations need to be sequenced. For example, one version of an application may need to be stopped and then another version started, perhaps because the resources used by the first version need to be released before the second version is started. This is the case in the *compare-change* operation of a *ConfigService*, when a request for a different version of a runnable item is part of a desired configuration. It is not possible to express this kind of sequence specification in Alloy. We are forced to create additional operations that compose the post conditions of operations we need to sequence. The language constructs of OCL (e.g. *if-then-else-endif*) allow explicit sequencing of operations, so this is not an issue with OCL.
- The Alloy specification suffers from a structural problem; definitions (i.e. *def* statements) and invariants (i.e. *inv* statements) of relations and subsets are included in the model after the declarations of these relations and subsets, often making it difficult to find associated constraints. In the new proposed version of Alloy ([Ja01a], [Ja01b]), all the information about a set is kept together, thus packaging relevant information.
- The Alloy and UML/OCL models of the ATM network monitoring system do not specify temporal constraints. The temporal constraints in the case study are simple, but they do need to be included in the specification. (These constraints are noted in Section 2.)

  Neither Alloy nor UML/OCL address the notion of time events directly, although UML does define sequence models. But sequence models are not sufficient to specify both temporal constraints of the ATM network monitoring system. One temporal constraint that can be expressed in sequence models is a time bound on individual configuration actions. Such a bound can be shown in UML sequence diagrams, however this capability is not integrated into OCL. However, sequence diagrams are used for instances of classes; they are not used as specifications unless they reference roles. The other temporal constraint, the minimum time between attempts to move to a new desired configuration, cannot be specified with sequence diagrams. This bound needs to be expressed as a time from the start of the last move to a new configuration to the start of the next move to a new configuration.

Another approach to specify temporal constraints is to add additional time-related concepts in the specification, such as temporal logic. We used three different types of temporal logic, and found that the most applicable for this problem were RTL-style occurrence relations. (The temporal logic systems we used are RTL [JMS98], TILCO [MN01], and intervals [MP92]). The RTL-style constraints were easy to develop, but, since event time is not part of UML/OCL or Alloy, we can only add them as comments in the overall model. (Examples of temporal constraints can be seen in the compare-change operation in the complete Alloy model of the case study system. This is available as www.cs.colostate.edu/~georg/alloy-atm.pdf.)

- Tool support for UML/OCL is minimal; research tools such as USE exist, but some only support a subset of the full UML/OCL language. In addition, the USE tool requires a model to be completely written in OCL, rather than allowing a combination graphical/textual representation of a model. Similarly, the Alloy tool requires a completely textual model of a system written in Alloy for its operation.

6 Conclusions

It is not possible to create a complete specification of software item run-time configuration management for the case study using either UML/OCL or Alloy, due to the lack of temporal operations. Other than this limitation, both UML/OCL and Alloy can be used to specify the other requirements of the run-time configuration management of the ATM/IP network monitoring system.

Both UML/OCL and Alloy need to be extended to include support for temporal specifications based on events. The simple time-related needs of this case study specification do not require extensive temporal operations; they are most easily specified with a notation that is based on events (e.g. RTL [JMS98], TILCO [MN01]) rather than intervals ([MP92]).

We prefer the current tool support for Alloy over that of UML/OCL. The Alloy tool implements the complete Alloy language and thus is a good match for Alloy models. It has syntactic, semantic, and analysis support for Alloy models. By contrast, the OCL tool we tested did not support the full OCL language. However, both tools require complete models in a textual format; they do not support any graphical input. While textual formats may be necessary for processing purposes, the automatic translation of graphical elements to text could be used to provide templates for more complete textual models.

In an industrial software development environment, some sort of verification is needed for specification and design models. In our experience, this often consists of peer review, and its quality is directly related to previous knowledge and experience of the participants. In order for modeling languages such as OCL or Alloy to be widely used, additional, more automated verification must be available since most architects/designers are not experienced in these languages.

This case study comparison requires further research to investigate the understandability of specifications develop in the two languages, as well as their ability to lead to system implementations that exhibit the required run-time configuration management behavior. The languages also need to be compared using case study systems in other problem domains.

Bibliography

[FK97] Fowler, M.; Scott, K.: *UML Distilled.* Addison-Wesley, 1997.

[Ha96] Hall, A.: "Using Formal Methods to Develop an ATC Information System." *IEEE Software* March 1996: 66-76.

[Ja99] Jackson, D.: "A Comparison of Object Modeling Notations: Alloy, UML, and Z." Http://sdg.lcs.mit.edu/~dnj/publications.html, August, 1999.

[Ja00a] Jackson, D.: "Automating First-Order Relational Logic." *ACM Foundations of Software Engineering* (2000).

[Ja00b] Jackson, D.: "Enforcing Design Constraints with Object Logic." *Static Analysis Symposium 2000* June/July 2000. Springer Verlag.

[Ja01a] Jackson, D.: "Alloy Revisited." IFIP 2.9, Hawks Cay, Florida, February, 2001.

[Ja01b] Jackson, D.: "Alloy: A Lightweight Specification Language Draft Language Notes." MIT Lab for Computer Science, January, 2001.

[Ja01c] Jackson, D.: "Alloy: A Micro Modelling Language." IFIP 2.3, Santa Cruz, January, 2001.

[JMS98] Jahanian, F.; Mok, A.; Stuart, D.: "Formal Specification of Real Time Systems." TR-88-25. Department of Computer Sciences, The University of Texas at Austin, 1988.

[JSS00] Jackson, D.; Schechter, I.; Shlyakhter, I.: "Alcoa: The Alloy Constraint Analyzer." *Proceedings International Conference on Software Engineerin* June 2000.

[La98] Larman, C.: *Applying UML and Patterns*. Prentice Hall, 1998.

[MN01] Mattolini, R.; Nesi, P.: "An Interval Logic for Real-Time System Specification." *IEEE Transactions on Software Engineering* March/April 2001.

[MP92] Manna, Z.; Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[OM01] OMG. "OMG Unified Modeling Language Specification." V1.4 draft, February, 2001.

[RG00] Richters, M.; Gogolla, M.: "Validating UML Models and OCL Constraints." *Proceedings UML 2000* (2000).

[WK99] Warmer, J.; Kleppe, A.: *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, March, 1999.

[Za00] *Parallel and Distributed Computing Handbook*. Edited by Albert Zomaya. McGraw-Hill, 1999.