ABSTRACT
An important part of the usability of a programming or specification language lies in the presence of supporting tools that are provided with the language, e.g., type checkers, debuggers and simulators. Development of such tools for domain-specific languages imposes a number of specific evolution requirements. Our contribution is twofold: First, we present an MSOS-based approach to automatic generation of formally specified type checkers for domain-specific languages. Second, we report on the application of our approach to Chi, a high level specification language for describing concurrent systems. The resulting type checker has been successfully integrated in the tool chain of the Chi language.

Categories and Subject Descriptors
F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type Structure; D.3.4 [Programming Languages]: Processors

General Terms Languages, Design

Keywords Type systems, MSOS, DSL, Code generation

1. INTRODUCTION
An important part of the usability of a programming or specification language lies in the presence of supporting tools that are provided with the language, e.g., compilers, debuggers and simulators. In the case of domain specific languages (DSLs) development of such tools requires addressing a number of specific challenges: tools should be suited for understanding by domain experts, support evolution and facilitate decision making by language developers.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires specific permission and/or a fee.

In this paper we describe our experiences developing a type checker for a high level specification language, Chi 2.0. The basic methodology we followed is shown in Figure 1. The bottom ellipse shows part of the basic structure of a compiler. The starting point is the source code of the program to be compiled, which is then parsed. The resulting Abstract Syntax Tree (AST) is then processed by the type checker, which produces a Decorated AST (DAST). Here, the word “decorated” refers to additional type information that is added to various nodes in the tree.

A type checker extracts information from the AST according to a type system, a description of the desired behaviour of the type checker. A type system can be seen as a set
of rules mapping AST fragments to a set of types. Traditional approaches to type checker construction fail to meet the requirements above: e.g., manually implemented type checkers are hard to modify, that is, they are ill-suited for evolution or decision-making support. In the case of Chi 2.0, the language developers were unsure if certain language elements were needed and what type rules would be most intuitive for these elements. In addition, they were unsure how certain type system concepts, like widening, should be implemented.

As Chi 2.0 developers had previous experience with specifying program behaviour using formal semantics [5] we proposed to the Chi 2.0 developers to generate a type checker from a formal specification. Our work has further been inspired by the fact that parsers have evolved from being mostly hand-written to mostly generated by grammar-based parser generator [20]. This way, the parser is easier to create and maintain, though it also creates a dependency on the parser generator and the resulting parser might be less customizable, in particular in the format the output is presented in.

As Figure 1 shows, we suggest a similar approach for type checkers. Similarly to the grammar describing syntactical well-formedness rules of a language, a type system is a set of well-formedness rules describing which terms are correct with respect to types. Once the type system has been formally defined, using a type checker generator one can generate a type checker in the same way parsers are generated from grammars by parser generators.

The remainder of the paper is organized as follows. In Section 2 we introduce Chi 2.0, the subject of our case study. In Section 3 we discuss the formalism we have chosen to specify the type system of Chi 2.0, MSOS, and our chosen notation, MSDF. Section 4 discusses our way of producing a type checker implementation based on the type system. Similarly to generated parsers we expect generated type checkers to be easier to create and maintain but more difficult to fine-tune. In Section 5 we review the challenges above and show how they have been addressed. Finally, Sections 6 and 7 discuss lessons learned and related work respectively, while Section 8 concludes and sketches directions for future work.

2. CHI 2.0
Chi 2.0 is a high level specification language for describing concurrent systems. The language has been successfully applied to solve real world industrial-sized problems [29, 6]. Originally created to design control of manufacturing systems, nowadays it is used to design control systems for complex sophisticated machines as well, using model-based engineering and supervisor synthesis. The designers of Chi 2.0 consider formal semantic specifications of languages they use and create to be very important. In particular, the dynamic semantics of Chi 2.0 have been formalized in SOS, a formalism well-known to the developers.

The language allows specification of pure discrete-event behaviour (where changes only take place during events), pure continuous time behaviour (where values of variables are decided by equations), or hybrid behaviour (a combination of discrete-event and continuous time behaviour). An example of the latter is for example a bottle filling line, where a discrete supervisor is used to control a liquid flow. An actual Chi specification consists of one or more parallel processes, that can communicate via synchronous actions, channels or shared variables, and differential equations that describe continuous behaviour.

With each new generation of the language, new concepts are added and existing ones are modified. Also, the application domain of the language is slowly shifting from discrete-event systems to hybrid systems, causing more changes to language syntax and its semantics. While earlier versions of Chi involved manipulations of integers, growing complexity of the systems being modelled required recent addition of such advanced language features as matrices, vectors and alternate algebras as well as support for implicit type widening, overloading, and templates. These changes increase the effort to implement a type checker exponentially. To keep the effort required at acceptable levels, an alternative approach was needed.

3. MSOS AND MSDF: SPECIFYING A TYPE CHECKER
Our approach starts from a formal specification of a type checker. To this end we advocate using MSOS [24] and MSDF [11].

MSOS was developed by Peter Mosses [25] in an attempt to simplify the notation of SOS and to reduce the coupling between rules. In particular, as a SOS specification grows, more and more rules require the knowledge of specific context information, such as lists of declared variables and constants. Such context information is represented in SOS by the *environment*. SOS requires the component structure of the environment to be indicated on each and every rule. Hence, as the number of environment components grows, all rules become more complex, although some of the rules do not need the knowledge of the environment at all. Additionally, if the environment contains components that are not mentioned by a rule, that rule cannot be applied, which makes it harder to combine or modify SOS specifications. MSOS removes this problem by storing environment information using transition labels.

A set of MSOS rules defines an arrow-labelled transition system, where the labels forms the arrows of the category. Note that this implies that transitions can be composed in an associative way and that there are identity transitions for each state. Details about the formal definition of MSOS can be found in [24]. In our previous work [8] we have shown that MSOS is well-suited for specifying type systems of evolving languages.

To represent MSOS rules in a form suited for automated processing we opt for MSOS Definition Format (MSDF) [11]. In addition to MSOS rules, MSDF also supports declarations, formulas and modules. *Declarations* describe algebraic data types and variables of those types. In our case, algebraic data types are used to represent the structure of the input tree, the type values and (possibly) the output tree. *Formulas* can then be used to give initial values to the variables or link them to each other. In a type system definition, initial values can be used to initialize the environment, e.g., by in-
Lines 3–6 contain declarations for the types Exp, Type and DType and variables of those types. The first defines possible expression constructs, in the form of a list of options separated by the “|” symbols. An expression in the listing above is, therefore, sumfold applied to three expressions, less applied to two expressions, dot applied to an identifier, an identifier, an integer or a real number. “E:Exp” means that variables with names consisting of E followed by zero or more digits are considered to be of type Exp. In the next line, the type Type defines three constants representing the basic types: bool for boolean values, int for integer values, real for real values and func for functions from lists of values of some type to a single value. As above, variables with names starting with T and followed by one or more digits are of the type T. We stress that while bool, int and real are sets of values, bool, int and real are constants representing the types of these values. In the last line, the type DType defines two constants describing continuous and behaviour respectively.

Lines 7–8 contain declarations for the types “SENV” and “DENV” and variables of those types. These declarations refer to the predefined “Map” type, and are used when referring to components from the environment.

Lines 10–33 contain the transition rules. The first two (Lines 10 and 12) are simple rules, without explicit preconditions. Implicitly, the types of the variables used restrict their values to integer and real constants, respectively. Because these rules are not affected by the environment, we do not mention it. Implicitly, this means the environment stays the same. In the case of the rules at Lines 14–16 and 18–20, the preconditions are based on the built-in lookup function. The lookup function is used to inspect the environment components, such as statenv and dynenv. Given a list of key-value pairs and a key that occurs in the one of the pairs, this function returns the corresponding value. If the key does not occur in the list of pairs, a special value “none” is returned. The rule in Lines 14–16 reads, thus, “If ID is known to have type T in the static type component of the environment, any reference to ID also has type T”; and the rule in Lines 18–20 “If ID is known to be a continuous real variable, then application of the dot operator results in a value of type real”. These rules both reference the environment. They gain access to the relevant information by extracting components from the label. If there are other components present in the environment which are inspected or changed by the rule, they remain unchanged, as indicated by the “---” marks.

The next transition rule (Lines 22–25) has four preconditions. The first precondition requires E1 to be of a valid type T1, i.e., transition E1 =|---> T1 to be possible. Similarly, the second precondition requires E2 to be of a valid type T2. The remaining preconditions use an auxiliary max function. Based on a partial order on types, max returns the larger of two types, if they are comparable. In the example we assume int < real. If the max function returns a type T3 and that type is not larger than real, we conclude that T1 and T2 are numerical types, and hence, E1 and E2 can be compared with the less operator.

The last transition rule (Lines 27–33) is an example of an even more complex rule, in this case of a fold based on the
addition operator. In its basic form, a fold is a higher-order function that repeatedly applies an operator to elements of a collection, resulting in an aggregate value. Chi 2.0 has a number of predefined folds, for a number of specific operators including addition. Additionally, Chi 2.0 folds have a filter step, that can be used to remove unwanted values from consideration, and a transformation step, that can be used to fold derived values, like the lengths of a collection of lists. When a fold expression is evaluated, it iterates over the collection represented by E1. The function represented by E2 is then applied to each value, and acts as a filter: if the result is “false” the value is discarded. Next, the function represented by E3 is applied to each value that passes the filter. The results of this function are combined into one value using the “addition” function, which is the result of the fold. When we look at folds from a typing perspective, this means that E1 needs to be of a collection type, so the fold can take elements from it. The “collection” predicate tests for this, while the “element” function retrieves the type of the elements of the collection. The next two components must have a function type that can accept the collection elements as parameters. Additionally, the function serving as a filter must return a boolean value, and the function that creates values to be combined must returns values that can be combined by, in this case, the “addition” operator. The type of the fold expression itself is the same as that type, as long as it is numeric.

4. TYPE CHECKER IMPLEMENTATION

Once the type system had been formally specified, the next step in the approach is the creation of a component to apply the type system to an AST. There are three general ways to obtain this component: the specification can be implemented by hand, executed by an interpreter or translated (compiled) to a different language already supported by an execution mechanism. Note that for the implementation of parsers all three options are used.

In our case study, we specifically want to experiment with type checker generation. We acknowledge that intrinsically, there is little reason to prefer an interpreted type checker over a generated implementation or vice versa. In order to make our transformation, we first needed to identify our source and target language, which should both be textual. As our source language we chose MSDF, as discussed in Section 3. This leaves the target language and the transformation technique to be determined.

4.1 Target language: Pyke

First, we choose our target language. Since the type checker has to be integrated in the entire suite of the Chi tools, it seemed sensible to choose the language that was used for those tools, i.e., Python. In addition to a smooth integration with other Chi tools, this allows us to benefit from a direct mapping of the MSOS labels to the concept of Maps native to Python. However, Python does not natively support the backtracking needed for cases when multiple transition rules can be applied. Rather than add this ourselves, we used Pyke [17].

Pyke is a Prolog-inspired inference engine operating on backwards-chaining rules that can interact directly with Python. Backwards-chaining rules are uniquely identified by means of labels and consist of the conclusion, indicated by use, and a (possible empty) set of premises, indicated by when. Identifiers can be used, e.g., to refer to rules during error reporting. Both the conclusion and each premise is a term.

During the derivation, the Pyke inference engine maintains a set of goals. For each goal the engine looks for a rule with the conclusion matching it. If such a rule is found, the goal is replaced by the set of premises corresponding to the conclusion. According to the result of the matching, variables appearing in the goal might be replaced by the actual values.

4.2 Transformation

To implement the transformation we opt for ASF+SDF [7]. ASF+SDF is a term-rewriting language successfully applied to implement program transformation and code generation in industrial cases [15, 32]. Alternative languages that could have been used are, e.g., Stratego [9] and Tom [5].

A crucial step in the process of generating the type checker is the ASF transformation that constructs the actual type checker code based on the MSDF specification. The result of the transformation is a set of Pyke rules, that, combined with the type-system-independent supporting Python code that provides support functions, like max, implement the type system as described in MSDF. The following sections describe how each of the four main constructs occurring in MSDF specifications is translated.

Imports. Recall that the main differences between MSDF and MSOS pertain to the introduction of modules and corresponding imports, and declarations. As a preprocessing step preceding the transformation we eliminate the imports by collecting all MSDF modules into a single specification. This step reduces the number of constructs that have to be handled and allows duplicate rules to be removed or combined before the transformation is carried out.

Formulas. One of the features that a Pyke knowledge base can have is an initialization section, a section of Python code where additional functions and variables can be declared and initialized. For example, if we wanted to provide a set of initial constants that can be used in the code, we could define the in the type system by using a formula to initialize a variable, called say “init-env”, with a map of constant names and their types. The transformation would then generate Python code in the initialization section that sets the variable to the desired value.

Declarations. Recall that declarations describe algebraic data types, and the main operation involving them is testing whether a given term matches a certain data type. When transforming MSDF declarations to Pyke we create a separate backwards-chaining rule for each case in a declaration. Hence, for the declaration in line 3–4 in Figure 2, we create six backwards-chaining rules. The rules created are similar and we illustrate only one of them, corresponding to dot(Id):
The first challenge required correctness of the tools supporting DSLs to be clear to the domain experts. We addressed this challenge by (1) separating the specification of the type checker (type system) from its implementation and (2) specifying the type checker in a formal language close to the popular SOS, well-known to the domain experts [5]. As SOS is a popular semantics specification approach for domain specific languages, e.g. Erlang [12] and GP [28], we believe that our approach can be applied to other domain specific languages as well.

The second challenge demands the approach to be suited for evolution, and the third one requires support of decision making by language designers. While these two challenges are quite different, our approach allows to address both of them in a uniform way. Recall that the main components of our approach are type system specification in MSOS and automatic generation of the type checker. In [8] we have shown that MSOS specifications are well-suited both for specification of type checking evolving languages and for decision making support. Automatic type checker generation allows to regenerate a type checker when the type system has evolved. Implementation of the transformation required merely 235 ASF+SDF transformation rules, amounting to 1174 lines of code. The MSDF grammar used has 115 production rules, and the Pyke grammar 280, 234 of which are part of an imported Python grammar. Hence there are only 46 Pyke-specific production rules.

Our final challenge required the generated tool to fit the existing tools available for the DSL. Since the entire suite of the Chi tools is implemented in Python, we have opted for generation of Pyke code.

6. LESSONS LEARNED

We have received a very positive feedback from the designers of Chi. The methodology applied in this paper has been chosen by them to support type checking of CIF [4], a new generation hybrid system specification language established by the members of the European network of excellence HYCON.

In addition to providing a component for the Chi 2.0 tool set, the effort described above aimed at practical evaluation of generative approach to type checker development.

- Design of an appropriate type system for Chi 2.0 required an iterative process. We expect that this is the case during the design for most DSLs. Choice for MSOS and code generation reduced the time needed to implement each iteration.
- Because we constructed the transformation that generates the code to implement the type checker from scratch, we were free to select a target language that suited our needs. By selecting Pyke, a knowledge engine based in the same language the other tools are written in, Python, we could benefit from logic programming capabilities without introducing a number of new dependencies on external software.
• Pyke is both well-suited for MSOS-based type checker generation due to the presence of rules, and for integration with other tools, due to closeness to Python. We expect other rule-based extensions of popular implementation languages such as, e.g., Jess [18] for Java and CLIPS for C, to be well-suited for MSOS-based type checker development as well.

However, we also noticed that smooth integration of the generated type checker with the Chi 2.0 tool set required the type checker to be aware of the representation of AST and DAST. Adding this knowledge to the transformation that generated the code was not a difficult problem, but it resulted in a transformation that cannot be easily reused for other projects and other target languages. A possible solution could be to investigate generic AST structures that can be used by a variety of DSLs at once.

7. RELATED WORK
Numerous ways of specifying a type system can be found in the literature. For example, type systems can be described using denotational semantics [10, 30]. Denotational semantics is based on functions that give the meaning or denotation of a program by linking inputs to the appropriate outputs. In this case, each node of an abstract syntax tree is linked to the appropriate type. Alternatively, [27] uses Structural Operational Semantics (SOS) to represent type systems. Unlike denotational semantics, operational semantics gives a meaning to terms by defining an abstract machine consisting of states connected by a transition relation. In SOS, the transition relation is defined by sets of rules allowing a step from one state to another if the rule’s preconditions are met. When used in type systems, the initial state is the AST, and the resulting state should be the desired type, or a DAST with type information in some or all nodes. Another form of operational semantics are evolving algebras. An evolving algebra consists of two parts, a partial, many-sorted algebra that describes the given program as the initial state, and a set of transition rules that describe transitions that allow state changes. Fundamentally similar in approach, evolving algebras mainly differ from SOS-based formalisms in that they have an imperative-programming style rather than the more set-theoretical style used by SOS. Industrial implementations of type systems are usually based on attribute grammars [1, 31, 15]. Attribute grammars consist of grammars where attributes have been defined for some or all nodes, usually in terms of other attributes. During type checking, attributes are evaluated as needed until the values for the desired attributes, like the type of an expression, is known. Finally, one can develop a special domain specific language for type systems [19].

We considered all the specification methods above as a basis for our type checker generator. Denotational semantics are clean and independent of implementation, but involve heavy mathematical machinery often surmounted by a highly specialized syntax [26]. Minor changes in the language might require a complete rewriting of the specification [23]. Since we target domain-specific languages we need a formalism that would be comprehensible for domain experts without background in formal modelling. As rapid evolution is not uncommon for domain-specific languages, the rigidity of the denotational semantics becomes a major problem. Attribute grammars are easier to understand, but focus on how to calculate type values, instead of on what the type should be. Moreover, attribute grammars usually require additional effort from the domain experts: even auxiliary data structures need to be implemented through production rules [16]. SOS is closer to natural reasoning, but can be very verbose and specifications tend to have high levels of coupling between rules. Finally, while developing a separate domain specific language is a valid option for type checker specification, the language developed cannot be reused for other forms of static semantic analysis. Moreover, application of such language requires a special learning effort from the domain experts. Evolving algebras share many of the problems above, in addition to being, in our opinion, less clear in general.

The idea of using a generative approach to construct type checkers can be found, e.g., in [22, 2]. Application of this idea to domain-specific languages, however, requires addressing the challenges stated in Section 1. These challenges render such approaches as [22, 2] not suited for domain-specific languages as these approaches were not conceived with the issues of language evolution and tool integration in mind. In particular, in [22], languages are divided into features. If two features are not related in a straightforward manner, extra effort is required to combine them. If a new feature is to be added to a languages that already has many features, this is obviously not desirable. TYPOL programs as described in [2] appear to suffer from the same problem, though the original paper does not state this explicitly.

8. CONCLUSIONS
In this paper we have presented our experiences with a novel approach to development of formally specified type checkers. We specified the type system for the specification language Chi 2.0 using MSOS and MSDF and generated a type checker in Pyke using ASF+SDF. MSOS is formal, well-suited for evolution and close enough to the popular SOS to be understood by the domain experts with limited programming experience. ASF+SDF allows us to generate a type checker and to reuse the generation process when the DSL evolves.

While we have focused on developing type checkers, we believe that our approach is applicable to other forms of static semantic analysis as well. As long as the desired static semantics can be specified using MSOS and MSDF, the implementation can be derived by means of a ASF+SDF generated transformation in a way similar to Section 4. For instance, because MSOS can handle dynamic semantics as well as static semantics, it should be possible to define partial evaluation problems, like constant propagation, in MSOS. Hence, one possible direction for future work consists in applying our approach to different kinds of static semantic analysis.

Another possible direction for future work consists in investigating alternative means of specification the type system. While MSOS has numerous advantages for specification of type systems while compared to such techniques as SOS, denotational semantics and attribute grammars (cf. Section 7), it still might be too complex for domain experts. One might
consider developing visual specification languages combining advantages of MSOS with intuitiveness of visual representation. Appropriateness of such a language should be investigated by means of comprehension studies.

References


